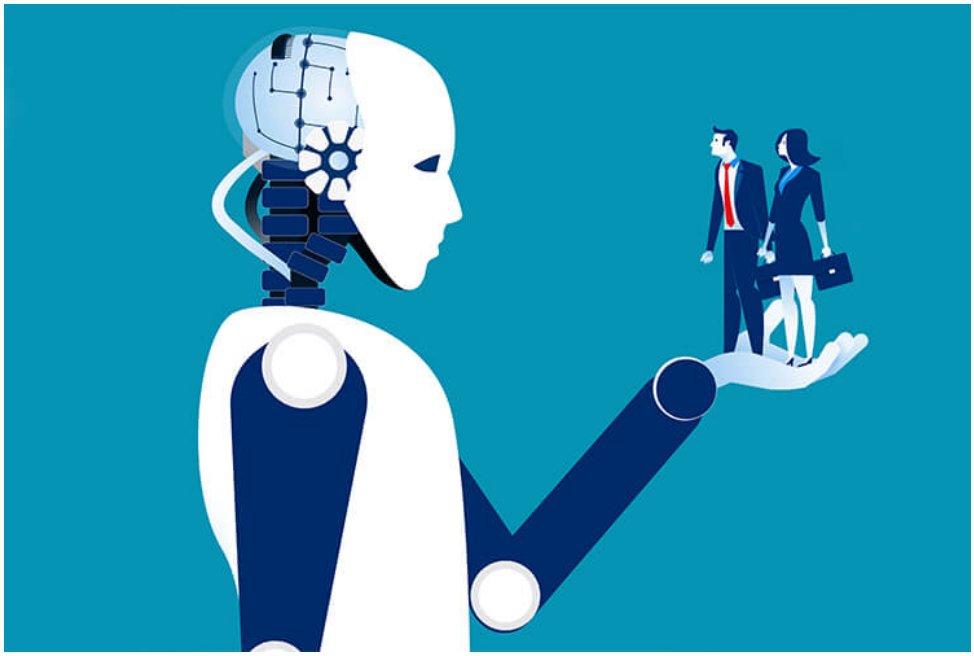


INTRODUKSJON TIL KUNSTIG INTELLIGENS

TDT4136

Mathilde Haugum

Dette er et kompendium for emnet TDT4136 Introduksjon til kunstig intelligens, holdt ved NTNU høsten 2019. Kompendiet er basert på boka *Artificial Intelligence – A modern Approach* av Stuart Russel og Peter Norvig.

Kapittel 1 – Introduksjon

Intelligensen er svært viktig for mennesker, og vi har lenge forsøkt å forstå hvordan vi tenker, altså hvordan en mengde materie kan oppfatte, forstå, forutsi og manipulere en verden som er mye større og mer kompleks. **Feltet for kunstig intelligens (AI) går videre, ved at det ikke bare forsøker å forstå intelligens, men også bygge intelligente enheter**. AI er et av de nyeste feltene innenfor vitenskap og engineering. Det omfatter en enorm mengde delfelt, slik som diagnostisering av sykdom, kjøring av biler, osv.

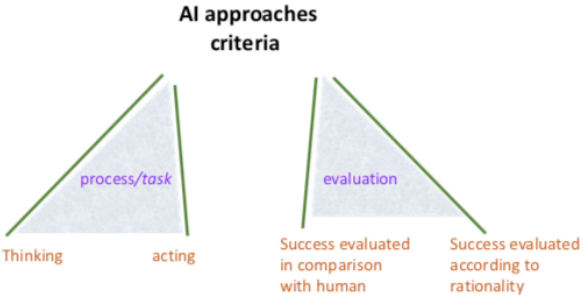
Hva er AI?

Det finnes ingen formell definisjon som dekker alle aspektene ved kunstig intelligens (AI). John McCarthy som er faren til begrepet *Artificial Intelligence*, definerte AI som «vitenskap og engineering forbundet med å lage intelligente maskiner, spesielt intelligente dataprogram. En lignende oppgave er å bruke datamaskiner for å forstå menneskelige intelligens, men AI trenger ikke å begrense seg til metoder som er biologiske observerbare. Intelligens er beregningene som er en del av evnen til å oppnå mål i verden. Mennesker, dyr og noen maskiner har ulik grad av intelligens». Tabellen under viser flere definisjoner.

Evaluering → ↓ Prosess	Menneskelig	Rasjonelt
Tenking	**Systemet tenker som mennesker** • Haugeland, 1985: «*Innsatsen for å få datamaskiner til å tenke ... maskiner med sinn, fullstendig og bokstavelig talt*» • Bellman, 1978: «*aktiviteter som vi assosierer med menneskelig tenking, slik som beslutningstaking, problemløsning, læring ...*»	**Systemet tenker rasjonelt** • Charniak, 1985: «*Studiet av mentale evner via bruken av beregningsmodeller*» • Winston, 1992: «*Studiet av beregninger som gjør det mulig å oppfatte, resonnere og handle*»
Handling	**Systemet handler som mennesker** • Kurzweil, 1990: «*prosessen av å skape maskiner som utfører funksjoner som krever intelligens når det utføres av mennesker*» • Rich & Knight, 1991: «*Studiet av hvordan man får datamaskiner til å gjøre ting, som mennesker fortiden kan gjøre bedre*»	**Systemet handler rasjonelt (= rasjonelle agenter)** • Poole, 1985: «*Beregningsintelligens er studiet av designet til intelligente agenter*» • Bellman, 1978: «*AI ... handler om intelligent oppførsel i gjenstander*»

Som vi kan se på tabellen, kan definisjonene av AI deles inn basert på to dimensjoner:

1. **Prosess** – ser på hvilke oppgaver definisjonen sier at AI system utfører. Definisjonene plasseres i toppen dersom oppgavene involverer tankeprosesser, mens de plasseres i bunnen hvis de involverer handling/oppførsel.
2. **Evaluering** – ser på hvordan definisjonen vurderer suksessen til AI system. Definisjonene plasseres til venstre hvis vurderingen er basert på gjengivelse av menneskelig ytelse, mens de plasseres til høyre hvis det er basert på **rasjonalitet**.



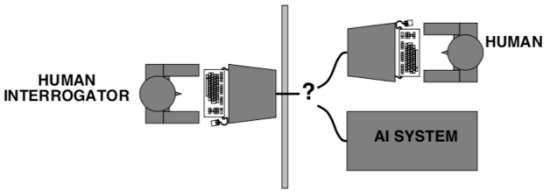
Rasjonalitet er evnen til å gjøre «det riktige» gitt det som er kjent. Det er altså et mål på ideell ytelse. En robot som er laget for å krysse veien uten å krasje, vil for eksempel være rasjonell hvis den kun passerer ved grønt lys. Hvis et helikopter faller ned og krasjer i roboten, vil den fortsatt være rasjonell, fordi det er ikke forventet at en rasjonell agent skal ta hensyn til alt. Det er ikke rasjonelt for roboten å sjekke om helikopter flyger i området før den krysser en vei. Det som er kjent er tilstanden til trafikklyset og posisjonen til roboten.

Basert på disse definisjonene, kan man si at det finnes to aspekter ved AI:

1. **Empirisk vitenskap** – involverer observasjon og hypoteser om menneskelig oppførsel, og brukes av menneskesentrerte definisjoner av AI. Man forsøker å forstå intelligente enheter ved å utvikle teorier som prøver å forklare og forutse egenskapene til disse enhetene. Funksjoner ved den menneskelige hjernen blir modulert og man forsøker å forklare ulike typer intelligens.
2. **Engineering og matematikk** – brukes av rasjonellsentrerte definisjoner av AI. Ekte problemer blir løst ved å bruke ideer om hvordan kunnskap kan representeres og brukes. Det lages intelligente enheter med intelligent oppførsel.

Vi ser nærmere på de fire tilnærmingene til kunstig intelligens.

1. Handle menneskelig – Turing Test tilnærming



En datamaskin har kunstig intelligens dersom den består Turing test (1950), der en menneskelig forhører stiller noen spørsmål og ikke klarer å avgjøre om svaret kommer fra en person eller en datamaskin. Turing test er altså en operasjonell test for intelligent oppførsel, siden det sjekker om maskiner kan oppføre seg intelligent. For å passere denne testen, må datamaskinen kunne prosessere naturlig språk (kommunisere), lagre kunnskap, resonnere (svare på spørsmål) og ha maskinlæring (tilpasning). Total turing test vurderer også sanseinntrykk og krever at datamaskinen har syn (oppfatter objekter) og robotikk (bevegelse). I dag blir denne testen lite brukt, siden det blir sett på som viktigere å studere underliggende prinsipper ved intelligens, istedenfor å kopiere et eksempel.

young woman: Men are all alike.
eliza: In what way?
young woman: They're always bugging us about something specific or other.
eliza: Can you think of a specific example?
young woman: Well, my boyfriend made me come here.
eliza: Your boyfriend made you come here?
young woman: He says I'm depressed much of the time.
eliza: I am sorry to hear you are depressed.
young woman: It's true. I'm unhappy.
eliza: Do you think coming here will help you not to be unhappy?
young woman: I need some help, that much seems certain.
eliza: What would it mean to you if you got some help?
young woman: Perhaps I could learn to get along with my mother.
eliza: Tell me more about your family.
young woman: My mother takes care of me.
eliza: Who else in your family takes care of you?
young woman: My father.
eliza: Your father.

Eliza er en datapsykoterapeut laget av Weizenbaum i 1966, og det er et eksempel på en maskin som handler menneskelig. Eliza fungerer ved å vri pasientenes uttalelse tilbake til dem, slik en ikke-direktiv psykoterapeut gjør. Maskinen forstår ikke egentlig påstandene til subjektet, siden det bare er en illusjon. Dette ble likevel ikke oppfattet av mange brukere. Et annet eksempel på en maskin som handler menneskelig er Sofia, som er en humanoid robot laget i Hong Kong i 2016.

2. Tenke menneskelig – Kognitiv modellering tilnærming

For å kunne avgjøre om et program tenker som et menneske, må vi første bestemme hvordan mennesker tenker. En slik **kognitiv modellering** kan gjøres på tre ulike måter: (1) introspeksjon, der man forsøker å fange egne tanker ettersom de kommer, (2) psykologiske eksperimenter, der man observerer en person i aksjon og (3) hjerneavbildning, der man ser hjernen i aksjon. Når man har en presis teori om minnet, kan teorien uttrykkes som et dataprogram. **Et dataprogram som er basert på teori om menneskelig tankeprosess, har kunstig intelligens dersom det har input-output oppførsel som matcher korresponderende menneskelig oppførsel.** Et eksempel er G.P.S. (General Problem Solver, 1961), som er laget slik at resonneringen til maskinen følger resoneringen til menneskelige subjekter som løser samme problem.

Den **kognitive revolusjonen** på 1960-tallet markerte overgangen fra behaviorisme til en informasjonsprosesserende psykologi, som forsøker å forklare at intern aktivitet i hjernen er et resultat av beregninger. En tilnærming er kognitiv vitenskap, som prøver å forutsi og teste oppførselen til mennesker (top-down). En annen tilnærming er kognitiv nevrovitenskap, som bruker nevrologisk data for å identifisere hjerneaktivitet (bottom-up). Begge disse er

forskjellig fra AI, men likheten er at de tilgjengelige teoriene foreløpig ikke forklarer eller fremkaller noe som ligner generell intelligens på menneskenivå.

3. Tenke rasjonelt – «Laws of thought» tilnærming

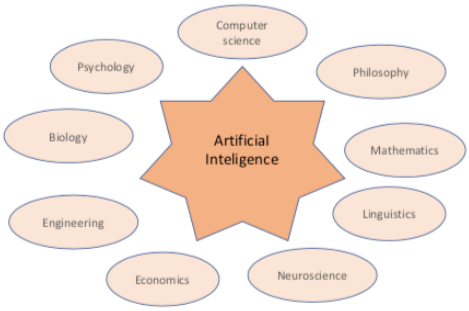
Aristotle formaliserte «rett tenking» ved å bruke en matematisk modell som kalles syllogisme. En syllogisme gir argumentstruktur som alltid gir riktig konklusjon, hvis den gis riktige premisser (eks: Sokrates er mann, alle menn er dødelige; derfor er Sokrates dødelig). Disse kalles ***Laws of thought*,** og de skulle styre driften av sinnet. Studiet av disse ble starten på ulike former for **logikk**, som er notasjon og utledningsregler for tanker. På 1900-tallet ble det utviklet en presis notasjon for påstander om alle typer objekter i verden og relasjonene mellom disse. I 1965 ble det laget program som kunne løse ethvert problem som var beskrevet i logisk notasjon (evig loop hvis ingen løsning eksisterer). **Et dataprogram har kunstig intelligens dersom det er bygget på slike program som kan løse alle problem gitt med logisk notasjon**. Dette markerer en direkte linje fra matematikk og filosofi til moderne AI.

Å tenke rasjonelt involverer å bruke logikk for å utlede korrekte slutninger. Denne tilnærmingen mener at AI system vil logisk resonnere til en konklusjon om at en gitt handling vil oppnå målet, og deretter handle basert på dette. Problemer med denne tilnærmingen er at det er komplisert å oversette fra uformell til formell kunnskap, noe intelligent oppførsel kan ikke formidles logisk og det er forskjell mellom å løse et problem i prinsipp og i praksis (systemet trenger veiledning i hvilke resonneringssteg som skal brukes først).

4. Handle rasjonelt – Rasjonell-agent tilnærming

En **agent** er noe som handler. En **rasjonell agent** er en agent som handler slik at den oppnår beste mulige utfall, eller beste forventede utfall ved usikkerhet. **Rasjonell handling er altså å gjøre den «riktige tingen» som er forventet å maksimere måloppnåelsen, gitt tilgjengelig informasjon og beregningsevner**. Rasjonell tenking kan være en del av å handle rasjonelt, siden rasjonell handling kan begynne med en logisk resonnering for å finne ut hvordan måloppnåelsen kan maksimeres. **Rasjonell tenking er likevel ikke en absolutt betingelse for rasjonell handling**, for det finnes situasjoner der rasjonell handling involverer ingen tenking (eks: reflekser). Et system som passerer Turing Test vil være en rasjonell agent, pga. egenskapene som kreves av testen.

En fordel med rasjonell-agent tilnærmingen er at den er **mer generell** enn *Laws of thought* tilnærmingen, siden korrekte slutninger er kun én av flere mulige mekanismer for å oppnå rasjonalitet. En annen fordel er at den er **mer mottagelig** til vitenskapelig utvikling enn tilnærmingene basert på menneskelig handling eller tenking. Dette skyldes at standarden for rasjonalitet er matematisk vell definert og fullstendig generell, mens menneskelig oppførsel og tenking kan påvirkes av flere faktorer. Perfekt rasjonalitet, dvs. alltid gjøre «det riktige», er ikke mulig å oppnå i kompliserte omgivelser, fordi det gir for høye krav på beregningene. Det er likevel vanlig å anta perfekt rasjonalitet som en god start for analysen.



Grunnlaget til kunstig intelligens

Datavitenskap (*computer science*) er det viktigste fagfeltet som ligger til grunn for AI, men som vi kan se på figuren er det flere fagfelt som har bidratt med ideer, synspunkter og teknikker:

- **Filosofi** – har bidratt med logikk, resonneringsmetoder, synet av minnet som fysisk system (følger logiske regler og

manipulerer kunnskap), metoder for å tilegne seg kunnskap, kobling mellom kunnskap og handling, språk og rasjonalisme (resonnering for å forstå verden).

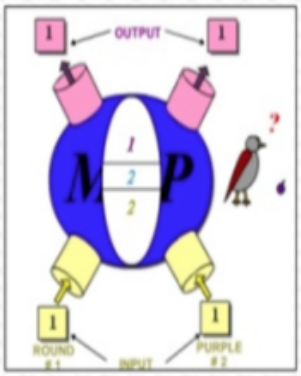
- **Matematikk** – har bidratt med formalisering av logikk og utregninger, og utformingen av algoritmer, teoremer og sannsynlighet.
- **Økonomi** – har bidratt med formell teori om hvordan man kan ta rasjonelle avgjørelser i ulike omgivelser, og hvordan nytte kan vurderes.
- **Nevrologi** – har bidratt med kunnskap om hvordan hjernen prosesserer informasjon, hvilken funksjon nevroner har og hvordan synapser fungerer som læremekanisme.
- **Psykologi** – har bidratt med beskrivelser av hvordan persepsjon (oppfattelse) og motorisk kontroll foregår, og utformingen av kognitiv psykologi (hjernen er en informasjons-prosesserende enhet) og kognitiv vitenskap (hjerneaktivitet skyldes beregninger)
- **Datavitenskap** – har bidratt med teknologi, programvare og maskinvare som er nødvendig for å bygge effektive datamaskiner
- **Kontrollteori og kybernetikk** – har bidratt med utvikling av gjenstander som kan kontrollere egne handlinger (endre oppførsel pga endringer i omgivelsen). Blant annet homeostatiske enheter (feedback brukes for å oppnå stabil adaptiv oppførsel), systemer som maksimerer objektiv funksjon, enkle optimale agenter, osv.
- **Lingvistikk** – har bidratt med definisjoner av forholdet mellom språk og tankeprosesser, kunnskapsrepresentasjon (hvordan kunnskap skal representeres slik at datamaskinen kan resonnere med den), grammatikk, syntaks og semantikk.

Det har lenge vært uenigheter om maskiner kan være intelligente. *Symbolic system hypothesis,* av Newell og Simon, sier at intelligens er substratnøytral, slik at alle fysiske symbolsystem har de nødvendige og tilstrekkelige midlene for generell intelligent handling. Motparten er *Biological substrate only,* av Searle, som sier at intelligens er substratavhengig og at materialet mennesket er laget av er fundamentalt for vår intelligens. Tenking er derfor kun mulig i spesielle maskiner som er levende og lagd av protein.

Historien til kunstig intelligens

Følgende er den historiske utviklingen av AI:

- **Svangerskapet til AI (1943)** – det første arbeidet innenfor AI ble utført av McCulloch og Pitts i 1943, når de lagde en boolean kretsmodell av hjernen vha kunstige nevroner. Målet var å forstå hvordan hjernen bruker nevroner for å produsere komplekse tanker. De lagde MCP nevroner og plasserte disse i et nettverk som kunne overføre informasjon via av/på-mekanismer. Nevronene kunne «skrus på» hvis de ble stimulert av tilstrekkelig antall nabonevroner. De viste at et nettverk av nevroner kunne beregne alle beregnbare funksjoner og implementere alle logiske forbindelser (eks: AND, OR). De foreslo også at slike nettverk kunne lære. Et eksempel er «*Bird example*», der en fugl vil bestemme om den skal spise et objekt eller ikke basert på to input om objektet: form (rund eller ikke) og farge (lilla eller ikke). Fuglen vil kun spise objektet hvis den er rund og lilla. MCP nevronet har to input (begge er 0 eller 1) og har en terskelverdi T, og output er en funksjon av disse (se figurer).



IF(sum(inputs))>= T
THEN Output=1

Object	Purple?	Round?	Eat?
Blueberry	1	1	1
Golf ball	0	1	0
Violet	1	0	0
Hot Dog	0	0	0

- **Turings bidrag (1950)** - Turing utførte mye arbeid som hadde stor påvirkning på AI-feltet, for eksempel skrev han artikkelen *Computer Machinery and Intelligence,* der han introduserte blant annet Turing testen og maskinlæring.

- **Fødselen av AI (1956)** – John McCarthy overbeviste Minsky, Shannon og Rochester til å bli med å arrangere et forskningsprosjekt ved Dartmouth College sommeren 1956, og dette ble starten på AI feltet. Formålet med prosjektet var å «ta utgangspunkt i antagelsen om at alle aspekter ved læring eller andre trekk ved intelligens kan i prinsippet beskrives så presist at man kan lage en maskin som simulerer det. Det vil gjøres et forsøk på å finne ut hvordan man kan lage maskiner som bruker språk, danner abstraksjoner og konsepter, løser problemer som for tiden kun kan løses av mennesker og forbedre seg selv». Dette prosjektet førte ikke til noen nye gjennombrudd, men hovedpersonene, som senere fikk stor påvirkning på AI-feltet, ble introdusert med hverandre. AI ble et separat felt, og ikke en del av kontrollteorien, operasjonsforskning, matematikk, osv. fordi til forskjell fra andre fagfelt har AI følgende egenskaper: (1) det kopierer menneskelige evner (eks: språkbruk og selvforbedring), (2) det er en forgreining av datavitenskap og (3) det forsøker å bygge maskiner som fungerer automatisk i komplekse omgivelser.

- **Tidig entusiasme, store forventninger (1952-1969)** – de første årene med AI var fulle med suksess. Datamaskiner ble tidligere sett på noe som bare kunne utføre aritmetikk og ikke noe mer, så det var derfor forbløffende når datamaskiner gjorde noe som helst smartere. I denne perioden ble det laget mange tidlige AI program, inkludert Samuels *Checkers program* (spilte dam), Newell og Simons *Logic Theorist* (resonneringsprogram) og G.P.S. (problemløser som tenker menneskelig) og Gelernters *Geometry Engine* (kunne bevise teorem). Newell og Simon formulerte *physical symbol system* hypotesen som sier at ethvert intelligent system opererer ved å manipulere datastrukturer som består av symboler. Denne hypotesen ble grunnlaget for det meste av suksessen til AI på 70- og 80-tallet. I 1958 utviklet John McCarthy Lisp, som ble det dominerende programmeringsspråket for AI de neste 30 årene. Han beskrev også det første fullstendige AI systemet, kalt *Advice Taker*, som brukte generell kunnskap om verden for å løse problem. Beskrivelsen viste at det er nyttig å ha en formell, eksplisitt representasjon av verden og evnen til å manipulere denne representasjonen med en deduktiv prosess (logisk resonnering med påstander for å nå konklusjon). I 1962 ble *perceptron convergence theorem* introdusert av Rosenblatt for å beskrive trening av enkle nevrale nettverk. I 1965 kom Robinson med en fullstendig algoritme for logisk resonnering

- **Skuffelsen (1966-1973)** – den lovende ytelsen til tidlige AI system, gjorde at man fikk en overdreven illusjon om fremtiden til AI. De fleste av disse AI systemene mislyktes når de ble testet på flere og vanskeligere problem. Et problem var at de fleste systemene visste ingenting om deres fagstoff, for eksempel maskiner som oversatte språk, hadde ingen bakgrunnskunnskap som trengs for å bestemme innholdet til noen setninger (eks: skjerp deg). Et annet problem var at AI systemene løste ikke-sporbare problem ved å prøve ut ulike kombinasjoner helt til en løsning ble oppdaget. Dette fungerte for mikroverdener som inneholdt få objekter, men ble vanskeligere når størrelsen økte. Man trodde at det handlet om å utvikle raskere maskinvare og større minne, men innså etterhvert at selv om programmet i prinsippet kan finne løsningen, betyr ikke det at programmet inneholder mekanismer som trengs for å finne løsningen i praksis. Et tredje problem var at det ble oppdaget begrensninger i strukturene som ble brukt for å generere intelligent oppførsel. Det ble vist at et perceptron kunne representere svært lite informasjon, noe som resulterte i at nesten all forskning på nevrale nettverk ble avsluttet.

- **Utvikling av kunnskapsbaserte system (1969-1979)** – de tidlige AI programmene var søkemekanismer som forsøkte å koble sammen elementære resonneringssteg for å finne en fullstendig løsning. Dette kalles svake metoder, siden de ikke kan løse større eller vanskeligere problem. Alternativet er å bruke mer kraftfull og domenespesifikk kunnskap som tillater større resonneringssteg og håndterer smalere områder. For å løse et vanskelig problem, må man nesten vite svaret allerede. Et eksempel på dette er DENDRAL programmet som ble laget i 1969, for å utlede molekylstrukturen fra informasjonen gitt av et massespektrometer. Systemet løste problemet ved å bruke den teoretiske kunnskapen som var formulert som regler. Dette ble starten på såkalte ekspertsystem, og det ble senere utviklet flere slike system (eks: MYCIN som diagnostiserer blodinfeksjoner vha 450 regler). Det ble en større vekt av applikasjoner som var rettet mot problemer i den virkelige verden.

- **AI blir industri (1980-)** – R1 ble laget i 1982 av McDermott, og det var det første suksessfulle, kommersielle ekspertsystemet. Det hjalp til med å konfigurere bestillinger av nye datamaskiner, og i 1986 bidro det med å redusere kostnader med 40 millioner dollar per år. Ekspertsystemene gjordet at AI industrien fikk en oppsving, og i 1988 hadde nesten alle store U.S. selskap egne AI grupper som brukte eller undersøkte slike system.

- **AI vinteren (1988-1993)** – mange selskap feilet ettersom de ikke klarte å tilfredsstille de ekstravagante løftene forbundet med ekspertsystemene.

- **Retur av nevrale nettverk (1986-)** – *back-propagation* lærealgoritmen ble først utviklet i 1969 av Bryson og Ho, og den ble gjenoppfunnet på midten av 80-tallet. Algoritmen ble brukt på mange læringsproblem i datavitenskap og psykologi, og resultatet ble spredt og førte til en ny begeistring for nevrale nettverk.

- **AI bruker vitenskapelig metode (1987-)** – i senere tid har det vært en revolusjon for innholdet og metodologien ved arbeid innenfor AI. Det har blitt mer vanlig å bygge på eksisterende teorier istedenfor å foreslå nye, basere påstander på grundige teorem eller eksperimentelle bevis istedenfor intuisjon og vise relevans til bruksområder i den virkelige verden istedenfor lekeeksempler (mikroverdener). AI ble etablert som en motstand mot begrensninger innenfor eksisterende felt som kontrollteori og statistikk, men denne isoleringen av AI ble fjernet. I 1995 ble det bestemt at AI skal bruke den vitenskapelige metoden, der hypoteser må utsettes for strenge, empiriske eksperimenter og resultatet må statistisk analyseres for at de skal aksepteres. Det skal være mulig å gjenta eksperimentet ved å bruke delt testdata og kode. I 1988 kom Pearl med artikkelen *Probabilistic Reasoning in Intelligent Systems*, og den førte til en ny akseptanse for statistikk og beslutningsteori innenfor AI.

- **Bruk av intelligente agenter (1995-)** – «*agents, agents, everywhere...*». Forskere begynte å se på agentperspektivet igjen, og Newell, Laird og Rosenbloom lagde SOAR som er en fullstendig agentarkitektur. Internett er et av de viktigste omgivelsene for intelligente agenter. AI teknologi er svært vanlig i nettbaserte applikasjoner, slik som søkemotorer og nettsideaggregatorer (samler relatert innhold og gir lenker til dette). En konsekvens av agentperspektivet er at tidligere isolerte delfelt av AI må reorganiseres for at deres resultat skal kobles sammen (eks: resonneringssystem må kunne håndtere usikkerhet, siden sansesystem ikke kan gi helt pålitelig informasjon om omgivelsene). En annen konsekvens er at AI får tettere kontakt med andre felt som håndterer agenter, slik som kontrollteori og økonomi.

- **Human-level AI tilbake på agendaen (2003)** – noen av grunnleggerne av AI, inkludert McCarthy og Minsky, mente at AI burde fokusere mindre på å skape forbedret versjoner av applikasjoner som er gode til spesifikke oppgaver (eks: kjøre en bil). I stedet mener de at AI bør returnere til røttene og lage maskiner som tenker, lærer og skaper. Denne innsatsen kalles human-level AI (HLAI) og vil kreve svært store kunnskapsbaser.

- **Tilgang til store datasett (2001-)** – gjennom historien til datavitenskap har fokuset vært på algoritmene, men i nyere arbeid innenfor AI har det blitt foreslått at for mange problem bør man heller fokusere på dataen. Dette skyldes en stadig større tilgang til svært store datakilder (eks: billioner med bilder på nettet). For eksempel har det blitt vist at en middelsgod algoritme med 100 millioner ord er bedre til å finne entydig betydning av ord enn den beste algoritmen med 1 millioner ord. Et problem innenfor AI er hvordan man skal uttrykke all kunnskap systemet trenger, noe som kalles kunnskapsflaskehals. Slike system indikerer at dette problemet kan løses med å gi nok data til læringsalgoritmer istedenfor håndkodet kunnskapsteknikk.

- **Ny skuffelse (2005-2010)** – AI systemene klarer ikke å møte forventningene og skuffer dermed igjen. Det er ikke mye verdsettelse for AI. Samtidig er det en oppsving av nye applikasjoner og mange AI applikasjoner er innebygd i infrastrukturen til all industri. Dette kan tyde på at AI vinteren kan gi etter for en ny vår.

Hovedparadigmer

Vi skiller mellom to hovedparadigmer innenfor AI:

1. **Good old fashioned AI (GOFAI)** = det meste av suksessen til AI på 70- og 80-tallet skyldes forskning basert på *Physical Symbol System* hypotesen til Newell og Simon, som sier at et fysisk symbolsystem har de nødvendige midlene for generell intelligent handling. Intelligens blir sett på som manipulering av symboler, og det spiller ingen rolle hvilket fysisk medium (eks: hjerne, papir eller datamaskin) som står bak manipuleringen. GOFAI har derfor et fokus på symbolske representasjoner, som kan representere situasjoner i den virkelige verden (eks: en blokk-verden). Kritikerne hadde følgende å si om GOFAI:
    i. Searle: man kan ikke si at et program (eller ethvert fysisk symbolsystem) forstår symbolene som det bruker, altså symbolene har ingen mening for maskinene.
    ii. Brooks: våre mest grunnleggende egenskaper innenfor bevegelse, overlevelse, persepsjon, osv. ser ikke ut til å kreve høynivå symboler, og bruken av slike symboler er mer komplisert og mindre suksessfull.
    iii. Harnad: definerte *symbol grounding problem*, som viser at en agent ikke oppfatter symboler, men at hjernen heller omdanner sanseinput til høyrenivå abstraksjoner (eks: symboler).
2. **Situated embodied AI (SEAI)** = tilnærmingene innenfor GOFAI viste seg å være skjøre og svært lite robuste når de ble brukt på problemer i den virkelige verden. Det viste seg å være vanskelig å definere en modell av verden, og Brooks kom med påstanden om at verden er dens egen beste modell. SEAI fokuserer på å ha en kropp, med blant annet motorikk, som er plassert i en fysisk omgivelse. Dette involverer nevrale nettverk, genetiske algoritmer, osv. og er ikke dekket av dette emnet.

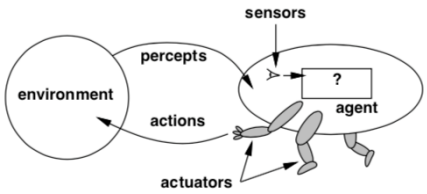
AI i dag

Det er vanskelig å gi et presist svar på hva AI kan gjøre i dag, fordi det er så mange aktiviteter i så mange delfelt. Noen eksempler på bruksområder er robotiserte kjøretøy (førerløs Google robotbil), talegjenkjenning (håndterer booking av flybilletter over telefonen), videospill (DEEP BLUE slo verdensmesteren i sjakk), bekjempelse av spam, maskinoversettelse (automatisk oversettelse fra arabisk til engelsk).

Kapittel 2 – Intelligente agenter

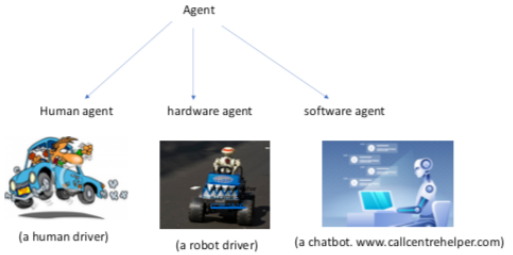
Rasjonelle agenter er sentrale i noen tilnærminger til kunstig intelligens, så i dette kapittelet skal vi se på ulike rasjonelle agenter som opererer i forskjellige omgivelser.

Agenter og omgivelser



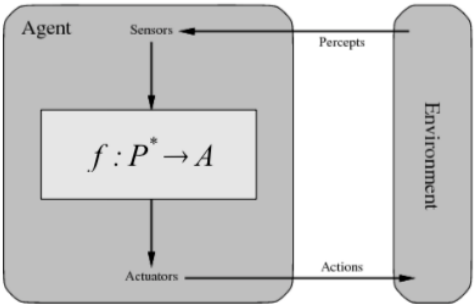
En agent er alt som kan oppfatte omgivelsene gjennom sensorer og handle på omgivelsene vha. aktuatorer (se figur). Det finnes flere ulike typer agenter:

- **Menneskelig agent** – sensorer er øyer, ører og andre organer, mens aktuatorer er hender, føtter, stemmebånd og andre kroppsdeler.
- **Robotagent** = sensorer kan være kameraer, infrarøde avstandsmålere, osv., mens aktuatorer er ulike motorer.
- **Programvare agent** = sensorer er tastatur, leser av filinnhold og mottaker av nettverkspakker, mens aktuatorer er skjermer, skriving av filer og sending av nettverkspakker.



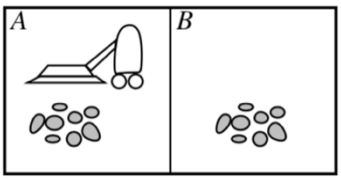
En persepsjon er det agenten oppfatter ved et bestemt øyeblikk, mens en perseptsekvens er alt agenten noensinne har oppfattet opptil det øyeblikket. Hvilken handling agenten velger ved et tidspunkt, kan avhenge av hele persepsjonssekvensen, men ikke av noe som ikke er oppfattet av agenten. **Agentens oppførsel beskrives av agentfunksjonen, som kartlegger enhver perseptsekvens (P) til en handling (A):**

$$f: P \rightarrow A$$



Kartleggingen til agentfunksjonen kan plasseres i en tabell, ved å prøve alle mulige perseptsekvenser og deretter se hvilken handling agenten utfører i respons til bestemte sekvenser (se figur 1). Agentfunksjonen viser altså hvordan de ulike perseptsekvensene fører til bestemte handlinger. **Agentprogrammet er den konkrete implementasjonen som kjører på en fysisk arkitektur for å produsere agentfunksjonen**. Det er viktig å skille mellom disse to begrepene; agentfunksjonen er en abstrakt matematisk beskrivelse, mens agentprogrammet er en implementasjon som kjører i et fysisk system.

Eksempel – Støvsugerverden



Støvsugerverden har to lokasjoner: firkant A og firkant B (se figur). Støvsuger-agenten kan oppfatte hvilken firkant den befinner seg i og om det er skittent i firkanten. Den kan velge å bevege seg til venstre eller høyre, suge opp skitten eller gjøre ingenting. Et eksempel på en agentfunksjon er: «Hvis nåværende firkant er skitten, sug; ellers flytt til annen firkant». Figur 1 viser en delvis tabell av denne agentfunksjonen, mens figur 2 viser agentprogrammet som implementerer den.

Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
⋮	⋮

Figur 1 - tabell for agentfunksjonen

```
function Reflex-Vacuum-Agent( [location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Figur 2– agentprogrammet som implementerer agentfunksjonen

God oppførsel – konseptet for rasjonalitet

En rasjonell agent bør velge handlingen som er forventet å maksimere ytelsesmålet, gitt en perseptsekvens og eventuell innebygd kunnskap. Det betyr at alle oppføringene til høyre i tabellen på forrige side er korrekte. Når en agent plasseres i en omgivelse, vil den generere en handlingssekvens basert på det den oppfatter. Denne sekvensen med handlinger gjør at omgivelsene går igjennom en sekvens av tilstander. Hvis denne tilstandssekvensen er ønskelig, så har agenten oppført seg bra og kan kategoriseres som en rasjonell agent. **Et ytelsesmål brukes for å evaluere sekvensen av tilstander omgivelsen går igjennom. En rasjonell agent vil utføre en handlingssekvens som maksimerer dette ytelsesmålet**.

Ytelsesmålet formes av målene ved designet, og det kan være vanskelig å bestemme et passende ytelsesmål for omgivelsene. For eksempel i støvsugerverden kan ytelsesmålet være mengde skitt som blir sugd opp i løpet av en periode. En rasjonell agent kan dermed maksimere dette ytelsesmålet ved å suge opp skittet, droppe det på gulvet, suge opp skittet, droppe det på gulvet, osv. Et bedre ytelsesmål vil belønne agenten for å ha et rent gulv, for eksempel ved å gi et poeng for hver ren firkant ved hvert timestep. **Det er bedre å designe ytelsesmål etter hva man ønsker i omgivelsene, enn basert på hvordan man tror agenten bør oppføre seg.** Ytelsesmålet kan også ta hensyn til tid, strømforbruk, støy, osv.

Rasjonalitet

Hva som er rasjonelt ved et tidspunkt vil avhenge av PEAS:

- **P.** ***Performance measure*** = ytelsesmålet som definerer suksesskriteriet
- **E.** ***Environment*** = agentens kunnskap om omgivelsen
- **A.** ***Actuators*** = handlingene agenten kan utføre vha. aktuatorer
- **S.** ***Sensors*** = agentens perseptsekvens som følge av det den oppfatter via sensorer

Dette gir følgende definisjon av en rasjonell agent:

> «For hver mulige perseptsekvens, vil en **rasjonell agent** velge handlingen som er forventet å maksimere ytelsesmålet, gitt bevisene som leveres av perseptsekvensen og enhver innebygd kunnskap agenten har»

Eksempel - støvsugerverden

Når vi skal avgjøre om støvsugeragenten er rasjonell, må vi først bestemme PEAS:

- **Ytelsesmålet**: ett poeng gis for hver ren firkant ved hvert timestep, over en levetid på 1000 timesteps.
- **Kunnskap om omgivelsene**: støvsugeren kjenner geografien til omgivelsene (A og B), men fordelingen av skitt og den initiale lokasjonen til agenten er ukjent. Rene firkanter forblir rene og suging vil rengjøre nåværende firkant. Handlingene *Left* og *Right* vil bevege agenten, så lenge det ikke fører til at agenten beveger seg utenfor omgivelsen.
- **Tilgjengelige handlinger**: de eneste tilgjengelige handlingene er *Left*, *Right* og *Suck*
- **Perseptsekvens**: agenten kan oppfatte sin lokasjon og om lokasjonen er skitten

Under disse verdiene for PEAS vil støvsugeragenten være rasjonell. Hvis ytelsesmålet inkluderer straff på ett poeng per bevegelse, vil støvsugeragenten være irrasjonell. Dette skyldes at den beveger seg mellom de to lokasjonene etter all skitten er rengjort og får dermed dårlig ytelsesmål. Under disse forholdene ville en rasjonell agent gjort ingenting når den er sikker på at alle firkantene er rene. Dersom rene firkanter kan bli skitten igjen, må den rasjonelle agenten av og til sjekke firkantene og rengjøre dem på nytt etter behov. Hvis geografien er ukjent må agenten i tillegg utforske omgivelsen. **Dette illustrerer hvordan rasjonalitet avhenger av PEAS**.

Allvitenhet, læring og autonomi

Det er viktig å skille mellom rasjonalitet og allvitenhet. En allvitende agent vet utfallet av sine handlinger og handler deretter, men **allvitenhet er umulig i realiteten**. I eksempelet med roboten på side 2, vil en allvitende robot vente med å krysse gaten, fordi den vet at et helikopter vil krasje på gaten. En robot som krysser gaten og krasjer med helikopteret kan fortsatt være rasjonell, fordi en rasjonell agent skal maksimere *forventet* ytelse og ikke *faktisk* ytelse. Rasjonalitet krever altså ikke allvitenhet, siden rasjonelle valg kun avhenger av perseptsekvensen ved det øyeblikket valget tas. Samtidig må man **sikre at perseptsekvensen inneholder nødvendig informasjon**. For eksempel må agenten se til venstre og høyre før den krysser gaten. Det er også viktig med **informasjonssamling, der handlingene utføres i en bestemt rekkefølge** og **roboten utfører eventuell utforskning av omgivelsene**, slik at perseptsekvensen blir riktig modifisert i fremtiden. For eksempel bør roboten se seg rundt før den krysser gaten. Dette vil hjelpe til med å maksimere forventet ytelse.

Det er også forventet at en rasjonell agent skal kunne lære av det den oppfatter. Agenten kan ha noe initial kunnskap om omgivelsene, men etterhvert som den får erfaring kan denne modifiseres og utvides. Noen agenter har fullstendig initial kunnskap om omgivelsene og trenger derfor ikke å oppfatte eller lære. Slike agenter er skjøre, for eksempel kan vi se på møkkbillen. Etter å ha gravd et rede og lagt sine egg, vil den hente en møkkball for å tette hullet. Hvis ballen fjernes på vei til redet, vil møkkbillen fortsette ruten og tette hullet med en ikke-eksisterende ball. Den oppfatter aldri at noe mangler, fordi evolusjonen har plassert en antagelse i oppførselen til billen og når den brytes resulterer det i mislykket oppførsel. Et annet eksempel er sphex veps, som vil flytte en larve til redet og deretter sjekke redet. Dersom man flytter på larven, mens vepsen sjekker redet kan man få vepsen til å gjenta samme prosess flere ganger. Den klarer ikke å lære at den initiale planen feiler og vil dermed ikke endre den.

En rasjonell agent bør være autonom, som vil si at den lærer for å kompensere for delvis eller ukorrekt tidligere kunnskap. **Den tar avgjørelser ut i fra nåværende situasjon og endrer planene sine deretter**. Dersom agenten avhenger av tidligere kunnskap og initial plan istedenfor det den oppfatter og lærer, er den lite autonom (eks: møkkbillen og sphex vepsen). En støvsugeragent som lærer seg å forutse når og hvor det blir mer skittent, vil gjøre en bedre jobb. Den rasjonelle agenten vil ikke være fullstendig autonom fra starten av, i stedet blir den gitt noe initial kunnskap og evnen til å lære. Når agenten har fått tilstrekkelig erfaring, kan oppførselen bli uavhengig av tidligere kunnskap. Dette gjør at man kan designe enkle rasjonelle agenter som kan fungere i ulike omgivelser.

Egenskaper ved omgivelsene

For å designe rasjonelle agenter har vi sett på definisjonen av rasjonalitet, men vi må også definere **oppgaveomgivelsene**, som er «problemene» de rasjonelle agentene skal «løse».

Spesifisering av oppgaveomgivelser – PEAS

Oppgaveomgivelsen omfatter ytelsesmålet, omgivelsen og agentens aktuatorer og sensorer. Som vi så på forrige side kan dette beskrives som **PEAS** (Performance, Environment, Actuators, Sensors). **Første steg i agentdesign er å spesifisere oppgaveomgivelsene (PEAS).** Vi ser på et eksempel med en automatisert taxi:

P. ***Performance measure*** – mulige ytelsesmål som den automatiserte taxien skal strebe etter, er å komme til riktig destinasjon, minimere forbruk av drivstoff, minimere kostnader eller tiden, minimere brudd av trafikklover og forstyrrelser, maksimere sikkerheten og komfort eller maksimere fortjeneste.

E. ***Environment*** – kjøremiljøet består av ulike typer veier, andre trafikanter, fotgjengere, dyr, veiarbeid, politibiler, dammer og vær. Taxien må også interagere med potensielle og faktiske kunder.

A. ***Actuators*** – inkluderer styring, akselerator, bremsing, tut og høyttaler eller display for å kommunisere med kunden.

S. ***Sensors*** – inkluderer en eller flere kameraer for å se veien, infrarød eller ekkoloddsensorer for å måle avstander, speedometer for å måle hastighet og akselerometer for å kontrollere kjøretøyet i svinger. Motorsensorer trengs for å kontrollere den mekaniske tilstanden til bilen, GPS sørger for at den vet hvor den skal kjøre og et tastatur eller mikrofon lar kunden spesifisere hvor de skal.

Tabellen viser grunnleggende PEAS elementer for flere agenttyper. Legg merke til at noen agenttyper opererer i kunstige omgivelser (eks: interaktiv språkveileder).

Agenttype	Ytelsesmål	Omgivelse	Aktuatorer	Sensorer
Medisinsk diagnosesystem	Sunne pasienter, reduserte kostnader	Pasient, sykehus, ansatte	Skjermvisning (spørsmål, tester, diagnoser, behandlinger, henvisninger)	Tastatur (symptomer, funn, pasienters svar)
Delplukkrobot	Prosent av deler som plasseres i riktig kasser	Transportbånd, kasser	Robotarm og -hånd	Kamera, leddvinkelsensorer
Raffineri-kontroller	Renhet, utbytte, sikkerhet	Raffineri, operatører	Ventiler, pumper, varmere, display	Temperatur, trykk, kjemiske sensorer
Internett shopping agent	Pris, kvalitet, effektivitet	Nettsider, leverandører, transportører	Display til bruker, URL, formutfylling	HTML sider (teks, grafikk, skript)

Egenskaper ved oppgaveomgivelser

Det er svært mange forskjellige oppgaveomgivelser som kan oppstå i AI, men disse kan kategoriseres basert på noen dimensjoner. **Disse dimensjonene vil avgjøre hvilken type oppgaveomgivelsen er, og dette har stor påvirkning på agentdesignet og hvilken teknikk som kan brukes for å implementere agenten**. Dimensjonene er:

Fullt eller delvis observerbar

Omgivelsen er fullt observerbar dersom sensorene kan detektere alle delene av tilstandene til omgivelsen som er relevante for valget som skal tas. Agenten slipper å opprettholde en intern tilstand for å holde oversikt over verden, siden den får all nødvendig informasjon fra sensorene. **Omgivelsen er delvis observerbar dersom deler av omgivelsen ikke kan sanses, pga støy, unøyaktighet eller manglende data** (eks: automatisert taxisjåfør kan ikke observere hvordan andre sjåfører tenker). Agenten må utføre informerte gjetninger om verden. Agenter uten sensorer har ikke-observerbare omgivelser.

Singel eller multiagent

Omgivelsen er singel-agent hvis det er ingen andre agenter eller andre agenter er en del av omgivelsen. Eksempler er en agent som løser kryssord eller en automatisert taxi i trafikk med andre automatiserte taxier. **Omgivelsen er multiagent hvis det er andre agenter med ytelsesmål som avhenger av handlingene til agenten (og motsatt)**. Eksempel er agent som spiller sjakk. Når man ser på omgivelsen til A, er det viktig at man kan skille mellom om entitet B er en annen agent eller et objekt som kun følger fysikklovene. Entitet B er en annen agent hvis den forsøker å maksimere et ytelsesmål som avhenger av handlingene til agent A. For eksempel ved sjakk, prøver spiller B å maksimere sitt ytelsesmål, noe som vil minimere ytelsesmålet til agent A (og motsatt). Vi skiller mellom konkurrerende og kooperativ multiagent omgivelser, der handlingen til en agent vil hhv. minimere og maksimere ytelsesmålet til andre agenter.

Deterministisk eller stokastisk

Omgivelsen er deterministisk dersom neste tilstand er fullstendig bestemt av nåværende tilstand og handlingen som utføres av agenten. **Omgivelsen er stokastisk dersom det er noe usikkerhet om utfallet til en handling, slik at det blir flere alternative utfall som kvantifiseres i form av sannsynligheter**. Dette inkluderer ikke usikkerhet som kun skyldes handlinger til andre agenter i multiagent omgivelser. For eksempel er sjakk deterministisk, selv om spilleren ikke klarer å forutsi neste trekk til den andre spilleren, fordi neste tilstand til brettet vil kun bestemmes av nåværende tilstand og trekket til spilleren. Hvis handlingen til motstanderen er det eneste som ikke kan forutsees (dvs. eneste usikkerhet), er agenten i en **strategisk omgivelse**. De fleste virkelige situasjonene er så komplekse at det er umulig å holde styr over alle uobserverte aspekter, så disse er stokastiske. For eksempel er taxikjøring stokastisk fordi det er mange usikre momenter i trafikken (eks: motorstopp, punktering, osv.). **Omgivelsen er usikker hvis den er verken fult observerbar eller deterministisk**.

Episodisk eller sekvensiell

Omgivelsen er episodisk dersom agentens opplevelse deles inn i atomiske episoder, der hver episode innebærer at agenten får en persept og utfører en enkel handling, og neste episode avhenger ikke av handlingene i tidligere episoder. Et eksempel er en agent som skal oppdage defekte deler på et samlebånd, siden den vil basere hver avgjørelse på nåværende del og tar ikke hensyn til tidligere avgjørelser. Nåværende avgjørelse vil heller ikke påvirke om neste del er defekt eller ikke. **Omgivelsen er sekvensiell dersom nåværende avgjørelse kan påvirke alle fremtidige avgjørelser**. Et eksempel er sjakk, der kortsiktige handlinger kan få langsiktige konsekvenser.

Statisk eller dynamisk

Omgivelsen er dynamisk dersom den kan endre seg mens agenten vurderer neste handling. Agenten blir kontinuerlig spurt hva den ønsker å gjøre, og hvis den ikke har bestemt seg enda, vil det tolkes som at den ønsker å gjøre ingenting. Et eksempel er taxikjøring, siden andre taxier kjører mens algoritmen bestemmer hva den skal gjøre. **Omgivelsen er statisk dersom den ikke endrer seg.** Det er enklere å håndtere statiske omgivelser, fordi agenten slipper å følge med på verden samtidig som den vurderer neste handling og den trenger ikke å bry seg om hvor lang tid det tar. Et eksempel er løsning av kryssord. **Omgivelsen er semidynamisk dersom omgivelsen ikke endrer seg over tid, men ytelsesscoren blir endret.** Et eksempel er sjakk, dersom man spiller med tid.

Diskret eller kontinuerlig

Omgivelsen er kontinuerlig hvis den har kontinuerlig tilstand, tid, persepter eller handlinger. Kontinuerlig-tilstand betyr at omgivelsen går gjennom en rekke kontinuerlige verdier (eks: lokasjon til taxi), kontinuerlig-tid betyr det at skjer jevnt over tid (eks: bevegelse til taxien), og kontinuerlig-handling betyr at handlingen utføres konstant og jevnt (eks: styring). **Omgivelsen er diskret hvis den har et endelig nummer av distinkte tilstander, tid, persepter eller handlinger**. For eksempel har sjakk et endelig antall ulike tilstander og et diskret sett med persepter og handlinger.

Kjent eller ukjent

Omgivelsen er kjent, dersom utfallene til alle handlingene er gitt, slik at agenten har mye kunnskap om hvordan omgivelsen fungerer. I dette tilfellet er det lett for agenten å ta gode avgjørelser. **Omgivelsen er ukjent, dersom agenten har lite kunnskap om omgivelsen og må derfor lære hvordan omgivelsen fungerer**. Det er viktig å skille mellom kjent/ukjent og fult/delvis observerbar. En kjent omgivelse kan være delvis observerbar, for eksempel kabal,

som er en kjent omgivelse (kjenner alle reglene), men delvis observerbar (ser ikke kortene som ikke er snudd enda). En ukjent omgivelse kan være fult observerbar, for eksempel i et nytt videospill der man ser hele spilltilstanden på skjermen, men vet ikke hvilke knapper som gjør hva før man prøver seg frem. Kjent/ukjent-dimensjonen blir ofte utelatt, siden det egentlig er en egenskap ved agenten og ikke ved omgivelsen. For eksempel sjakk kan være både kjent og ukjent avhengig av om spilleren kan alle reglene eller ikke.

Oppsummert

Hvilken type oppgaveomgivelse agenten befinner seg i, vil ha stor påvirkning på designet til agenten. **Den virkelige verden er delvis observerbar, multiagent, stokastisk, sekvensiell, dynamisk og kontinuerlig.** Tabellen under viser egenskaper ved noen ulike omgivelser

Omgivelse	Observerbar	Deterministisk	Episodisk	Diskret	Dynamisk	Agent
Kryssord	Fult	Deterministisk	Sekvensiell	Diskret	Statisk	Singel
Sjakk m/klokke	Fult	Deterministisk	Sekvensiell	Diskret	Semidynamisk	Multi
Poker	Delvis	Stokastisk	Sekvensiell	Diskret	Statisk	Multi
Taxikjøring	Delvis	Stokastisk	Sekvensiell	Kontinuerlig	Dynamisk	Multi
Samlebånd robot	Delvis	Stokastisk	Episodisk	Kontinuerlig	Dynamisk	Singel
Medisinsk diagnose	Delvis	Stokastisk	Sekvensiell	Kontinuerlig	Dynamisk	Singel

Tabellen inkluderer ikke kjent/ukjent, siden det egentlig ikke er en egenskap ved omgivelsen, men heller ved agenten. Egenskapene avhenger av hvordan oppgaveomgivelsen er definert. For eksempel ved medisinsk diagnose, kan det hende systemet er multiagent hvis det skal ta hensyn til trassige pasienter og skeptiske ansatte. Prosessen er episodisk hvis det involverer å velge en diagnose gitt en liste med symptomer, mens den er sekvensiell dersom det også involverer tester, evaluering av progresjon til behandlingen, osv.

Strukturen til agenter

Et **agentprogram** implementerer agentfunksjonen som kartlegger persepter til handlinger. Dette programmet kjører på et fysisk system med sensorer og aktuatorer, og dette kalles **arkitekturen**. Vi har at:

$$\textbf{agent} = \textbf{arkitektur} + \textbf{program}$$

Programmet vi velger må passe arkitekturen. For eksempel hvis programmet anbefaler handlingen *Walk*, er det viktig at arkitekturen har føtter. Generelt sett vil arkitekturen (1) sende persepter fra sensorene til programmet, (2) kjøre programmet og (3) sende valg av handlinger fra programmet til aktuatorene. Vi ser nærmere på ulike typer agentprogram.

Agentprogram

Agentprogram tar nåværende input fra sensorene og returnerer en handling til aktuatorene. Input er kun nåværende persept, fordi det er det eneste som er tilgjengelig fra omgivelsen. Hvis handlingen til agenten skal avhenge av hele perseptsekvensen, må agenten huske persepter.

```
function Table-driven-Agent( [percepts ]) returns an action
    static: table, maps percepts to actions

 append percept to the end of percepts
 action← Lookup(percepts, table)
```

Figuren viser et enkelt agentprogram som lagrer perseptsekvensen og bruker denne som indeks i en tabell med handlinger, for å finne neste handling. Denne tabellen representerer agentfunksjonen som kartlegger alle mulige perseptsekvenser til handlinger. Ved å bruke en perseptsekvens som indeks, vil man dermed returnere en passende handling**. For å bygge en rasjonell agent på denne måten, må vi lage en tabell som inneholder passende handling for alle mulige**

perseptsekvenser. Denne tabelldrevne fremgangsmåten er **umulig å bruke i praksis**, fordi hvis det er $|P|$ antall persepter og agenten har livstid T, vil tabellen inneholde $\sum_{t=1}^{T}|P]^T$ elementer som må søkes gjennom. Tabellen vil altså være svært stor, selv for relativt enkle omgivelser. Ingen fysiske agenter har nok plass til å lagre slike tabeller, designeren har ikke tid til å lage tabellen, ingen agent kan lære alle riktige tabelloppføringer fra erfaring og selv om omgivelsen kan gi en gjennomførbar tabell, vil ikke designeren få veiledning i hvordan den skal fylles.

Det tabelldrevne agentprogrammet gjør det vi ønsker, men det er ikke mulig å implementere denne i praksis. **Målet ved AI er å skrive agentprogram som produserer ønsket rasjonell oppførsel fra liten kode, istedenfor en stor tabell**. Fire agentprogram som følger dette prinsippet, med økende grad av generalitet er:

1. Simple reflex agents
2. Model-based reflex agents
3. Goal-based agents
4. Utility-based agents

Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
⋮	⋮

1. Simple reflex agents

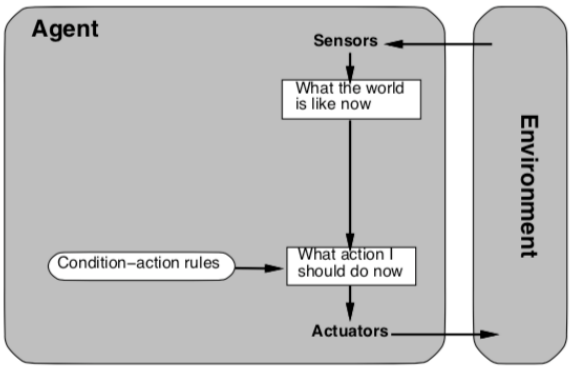
Simple reflex agents er den enkleste agenttypen, siden de velger handlinger basert på nåværende persept og ignorerer resten av perseptsekvensen. Figuren til høyre viser tabellen til agentfunksjonen for en simple reflex støvsugeragent, mens figuren til venstre viser agentprogrammet for denne agenten. Som vi kan se på figurene er handlingsvalget kun basert på nåværende lokasjon og om den lokasjonen er skitten. Programmet er mindre enn tabellen, noe som skyldes at antall muligheter reduseres fra 4^T til 4 ved å ignorere perseptsekvensen, og når nåværende lokasjon er skitten vil ikke handlingen avhenge av lokasjonen. **Simple reflex agent gir en stor reduksjon i antall mulige handlinger, fra $\sum_{t=1}^{T}|P]^T$ til $|P|$.**

```
function Reflex-Vacuum-Agent( [location,status]) returns an action

    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Simple reflex oppførsel foregår også i mer komplekse omgivelser. For eksempel når man kjører taxi, bør man begynne å bremse når bilen foran bremser. **Simple reflex agents er implementert vha *condition-action* regler.** For eksempel for taxien vil regelen være:

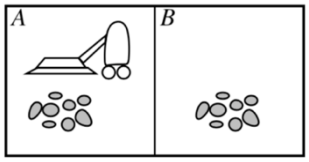
if bil-foran-bremser **then** start-bremsing

Figuren til høyre viser strukturen til simple reflex agents, mens figuren under viser et generisk agentprogram. Programmet har en rekke *condition-action* regler, bruker perseptet for å finne den første regelen der betingelsen oppfylles og utfører handlingen til denne regelen. ***Condition-action* reglene lar altså agenten koble persepter til handlinger**.




```
function Generic-SimpleReflexAgent( percept) returns an action

  static rules
  state ← Interpret(percept)
  rule ← Rule-match(state)
  action ← Rule-action(rule)
  return action
```

Ulempen med simple reflex agents er at de har begrenset intelligens. **Agenten fungerer bare hvis omgivelsen er fult observerbar, slik at den kan gjøre riktig avgjørelse kun basert på nåværende persept**. I eksempelet med taxien, kan det hende kameraet ikke klarer å



oppfatte alle typer bremselys, slik at det ikke alltid er mulig å bruke et enkelt bilde for å bestemme om bilen foran bremser. En simple reflex agent som kjører bak en slik bil vil enten bremse kontinuerlig eller ikke i det hele tatt. **Simple reflex agenter vil ofte ende opp i uendelige looper i delvis observerbare omgivelser**. For eksempel hvis vi fjerner lokasjonssensoren hos støvsugeragenten, vil den ende opp i en uendelig loop der den gjentatt forsøker å bevege seg til venstre når den starter i lokasjon A eller til høyre når den starter i lokasjon B. **Agenten kan flykte fra slike uendelige looper vha randomiserte handlinger**. For eksempel hvis perseptet til støvsugeragenten er *Clean*, vil den velge tilfeldig mellom *Left* og *Right*. **Randomisert oppførsel kan være rasjonelt i noen multiagent omgivelser, men det er som regel ikke rasjonelt i singel-agent omgivelser**. I de fleste tilfeller er det bedre med en mer sofistikert deterministisk agent.

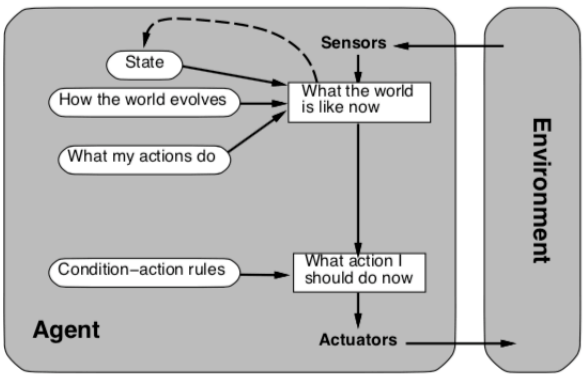
2. Model-based reflex agents

Delvis observerbare omgivelser kan håndteres ved å la agenten opprettholde en intern tilstand som avhenger av perseptsekvensen, slik at den kan holde rede på delen av verden den ikke kan se. For eksempel når en automatisert taxi skal bytte fil, bør den kunne holde rede over hvor andre biler befinner seg, hvis den ikke kan se alle med en gang (eks: blindsoner). For å oppdatere denne interne tilstanden over tid må agentprogrammet ha kunnskap om:

1. **Hvordan verden utvikler seg uavhengig av agenten**. Eksempel: forbikjørende bil kommer stadig nærmere
2. **Hvordan handlinger til agenten påvirker verden**. Eksempel bil kjører til høyre når rattet vendes med klokka

Denne kunnskapen om «hvordan verden fungerer» kan implementeres i enkle Boolean kretser eller fullstendige vitenskapelige teorier, og det kalles en **modell** av verden. **En agent som bruker en slik modell, kalles en modell-basert agent**.

Figuren til høyre viser strukturen til en model-based reflex agents, der vi kan se at **agenten oppdaterer beskrivelsen av nåværende tilstand ved å kombinere perseptet med den gamle interne tilstanden og agentens modell for hvordan verden fungerer**. Figuren under viser agentprogrammet, der funksjonen **UPDATE-STATE** er ansvarlig for å oppdatere den interne tilstanden. Hvordan modellen og tilstanden er representerer varierer. **Valget av handlingen gjøres på samme måte som Simple reflex agents**.



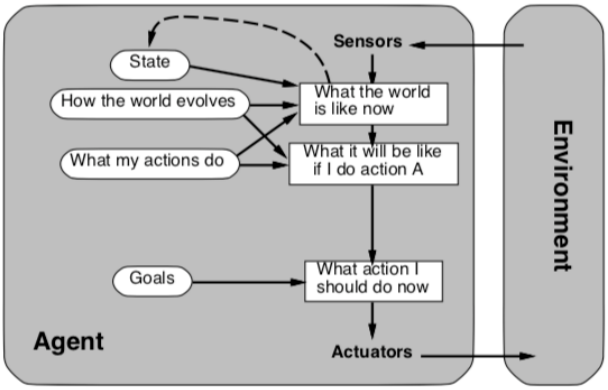

```
function REFLEX-AGENT-WITH-STATE(percept) returns an action
  static: state, a description of the current world state
          rules, a set of condition-action rules
          action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept)
  rule ← RULE-MATCH(state, rule)
  action ← RULE-ACTION[rule]
  return action
```

I delvis observerbare omgivelser er det vanskelig for agenten å nøyaktig avgjøre nåværende tilstand, uansett hvilken type modell som brukes. Denne avgjørelsen kan derfor måtte inkludere resonnering under usikkerhet, der agenten «gjetter» nåværende tilstand. En automatisert taxi vil for eksempel ikke kunne vite hva som er foran en stor trailer som har stoppet og kan bare gjette hva som forårsaker ventingen. **Usikkerhet ved nåværende tilstand kan være uunngåelig, men agenten må fortsatt ta en avgjørelse.**

3. Goal-based agents

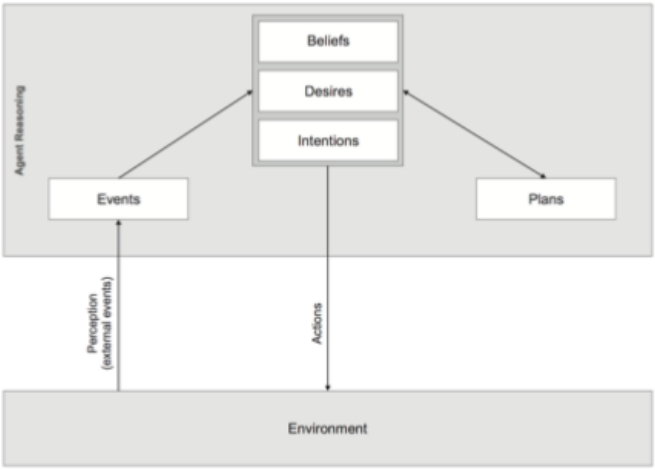
Det er ikke alltid tilstrekkelig å vite noe om nåværende tilstand for å bestemme hva som skal gjøres. For eksempel ved et veikryss, kan taxien kjøre til venstre, høyre eller rett frem, men

riktige avgjørelse avhenger av hvor taxien skal. I tillegg til en beskrivelse av nåværende tilstand, trenger agenten informasjon om **målet** som beskriver ønsket sitasjon (eks: destinasjon til taxipassasjer). Figuren til høyre viser strukturen til **goal-based agents, som kombinerer modellen med informasjon om målet for å velge handlinger som gjør at agenten oppnår målet.**



Goal-based agents har altså en eksplisitt representasjon av mål, slik at agenten vet hvilke tilstander som er ønskelig. Disse agentene bruker resonnering rundt mål for å velge handlinger. Dette valget kan være rett frem, for eksempel hvis en enkel handling resulterer i oppnåelse av målet. Det kan også involvere lange sekvenser av handlinger, med søking (kap. 3, 5) og planlegging (kap. 10, 11), for å finne ut hvordan målet kan oppnås. **Den største forskjellen mellom goal-based agents og reflex agents (både simple- og model-based), er at goal-based agents vurderer fremtiden når den tar et valg**. Valg av handling i goal-based agents er ikke basert på *condition-action* regler. Informasjon om fremtiden er ikke eksplisitt representert i reflex-agents, fordi persepter blir direkte kartlagt til handlinger i *condition-action* reglene. Reflex-agents vil svinge til høyre når den befinner seg ved en bestemt lokasjon, mens goal-based agents vil velge retning avhengig av hvor den skal.

Goal-based agenter er mindre effektive, men mer fleksible, fordi det er enklere å endre kunnskapen som støtter avgjørelsen siden den er eksplisitt representert. For eksempel hvis det starter å regne, kan agenten oppdatere sin kunnskap om hvor effektivt den kan bremse, noe som fører til at all relevant oppførsel blir automatisk tilpasset denne nye betingelsen. For en reflex-agents vil dette kreve at man skriver om flere av *condition-action* reglene. For at taxien skal gå til en ny destinasjon, krever goal-based agents at vi spesifiserer det nye målet, mens reflex-agents krever at vi endrer alle reglene for når agenten skal snu til høyre/venstre og når den skal kjøre rett frem.



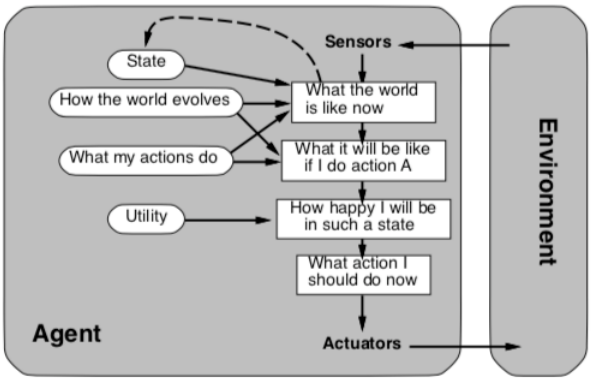
Et eksempel på goal-based agents er **BDI (*Belief Desire Intention*)** agenter som bruker en **mental tilstand** som grunnlag for resonnering. De tre viktigste mentale holdningene er tro (*belief*), ønske (*desire*) og intensjon. Resonneringen kalles praktisk resonnering (motpart til deduktiv resonnering)

4. Utility-based agents

I de fleste omgivelsene er det ikke tilstrekkelig med mål for å sikre oppførsel med høy kvalitet. Mange handlingssekvenser vil for eksempel få taxien til destinasjonen, men noen er raskere, tryggere, mer pålitelig og billigere enn andre. Mål gir kun en binær forskjell mellom «happy» og «unhappy» tilstander, men et mer generelt ytelsesmål bør si hvor «happy» de ulike tilstandene gjør at agenten blir. Istedenfor «happy» bruker vi begrepet ***utility* (nytte)**. **I tilfeller der mål kan nås på flere måter, kan utility verdier brukes for å velge mellom disse.**

Ytelsesmålet gir score til enhver sekvens av tilstander, så dette kan brukes for å skille mellom mer eller mindre ønskelige måter å nå målet. **Utility funksjonen er en internalisering av ytelsesmålet, så den kartlegger en tilstand til et reelt nummer, for å gi score til tilstanden** (dvs. bestemme hvor nyttig den er for agenten). **En rasjonell agent vil velge handlingen som maksimerer forventet utility hos utfallet (dvs. gir mest nyttig utfall).** Det er ikke alltid kjent

hvor nyttig en handling eller et utfall er (eks: delvis observerbar og stokastisk omgivelse), så derfor bruker vi «forventet» utility. Agenten ser på hvilken nytte den forventer at utfallet har, gitt sannsynligheter og nytte for hvert utfall. I tillegg vil **utility reflektere agentens preferanser**, for eksempel for en taxisjåfør kan sikkerhet være viktig, mens for en annen kan tid være viktig. Figuren viser strukturen til utility-based agents.



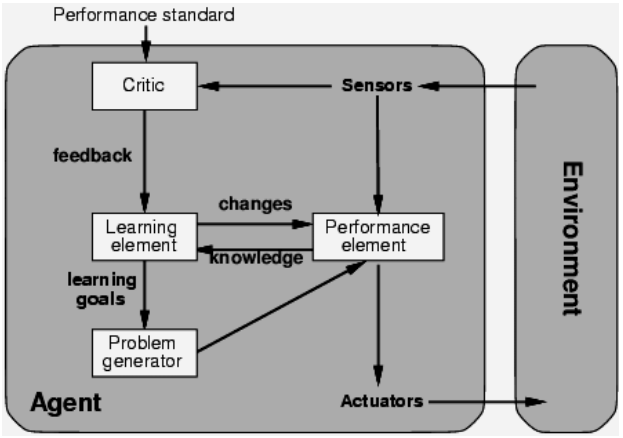
En reflex-agent kan også være rasjonell, men fordelen ved utility-based agents er at de er mer fleksible og kan lære. I følgende tilfeller vil ikke en goal-based agent nødvendigvis ta rasjonelle valg, så en utility-based agent er bedre:

1. **Det er motstridende mål som ikke kan oppfylles fullstendig** (eks: sikkerhet og hastighet). Utility funksjonen vil gi passende kompromiss
2. **Det er flere mål og det er ikke sikkert at noen av dem kan oppnås**. Utility funksjonen lar agenten se på sannsynligheten for suksess mot prioriteten til målet

Ulempen er at det er vanskelig å lage slike agenter som lager modeller og velger handlinger som maksimerer utility.

Lærende agenter

Vi har sett på hvordan agentprogram velger handlinger, men ikke hvordan de blir laget. Et alternativ er å skrive programmene for hånd, men den foretrukne metoden er ofte å lage lærende maskiner og deretter lære dem. En fordel med læring, er at agentene kan plasseres i omgivelser som er ukjente i begynnelsen, og etterhvert blir de stadig mer kompetente. En lærende agent kan deles inn i fire hovedkomponenter:



1. ***Performance element*** = velger ekstern handling. Dette utgjør «agenten», som tar inn persepter og velger handlinger.
2. ***Learning element*** = ansvarlig for forbedringen. Den får feedback fra *Critic* og bruker dette for å bestemme hvordan *Performance element* kan forbedres.
3. ***Critic*** = gir tilbakemelding til *Learning element* om hvor bra agenten gjør det sammenlignet med en fast ytelsesstandard. Persepter gir ingen indikasjon på om agenten er suksessfull. For eksempel i sjakk kan perseptet gi at spilleren har sjakkmatt på motstanderen, men det er ytelsesstandarden som gir at dette er bra.
4. ***Problem generator*** = foreslår handlinger som fører til nye og informative opplevelser. Hvis *performance element* får velge selv vil den alltid velge handlingen som er best, gitt det den vet. Hvis agenten er villig til å utforske og kanskje utføre noen subooptimale handlinger på kort sikt kan det hende den oppdager handlinger som er mye bedre på lang sikt.

For eksempel kan vi se på en automatisert taxi. Taxien bruker *performance element* for å kjøre på veien. Når taxien utfører en skarp sving over tre kjørefelt, observerer *Critic* at andre sjåfører viser et sjokkerende språk og sender dette til *Learning element*. Fra denne opplevelsen kan *Learning element* formulere en regel som sier at dette var en dårlig handling, og *Performance element* endres ved å installere den nye regelen. *Problem generator* kan identifisere oppførselsområder som trenger å forbedres og foreslår utforskning basert på det. For eksempel kan den foreslå å prøve bremsene på ulike veityper.

Læring som endrer modellen til agenten kan være basert på persepter, for eksempel vil to etterfølgende persepter av omgivelsen la agenten lære «How the world envolve» (se

tidligere figurer). Læring som endrer utility funksjonen må derimot inkludere ytelsesstandarden, siden det er mer kompleks og agenten må få tilbakemelding om en handling er suksessfull for å avgjøre nytten. Man kan si at ytelsesstandarden brukes for å gi tilbakemelding på kvaliteten til agentens oppførsel, ved at den bruker deler av perseptet som belønning (eller straff). For eksempel kan manglende tips fra taxipassasjer, bety at kjøreturen var ubehagelig og lite nyttig. Dermed kan agenten lære at voldsomme manøvrer ikke bidrar til å maksimere forventet utility.

Kapittel 3 – Løse problemer ved søking

Noen problemer har rett-frem løsninger som involverer bruk av formler eller velkjente prosedyrer. Andre problemer krever søk, der det er ingen standardisert metode og alternativer blir utforsket systematisert for å løse problemet. Antall alternativer ved søk kan være svært stor, til og med uendelig.

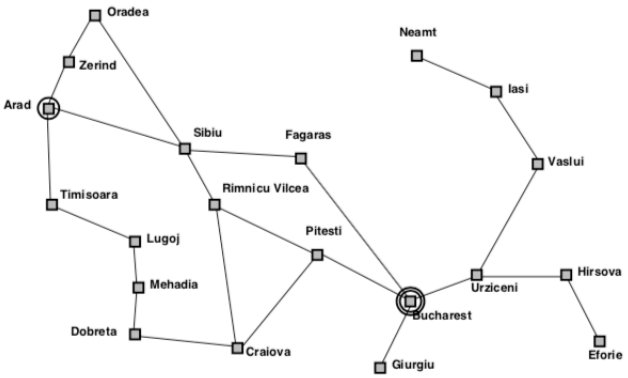
Problemløsende agenter

En problemløsende agent er en goal-based agent, som forsøker å løse et problem vha. et «*formulate, search, execute*» design. Figuren viser agentprogrammet, der den (1) formulerer et mål og et problem som skal løses, (2) kaller en søkealgoritme for å finne en løsning og (3) bruker løsningen for å bestemme handlingssekvensen. Vi skal nå se nærmere på de tre fasene.

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH( problem)
      if seq = failure then return a null action
  action ← FIRST(seq, state)
  seq ← REST(seq, state)
  return action
```

1. Formulering av mål og problem

Rasjonelle agenter forsøker å maksimere deres ytelsesmål, noe som kan forenkles hvis agenten har et mål den prøver å oppnå. Ytelsesmålet kan inneholde mange faktorer, slik at valget om hvilke handlinger som skal utføres blir kompleks og involverer mange avveininger. I slike tilfeller kan et mål brukes for å avslå handlinger som vil gjøre at målet ikke nås, slik at valget blir forenklet. Ved å spesifisere et mål kan man altså begrense hvilke handlinger agenten må vurdere. **Målformulering, basert på nåværende situasjon og ytelsesmålet, er første steg i problemløsningen**. For eksempel hvis agenten er på ferie i Romania kan ytelsesmålet være å sole seg, spise god mat, osv., mens målet kan være at man skal reise til Bucharest. Alle handlinger som gjør at man ikke når Bucharest innen tidsfristen blir avslått.



Agentens oppgave er å finne ut hvordan den skal oppføre seg for å nå måltilstanden. Først må agenten utføre en **problemformulering**, der den bestemmer hvilke handlinger og tilstander den skal vurdere. For eksempel hvis målet er å reise til Bucharest, kan ikke handlingene som vurderes være på formen «beveg venstre fot», fordi det blir for mange steg og for mye usikkerhet. Handlingene kan i stedet være på nivået av å kjøre fra en by til en annen, slik at hver tilstand går ut på å være i en by.

2. Søk

Agenten vil ikke vite hva den skal gjøre, hvis den har flere tilstander å velge mellom (ingen er måltilstanden) og den har ikke noe informasjon om utfallet av å velge ulike tilstander (dvs. usikker om en tilstand fører raskere til målet). For eksempel hvis agenten er i Arad vil den ikke vite om det er best å kjøre til Zerind, Sibiu eller Timisoara. Hvis omgivelsen er ukjent, må agenten velge tilfeldig. Hvis omgivelsen er kjent, dvs. agenten har informasjon om tilstandene (eks: kart over Romania), kan agenten bruke denne informasjonen for å lage hypotetiske baner med handlinger og se om de fører til målet. Når agenten har funnet en bane til målet, kan den nå målet ved å utføre handlingene langs denne banen. For eksempel når agenten finner en bane til Bucharest, kan den nå Bucharest ved å kjøre gjennom byene langs banen. **En agent som har flere umiddelbare alternativer med ukjent verdi, kan bestemme hva den skal gjøre ved å undersøke fremtidige handlinger som til slutt fører til**

en tilstand med kjent verdi. Dersom omgivelsen er observerbar, diskret, kjent og deterministisk vil løsningen være en sekvens med handlinger. **Prosessen av å lete etter en sekvens med handlinger som når målet, kalles søk**. En søkealgoritme vil ta et problem som input og returnerer en løsning i form av en handlingssekvens.

3. Utføring

Når løsningen er funnet kan de anbefalte handlingene utføres, noe som kalles *execution* fasen. Handlingene i handlingssekvensen vil utføre en om gangen. Når agenten utfører løsningen vil den ignorere perseptene, fordi den vet på forhånd hva de vil være. Når løsningen har blitt utført vil agenten formulere et nytt mål.

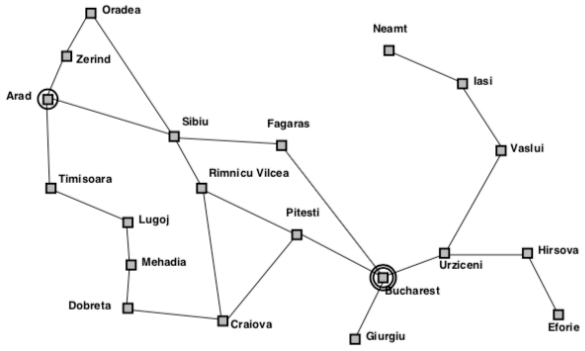
Veldefinerte problemer og løsninger

Et problem kan defineres av fem komponenter:

1. **Tilstander** = gir hva de ulike tilstandene er (eks: byer)
2. **Initial tilstand** = tilstanden der agenten starter (eks: Arad).
3. **Handlinger** = hvilke handlinger som er tilgjengelig for agenten som befinner seg i en bestemt tilstand (eks: fra $In(Arad)$ vil tilgjengelige handlinger være $\{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$
4. **Overgangsmodell** = gir hva som skjer når man utfører en handling fra en tilstand. For eksempel vil $Result\big(In(Arad), Go(Zerind)\big) = In(Zerind)$. Suksessorfunksjonen beskriver alle tilstandene som kan nås fra nåværende tilstand vha. en handling.
5. **Måltest** = bestemmer om nåværende tilstand er måltilstanden, som kan være eksplisitt (eks: agenten er i Bucharest) eller implisitt (eks: sjakkmatt)
6. **Banekostnad** = hver bane har en numerisk kostnad som reflekterer ytelsesmålet (eks: antall km til Bucharest). Kostnaden til banen vil være lik summen av kostnadene til individuelle handlinger langs banen. Stegskostnaden for å utføre handling a for å gå fra tilstand x til y, er $c(x, a, y)$ og det antas at den er positiv.

Disse elementene definerer et problem og kan samles i en enkelt datastruktur som kan gis som input til en problemløsende algoritme. **En løsning på problemet er en handlingssekvens som fører fra den initiale tilstanden til en måltilstand**. Kvaliteten til løsningen bestemmes av banekostnaden, og en optimal løsning vil ha den laveste banekostnaden.

Den initiale tilstanden, handlingene og overgangs-modellen vil avgjøre tilstandsrommet, som er settet av alle tilstander som kan nås fra den initiale tilstanden via ulike handlingssekvenser. Tilstandsrommet kan brukes for å lage en rettet graf, der nodene representerer tilstander og kantene representerer handlinger.



Formulering av problem – abstraksjon

Dersom vi bruker komponentene over til å formulere problemet av å komme seg til Bucharest, vil vi fortsatt ha en modell (dvs. abstrakt matematisk beskrivelse) og ikke den virkelige beskrivelsen. For eksempel vil ikke tilstanden «Arad» si noe om været, utsikten, reisekameratene, osv. Disse betraktningene blir ikke inkludert i tilstandsbeskrivelsen fordi de er ikke relevante for problemet av å finne en rute til Bucharest. **Prosessen der man fjerner detaljer fra en representasjon kalles abstraksjon**. Den virkelige verden er svært komplisert, så derfor er det nødvendig å abstrahere tilstandsrommet. Det er også nødvendig å abstrahere handlingene. Dette går ut på å utelukke detaljer ved handlingen (eks: forbruke drivstoff, forurense), se bort fra detaljerte handlinger (eks: skru på radio, se ut av vindu) og spesifisere nivået for handlingen (eks: «kjøre mellom byer» istedenfor «svinge»).

De abstrakte tilstandene og handlingene vi har valgt vil korrespondere med en kompleks kombinasjon av virkelige tilstander og handlinger. En løsning på det abstrakte problemet kan være en bane fra Arad til Sibiu til Rimnicu Vilcea til Pitesti til Bucharest. Denne abstrakte løsningen vil korrespondere med et stort antall mer detaljerte baner. For eksempel kan vi radioen være på mellom Sibiu til Rimnicu Vilcea, og av resten av turen. **Abstraksjonen er gyldig dersom enhver abstrakt løsning kan utvides til en løsning i den ekte verden**. En betingelse for abstraksjonen på forrige side er dermed at for enhver virkelig tilstand som er i Arad vil det være en virkelig bane til en tilstand som er i Sibiu, osv. **Abstraksjonen er nyttig dersom det er enklere å utføre de abstrakte handlingene, sammenlignet med det originale problemet**. En god abstraksjon innebærer dermed å fjerne så mange detaljer som mulig, mens man bevarer gyldighet og sikrer at de abstrakte handlingene er lette å utføre. Resultatet er en god modell av problemet.

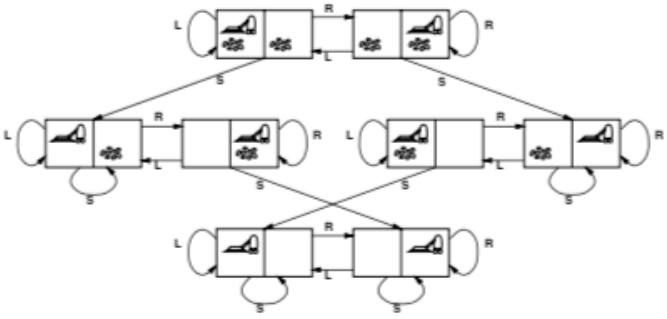
Problemeksempler

Lekeproblem brukes for å illustrere ulike problemløsende metoder, siden de kan beskrives presist og nøyaktig. Ved **virkelig-verden problem** er løsningene viktigere og det finnes som regel ingen beskrivelse som alle er enig om.

Eksempler på lekeproblem

Et eksempel på et lekeproblem er støvsugerverden, der problemet kan formuleres slik:

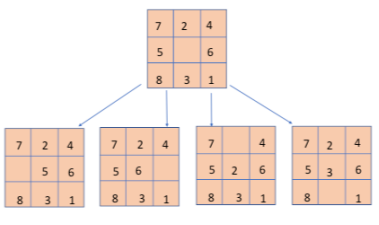
1. **Tilstander** = ulike lokasjoner av støvsugeren og skitt. Det er to mulige lokasjoner for støvsugeren og hver av lokasjonene kan være skitten eller ikke. Dermed er det $2 * 2^2 = 8$ mulige tilstander. Et større rom med n lokasjoner vil ha $n * 2^n$ mulige tilstander.
2. **Initial tilstand** = enhver tilstand kan være initial tilstand
3. **Handlinger** = når agenten er i en tilstand, har den fire tilgjengelig handlinger: *Left, Right*, *Suck* og *NoOp*.
4. **Overgangsmodell** = handlingene har forventet effekt, bortsett fra *Left* når støvsugeren er i venstre lokasjon, *Right* når støvsuger er i høyre lokasjon og *Suck* når lokasjonen er ren, som har ingen effekt.
5. **Måltest** = sjekker om alle lokasjonene er rene
6. **Banekostnader** = antall steg i banen, siden hvert steg koster 1 poeng (0p for *NoOp*)



Figuren viser tilstandsrommet for problemet (dvs. initial tilstand, handlinger og overgangsmodell). Legg merke til at tilstandsrommet viser initial tilstand (kan være alle tilstander), tilgjengelige handlinger og overgangsmodellen (hvilken tilstand man entrer ved å utføre en bestemt handling fra en bestemt tilstand). Til forskjell fra den virkelige verden har dette problemet diskrete lokasjoner, diskret skitt, pålitelig rengjøring og det blir aldri med skittent.

Et annet eksempel på et lekeproblem er 8-puzzle, der problemet kan formuleres slik:

1. **Tilstander** = bestemt plassering av de åtte brikkene og den tomme plassen (ignorerer mellomliggende posisjoner = abstraksjon)
2. **Initial tilstand** = enhver tilstand kan være initial tilstand
3. **Handlinger** = den tomme plassen kan flyttes *Left, Right, Up* eller *Down*.
4. **Overgangsmodell** = hvis den tomme plassen flyttes til en lokasjon vil den bytte plass med brikken som opprinnelig var i denne lokasjonen



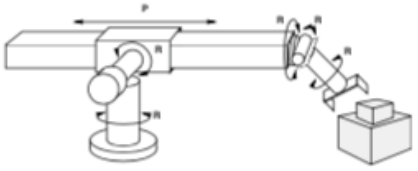
5. **Måltest** = sjekker om tilstanden matcher forhåndsbestemt mål
6. **Banekostnad** = antall flytt av den tomme plassen, siden hver flytt koster 1 poeng

Eksempler på virkelig-verden problem

Et eksempel på et virkelig-verden problem er finne-rute problemet som brukes innenfor flere ulike områder (eks: nettsider, flyselskap-planlegging, osv.). For en flyreise-planlegger, kan problemet formuleres slik:

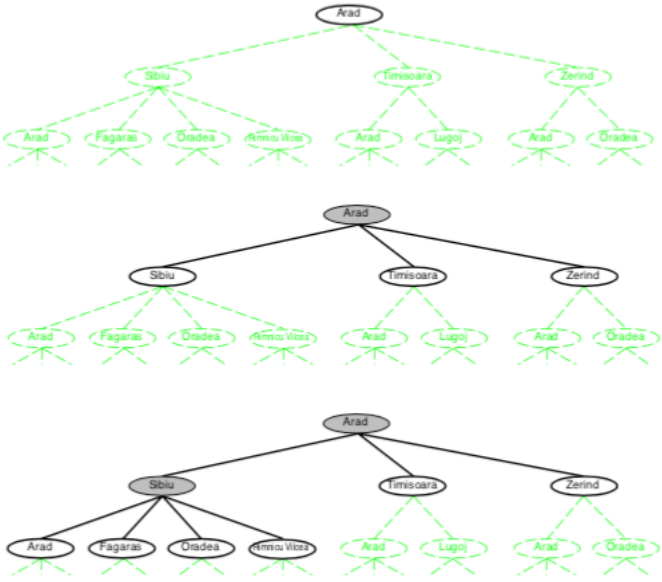
- **Tilstander** = ulike lokasjoner (eks: flyplass) og tid.
- **Initial tilstand** = lokasjonen og tiden brukeren oppgir
- **Handlinger** = ta et fly fra en lokasjon til en annen som går etter et bestemt tidspunkt
- **Overgangsmodell** = destinasjonen og tidspunkt for landing blir ny lokasjon og tid
- **Måltest** = er vi ved endelig destinasjon oppgitt av brukeren?
- **Banekostnad** = kan avhenge av kostnaden, ventetid, flytid, type fly, osv.

Et annet eksempel på et virkelig-verden problem er robotmontering, der problemet kan formuleres slik:



- **Tilstander** = vinkelen til robotleddene og deler av objektet som er montert
- **Initial tilstand** = ingen deler av objektet er montert sammen
- **Handlinger** = øke eller minke vinkelen til de ulike leddene
- **Overgangsmodell** = etter bevegelse av leddene og objektdeler når vi en ny tilstand
- **Måltest** = er objektet fullstendig montert?
- **Banekostnad** = monteringstiden

Tilstandsrom søk

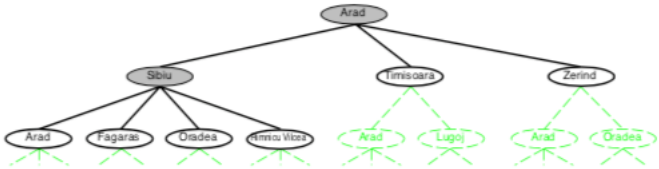


En løsning er en handlingssekvens, så søkealgoritmer vil vurdere ulike handlingssekvenser. **De mulige handlingssekvensene vil starte ved den initiale tilstanden og danner et søketre, der den initiale tilstanden er roten, grenene er handlinger og nodene er tilstander i tilstandsrommet**. Figuren viser hvordan søketreet blir laget for problemet av å finne en rute fra Arad til Bucharest. Her vil roten være *In(*Arad), og siden dette ikke er måltilstanden vil vi utvide tilstanden ved å bruke alle tilgjengelige handlinger hos nåværende tilstand. Vi får tre nye barnenoder: *In(*Sibiu), *In*(Timisoara) og *In*(Zerind). Deretter må vi velge hvilken av disse nodene som skal være neste tilstand, og hvordan dette gjøres skiller de ulike søkealgoritmene fra hverandre. For eksempel kan vi velge Sibiu først og siden dette ikke er måltilstanden vil vi utvide denne til fire nye barnenoder. Disse vil sammen med *In*(Timisoara) og *In*(Zerind) representere **bladnodene** til søketreet som utgjør **fronten (*frontier*)** til treet. **Prosessen av å utvide nodene vil fortsette helt til man finner en løsning eller til det er ingen flere noder å utvide**.

Figuren viser den generelle **TREE-SEARCH algoritmen** som gir den grunnleggende strukturen som brukes av alle søkealgoritmer. Forskjellen mellom søkealgoritmene er i hovedsak **søkestrategien** som avgjør hvilken node som skal utvides neste gang.

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

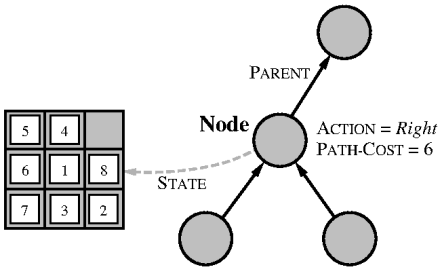
Merk: utvidelse av en node er det samme som å lage etterkommere (*successors*) hos allerede-utforsket tilstander



På figuren kan vi se at søketreet inkluderer banen fra Arad til Sibiu og tilbake til Arad. *In*(Arad) er en **gjentatt tilstand** i søketreet og lages av en **loopbane**. Dersom man inkluderer slike baner, vil det fullstendige søketreet bli uendelig. Vi trenger ikke å ta hensyn til disse, fordi banekostnaden er additiv og stegskostnaden er positiv. En bane fra initial tilstand til måltilstand vil derfor aldri bli bedre ved å inkludere en loopbane. Loopbaner er et tilfelle av **overflødige baner**, som vil eksistere når det finnes mer enn en handlingssekvens som fører til måltilstanden. I problemer der handlinger kan reverseres (eks: 8-puzzle) er det umulig å unngå overflødige baner. I slike tilfeller sier man at «algoritmer som glemmer historien sin, er dømt til å gjenta den». **Måten å unngå utforskning av overflødige baner (inkludert loopbaner), er altså å huske hvor man har vært**. For å oppnå dette kan vi legge til datastrukturen *fringe*, som vil huske alle noder som har blitt utvidet (*fringe* kan være FIFO, LIFO eller prioritetskø eller hashtabell). Noder som nylig har blitt generert, blir kastet dersom de allerede er i *fringe*. Den nye algoritmen kalles **GRAPH-SEARCH** (se figur). Denne algoritmen vil vokse et tre på tilstandsrommet og fronten separerer tilstandsrommet i et utforsket og ikke-utforsket område. **Søkealgoritmen vil systematisk utforske tilstandene i tilstandsrommet, helt til den finner en løsning.**

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(START-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in fringe or closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Strukturen til søkealgoritmer



En søkealgoritme må holde styr over søketreet som blir konstruert. Hver node n i treet har følgende komponenter:

- n**.STATE** = tilstanden i tilstandsrommet som noden korresponderer med.
- n**.PARENT** = foreldrenoden
- n**.ACTION** = handlingen som ble brukt på foreldrenoden
- n**.PATH-COST** ($g(x)$) = kostnaden for banen fra initial node til denne noden

Her kan vi se forskjellen mellom en node og en tilstand. En tilstand er en representasjon av en fysisk konfigurasjon fra verden (eks: by), mens en node er en datastruktur som utgjør en del av søketreet og har komponentene over. Tilstander har ikke foreldre, barn, handling eller banekostnad. To noder kan inneholde samme tilstand dersom tilstanden lages av to ulike baner. Figuren til høyre viser algoritmen for hvordan en node blir utvidet og barnenodene lages. Legg merke til DEPTH som representerer antall noder til roten, og SUCCESOR-FN som brukes for å hente tilstandene som barnenodene korresponderer med.

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action, result)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Mål av problemløsningsytelse

En søkestrategi bruker en bestemt rekkefølge når den bestemmer hvilken node som skal utvides. For å evaluere de ulike søkestrategiene ser vi på følgende dimensjoner:

- **Fullstendig** = vil den alltid finne en løsning, hvis løsningen eksisterer?
- **Optimal** = vil den finne den optimale løsningen? (dvs. lavest banekostnad)
- **Tid** = hvor lang tid bruker den på å finne en løsning?
- **Rom** = hvor mye minne trengs for å gjennomføre søket?

Tid og rom blir vurdert mht. et mål på vanskelighetsgraden til problemet, som gis av størrelsen til tilstandsrommet/grafen. I AI blir denne grafen implisitt representert av initial tilstand, handlinger og overgangsmodellen, og er ofte uendelig stor. Tid og rom blir derfor vurdert ut i fra:

- b = maksimal *branching* faktor i søketreet
- d = dybden til lavest-kostnad løsning (dvs. kan nås med lavest banekostnad)
- m = maksimal lengde til en bane i tilstandsrommet (kan være ∞)

Tid blir ofte målt som antall noder som genereres i løpet av søket, mens rom blir ofte målt som maksimalt antall noder som lagres i minnet. **Søkekostnaden** er som regel avhengig av tiden (kan også inkludere rom), mens den **totale kostnaden** er søkekostnaden pluss banekostnaden til løsningen.

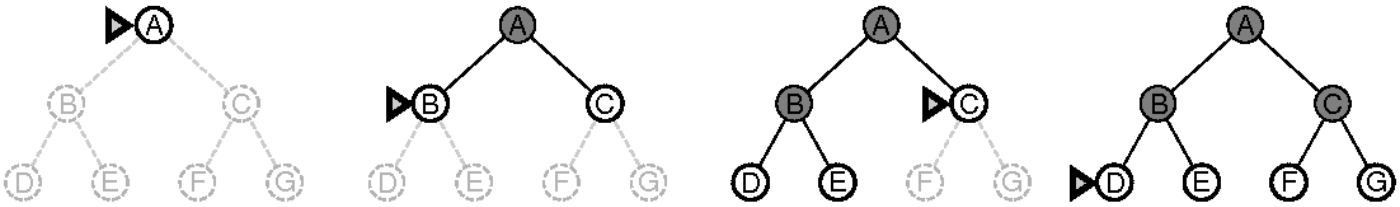
Uinformert søkestrategier

Uinformerte søkestrategier bruker kun informasjonen som er tilgjengelig i definisjonen av problemet. De kan utvide noder og skille mellom en måltilstand og en ikke-måltilstand. Forskjellen mellom søkestrategien er rekkefølgen for utvidelsen av nodene. Strategier som vet om en ikke-måltilstand er «mer lovende» enn andre kalles **informerte** eller **heuristiske søkestrategier** (mer senere).

Bredde-først søk (BFS)

Bredde-først søk (BFS) vil utforske tilstandsrommet horisontalt, ved at den utvider alle noder ved en bestemt dybde, før den den utvider noen noder ved neste dybde (dvs. barnenodene). Den neste noden som utvides vil altså være den ikke-utforskete noden med lavest dybde. **Dette oppnås ved å bruke en FIFO kø som lagrer nodene som har blitt oppdaget**. Nye noder vil plasseres i enden av køen, slik at de gamle nodene blir utvidet først. **Måltesten utføres på en node når den legges til i køen (reduserer tiden før løsningen finnes). For å unngå overflødig utforskning vil BFS kun legge til noder som ikke allerede er utvidet**. Når en node plukkes fra FIFO-køen for å utvides blir den lagt til et *explored*-sett, og det er kun barnenoder som ikke er i dette settet som blir lagt til FIFO-køen. Dvs. BFS sikrer at en node ikke blir utvidet flere ganger.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```



Evaluering av BFS gir følgende dimensjoner:

- **Fullstendig** = BFS er fullstendig, hvis løsningen er ved den endelige dybden d og branching-faktor b er endelig
- **Optimal** = når BFS finner målnoden, vil dette være den grunneste løsningen, fordi noder ved lavere dybde har allerede feilet måltesten. Dette er likevel ikke det samme som den optimale løsningen, fordi det avhenger av banekostnaden. For at BFS skal finne den optimale løsningen må banekostanden være en funksjon av dybden til noden. Dette er tilfellet dersom alle steg har samme kostnad.

- **Tid** = for et uniformt tre, der alle tilstander har b etterfølgere og løsningen ligger ved dybde d, vil antall noder som utforskes være $1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$, siden først må roten utforsker, deretter må b etterfølgere utforskes, og for hver av disse må b etterfølgere utforskes, osv.
- **Rom** = antall noder som lagres i minnet vil være $O(b^d)$, som er skremmende mye.

Depth	Nodes	Time	Memory
2	110	1.1 milliseconds	107 kilobytes
4	11,110	111 milliseconds	10.6 megabytes
6	10^6	11 seconds	1 gigabytes
8	10^8	19 minutes	103 gigabytes
10	10^{10}	31 hours	10 terabytes
12	10^{12}	129 days	1 petabytes
14	10^{14}	35 years	99 petabytes
16	10^{16}	3,500 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 100,000 nodes/second; 1000 bytes/node.

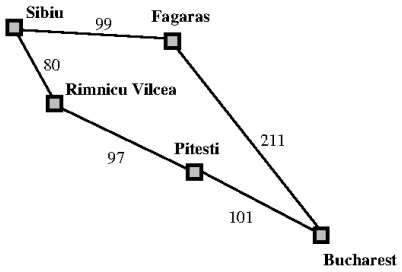
Figuren viser krav for tid og rom for ulike dybder, der branching-faktor er $b = 10$, hver node krever 1000 bytes lagring og 100 000 noder kan lages per sekund. **Problemet med BFS er at det krever mye rom**, fordi man kan vente 13 dager på løsningen til et viktig problem, men ingen PC har 1 petabyte med lagringsplass. Tid vil fortsatt være en viktig faktor, for eksempel vil det ta 3500 år å finne en løsning når dybden er 16.

Uniform-kostnad søk

Uniform-kostnad søk vil utvide noden med lavest banekostnad ($g(n)$), ved at den lagrer *frontier* som en prioritetskø ordnet etter lavest banekostnad. Husk at banekostnaden er kostnaden fra roten til noden (dvs. må summere kostnadene opp til noden). Måltesten blir brukt når noden utvides, for å sikre at løsningen den finner er på den optimale banen. Dersom algoritmen finner en bane til en node ved fronten som har lavere kostnad, vil den erstatte denne noden. **Hvis alle stegskostnadene er like vil uniform-kostnad søk være lik BFS**, bortsett fra at BFS stopper når den oppdager målnoden, mens uniform-kostnad søk stopper når den utvider målnoden (tar derfor litt lengre tid).

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Et eksempel er problemet av å reise fra Sibiu til Bucharest (se figur). Minste-kostnad node av etterfølgerne er Rimnicu Vilcea, så denne utvides slik at Pitesti legges til med kostnad 177 (= 80 + 97). Minste-kostnad node er nå Fagaras, og når denne utvides blir Bucharest lagt til med kostnad 310 (= 99 + 211). Dette er målnoden, men søket fortsetter siden måltesten ikke utføres før noden utvides. Minste-kostnad node er nå Pitesti, så denne utvides og algoritmen oppdager Bucharest med kostnad 278 (= 177 + 101). Siden denne banen har lavere kostnad enn den som ble funnet tidligere, blir Bucharest lagt til på nytt. Minst-kostnad node er nå Bucharest og når denne utvides, blir det oppdaget at målet er nådd.

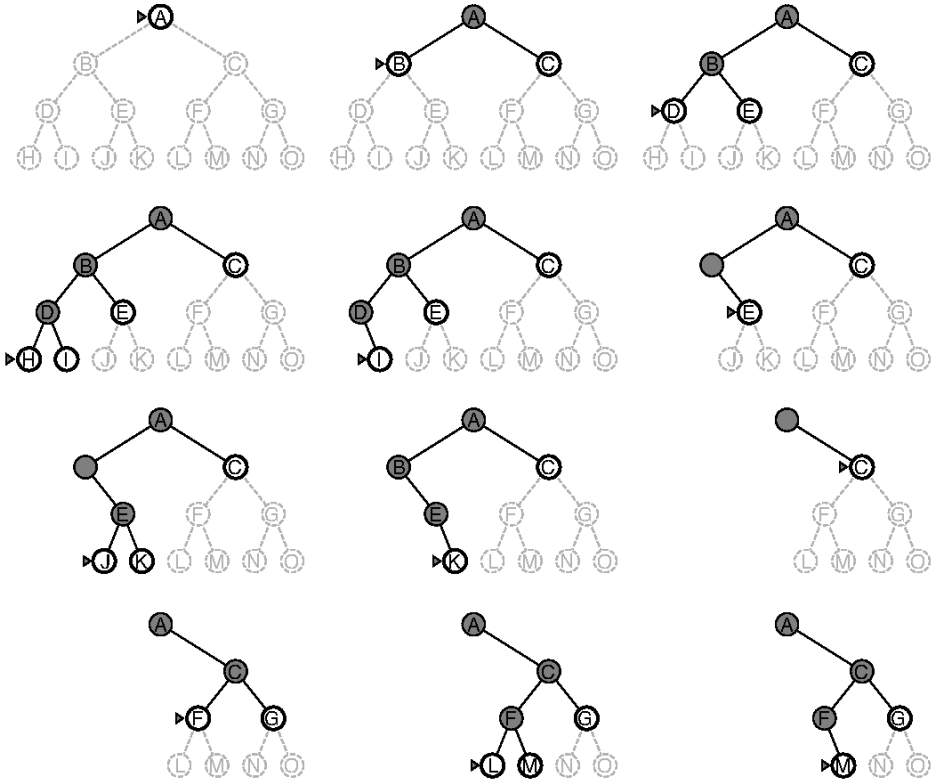


Evaluering av uniform-kostnad søk gir følgende dimensjoner:

- **Fullstendig** = Ja, hvis stegskostnadene er positive og det finnes ingen loop som har 0 kostnad. Dette er tilfellet hvis alle kostnader er større enn en liten positiv konstant ϵ.
- **Optimal** = Ja, fordi når node n velges for utviding, vil den optimale banen til denne noden være funnet (søket velger alltid noder med lavest kostnad, så banen til n består av de billigste mulige stegene) og stegskostnadene er større enn 0, slik at baner blir mer kostbare ved å legge til flere noder. Når en målnode utvides, vil banen derfor være optimal.
- **Tid** = hvis C^* er kostnaden til den optimale løsningen og alle kostnader er minst ϵ, vil antall noder som utforskes være $O(b^{C^*/\epsilon})$, som kan være mye større enn b^d.
- **Rom** = også $O(b^{C^*/\epsilon})$, siden alle noder som utforskes blir lagret i minnet

Dybde-først søk (DFS)

Dybde-først søk (DFS) vil utforske tilstandsrommet vertikalt, ved at den utvider alle noder langs en bane, før den utvider noen noder ved neste bane. Den neste noden som utvides vil være den ikke-utforskete noden med høyest dybde. **Dette oppnås ved å bruke en LIFO kø som lagrer nodene som har blitt oppdaget.** Nye noder vil plasseres i starten av køen, slik at de nye nodene blir utvidet først. **Hvis algoritmen når en node som ikke har en etterfølger, vil den gå tilbake til forrige dypeste node som har ikke-utforsket etterfølgere og fortsette søket fra denne**. Måltesten utføres på en node når den ligge til i køen (reduserer tiden før løsningen finnes). DFS kan være basert på TREE- eller GRAPH-SEARCH strukturen eller en rekursiv funksjon som kalles seg selv på barnenodene.



Evaluering av DFS gir følgende dimensjoner:

- **Fullstendig** = Nei, fordi DFS vil feile hvis treet har uendelig dybde (eks: loop). Den kan bli fullstendig i en endelig tilstandsrom ved å modifisere TREE-SEARCH versjonen slik at den sjekker nye tilstander opp mot de som er langs banen fra roten til nåværende node. Dette krever ingen ekstra minnekostnad, men det vil ikke unngå overflødige baner. Alternativt kan man bruke GRAPH-SEARCH versjonen, som vil være fullstendig i endelige tilstandsrom. Denne versjonen vil likevel kreve eksponentielt rom, siden den vil huske alle nodene den har oppdaget. Begge feiler i uendelige tilstandsrom.
- **Optimal** = Nei, den vil returnere første løsning den finner, uansett hva banekostnaden er
- **Tid** = antall noder som utforskes er $O(b^m)$, der m er maksimal dybde i treet. Dette er svært dårlig hvis $m \gg d$ (dvs. dypt søketre og grunn løsning), men kan slå BFS hvis $m \approx d$ (dvs. tett søketre og dyp løsning). Hvis tilstandsrommet er ∞ kan m være ∞
- **Rom** = For GRAPH-SEARCH versjonen er det ingen fordel, siden den må huske alle nodene. TREE-SEARCH versjonen trenger kun å huske nodene langs banen fra roten til nåværende node og deres ikke-utforsket etterfølgere. Når alle etterfølgerne til en node har blitt utforsket, kan den slettes fra minnet. Når branching-faktoren er b og maksimal dybde er m, vil antall noder i minnet derfor være $O(bm)$.

Mindre bruk av rom har gjort at DFS blir brukt av mange områder innenfor AI. Ved dybde $d = 16$ vil BFS kreve 10 exabytes, mens DFS krever 156 kilobytes.

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Dybde-begrenset søk

Dybde-begrenset søk er dybde-først søk med en begrensning på dybden (***l***), og noder ved denne dybden blir behandlet som om de ikke har noen etterfølgere. Dette løses problemet ved uendelige tilstandsrom. Søkealgoritmen kan være rekursiv, ved at den kaller på seg selv for barnenodene som utvides (se figur). Her vil *cutoff* være at det ikke er noen løsning innenfor dybdebegrensningen. Evaluering av Dybde-begrenset søk gir følgende dimensjoner:

- Fullstendig = Nei, den vil ikke finne løsningen hvis $l < d$, dvs. den grunneste løsningen er dypere enn begrensningen.
- Optimal = Nei, den vil returnere første løsning den finner, uansett hva banekostnaden er
- Tid = antall noder som utforskes er $O(b^l)$
- Rom = antall noder som lagres i minnet er $O(bl)$

Dybdebegrensningen kan noen ganger baseres på kunnskap om problemet. For eksempel i problemet av å reise fra Arad til Bucharest, er det 20 byer. Derfor vet vi at løsningen inkluderer maksimalt 19 byer. Derfor kan vi velge $l = 19$. Ved å se på kartet kan vi observere at alle byer kan nås fra en annen by via maksimalt 9 steg. Dette kalles diameteren til tilstandsrommet, og vil være en god dybdebegrensning: $l = 9$.

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
  inputs: problem, a problem

  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH( problem, depth)
    if result ≠ cutoff then return result
  end
```

Iterativ fordypende søk (IDS)

Iterativ fordypende søk gjentar dybde-begrenset søk for en gradvis utvidet dybdebegrensning, helt til den finner det grunneste målet ved dybde $l = d$. Den kombinerer fordelene ved BFS og DFS. I likhet med DFS vil antall noder som lagres i minnet være $O(bd)$, mens i likhet med BFS vil søkealgoritmen være fullstendig hvis branching-faktor er endelig og optimal hvis banekostnadene er ikke-reduserende funksjoner av dybden. Figuren under viser fire iterasjoner av søkealgoritmen. Iterativ fordypende søk er mye brukt når søkerommet er stort og dybden til løsningen er ukjent.

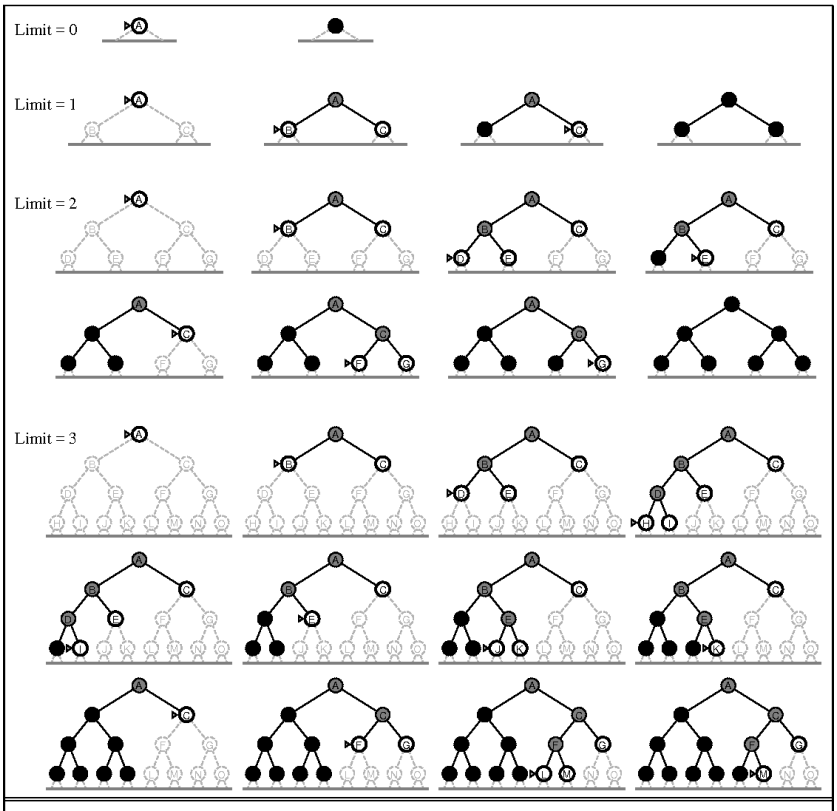
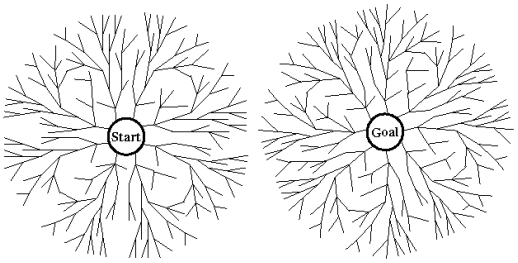


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Evaluering av Iterativ fordypende søk gir følgende dimensjoner:

- **Fullstendig** = Ja, hvis branching-faktor er endelig (lik BFS)
- **Optimal** = Ja, hvis banekostnader er ikke-reduserende funksjon av dybde (lik BFS)
- **Tid** = antall noder som utforskes er $(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d = O(b^d)$, siden roten blir gjentatt $(d+1)$ ganger, neste nivå blir gjentatt $(d+1) - 1 = d$ ganger, osv. og nivå ved dybde d blir laget 1 gang. Den bruker altså like lang tid som BFS (kostnaden for å gjenta de øvre nivåene er liten).
- **Rom** = antall noder som lagres i minnet er $O(bd)$ (lik DFS)

Bidireksjonalt søk (ikke F)



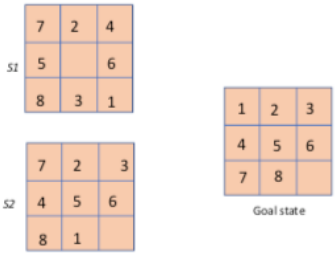
Bidireksjonalt søk innebærer at søket blir to søk blir startet samtidig, en går forover fra den initiale tilstanden og en går bakover fra målet. Håpet er at de to skal møtes i midten (se figur). Hvis det kan brukes vil det gi fullstendig og optimal søk med tid og rom på $O(b^{d/2}) = O(\sqrt{b^d})$ (= reduserer tid og rom med kvadratroten). Måltesten blir erstattet med en test som sjekker om de to frontene krysser. Hvis det er tilfellet har algoritmen funnet en løsning. Bidireksjonalt søk krever en metode for å finne forgjenger-noder, noe som kan være komplisert.

Sammenligning av uinformerte søkestrategier

Tabellen under viser sammenligningen av søkestrategiene for TREE-SEARCH versjonen (dvs. DFS vil ikke huske alle noder).

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*a	Yes*a,b	No	No	Yes*a
Time	b^d	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^d	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*c	Yes	No	No	Yes*c

*a) breadth-first complete if **b** is finite
*b) uniform-cost complete if step cost is $\geq \epsilon > 0$
*c) optimal if step costs are equal. Also if path cost non-decreasing fn. of depth.



For en uinformert søkestrategi, vil tilstand S1 og S2 være to noder som er like mye verdt (eks: samme nivå i søketreet). Uinformerte søkestrategier bruker kun posisjonen til nodene i treet og ikke tilstandsbeskrivelsene. For en informert (heuristisk) søkestrategi vil S1 være mer lovende enn S2, fordi den teller antall brikker som er feil. Heuristiske strategier bruker informasjonen i tilstandsbeskrivelsen for å finne løsningen. Vi skal se nærmere på disse.

Informerte (heuristiske) søkestrategier

Informerte (heuristiske) søkestrategier kan bruke problemspesifikk kunnskap for å finne løsninger mer effektivt. Den generelle tilnærmingen kalles **best-først søk,** som er en instans av TREE-SEARCH eller GRAPH-SEARCH algoritmene, der en evalueringsfunksjon $(f(n))$ brukes for å bestemme hvilken node som skal utvides. Evalueringsfunksjonen representerer et kostnadsestimat, så noden med lavest evaluering blir utvidet først. Dette gjøres ved å la *frontier* være en prioriteringskø der node med lavest $f(n)$ velges først. **Valget av $f(n)$ vil bestemme søkestrategien**. **De fleste evalueringsfunksjonene vil involvere den heuristiske funksjonen $h(n)$, som er den estimerte kostnaden hos den billigste banen fra tilstanden ved node n til måltilstanden**. Heuristiske funksjoner regnes som problemspesifikk kunnskap, og de har følgende begrensning: $h(n) = 0$ hvis n er målnoden.

Grådig best-først søk

Grådig best-først søk prøver å finne løsningen raskere, ved å utvide noden som ser ut til å være nærmest målet. Den vil evaluere noder ved å kun bruke den heuristiske funksjonen: $f(n) = h(n)$. Vi ser på finne-rute problemet i Romania, og bruker heuristikken for *straight-line distance*, som vi kaller h_{SLD}. Tabellen viser noen verdier for denne heuristikken (merk: kan ikke regnes ut fra definisjonen av problemet). Algoritmen vil først utvide Sibiu, fordi denne er nærmere Bucharest enn Zerind og Timisoara (dvs. har mindre h_{SLD}). Deretter vil den utvide Fagaras fordi den er nærmest Bucharest. Dette vil generere node for Bucharest og dermed er løsningen funnet. For dette problemet vil grådig best-først søk finne en løsning uten å utvide noden noder som ikke er på løsningsbanen. Søkekostnadene er dermed minimerte. Løsningen er likevel ikke optimal, fordi den er 32 km lengre enn banen via Rimnicu Vilcea og Pitesti. Dette illustrerer hvorfor den kalles «grådig», siden den prøver å komme så nært som mulig ved hvert steg.

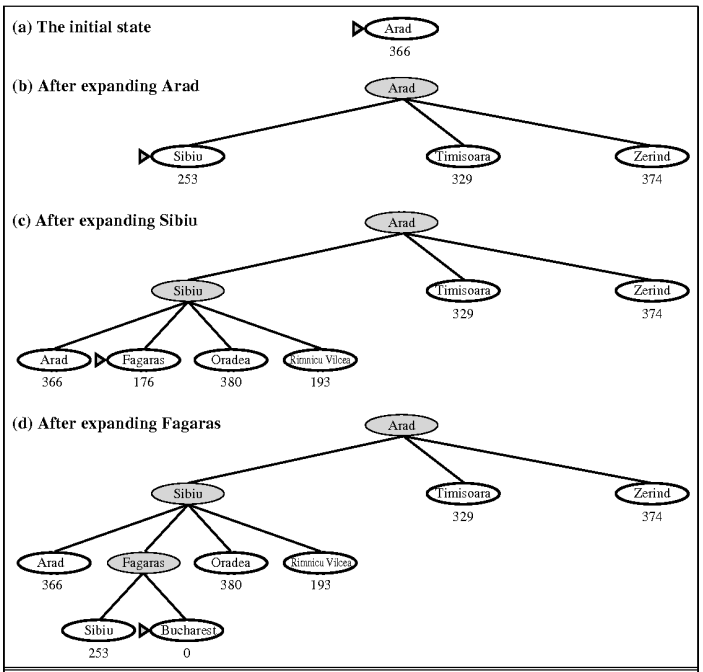
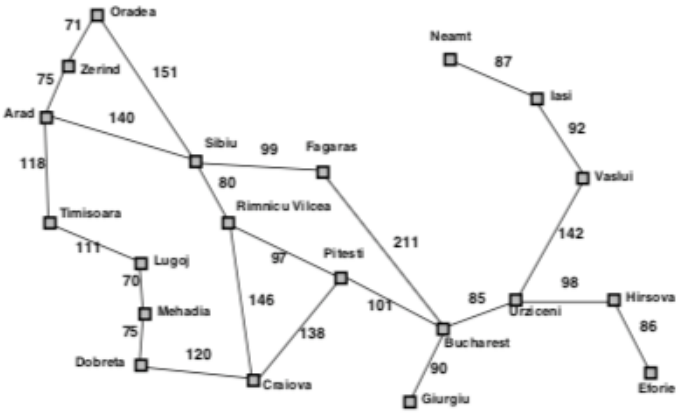


Figure 3.23 Stages in a greedy best-first tree search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.



Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Grådig best-først søk er ikke fullstendig, selv om tilstandsrommet er endelig. For eksempel kan vi se på problemet av å komme seg fra Iasi til Fagaras. Heuristikken foreslår av Neamt bør utvides først, fordi det er nærmes Fagaras, men dette er en blindvei! Løsningen er å først gå via Vaslui, et steg som er lengre unna målet. Algoritmen vil aldri finne denne løsningen, og den vil heller entre en loop der den går fra Iasi til Neamt til Iasi, osv. (siden Iasi er nærmere målet enn Vaslui). **Worst-case tid og rom for treversjonen av grådig best-først søk er $O(b^m)$, der m er maksimal dybde hos søketreet. Disse kan reduseres betraktelig dersom man velger en god heuristisk funksjon.**

A* søk – minimering av total estimert kostnad

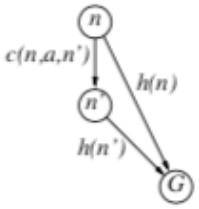
A* søk uttales «A-stjerne søk»

A* søk evaluerer noder ved å se på $f(n) = g(n) + h(n)$, altså den kombinerer banekostnaden for å komme seg fra roten til noden og estimert kostnad for å komme seg fra noden til målnoden. $f(n)$ blir dermed den estimerte kostnaden for den billigste løsningen gjennom n. Ideen bak A* er å unngå å utvide baner som allerede er dyre. Dersom den heuristiske funksjonen oppfyller visse betingelser, vil **A* søk være både fullstendig og optimal**. Algoritmen er identisk til UNIFORM-COST SEARCH, bare at den bruker $g + h$ istedenfor g (dvs. de er identiske hvis $h = 0$).

Optimalitet hos A* – begrensninger

For at A* skal være optimal, må den oppfylle noen betingelser, som er:

- **For tresøk-versjonen (dvs. overflødige baner):** $h(n)$ må være en ***admissible* heuristisk**, som vil si at den **aldri overestimerer** kostnaden for å nå målet (og $h(n) \geq 0$, slik at $h(G) = 0$ for ethvert mål G). Siden $g(n)$ er den faktiske banekostnaden, vil dette bety at $f(n) = g(n) + h(n)$ aldri vil overestimere den sanne kostnaden til en løsning som ligger langs nåværende bane via n. Admissible heuristisk er optimistisk, fordi den tenker at kostnaden av å løse problemet er mindre enn det den faktisk er. Et eksempel er h_{SLD}, som er admissible fordi den korteste linjen mellom A og B vil være en rett linje, så derfor kan ikke dette være et overestimat.
- **For grafsøk-versjonen (dvs. ingen overflødige baner):** $h(n)$ må være en **konsistent heuristikk,** som vil si at for alle noder n med etterfølger n' laget av handling a, vil estimert kostnad for å nå målet fra n ikke være større enn stegskostnaden for å nå n' pluss estimert kostnad for å nå målet fra n': $h(n) \leq c(n, a, n') + h(n')$. Dette er en form for trekantulikhet, som sier at en side av trekanten ikke kan være lengre enn summen av de to andre sidene (se figur). Konsistens er en sterkere betingelse enn admissible (dvs. hvis h er konsistent er den også admissible), men de fleste admissible heuristikker er også konsistent (dvs. det holder at $h(n)$ er en av delene).

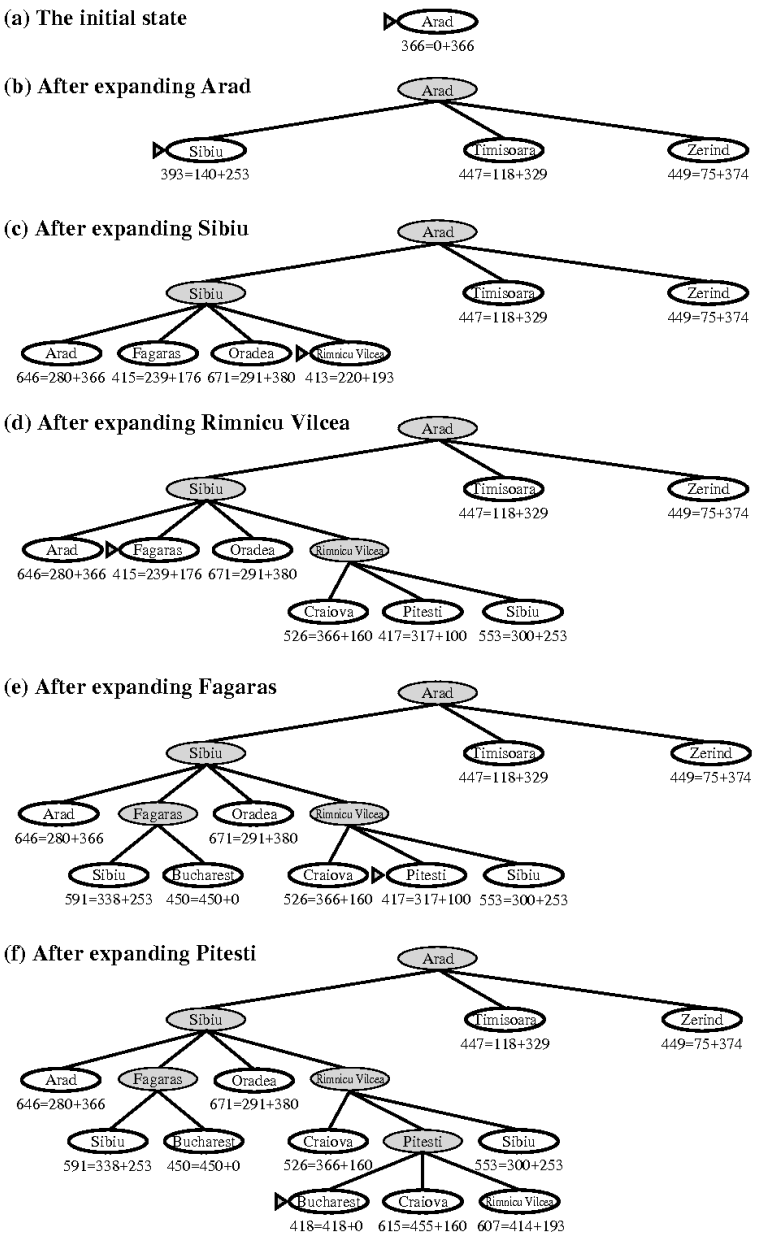


- **Branching-faktor er endelig**
- **Stegskostnadene er positive og større enn 0 (≥ 0)**

Figuren viser hvordan A* søk kan brukes for å løse problemet av å komme seg fra Arad til Bucharest. Her blir $f(n) = g(n) + h_{SLD}(n)$ brukt for å bestemme banen, der h_{SLD} er admissible. Legg merke til at Bucharest først blir oppdaget i fronten ved steg e, men den blir ikke valgt for utviding fordi den har $f = 450$, som er større enn for Pitesti der $f = 417$. Dette kan tolkes som at det kanskje er en løsning via Pitesti som har kostnad 417, slik at det ikke er bra nok å velge løsningen som har kostnad 450.

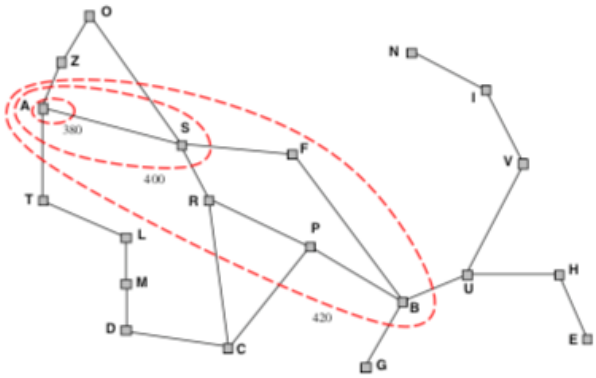
Subtreet til noder som ikke blir utvidet (eks: Timisoara) sies å bli beskjært (*pruned*). Siden h_{SLD} er admissible, kan algoritmen ignorere disse subtrærne og fortsatt garantere optimal løsning.

På neste side skal vi se hvorfor A* søk gir den optimale løsningen.



Optimalitet hos A*

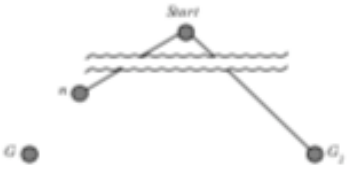
Hvis heuristikken er admissible eller konsistent, vil sekvensen av noder som utvides av A* ha økende $f(n)$ verdi (bevis under)**. Den første målnoden som utvides vil derfor være en optimal løsning, siden $f(n)$ er den sanne banekostnaden for målnodene ($h(G) = 0$) og alle noder som oppdages senere vil være minst like dyre (siden f øker)**. Dette gjør at vi kan tegne **konturer** i tilstandsrommet og merke disse med f-verdier. Kontur i vil inneholde alle nodene som har $f(n) \leq i$, og kontur $i + 1$ vil inneholde større f-verdier. Figuren viser et eksempel. Legg merke til at alle konturene dekker startnoden. **Vi kan se på A* søket som en prosess som legger til f-konturer med stadig større f-verdi.** Hvis $h = 0$ (dvs. uniform-kostnad søk) vil konturene være sirkulære rundt startnoden. **Hvis heuristikken er bedre, vil konturene strekke seg mot målnoden og blir smalere rundt den optimale banen. Hvis C^* er kostnaden til den optimale løsningen, har vi at:**



- **A* vil utvide alle nodene med $f(n) < C^*$**
- **A* vil utvide noen noder med $f(n) = C^*$**
- **A* vil utvide ingen noder med $f(n) > C^*$**

Dvs. A* vil utvide nodene som er innenfor konturen fordi disse vil være langs den optimale banen til målnoden. For at sekvensen av noder som utvides av A* vil ha økende f-verdi, krever det at heuristikken er admissible eller konsistent. På neste side ser vi hvorfor.

*Optimalitet for tresøk-versjon av A**



Tresøk-versjonen av A* er optimal hvis $h(n)$ er admissible. Heuristikken er admissible hvis den aldri overestimerer kostnaden for å nå målet og $h(n) \geq 0$, slik at $h(G) = 0$ for ethvert mål G. For å vise hvorfor det er nødvendig at heuristikken er admissible, ser vi på situasjonen der G_2 er en suboptimal målnode og n er en ikke-utvidet node langs banen til den optimale målnoden G (se figur). Vi ønsker å vise at $f(n) < f(G_2)$, slik at A* vil velge å utvide n. Altså, at A* velger banen som fører til den optimale løsningen.

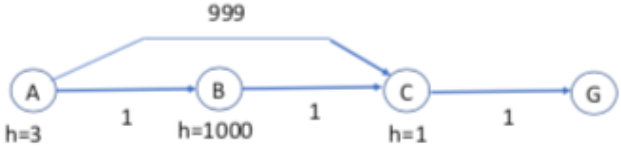
Vi har at $f(G_2) = g(G_2)$ og $f(G) = g(G)$, siden $h(G_2) = h(G) = 0$ (målnoder). Siden G_2 er suboptimal vil:

$$g(G) < g(G_1) \quad \Rightarrow \quad f(G) > f(G_2)$$

Hvis $h^*(n)$ er den sanne billigste kostnaden fra n til målet og $h(n)$ er den estimerte, vil det at $h(n)$ er admissible bety at: $h(n) \leq h^*(n)$. Dette skyldes at en admissible heuristikk aldri vil overestimere, men kan underestimere kostnaden. Siden $g(n)$ er kostnaden fra starten til noden og $h^*(n)$ er kostnaden fra noden til målet, vil $f(G) = g(n) + h^*(n)$. Dersom vi kombinerer dette med resultatet over får vi at:

$$g(n) + h(n) \leq g(n) + h^*(n)$$
$$f(n) \leq f(G)$$
$$f(n) < f(G_2)$$

Dermed vil A* velge å utvide n og vil fortsette helt til den når den optimale løsningen, G. **Tresøk-versjonen av A* er altså optimal hvis heuristikken er admissible, fordi det gjør at noder som kommer etter den optimale løsningen vil ha høyere f-verdi**.



Figuren til venstre illustrerer hvordan A* tresøk ikke vil kunne finne den optimale løsningen dersom heuristikken ikke er admissible. $h(B) = 1000$ er et overestimat, som gjør at A* velger den suboptimal banen.

*Optimalitet for grafsøk-versjon av A**

Grafsøk-versjonen av A* er optimal hvis $h(n)$ er konsistent. Heuristikken er konsistent hvis $h(n) \leq c(n, a, n') + h(n')$, der n' er etterfølgeren til n. For å vise hvorfor det er nødvendig at heuristikken er konsistent, kan vi bruke lignende argumentet som for uniformt-kostnad søk, der g er erstattet med f (s. 25). Beviset har to deler:

1. **Hvis $h(n)$ er konsistent, vil $f(n)$ ikke reduseres langs banen**: hvis n' er en etterfølger til n, vil $g(n') = g(n) + c(n, a, n')$. Dermed vil:

$$f(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n) \quad \Rightarrow \quad f(n') \geq f(n)$$

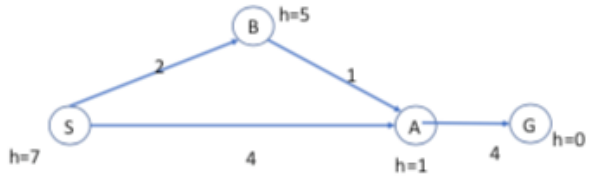
   Siden $h(n) \leq c(n, a, n') + h(n')$ (siden den er konsistent). f-verdien vil altså ikke reduseres langs banen av noder.
2. **Når A* utvider node n, vil den optimale banen til denne noden være funnet:** søket velger alltid noder med lavest f, så hvis en annen node har lavere f, ville den optimale banen ha gått via denne noden og til n. Uansett vil banen til n være optimal (ha minst mulig kostnad).

Fra disse observasjonene, følger det at sekvensen av noder som utvides av A* vil ha en stadig økende $f(n)$. Første målnode som utvides vil derfor være en optimal løsning, fordi senere løsninger vil ha større f-verdier.

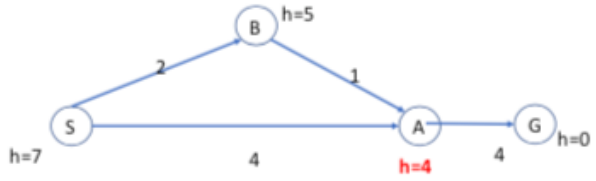
A pseudokode*

```
Start.g = 0;
Start.h = heuristic(Start)
FRONTIER = {Start}
CLOSED = {empty set}
   WHILE FRONTIER is not empty
        N = FRONTIER.popLowestF()
        IF state of N= GOAL RETURN N
        add  N to CLOSED
        FOR all children M of N not in CLOSED:
                M.parent = N
                M.g = N.g + cost(N,M)
                M.h = heuristic(M)
                add M to FRONTIER
        ENDFOR
   ENDWHILE
```

Figuren viser pseudokoden for **grafsøk-versjonen av A*.** Legg merke til at FRONTIER vil inneholde nodene ved fronten, mens CLOSED inneholder nodene som er utvidet. Legg også merke til **noder som allerede er blitt utvidet vil ikke kunne legges til fronten på nytt**. Dette er nødvendig for at A* søket skal være en grafsøk (noder blir kun utvidet én gang). For å unngå behovet for å utvide noder som er i CLOSED, er det viktig at heuristikken er konsistent. Dette vil sikre at A* algoritmen finner den optimale banen. Når en ny node blir generert vil den kastes hvis den er i CLOSED, mens hvis den allerede er i FRONTIER vil den beholdes hvis den har lavere f-verdi.



Dette er et eksempel der heuristikken ikke er konsistent, siden $h(S) = 7 > c(S, a, A) + h(A) = 4 + 1 = 5$. Her vil $f(A) = 5$, $f(B) = 7$ og $f(G) = 8$. A* vil først utvide A og deretter vil den utvide B. Her vil den møte et problem, siden neste steg vil være å utvide A, men denne er allerede utvidet (dvs. i CLOSED). Dermed vil den ikke finne den optimale banen til målet.



Dette er et eksempel der heuristikken er konsistent, siden $h(S) = 7 \leq c(S, a, A) + h(A) = 4 + 4 = 8$. Her vil $f(A) = 8$, $f(B) = 7$ og $f(G) = 8$. A* vil først utvide B, deretter vil den utvide A og til slutt vil den utvide G. Dermed har den funnet den optimale løsningen. Den slipper å utvide A på nytt, fordi heuristikken er konsistent.

Hvis A* søket er et grafsøk, vil er det altså nødvendig at heuristikken er konsistent, for at søket skal kunne finne den optimale løsningen. Grafsøk innebærer at noder blir utvidet (dvs. plassert ved fronten) kun én gang.

A* er effektivt optimal

A* er effektivt optimal for enhver heuristikk som er konsistent, noe som betyr at det ikke finnes noen andre optimale algoritmer som vil utvide færre noder enn A*. Unntaket er at A* kan være uheldig i hvordan den velger mellom noder som har $f(n) = C^*$. A* er effektivt optimal fordi en algoritme som ikke utvider alle nodene med $f(n) < C^*$, vil risikere å gå glipp av den optimale løsningen.

Evaluering av A* søk

Evaluering av A* søk gir følgende dimensjoner:

- **Fullstendig** = Ja, så lenge det er endelig mange noder som har $f(n) < C^*$ (C^* er kostnad til optimal løsning), siden A* vil utvide alle disse nodene. Dette vil være tilfellet hvis stegskostnadene er større enn ϵ (endelig kostnad) og branching-faktoren b er endelig.
- **Optimal** = Ja, hvis heuristikken er admissible eller konsistent, branching-faktoren er endelig og sekvenskostnadene er større enn 0.
- **Tid** = $O(b^{\epsilon d})$, der d er dybden til optimal løsning og $\epsilon = (h^* - h)/h^*$ er den relative feilen til heuristikken (krever at stegskostnad ikke blir endret). Husk at h^* er den faktiske kostnaden for å komme seg fra roten til målet, mens h er den estimerte kostnaden. Tiden vil avhenge av antagelsene som er gjort om tilstandsrommet (eks: flere nesten-optimale mål vil føre til ekstra kostnad).
- **Rom** = b^d, siden den lagrer alle nodene i minnet (største ulempe ved A*)

Bruken av tid og rom gjør at det ofte er upraktisk å bruke A* for å finne den optimale løsningen. Det blir ofte brukt for å finne suboptimale løsninger eller det kan brukes heuristikker som er mer nøyaktig, men ikke helt admissible. **Uansett, vil bruken av god heuristikk gi enorme forbedringer sammenlignet med bruken av uinformert søk**.

Minne-begrenset heuristisk søk

For å redusere minnekravet til A*, kan følgende versjoner brukes:

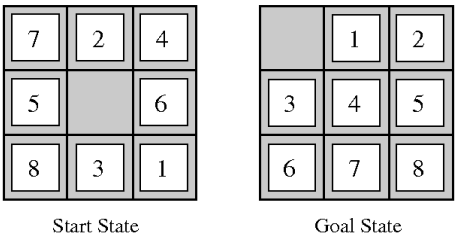
- **IDA* (iterative-deepening A*)** = bruken av iterativ fordyping i heuristisk søk. Begrensningen blir satt på f-kostnaden og ved hver iterasjon vil den være den minste $f$verdien til nodene som overgikk forrige begrensning. Den er utsatt for overflødige baner og gjentatt utviding siden den husker lite mellom iterasjoner.
- **Rekursiv best-først søk (RBFS)** = en rekursiv algoritme som bruker variabelen $f_$limit for å holde styr over f-verdien til den beste alternative banen som er tilgjengelig. Hvis nåværende node har større f-verdi, vil den gå til den alternative banen, samtidig som den oppdaterer $f_$limit (settes lik minst f-verdi blant alle frontnoder) og setter f-verdien til foreldrenoder lik den minste f-verdien til deres barnenoder. Er optimal hvis $h(n)$ er admissible, men den er utsatt for overflødige baner og gjentatt utviding siden den husker lite
- **SMA* (simplifisert minne-begrenset A*)** = fungerer som A*, helt til minnet blir fult av noder og den dropper den verste bladnoden som har høyest f-verdi. Når den dropper en node vil den lagre f-verdien i foreldrenoden, slik at forgjengeren til et droppet subtre vil huske den beste banen i subtreet. SMA* vil regenerere subtreet hvis det viser seg at det er bedre enn alle andre baner den har sett på. Hvis alle nodene har samme f-verdi vil den utvide den nyeste bladnoden og slette den eldste. Den er fullstendig og optimal hvis d kan nås vha det gitte minnet (hvis ikke blir problemet uløselig).

Man kan også lage agenter som kan lære å søke bedre. Disse vil utføre vanskelige problemer og vil feile, og ved erfaring kan de lære hvilke typer subtrær de ikke bør utforske.

Heuristiske funksjoner

Vi ser på heuristikken hos 8-puzzle for å vurdere heuristiske funksjoner. 8-puzzle går ut på å flytte brikkene horisontalt eller vertikalt inn i det tomme rommet, helt til konfigurasjonen matcher målkonfigurasjonen. Gjennomsnittlig løsningskostnad er 22 steg, mens branching-faktor er 3 (4 når tom brikke er i midten, 2 når den er i et hjørne og 3 når den er ved en kant). Et uinformert tresøk med dybde 22, vil ha $3^{22} \approx 3.1 * 10^{10}$ tilstander, mens et grafsøk

vil ha omtrent 170 000 distinkte tilstander. For å finne løsningen raskere vil vi bruke en **god heuristisk funksjon**. For eksempel kan vi finne den optimale løsningen ved å bruke A* og en admissible heuristisk funksjon, som aldri overestimerer antall steg til løsningen. Det finnes flere slike heuristikker, for eksempel:



- h_1 = antall brikker som er feil plassert. På figuren er alle brikkene plassert feil, så $h_1(\text{start}) = 8$. Dette er en admissible heuristikk, siden alle brikker som er plassert feil må flyttes minst én gang
- h_2 = summen av avstandene mellom brikkenes nåværende posisjon og deres målposisjon. Brikker kan ikke beveges horisontalt, så vi teller antall horisontale og vertikale avstander (Manhattan distance). For starttilstanden vil $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$. Dette er en admissible heuristikk, siden alle brikker som er plassert feil må flyttes til målposisjonen.

Siden den sanne løsningskostnaden er 26, vil verken h_1 eller h_2 være overestimat.

Hvordan heuristisk nøyaktighet påvirker ytelsen

Effektiv branching faktor b^* kan brukes for å karakterisere kvaliteten til en heuristisk funksjon. Hvis A* utforsker N noder, vil b^* gi hvilken branching faktor et uniformt tre med dybde d må ha for å inneholde $N + 1$ noder:

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

For eksempel hvis A* finner en løsning ved dybde 5 og bruker 52 noder, vil $b^* = 1.92$. **En god heuristisk funksjon vil ha effektiv branching faktor rundt 1, fordi dette vil la relativt store problemer løses med fornuftig kostnad**. Dette kan brukes for å teste heuristikkene h_1 og h_2, ved at vi lager 1200 problemer med løsningslengde på 2 til 24, og løser disse med iterativ fordypende søk (IDS, s. 27) og A* søk. Tabellen gir gjennomsnittlig antall noder som utforskes av hver strategi og effektiv branching faktor. Resultatet gir at h_2 er bedre enn h_1, og at det er mye bedre å bruke A* enn IDS. Selv for små problemer ($d = 12$), vil informert søk med h_2 være 50 000 ganger mer effektivt enn uinformert søk.

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

En annen måte å sammenligne kvaliteten til heuristiske funksjoner, er ved å se på hvilken funksjon som **dominerer**. For de definerte heuristikkene vil h_2 dominere h_1, fordi for enhver node n, så vil $h_2(n) \geq h_1(n)$. **Dominering handler om effektivitet, fordi A* som bruker h_2 vil aldri utvide flere noder enn A* som bruker h_1** (bortsett fra hvis det er flere noder som har $f(n) = C^*$). Alle noder som har $f(n) < C^*$ vil bli utvidet, noe som betyr at A* vil utvide alle nodene som har $h(n) < C^* - g(n)$. Jo større $h(n)$ er, desto færre noder vil altså bli utvidet. Alle nodene som blir utvidet av A* søk med h_2, vil også bli utvidet av A* søk med h_1, og h_1 kan kanskje forårsake at andre noder blir utvidet i tillegg. **Det er derfor generelt bedre å bruke heuristikker med høy verdi, så lenge den er konsistent og beregningstiden for heuristikken ikke er for høy**.

Dersom det finnes flere admissible heuristikker h_1, h_2, ..., h_m, vil den beste heuristikken være $h(n) = \max\{h_1(n), \ldots, h_m(n)\}$, siden den vil dominere de andre heuristikkene og er admissible.

Generering av admissible heuristikk fra relakserte problem

Et relaksert problem er et problem som har færre restriksjoner på handlingene, og det kan brukes for å lage heuristikk. For eksempel har h_1 og h_2 blitt laget for forenklede versjoner av 8-puzzle problemet. Hvis reglene hadde blitt endret slik at hver brikke kunne ha blitt plassert hvor som helst på brettet, selv om det er brikker der fra før av, ville h_1 gitt antall steg i den korteste løsningen. Hvis brikkene kunne ha blitt flyttet et steg i alle retninger, selv om de er okkupert, ville h_2 gitt antall steg i den korteste løsningen. **Når det blir fjernet restriksjoner i problemet, vil det bli lagt til flere kanter i grafen til tilstandsrommet**. En optimal løsning av det originale problemet vil derfor også være en løsning i det relakserte problemet, men det relakserte problemet kan ha en løsning som er billigere dersom de ekstra kantene gir snarveier. **Kostnadene til en optimal løsning i det relakserte problemet vil være en admissible heuristikk for det originale problemet**. **Heuristikken vil også være konsistent, siden det er en eksakt kostnad i det relakserte problemet og må dermed følge triangelulikheten**.

Hvis problemet er: «En brikke kan bevege seg fra A til B hvis, A er horisontal eller vertikal nær B og B er blank», kan vi lage tre relakserte problem ved å fjerne en eller begge betingelsene:

1. En brikke kan bevege seg fra A til B hvis A er horisontal eller vertikal nær B
2. En brikke kan bevege seg fra A til B hvis B er blank
3. En brikke kan bevege seg fra A til B

Fra 1 kan vi finne h_2, siden h_2 vil gi scoren hvis man kan flytte brikker til naboposisjonen, uten å må ta hensyn til at posisjonen man flytter til er blank. Fra 3 kan vi finne h_1, siden h_1 vil gi scoren hvis man kan flytte brikker direkte til deres målposisjon. **Det er viktig at de relakserte problemene kan løses uten søk, fordi hvis det er vanskelig å løse vil det ta for lang tid å regne ut verdiene til den heuristiske funksjonen.**

Generering av admissible heuristikk fra delproblemer

Løsningen til delproblemer hos et problem kan brukes for å lage admissible heuristikk. For eksempel for 8-puzzle kan et delproblem være å gå brikke 1, 2, 3 og 4 til målposisjon. Kostnaden til den optimale løsningen hos dette delproblemet vil være en nedre grense på kostnaden hos det fullstendige problemet. Dette vil i noen tilfeller være mer nøyaktig enn Manhattan distance. Ved mønster databaser blir kostnadene for alle mulige delproblemer lagret, slik at når søket løser det originale problemet kan det regne ut heuristikken ved å hente frem kostnaden til korresponderende delproblem.

Lære heuristikker fra erfaring

En agent kan lære seg å lage heuristiske funksjoner ved å løse problemet mange ganger og dermed lære fra erfaring. For eksempel kan agenten løse mange 8-puzzle og hver optimal løsning vil gi mulighet for å lære $h(n)$. Dette kan gjøres vha. nevrale nett, avgjørelsestrær og andre metoder.

Kapittel 4 – Forbi klassisk søk

I kapittel 3 så vi på problemer med observerbar, deterministisk og kjent omgivelse, der løsningen er en sekvens av handlinger. I dette kapittelet vil vi se hva som skjer når disse antagelsene blir relaksert.

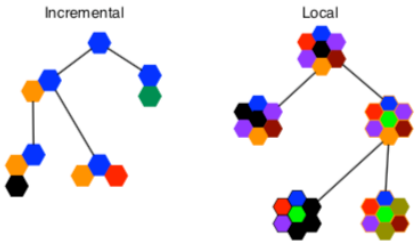
4.1 Lokale søkealgoritme og optimaliseringsproblem

Søkealgoritmene vi har sett på frem til nå har funnet løsningen ved å systematisk utforske tilstandsrommet. Dette gjøres ved å holde en eller flere baner i minnet og registrere hvilke alternativer som har blitt utforsket. Når målet er funnet, vil banen også utgjøre en del av løsningen. **I mange optimaliseringsproblem er banen irrelevant, og det er kun måltilstanden som vil være løsningen**. Det er den endelige konfigurasjonen som er nyttig og ikke hvordan denne ble oppnådd. **Lokal søkealgoritmer bruker kun en enkel node og vil bevege seg til naboer av denne noden**. Banen som følges i søket blir som regel ikke bevart. Lokal søk er ikke systematisk, men de har to fordeler:

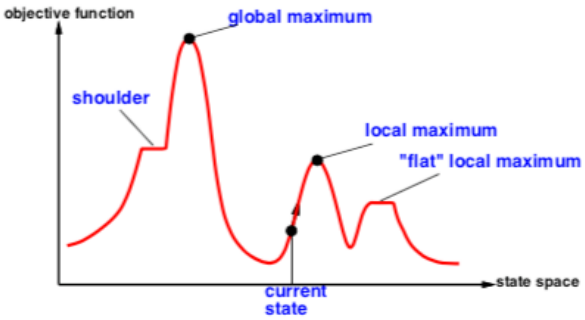
1. **Lav tidskompleksitet = de bruker veldig lite minne** (ofte konstant mengde), siden de ikke tar vare på baner som fører tilbake til starttilstanden.
2. **Tilfredsstillende = de kan finne fornuftige løsninger i store eller uendelige (kontinuerlige) tilstandsrom**, der systematiske algoritmer er upraktiske.

Tidskompleksiteten til lokale søkealgoritmer varierer, men nylig arbeid indikerer at det gir store forbedringer sammenlignet med inkrementell søk for problemer der den optimale løsningen er tett (dvs. ved liten dybde d). Lokal søk krever at representasjonene er lette å finjustere slik at det kan lages naboer i søkerommet. For å evaluere løsninger bruker lokal søk en objektivfunksjon, som ligner heuristikk, men krever mindre gjetningsarbeid.

Lokale søk er nyttige for å løse optimaliseringsproblem, der målet er å finne den beste tilstanden ved å optimalisere en objektivfunksjon (målfunksjon). For eksempel hvis objektivfunksjonen er reproduksjonsevne, kan evolusjonsteorien ses på som et forsøk å optimalisere denne for å skape individer med best sjanse til å overleve. Dette er et problem som ikke kan løses av systematisert søk, for det er ingen banekostnad eller måltest. I et optimaliseringsproblem vil tilstandsrommet være et sett med «fullstendige» konfigurasjoner, og lokal søk brukes for å finne den optimale konfigurasjonen eller konfigurasjonen som oppfyller bestemte begrensninger. **For å løse optimaliseringsproblem kan man også bruke iterativ forbedrende algoritmer, som tar vare på nåværende tilstand og forsøker å forbedre den (se figur).**



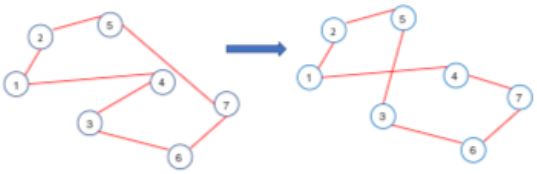
Lokale søkealgoritmer utforsker **tilstandsrom landskapet** (se figur), der x-verdiene er tilstander og y-verdiene er objektivfunksjonen eller heuristiske kostnaden. Hvis y-verdien er objektivfunksjon, vil målet være å finne den høyeste verdien, gitt av global maksimum. Hvis y-verdien er heuristisk kostnad, vil målet være å finne den laveste verdien, gitt av global minimum. **En fullstendig lokal søkealgoritme vil alltid finne et mål hvis det eksisterer, men en optimal lokal søkealgoritme vil alltid finne en global minimum eller maksimum**.



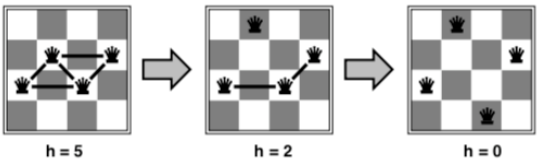
Problemeksempler

Noen eksempler på problem som kan løses med lokal søkealgoritme er:

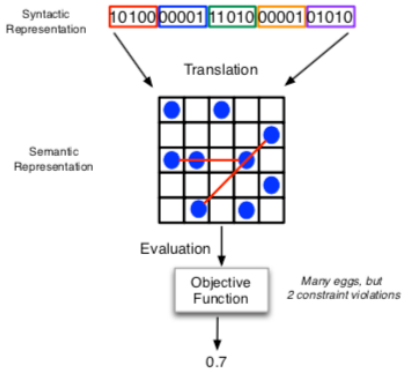
- **Travelling Salesperson Problem (TSP)** = finn billigste reise som besøker hver by én gang og returnerer til startbyen. Siden veiene mellom byene kan ha ulike kostnader, vil det være løsninger som er billigere enn andre. For å finne den optimale løsningen kan vi starte med en fullstendig reise (dvs. en løsning) og deretter utføre parvise utvekslinger for å finne en billigere løsning. Varianter av denne tilnærmingen kan brukes for å raskt finne løsningen som er innenfor 1% av optimalitet med flere tusen byer.

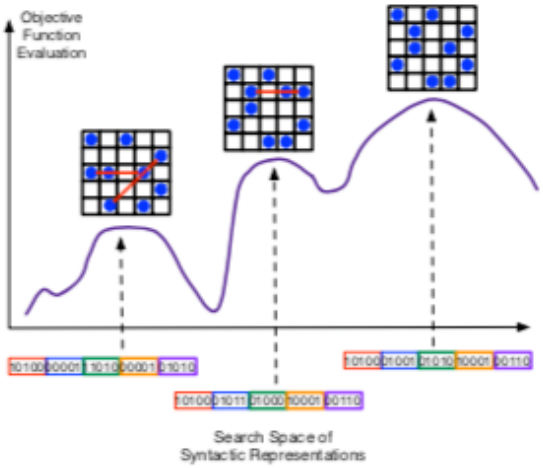


- ***n*-queens** = plasser n dronninger på et $n \times n$ brett, slik at ingen er på samme rad, kolonne eller diagonal. Dette problemet kan løses ved å plassere dronningene i hver sin kolonne og deretter flytte dem innenfor kolonnen for å redusere konflikter. Denne metoden vil nesten alltid løse problemet raskt og for store n.



- **Egg kartong problemet** = plasser så mange egg som mulig i $M \times N$ kartongen, men aldri mer enn K egg langs enhver horisontal, vertikal eller diagonal linje. Som regel vil $K = 2$. Som vi ser på figurene under kan dette representeres med syntaks (1'ere og 0'ere. Dette er et optimaliseringsproblem som kan løses med lokal søk, ved å optimalisere objektivfunksjonen.





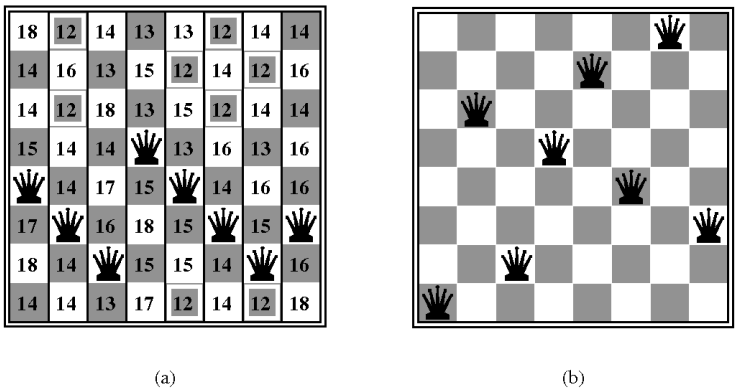
Figuren til venstre illustrerer hvordan tilstandsrom landskapet kan brukes for å evaluere løsninger på optimaliseringsproblemet. Jo høyere objektivfunksjon, desto bedre er løsningen. De to løsningene til venstre bryter begrensningen om å ha $K = 2$ egg langs hver linje og vil derfor ha lavere objektivitetsfunksjon. **Løsningen til høyre bryter ikke begrensningen og er derfor en optimal løsning som ligger ved global maksimum av objektivfunksjonen**.

Hill-climbing søk

Hill-climbing søkealgoritmen er en loop som vil kontinuerlig bevege seg i retningen som vil øke verdien, slik at den går oppover i landskapet. Denne verdien kan være verdien til en objektivfunksjon eller en heuristisk kostnad (verdien vil reduseres, går nedover). **Den vil terminere når den når en topp, der ingen nabo har høyere verdi.** Algoritmen vil ikke ta vare på noe søketre, så datastrukturen til nodene trenger kun å bestå av tilstanden og verdien til objektivfunksjonen. Søket ser ikke lengre frem enn den umiddelbare naboen til nåværende tilstand. Dette tilsvarer **å finne toppen til Mount Everest i tykk tåke, samtidig som man lider av hukommelsestap.**

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
   end
```

Hill-climbing søk kan brukes for å løse 8-queen problemet, ved å bruke en fullstendig-tilstand formulering, der de 8 dronningene plasseres i hver sin kolonne. For en tilstand (dvs. en bestemt konfigurasjon av brettet) vil den heuristiske funksjonen være antall dronningpar som angriper hverandre. Etterfølgerne vil være alle mulige tilstander som genereres av å flytte en enkel dronning til en annen posisjon i samme kolonne. Figur a viser en tilstand med $h = 17$ og de heuristiske kostnadene hos etterfølgerne. Hill-climbing søk ønsker å minimere den heuristiske kostnaden og vil derfor flytte en av dronningene til en posisjon med $h = 12$ (velger tilfeldig når det er flere). Dette blir gjentatt helt til vi når situasjonen på figur b der $h = 1$ (et angrep) og alle etterfølgerne høyere kostnad. Det er ikke en optimal løsning med global minimum ($h = 0$), men det er en lokal minimum som ikke kan reduseres videre.



Hill climbing blir kalt grådig lokal søk, fordi den vil velge den beste nabotilstanden uten å tenke på hva som skjer videre. Dette vil gi rask progresjon mot løsningen i glatte landskap, men hill climbing kan **sette seg fast** som følger av:

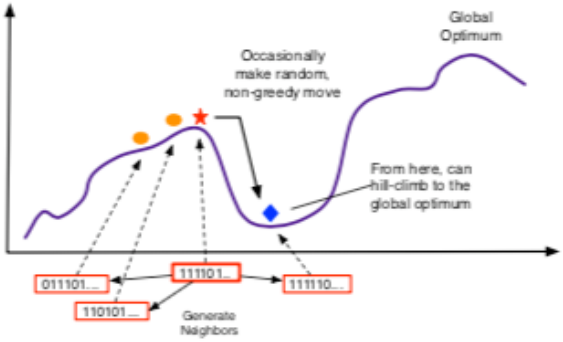
1. **Lokal maksimum** = høyere enn nabotilstander, men lavere enn global maksimum (motsatt for minimum). Hill-climbing søk som når en lokal maksimum vil dras opp mot toppen og vil deretter sette seg fast. Dette illustreres av eksempelet over.
2. **Rygger** = en sekvens av lokale maksimale verdier som er vanskelig å navigere
3. **Platå** = et flatt område som kan være en flat lokal maksimum uten noen oppoverbakke, eller en skulder der verdien plutselig endres og progresjon er mulig.

I hvert tilfelle vil algoritmen nå et punkt der det ikke blir gjort noen progresjon. Hill climbing søkealgoritmen vil ofte sette seg fast i 8-queen problemet (86%), men hvis den er rask og vil bare bruke 4 steg når den finner en løsning og 3 steg før den setter seg fast. For å unngå at Hill climbing søket setter seg fast kan vi tillate **sidelengs bevegelse**, men dette kan føre til uendelige løkker hvis søket når en platå som ikke er en skulder. For å unngå dette kan vi legge inn en begrensning på antall sidelengs bevegelser som kan utføres etter hverandre. Hvis det brukes i 8-queen problemet vil kun 6% av søkene sette seg fast, men det vil øke kostnadene siden det krever 21 steg for å finne løsning og 64 for å sette seg fast. Det har blitt laget flere varianter av hill climbing søkealgoritmen, for eksempel **random-restart hill climbing** som prøver på nytt med tilfeldige starttilstander helt til den finner en løsning og **stokastisk hill climbing** som velger tilfeldig blant bevegelsene som går oppover.

Simulert annealing (SA)

Simulert annealing (SA) fungerer som hill-climbing med *jiggle*. Hill climbing søkealgoritmen er effektiv, men ikke fullstendig siden den aldri går nedover og kan dermed sette seg fast i lokale maksimum. En tilfeldig gåtur vil finne global maksimum og er dermed fullstendig, men det er svært ineffektivt. SA forsøker å kombinere disse, slik at den oppnår både fullstendig og effektiv søk. Dersom vi forsøker å få en ping-pong ball til å legge seg i det dypeste hullet på en humpete overflate, kan det hende ballen vil legge seg i et lokalt minimum. Dette kan

løses ved å riste overflaten så hardt at ballen hopper ut av lokal minimum. Samtidig må vi passe på at vi ikke rister så hardt at den hopper ut av globalt minimum, hvis den befinner seg der. **For å beskrive mengde *jiggle* bruker SA en temperaturparamenter. I starten av søket vil SA bruke en høy temperatur (dvs. sterk *jiggle*) og deretter vil den gradvis redusere temperaturen.** Denne jiggle vil gjøre at søket kommer seg over rygger og platåer.



I algoritmen blir dette implementert ved at den tillater noen «dårlige» bevegelser, men vil gradvis redusere størrelsen og frekvensen til disse. Neste node blir altså valgt tilfeldig og dersom denne vil forbedre situasjonen blir den alltid akseptert. Hvis ikke er det en viss sannsynlighet for at den aksepteres og denne sannsynligheten blir deretter redusert. Hvor mye sannsynligheten reduseres avhenger av hvor «dårlig» bevegelsen var og temperaturen, som gradvis reduseres av *schedule*. Hvis temperaturen blir reduserte tilstrekkelig sakte, vil sannsynligheten for at søket finner en global optimum være nær 1.

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```

Lokal *beam* søk

Lokal *beam* søk (strålesøk) holder styr over k tilstander istedenfor kun én. Den vil begynne med k tilfeldig tilstander og ved hvert steg vil den generere etterfølgerne til alle k tilstandene. Algoritmen vil stoppe hvis en av disse er målet, hvis ikke vil den velge de k beste etterfølgerne og gjenta prosessen. **Dette er ikke det samme som k uavhengig søk som utføres parallelt, fordi søketråder som finner gode etterfølgere vil rekruttere de andre trådene, slik at ressursene flyttes til der det er mest progresjon.**

Et problem ved lokal beam søk er at alle k tilstander ender opp i samme lokale optimum, slik at det blir en dyr versjon av hill climbing. For å unngå dette kan man bruke **stokastisk beam søk, som tilfeldig velger k verdier, men høye k-verdier har større sannsynlighet for å velges**. Dette er en analogi til naturlig seleksjon, der verdien til et avkom (etterfølger) fra en organisme (tilstand), vil avgjøre hvor mye det vil bidra til neste generasjon.

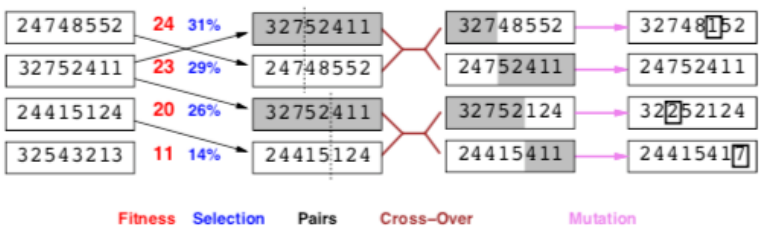
Genetisk algoritme (GA)

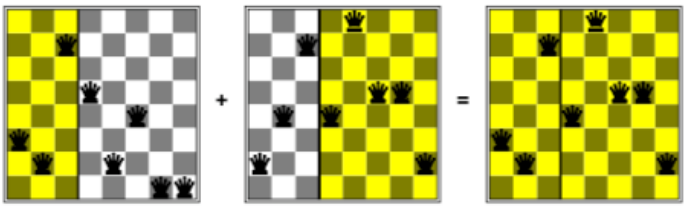
Genetisk algoritme er en variant av stokastisk beam søk, der etterfølgertilstanden lages ved å kombinere to foreldretilstander. Analogien til naturlig seleksjon gjelder fortsatt, men nå er det seksuell reproduksjon istedenfor aseksuell. **GA vil begynne med k tilfeldig genererte tilstander som kalles populasjonen, og hver tilstand (individ) blir representert som en string over et endelig alfabet (ofte binært)**. For eksempel i 8-queen problemet blir tilstanden representert som 8 tall, der hvert tall gir posisjonen til dronningen i kolonnen (1-8 fra bunnen). Figuren viser en populasjon med fire individer som representerer hver sin 8-queen tilstand. De ulike stegene i GA er:

1. **Fitness function** = hvert individ blir vurdert basert på verdien hos objektivfunksjonen. Jo høyere verdi, desto bedre tilstand. For 8-queen problemet vil dette derfor være antall ikke-angripende dronningpar.
2. **Seleksjon** = tilstander blir valgt ut for reproduksjon avhengig av fitness scoren, som er sannsynligheten for å brukes i produksjonen av neste generasjon. Den vil være

prosentandelen av total score som tilstanden utgjør. For 8-queen problemet vil for eksempel 11 bli 14%, siden $11/(24+23+20+11) = 0.14$.

3. **Par** = to par blir tilfeldig valgt, men høyere fitness score øker sannsynligheten for å velges. For 8-queen problemet ser vi at en tilstand blir valgt to ganger, fordi den har høy fitness score.
4. **Cross-over** = et cross-over punkt i stringen blir tilfeldig valgt. De to foreldrestringene blir krysset, ved at en barnestring får start av en forelder og slutt av en annen, og den andre barnestringen får motsatt. For 8-queen problemet blir cross-over punktet etter tredje tall for første par og etter 5 tall for andre par.
5. **Mutasjon** = hver lokasjon i barnestringene blir utsatt for tilfeldig mutasjon som vil skje med en liten sannsynlighet. For 8-queen problemet vil dette innebære å tilfeldig velge en dronning å flytte den til en tilfeldig plass i kolonnen





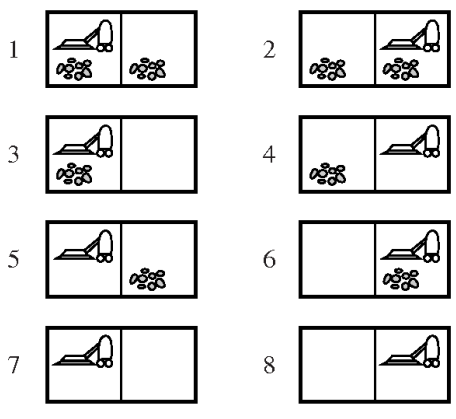
I begynnelsen vil populasjonen ofte bestå av individer som er svært ulike, slik at cross-over operasjonen produserer tilstander som er veldig ulike foreldrene. Etterhvert vil individene bli mer like og stegene blir mindre. I likhet med stokastisk beam search, vil genetisk algoritme kombinere en oppovertendens med tilfeldig utforskning og utveksling av informasjon blant parallelle søketråder. **Fordelen ved GA skyldes cross-over funksjonen, men denne vil kun være fordelaktig hvis stringene er meningsfulle komponenter**. Dvs. det er ikke like nyttig å bruke cross-over alene. Cross-over brukes for å kombinere blokker med nyttige funksjoner. For eksempel kan en blokk være å plassere dronningene i posisjon 2, 4 og 6, og når dette kombineres med andre blokker kan det hende man finner en løsning.

4.3 Søk med ikke-deterministiske handlinger

I kapittel 3 var omgivelsene fullt observerbare og deterministiske, slik at agenten visste effekten av handlingene sine. Dette gjør at agenten vil vite hvilken tilstand som resulterer av en bestemt handlingssekvens og hvilken tilstand den befinner seg i. Perseptene blir kun brukt for å bestemme den initiale tilstanden. **Når omgivelsen er delvis observerbar og/eller ikke-deterministisk, blir persepter mer nyttige.** I en delvis observerbar omgivelse vil agenten bruke persepter for å bestemme hvilken tilstand den er i, mens i en ikke-deterministisk omgivelser blir persepter brukt for å bestemme hvilke av de mulige utfallene hos handlingene som har skjedd. De fremtidige handlingene til agenten vil derfor avhenge av perseptene. **Løsningen til et problem er en beredskapsplan (eller strategi) som gir hva som skal gjøres avhengig av perseptene som mottas**. Det er altså ikke en handlingssekvens

Uberegnelig støvsugerverden

Vi ser på støvsugerverden, som har åtte tilstander i tilstandsrommet (se figur) og handlingene *Left, Right* og *Suck*. Målet er å rengjøre all skitten. Hvis omgivelsen er observerbar, deterministisk og kjent, er det lett å løse problemet med algoritmene fra kapittel 3 og løsningen er en handlingssekvens. Vi skal nå se på en ikke-deterministisk omgivelse med en uberegnelig støvsuger, som fungerer som følger:



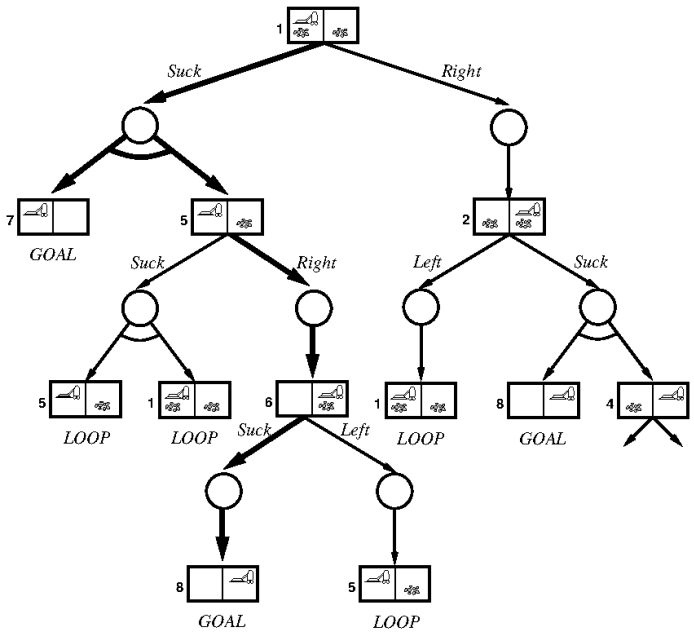
- Når *Suck* brukes på en ren lokasjon, vil den noen ganger skitne til lokasjonen
- Når *Suck* brukes på en skitten lokasjon, vil den rengjøre lokasjonen og noen ganger vil den også rengjøre nabolokasjonen

I dette tilfellet vil $Result$ i overgangsmodellen (s. 20) returnere et sett med mulige utfallstilstander, istedenfor en enkel tilstand. For eksempel vil $Results(1, Suck) = \{5, 7\}$. Løsningen på problemet vil ikke være en handlingssekvens, for hvis vi for eksempel starter i tilstand 1 er det ingen enkel handlingssekvens som løser problemet. Løsningen vil i stedet være en beredskapsplan, slik som: $[Suck, if\ State = 5\ then\ [Right, Suck]\ else\ [\,]]$. **Løsningen til en ikke-deterministisk problem kan altså inneholde nøstede if-then-else påstander.** Den er et tre istedenfor en sekvens, slik at handlinger kan velges basert på det som skjer ila utførelsen og man når måltilstanden uansett. Mange av problemene i den virkelige verden er ikke-deterministiske, fordi man klarer ikke å forutsi nøyaktig hva som vil skje.

AND-OR søketrær

For å finne beredskapsplanen til ikke-deterministiske problemer vil vi også her konstruere et søketre, men det har litt andre egenskaper. I en deterministisk omgivelse, vil branching skyldes **OR noder,** der agenten velger å utføre handling *Left* OR *Right* OR *Suck*. I en ikke-deterministisk omgivelse, vil branching i tillegg skyldes **AND noder**, der omgivelsene bestemmer utfallet til hver handling. For eksempel vil *Suck* handlingen i tilstand 1 kunne føre til tilstand 5 AND tilstand 7. Disse nodene vil alternere, slik at vi får et **AND-OR søketre** (se figur). En løsning vil være et subtre der:

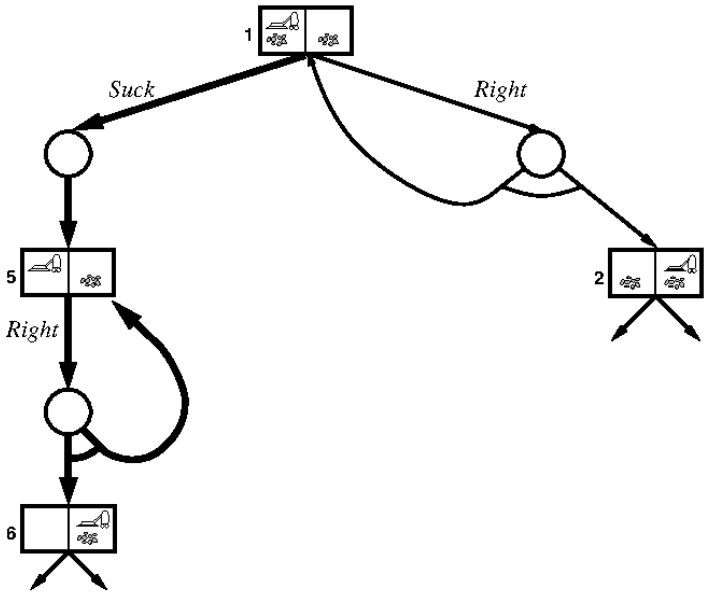
1. Alle bladnoder er en måltilstand
2. Ett utfall er gitt ved OR-noder
3. Alle utfall er gitt ved AND-noder



Løsningen er markert med fete linjer på figuren, og den korresponderer med $[Suck, if\ State = 5\ then\ [Right, Suck]\ else\ [\,]]$. For å finne løsningen i AND-OR grafer kan man bruke grafsøk med dybde-først, bredde-først, best-først, A*, osv., men det krever noen modifikasjoner.

Prøv, prøv igjen

Den glatte støvsugerverden er identisk til den vanlige støvsugerverden, bortsett fra at bevegelseshandlinger noen ganger feiler. For eksempel kan handlingen *Right* utført i tilstand 1 føre til tilstandene $\{1, 2\}$. Figuren viser deler av AND-OR søkegrafen. I dette tilfellet vil vi ha en **syklusløsning** som går ut på å gjenta *Right*, helt til det fungerer. I dette tilfellet vil løsningen være $[Suck, while\ State = 5\ do\ Right\ ]$. En syklusplan vil være en løsning dersom alle bladnoder er en måltilstand og en bladnode kan nås fra alle punkter i planen. En agent som



utfører en syklusløsning vil tilslutt nå målet, så lenge hvert utfall i den ikke-deterministisk handlingen til slutt vil skje. Dvs. så lenge støvsugeren til slutt vil klare å bevege seg til høyre.

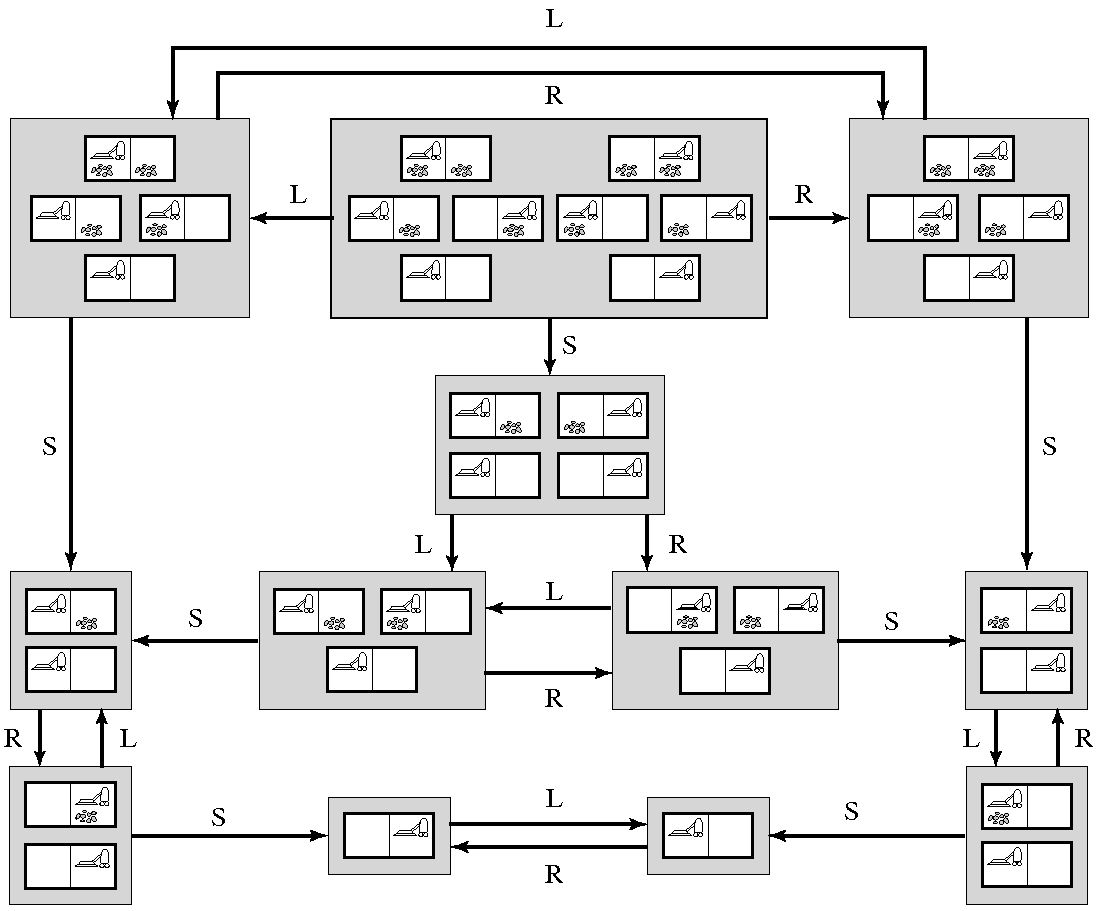
4.4 Søk med delvis observasjoner

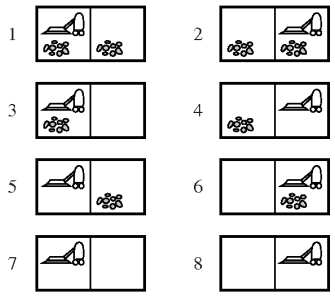
En omgivelse som er delvis observerbar betyr at perseptene til agenten ikke er tilstrekkelig for å bestemme den nøyaktige tilstanden. For å løse delvis observerbare problem blir det brukt såkalte **trotilstander** (*belief state*). En trotilstand vil representerer hva agenten tror om tilstanden den kan befinne seg i, gitt handlingssekvensen og perseptene den har oppfattet frem til det punktet.

Søk med ingen observasjoner

I sensorløse problem vil perseptene gi ingen informasjon. Slike problem kan ofte løses og sensorløse agenter er nyttige siden de ikke avhenger av at sensorene fungerer som de skal. I en sensorløs versjon av støvsugerverden, vil støvsugeren vite geografien til verden, men den vil ikke vite hvilken lokasjon den er i eller hvordan skitten er distribuert. Den initiale tilstanden vil være et av elementene i {1, 2, 3, 4, 5, 6, 7, 8}. Hvis agenten utfører handlingen *Right,* vil den være et av elementene i {2, 4, 6, 8}, altså vil den nå ha mer informasjon. Handlingssekvensen [*Right, Suck*] vil alltid ende i en av {4, 8}, mens handlingssekvensen [*Right, Suck, Left, Suck*] vil alltid nå måltilstanden 7, uansett hva starttilstanden er. Vi sier at agenten kan **tvinge** verden inn i tilstand 7.

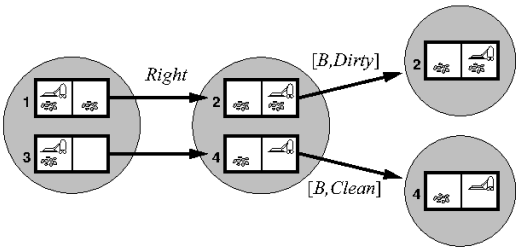
For å løse sensorløse problem vil vi søke i rommet av trotilstander istedenfor fysiske tilstander, og i dette rommet vil problemet være **fullt observerbart**, siden agenten alltid kjenner sin egen trotilstand. **Løsningen vil være en handlingssekvens, siden perseptene etter handlingen alltid er tomme og er dermed forutsigbare**. Dette er tilfellet, selv om omgivelsen er ikke-deterministisk. Figuren under viser rommet av trotilstander. Agenten vil befinne seg i en bestemt trotilstand, men vil ikke vite hvilken fysisk tilstand den er i. Den initiale trotilstanden er den øvre boksen i midten. **Standard søkealgoritmer kan brukes på rommet av trotilstander for å løse sensorløse problem**. Inkrementelle algoritmer kan brukes for å lage løsninger for hver tilstand innenfor trotilstanden, noe som ofte vil være mer effektivt. I noen tilfeller er det ikke mulig å finne en løsning uten persepter (eks: 8-puzzle).





Søk med observasjoner

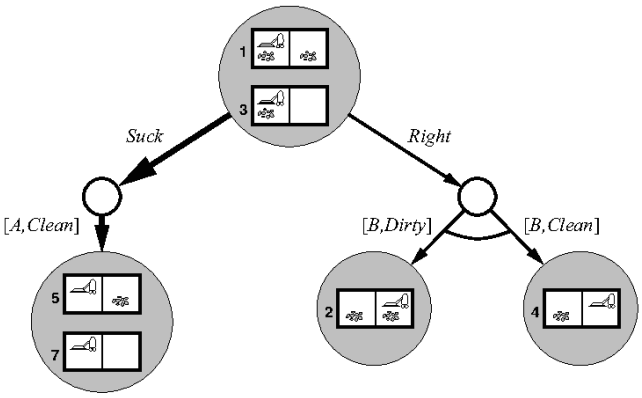
Når omgivelsen er delvis observerbar, må vi spesifisere hvordan omgivelsen lager persepter for agenten. For eksempel i lokal-sansing støvsugerverden kan agenten ha en sensor som detekterer lokasjonen og lokal skitt, men ikke skitt i andre lokasjoner. **Den formelle problemspesifikasjonen inkluderer $PERCEPT$-funksjonen, som returnerer perseptene som mottas i en gitt tilstand** (vil være et sett for ikke-deterministisk omgivelse). I støvsugerverden vil $PERCEPT(1) = [A, Dirty]$ for en delvis observerbar omgivelse, $PERCEPT(1) = 1$ for en fullt observerbar omgivelse og $PERCEPT(1) = null$ i en sensorløs omgivelse. **Når omgivelsene er delvis observerbare vil det ofte være tilfellet at flere tilstander kan produsere samme persepter**. For eksempel kan persept $[A, Dirty]$ produseres av tilstand 1 og 3. Hvis dette blir gitt som en initial persept, vil initial trotilstand være $\{1, 3\}$.



På figuren kan vi se et eksempel på en overgang i lokal-sansing støvsugerverden. Når agenten utfører en handling vil den gå fra en trotilstand til en annen trotilstand med to fysiske tilstander. Etter overgangen er det to mulige persepter $[B, Dirty]$ og $[B, Clean]$ og hvilken av disse som oppfattes av sansene vil avgjøre hvilken trotilstand agenten ender i. Størrelsen til trotilstanden kan ikke reduseres i en overgang, fordi observasjoner vil kun redusere usikkerheten over hvilken tilstand agenten befinner seg i.

Løse delvis observerbare problemer

Løsningen til et delvis observerbart problem vil være en beredskapsplan og den kan finnes ved å bruke en AND-OR søkealgoritme på trotilstand-problemet. For eksempel for lokal-sansing støvsugerverden med initial persept $[A, Dirty]$ vil løsningen være $[Suck, Right, if\ b\_state = \{5\}\ then\ suck\ else\ [\,]]$. Legg merke til at beregningsplanen tester trotilstanden istedenfor den fysiske tilstanden, fordi i en delvis observerbar omgivelse kan ikke agenten utføre en løsning som krever at den tester den faktiske tilstanden.



Agent i delvis observerbare omgivelser

En problemløsende agent for delvis observerbare omgivelser vil ligne den for fullt observerbare omgivelser (kapittel 3), ved at den vil formulere et problem, kalle på en søkealgoritme (eks: AND-OR grafsøk) for å finne løsningen og utføre løsningen. Det er to hovedforskjeller:

1. **Løsningen er en beredskapsplan med if-then-else påstander**, istedenfor en handlingssekvens (unntak: sensorløs)
2. **Agenten må opprettholde trotilstander**, ettersom den utfører handlinger og mottar persepter. Dette er en av kjernefunksjonene i delvis observerbare omgivelser.

Kapittel 5 – Motstandersøk

Dette kapittelet ser på problemer som dukker opp når vi forsøker å planlegge på forhånd i en verden der andre agenter planlegger mot oss.

Spill

I en multiagent omgivelse vil det være flere agenter som utfører handlinger og påvirker ytelsesmålet til hverandre (s. 11). En agent som befinner seg i en slik omgivelse må derfor ta hensyn til handlingene til andre agenter, siden det kan påvirke agentens problemløsende prosess. Dette kapittelet ser på konkurrerende omgivelser, der målene til agentene er motstridende og det oppstår **motstandersøk problemer**, også kjent som **spill.** Et spill kan være:

- **Deterministisk eller stokastisk** = i et deterministisk spill vil spilleren kjenne utfallet til alle handlinger, mens i et stokastisk/sjanse spill vil noen handlinger ha sannsynlige utfall (eks: rulle terning).
- **Perfekt eller imperfekt info** = i et spill med perfekt informasjon vil spilleren ved ethvert tidspunkt kjenne tilstanden til spillarenaen og de andre spillerne. For et spill med imperfekt informasjon vil noe informasjon om arenaen eller andre spillere være utilgjengelig.

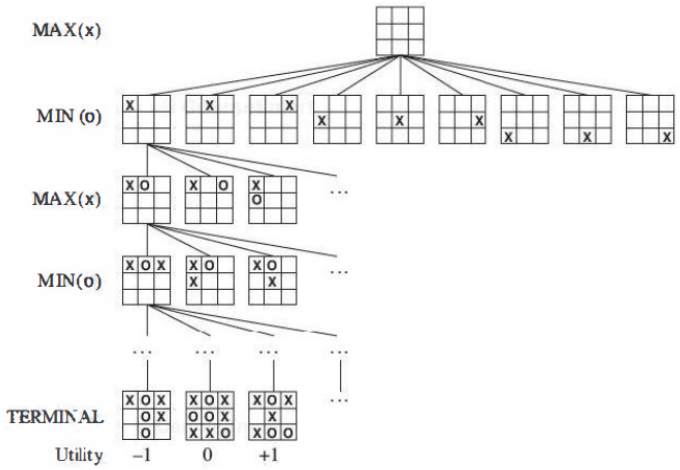
Figuren viser noen eksempler, for eksempel vil sjakk være deterministisk med perfekt informasjon, siden sjakkspilleren vil vite utfallet av enhver handling og har full informasjon om tilstanden til brettet og den andre spilleren. Poker er derimot et sjansespill med imperfekt informasjon, siden pokerspilleren ikke vet utfallet når kort deles ut og tilstanden til de andre spillerne er ukjent.

I spillteori (*game theory*) blir alle multiagent omgivelser behandlet som spill, så lenge agentene har en signifikant påvirkning på hverandre (konkurrerende eller samarbeidende). I AI er det vanligst at spillet er deterministisk og fullt observerbar med perfekt informasjon, der to agenter alternerer mellom å handle og nytteverdiene ved slutten av spillet vil være like store og motsatte (dvs. en vinner og en taper). Tilstanden til spill er som regel lett å representere og ett sett med regler vil ofte begrense hvilke handlinger agentene får utføre. I spill blir ineffektivitet straffet, for eksempel vil et ineffektivt sjakkprogram tape fordi det bruker for lang tid på å bestemme neste trekk.

Vi ser på spill med to spillere: MAX og MIN, med alternerende handlinger, der MAX får starte. Ved slutten av spillet vil vinneren få poeng, mens taperen blir straffet. **Et spill kan formelt defineres som en type søkeproblem med følgende elementer:**

- $\boldsymbol{S_0}$ = initial tilstand, som gir hvordan spillet er satt opp ved starten
- $\boldsymbol{PLAYER(s)}$ = gir hvilken spiller som får handle i en tilstand
- $\boldsymbol{ACTION(s)}$ = gir settet av lovlige handlinger i en tilstand
- $\boldsymbol{RESULT(s, a)}$ = overgangsmodellen, som gir resultatet til en handling
- $\boldsymbol{TERMINAL\text{-}TEST(s)}$ = en terminal test som er sann når spillet er over
- $\boldsymbol{UTILITY(s, p)}$ = nyttefunksjonen (objektivfunksjonen) som gir poenget som spiller p mottar når spillet er over. For eksempel i sjakk vil spillet være seier, tap eller uavgjort med verdiene +1, 0 eller ½. I **zero-sum spill** vil total verdi være den samme for alle instanser av spillet. Dette er tilfellet i sjakk, siden den totale verdien alltid er 1 (= 1+0, 0+1, ½+½)

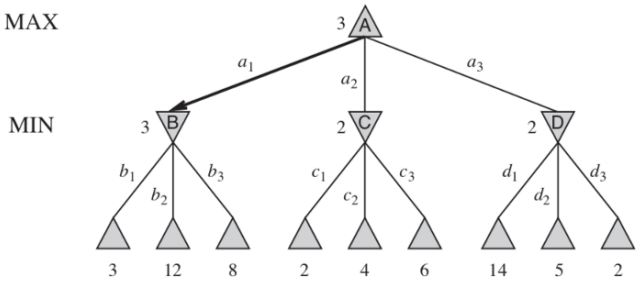
Spilltreet er et søketre der nodene er spilltilstander og kantene er bevegelser, og det er definert av den initiale tilstanden, $ACTIONS$ og $RESULT$. Forskjellen fra søketrær hos standard søk, er at **ved hvert nivå vil det alternere mellom hvilken av de to spillerne som har kontroll og utfører en bevegelse**. Figuren viser deler av spilltreet for tic-tac-toe. **Spilleren som vi ønsker skal vinne, vil starte spillet**. I dette tilfellet er det MAX, og fra den initiale tilstanden kan MAX velge mellom ni ulike handlinger. Spillet vil alternere mellom at MAX plasserer en X og MIN plasserer en O, helt til vi når bladnodene som korresponderer til terminale tilstander, der en spiller har tre på rad eller brettet er fullt. **Nummeret på hver bladnode gir nytteverdien (*utility*) for MAX-spilleren ved den terminale tilstanden. Høye verdier er derfor bra for MAX og dårlig for MIN**. **Målet med søket er å maksimere nytteverdien til spilleren vi ønsker skal vinne.** Spilltreet vil ofte være veldig stort, slik at det er umulig å lage i den fysiske verden.



Optimale avgjørelser i spill

I et normalt søk vil den optimale løsningen være en handlingssekvens som fører til en terminal tilstand der MAX vinner. I motstandersøk vil MIN forsøke å motarbeide dette. MAX må derfor finne en betinget **strategi** som gir hvordan MAX skal handle i den initiale tilstanden og hvordan MAX skal handle i respons til MIN sine handlinger. **Løsningen til et motstandersøk (spill) vil altså være en strategi som gir hvordan MAX skal handle i den initiale tilstanden og i respons til handlingene MIN utfører.** Her antar vi at motstanderens handlinger er uforutsigbare (hvis ikke vil løsningen være en handlingssekvens). En optimal strategi vil føre til et utfall som er minst like bra som enhver annen strategi, når man spiller mot en motstander som ikke gjør feil.

For å vise hvordan man kan finne den optimale strategien, skal vi se på et zero-sum spill kalt two-ply (se figur). Dette spillet er over når MAX og MIN har utført ett trekk hver. De mulige handlingene for MAX ved roten er A_1, A_2 og A_3, og MIN kan respondere med A_{11}, A_{12}, osv.. Nytteverdien ligger mellom 2 og 14.



Dersom et spilltre er gitt, kan den optimale strategien bestemmes ved å bruke minimax verdien til hver node ($MINIMAX(n)$). Minimax verdien er nytteverdien hos MAX for å være i den korresponderende tilstanden, gitt at begge spillerne vil spille optimalt fra denne noden til enden av spillet. For en terminal tilstand vil minimax verdien være nytteverdien. MAX vil foretrekke å gå til en tilstand med maksimal minimax verdi, mens MIN vil foretrekke å gå til en tilstand med minimal minimax verdi:

$$MINIMAX(s) = \begin{cases} UTILITY(s) & if\ TERMINAL\_TEST(s) \\ \max(MINIMAX(Result(s,a)) & if\ PLAYER(s) = MAX \\ \min(MINIMAX(Result(s,a)) & if\ PLAYER(s) = MIN \end{cases}$$

For two-ply spilltreet vil minimax-verdiene til terminaltilstandene være nytteverdiene som gis av $UTILITY(s)$ funksjonen til spillet. Den første MIN-noden, markert som B, har tre etterfølgere med verdi 3, 12 og 8, slik at dens minimax verdi er 3. MAX-noden ved roten har også tre etterfølgere med verdi 3, 2 og 2, slik at dens minimax verdi er 3. Ved roten vil

minimax avgjørelsen være a_1, siden dette er det optimale valget for MAX som fører til tilstanden med størst minimax verdi. Her har vi antatt at MIN spiller optimalt, slik at den maksimerer *worst-case* utfallet for MAX. Hvis MIN ikke spiller optimalt vil MAX gjøre det enda bedre. Det kan finnes strategier som fungerer bedre for suboptimale motstandere, men disse er verre for optimale motstandere.

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game

  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
  return v
```

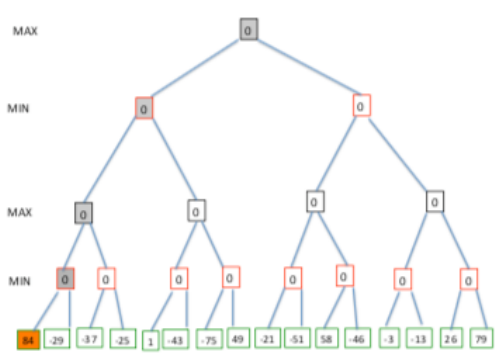
Minimax algoritmen

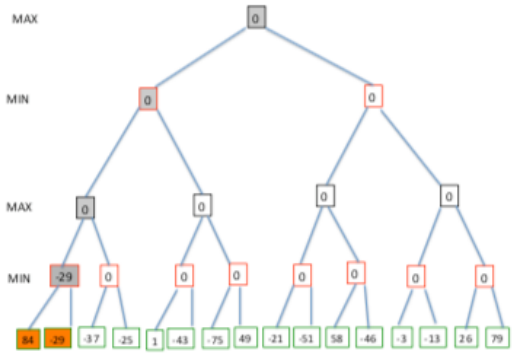
Minimax algoritmen vil bestemme minimax avgjørelsen ved å rekursivt regne ut minimax verdiene til alle etterfølgertilstandene. Algoritmen bruker **en dybde-først utforskning** av spilltreet. Den beveger seg nedover treet helt til den når terminaltilstander, der minimax verdiene er lik nytteverdiene. Deretter vil den bruke disse for å rekursivt finne minimax verdiene til forgjengernodene. MIN-nodene vil velge den minste minimax verdien blant alle etterfølgerne, mens MAX-nodene vil velge den største minimax verdien blant alle etterfølgerne. Evaluering av minimax algoritmen gir følgende dimensjoner:

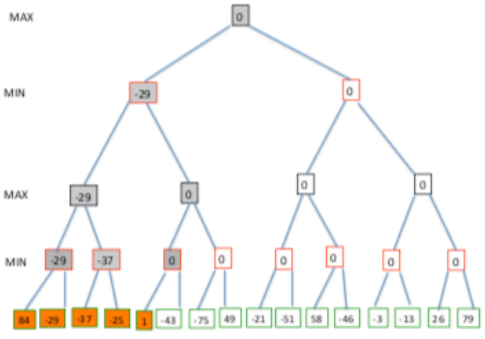
Minimax-algoritmen brukes på zero-sum spill med perfekt informasjon (dvs. fullt observerbare). Kan være to eller flere spillere.

- **Fullstendig** = Ja, hvis treet er endelig
- **Optimal** = Ja, hvis motstanderen spiller optimalt
- **Tid** = $O(b^m)$, siden den bruker DFS utforskning
- **Rom** = $O(bm)$, siden den bruker DFS utforskning

For sjakk vil $b \approx 35$ og $m \approx 100$ for «fornuftige» spill, så en eksakt løsning er ikke alltid mulig (krever $O(35^{10})$ noder). Senere skal vi se at det ikke er nødvendig å utforske alle baner. Figurene under viser et eksempel på hvordan minimax algoritmen fungerer.







Den optimale strategien er markert i gult.

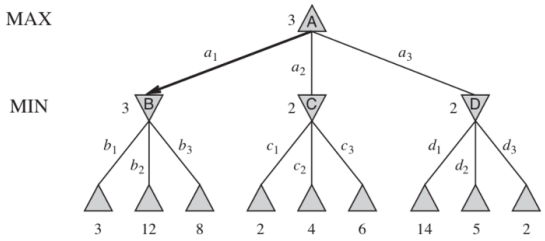
Minimax algoritme for flere spillere

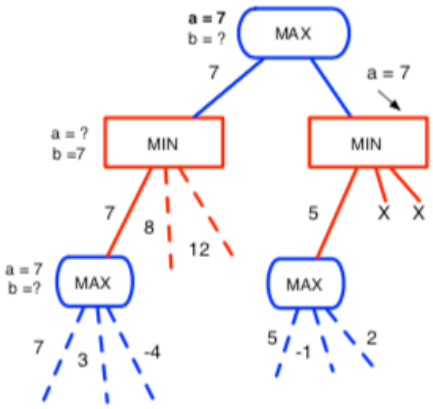
Mange populære spill tillater mer enn to spillere. Hvis vi skal bruke minimax algoritmen for å finne den optimale strategien i spill med flere spillere må vi utføre noen endringer. For et spill med tre spillere: A, B og C må minimax verdiene representeres som en **vektor** $\langle v_a, v_b, v_c \rangle$ ved hver node. For terminale tilstander vil denne gi nytteverdien for alle spillerne. **Nodene vil velge handlingen som vil maksimere deres minimax verdi, altså etterfølgeren der deres verdi i vektoren er maksimert**. For eksempel for noden markert med X i figuren, er det spiller C som skal utføre en handling som vil føre til etterfølger med vektor $\langle 1, 2, 6 \rangle$ eller $\langle 4, 2, 3 \rangle$. For å maksimere sin minimax verdi, vil spiller C utføre handlingen som fører til etterfølgeren med minimax verdi $\langle 1, 2, 6 \rangle$. Det samme gjelder de andre nodene. Det kan også oppstå **allianser** mellom spillere.

Alpha-Beta pruning

Problemet med minimax algoritmen er at antall spilltilstander den må utforske er eksponentiell med dybden til treet. Denne eksponenten kan ikke elimineres, men den kan halveres, siden det er mulig å bestemme riktig minimax avgjørelse uten å se på alle noder i spilltreet. **Alpha-Beta pruning vil returnere samme handling som minimax, men den ser bort fra greiner som ikke kan påvirke den endelige avgjørelsen.**

Alpha-Beta pruning kan brukes for å finne minimax avgjørelsen hos two-ply spillet uten å evaluere to av bladnodene. Algoritmen vil først finne at handling a_1 vil føre til minimax verdi 3, og når den deretter finner at handling c_1 gir minimax verdi 2, vet den at MAX-noden vil aldri velge handlingen som fører til MIN-node C, siden $3 > 2$. Derfor vil ikke algoritmen se på bladnodene med nytteverdi 4 og 6.





Figuren viser enda et eksempel. DFS utforskningen gjør at algoritmen vil se på venstre subtre først, der resultatet er at venstre MIN-noden får minimax-verdi 7. Dette blir lagret som en midlertidig minimax verdi i MAX-noden. Algoritmen vil deretter se på høyre subtre. Den ene etterfølgeren til høyre MIN-noden gir en minimax verdi på 5. Siden venstre MIN-node alltid vil velge den minst verdien, vil minimax verdien være maksimalt 5. MAX-noden vil derfor aldri velge denne MIN-noden, fordi den vil heller velge venstre MIN-node med minimax verdi $7 > 5$. Derfor er det ikke nødvendig å undersøke verdiene hos etterfølgerne markert med X. Dette kan være store subtrær, som ikke trenger å genereres fordi de vil ikke påvirke valget ved roten.

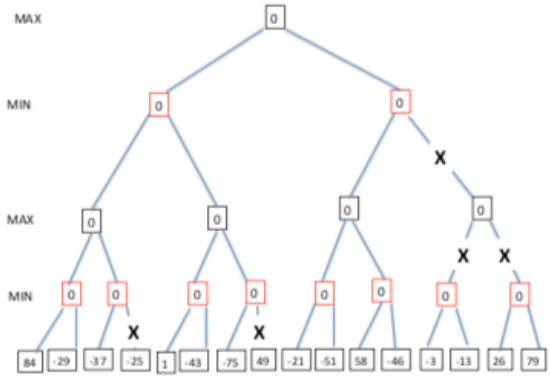
Alpha-Beta pruning bruker:

- **α = den største minimax verdien vi har funnet så langt blant etterfølgerne til en MAX-node.** Når det oppdages en MIN-node med større minimax verdi, blir α satt lik denne verdien (α er derfor nedre grense underveis i søket). Det brukes for pruning av MIN-noder: **en MIN-node blir ikke videre utforsket dersom det oppdages at den har en etterfølger som er mindre enn α.**
- **β = den minste minimax verdien vi har funnet så langt blant etterfølgerne til en MIN-node**. Når det oppdages en MAX-node med mindre minimax verdi, blir β satt lik denne verdien (β er derfor øvre grense underveis i søket). Det brukes for pruning av MAX-noder: **en MAX-node blir ikke videre utforsket dersom det oppdages at den har en etterfølger som er større enn β.**

Disse verdiene blir delt mellom nodene i spilltreet.

Figuren viser algoritmen for Alpha-Beta søk, som tar inn en initial tilstand og returnerer minimax avgjørelsen. Den begynner med å kalle på MAX-VALUE siden det er MAX-noden som utfører minimax avgjørelsen. Denne funksjonen vil returnere den største minimax verdien blant MIN-node etterfølgerne. Algoritmen gjør dette vha rekursive kall helt til den når en terminal tilstand der TERMINAL-TEST vil være true. Legg merke til "if $v \geq \beta$ then return v" og "if $v \leq \alpha$ then return v", som implementerer selve pruningen. Når det rekursive kallet blir returnert, vil det ikke skje noe videre utforskning i denne delen av spilltreet.

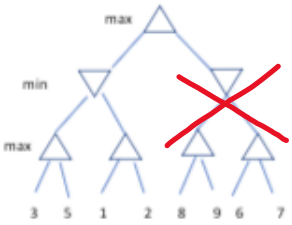
```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE (state, −∞ +∞)
    return the action a in ACTIONS(state) with value v))
function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```



Figuren viser hvordan Alpha-Beta søk ville ha redusert antall tilstander som blir utforsket under løsningen av problemet på side 46. **Pruningen vil ikke påvirke det endelige resultatet.**

Rekkefølgen til handlingene

Effektiviteten til pruningen er svært avhengig av rekkefølgen til utforskningen av tilstandene, siden det avgjør rekkefølgen til oppdagelsen av nytteverdiene (eks: best-først, tilfeldig). Dersom de «dårlige» etterfølgerne blir generert først, vil det ikke være noen pruning og alfa-beta søket bruker like lang tid på å finne løsningen som minimax (dvs. $O(b^m)$). **Dersom de derimot har «perfekt rekkefølge» (best-først) kan tiden halveres til $O(b^{m/2})$, siden man kan se bort fra halve spilltreet (se figur).** Dette gjør at alpha-beta kan bruke like langt tid som minimax søk på å løse et spilltre som er nesten dobbelt så dypt. Ved å bruke riktig rekkefølge på undersøkelsen av tilstandene, kan den løselige dybden altså dobles. Hvis etterfølgerne blir undersøkt i tilfeldig rekkefølge vil tidskompleksiteten bli $O(b^{3m/4})$.



Det er ikke mulig å alltid velge de beste etterfølgerne i praksis, så $O(b^{m/2})$ er en teoretisk grense. Det finnes likevel taktikker for å komme nær denne grensen. For eksempel kan man bruke **dynamisk bevegelsesorden skjema**, der man forsøker å utføre bevegelser som var de beste tidligere i spillet (samme trussel vil ofte komme flere ganger). Man kan også bruke iterativt fordypende søk (IDS) for å få mer informasjon om nåværende bevegelse (gir relativt lite økning i kostnad). De beste bevegelsene kalles *killer moves*. En annen taktikk er å lage en hashtabell kalt **transposisjonstabell**, som vil lagre tilstander, slik at hvis de oppstår på nytt slipper algoritmen å utføre beregningene på nytt.

Ufullkomne beslutninger i sanntid

Både minimax og alfa-beta må søke helt ned til de terminale tilstandene, noe som ofte ikke er praktisk fordi bevegelser må utføres innen en fornuftig tid. **I stedet kan søket avsluttes tidligere ved at det bruker en heuristisk evalueringsfunksjon på tilstandene, slik at de ikke-**

terminale nodene blir omdannet til terminale bladnoder. Minimax og alfa-beta blir dermed endret på to måter:

1. **UTILITY-funksjonen erstattes med en heuristisk evalueringsfunksjon, EVAL, som vil estimere nytteverdien til en posisjon**
2. **TERMINAL-TEST erstattes med en cutoff test som bestemmer når EVAL skal brukes**

Vi får følgende definisjon av heuristisk minimax for tilstand s og maksimum dybde d:

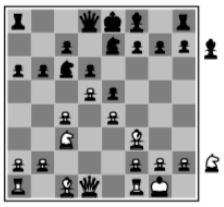
$$H\_MINIMAX(s,d) = \begin{cases} EVAL(s) & if\ CUTOFF\_TEST(s,d) \\ \max(H\_MINIMAX(Result(s,a), d+1) & if\ PLAYER(s) = MAX \\ \min(H\_MINIMAX(Result(s,a), d+1) & if\ PLAYER(s) = MIN \end{cases}$$

Evalueringsfunksjoner

En evalueringsfunksjon vil gi et estimat på forventet nytteverdi (*utility*) i spillet fra en gitt posisjon, slik en heuristisk funksjon vil gi et estimat på avstanden til målet. **Evalueringsfunksjonen må ordne de terminale tilstandene på samme måte som UTILITY-funksjonen, ved å evaluere seier bedre enn uavgjort, som igjen evalueres bedre enn tap**. Hvis ikke vil agenten som bruker evalueringsfunksjonen kunne tape, selv om den ser hele veien til slutten av spillet. Det er også viktig at **beregningene ikke tar for lang tid**, for hele poenget med evalueringsfunksjonen er at søket skal gå raskere. **Evalueringsfunksjonen hos ikke-terminale tilstander bør være sterkt korrelert med de faktiske sjansene for å vinne** (dvs. den bør gjette det endelige utfallet, gitt den begrenset mengden beregninger den kan gjøre for hver tilstand).

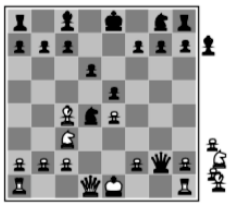
De fleste evalueringsfunksjonene fungerer ved å regne ut ulike egenskaper ved en tilstand, for eksempel i sjakk kan det være antall hvite brikker, svarte brikker, hvite dronninger, osv. **Evalueringsfunksjonen vil deretter regne ut det numeriske bidraget til hver egenskap og kombinerer disse for å finne den totale verdien**. For eksempel i sjakk vil hver brikke ha en materialverdi (eks: bonde = 1, dronning = 9) og disse verdiene blir summert for å finne evalueringen av en posisjon. Dette kalles en **vektet lineær evalueringsfunksjon**:

$$EVAL(s) = w_1 f_1 + w_2 f_2 + \cdots + w_n f_n$$



Black to move

White slightly better



White to move

Black winning

Der w_i er vekten og f_i er en egenskap ved posisjonen. I sjakk kan f_i være antallet av hver type brikke, mens w_i kan være materialverdiene. For eksempel kan $w_1 = 9$ og $f_1(s) = \#\text{hvite dronninger} - \#\text{svarte dronninger}$. På figuren kan vi se eksempler der denne evalueringsfunksjonen har blitt brukt for å bestemme om hvit eller svart spiller ser ut til å vinne. Denne evalueringen antar at bidraget til hver egenskap er uavhengig av verdiene til andre egenskaper. Når biskopen får verdi 3 har man ignorert av biskoper blir mer verdifulle i siste del av spillet, når de har mye plass å bevege seg.

Cutoff søk

Neste steg er å modifisere alfa-beta søkealgoritmen, slik at den kaller på den heuristiske EVAL-funksjonen når søket skal kuttes av. Dette gjøre ved å erstatte TERMINAL-TEST med CUTOFF-TEST. **Cutoff-funksjonen vil ta inn en dybde som representerer hvor dypt i spilltreet algoritmen vil søke, før den avslutter søket og utfører den heuristiske evalueringen. Det er ulike måter å bestemme denne dybden**. Den enkleste er å sette en fast dybde d, som vil gjøre at en bevegelse returneres innenfor den gitte tiden. En mer robust måte er å bruke **iterativt fordypende søk** (IDS), som vil returnere bevegelsen til det

dypeste fullførte søket når tiden er over. Disse tilnærmingene kan føre til feil, siden de ikke tar hensyn til om neste bevegelse vil innebære store endringer i de heuristiske verdiene. **Ved hvilende søk (*quiescence search*) blir evalueringsfunksjonen kun brukes på hvilende posisjoner, som er tilstander der det er lite sannsynlig at verdien vil svinge voldsomt i nær fremtid.** For eksempel i sjakk vil en posisjon der det er mulig å utføre gunstige fangst (eks: ta dronningen med bonde) ikke være hvilende, og søket vil dermed ikke stoppe ved denne posisjonen. Søket vil utvide denne posisjonen helt til den finner en hvilende posisjon. Det finnes også andre effekter cutoff må ta hensyn til.

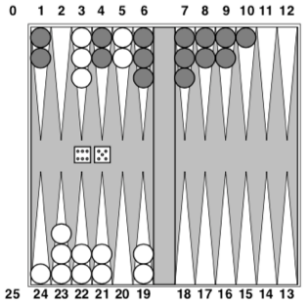
Forward pruning

Forward pruning er når noen bevegelser ved en node blir umiddelbart kuttet bort uten videre betraktning. Når mennesker spiller sjakk vil de vurdere noen få bevegelser fra hver posisjon. En tilnærming til forward pruning er **beam søk**, som ved hvert trekk vil kun se på de n beste bevegelsene med høyest evalueringsfunksjon. Dette er en farlig strategi, fordi det kan hende at den beste bevegelsen blir kuttet vekk. For å redusere sannsynligheten for dette, kan man bruke statistikk fra tidligere erfaring (PROBCUT programmet). Et grunt søk blir utført for å finne en minimax verdi hos en node og tidligere erfaring brukes for å bestemme hvor sannsynlig det er at denne verdien er utenfor (α, β) vinduet. Dette programmet kan gjøre feil, men det vil bruke mye mindre tid.

Dersom alle disse teknikkene kombineres, vil det være mulig å lage et program som kan spille sjakk relativt bra.

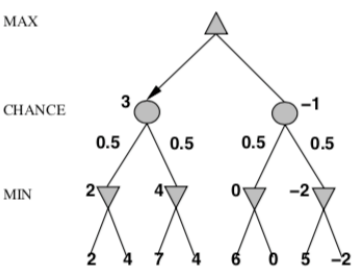
Søk vs. oppslag

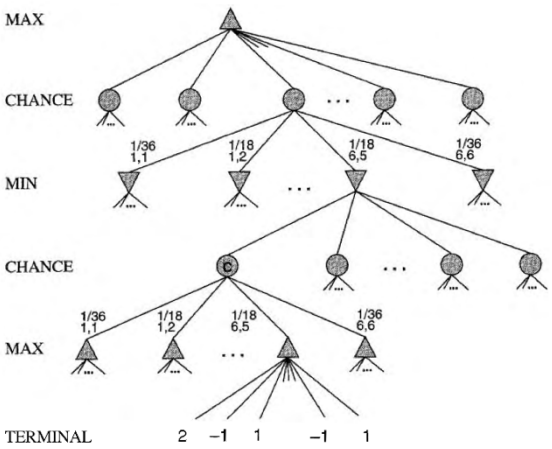
I starten av sjakk, er det vanlig at programmet bestemmer åpningen ved slå opp i samlinger av åpninger fra bøker laget av menneskelige eksperter. Programmet kan også bruke statistikk om hvilke åpninger som oftest fører til seier. Etter 10 trekk vil spillet som regel entre en tilstand som er sjeldent sett, og da vil programmet bytte fra oppslag til søk. Ved enden av spillet vil det bli færre mulige posisjoner, slik at det igjen blir mulige å utføre oppslag.



Stokastiske spill

Stokastiske spill inkluderer tilfeldige elementer (eks: terningkast), som gjør at utfallet ved handlinger blir noe usikkert. Backgammon er et eksempel på et stokastisk spill som kombinerer hell og ferdigheter, ved at antall lovlige bevegelser i spillerens tur bestemmes av et terningkast. I dette tilfellet er det ikke mulige å lage et standard spilltre. **For stokastiske spill vil spilltreet inkludere sjansenoder i tillegg til MAX og MIN nodene.** Disse vises som sirkler i spilltreet (se figur). **Banene ut fra en sjansenode representerer de mulige utfallene, og de er merket med utfallet og sannsynligheten for utfallet**. For eksempel ser vi at terningkastet kan ha utfallet (1, 1) med sannsynlighet 1/36, mens utfallet (1, 2) kan skje med sannsynlighet 1/18. Figuren til venstre viser et forenklet stokastisk tre for mynt-flipping, der utgående baner fra sjansenoder er kun merket med sannsynlighetene.





Algoritmen for stokastiske spill

```
...
if state is a MAX node then
        return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a MIN node then
        return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a chance node then
        return sum of ..... EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
alternative: ...
```

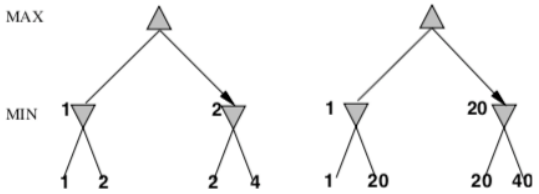
I et stokastisk spill vil sjansenodene gjør det slik at posisjonene ikke vil ha bestemte minimax verdier. For å finne den beste bevegelsen, må vi i stedet regne ut den forventede verdien til posisjonen, som er gjennomsnittlig verdi blant alle utfallene til sjansenodene. **Spill med sjansenoder vil bruke en expectiminimax verdi, som tilsvarer minimax verdien i deterministiske spill.** Algoritmen vil fungere som før, bortsett fra at den må kunne håndtere sjansenodene. Hvis tilstanden er en sjansenode, vil den regne ut den forventede verdien ved å summere verdiene til alle utfallene, vektlagt med sannsynligheten til hvert utfall. For å finne verdiene må algoritmen reise rekursivt nedover spilltreet til den når terminal tilstander, der den vil bruke nytteverdiene og forplante disse oppover. MAX-noden velger sjansenoden med størst expectiminimax, mens MIN-noden velger sjansenoden med minst expectiminimax:

$$EXPECTIMINIMAX(s) = \begin{cases} UTILITY(s) & if\ TERMINAL\_TEST(s) \\ \max(EXPECTIMINIMAX(RESULT(s,a)) & if\ PLAYER(s) = MAX \\ \min(EXPECTIMINIMAX(RESULT(s,a)) & if\ PLAYER(s) = MIN \\ \sum_r P(r)EXPECTMINIMAX(RESULT(s,r)) & if\ PLAYER(s) = CHANCE \end{cases}$$

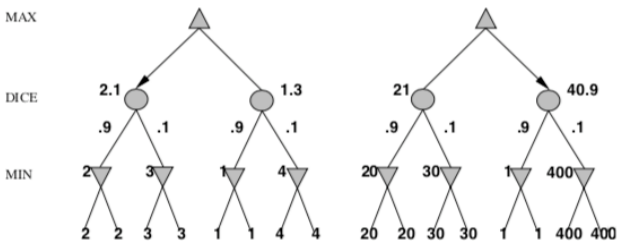
Her vil r være et mulig utfall.

Evalueringsfunksjoner i stokastiske spill

For evalueringsfunksjonen ved deterministiske spill er det ikke så viktig at verdiene til UTILITY-funksjonen er nøyaktige, fordi det er kun rekkefølgen som er vesentlig. **Evaluering i deterministisk spill bruker en ordinal *utility* funksjon**. På figuren kan vi se at høyre handling vil være den beste uansett om nytteverdiene er [1, 2, 4] eller [1, 20, 40].



I likhet med minimax, kan expectiminimax søket tilnærmes ved å avslutte ved et punkt og deretter bruke en evalueringsfunksjon på posisjonen for å finne et estimat på nytteverdien. Forskjellen er at man må være mer forsiktig med evalueringsverdiene, så man kan ikke bare gi høyere verdier til bedre posisjoner. **I stokastiske spill blir nytteverdiene brukt for å regne ut expectiminimax, som avgjør hvilken bane algoritmen følger**. Hvis verdiene ikke er nøyaktige vil det føre til at sjansenoden får feil expectiminimax verdi, noe som kan resultere i at algoritmen velger feil bane i spilltreet. **Evaluering i stokastiske spill må være proporsjonal med forventet payoff.** På figuren ser vi at venstre handling er bedre når nytteverdiene er [1, 2, 3, 4], mens høyre handling er bedre når nytteverdiene er [1, 20, 30, 400]. Programmet vil altså oppføre seg helt ulikt hvis vi endrer skalaen til noen av evalueringsverdiene.

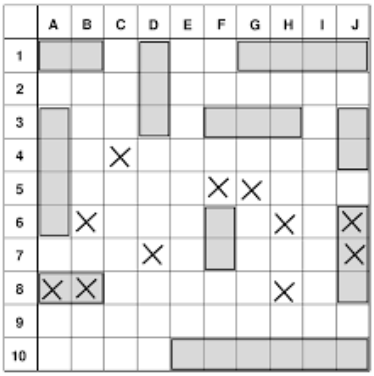


Hvis programmet vet utfallet av alle terningkast i spillet, vil spillet være deterministisk og kan løses av minimax på tiden $O(b^m)$**. I et stokastisk spill vil expectiminimax i tillegg vurdere alle sekvensene til de n utfallene, slik at det tar $O(b^m n^m)$ tid for å finne løsningen**. Den ekstra kostnaden gjør at dybden til expectiminimax søk blir begrenset. Dette er et generelt problem når usikkerhet blir en del av problemet. Det kan forbedres ved å bruke en form for alfa-beta pruning på sjansenodene (kan regne ut gjennomsnitt uten å se på alle verdiene).

Delvis observerbare spill

Delvis observerbare spill kan innebære at eksistensen og plasseringen av fiender er ukjent, slik at den bruker spioner og speidere for å samle informasjon og hemmeligheter og bløffer for å forvirre motstanderen.

Deterministiske, delvis observerbare spill



I deterministiske, delvis observerbare spill skyldes usikkerheten i tilstanden at man mangler informasjon om motstanderens valg. For eksempel i Battleship-spillet kan man ikke se hvordan den andre spilleren har plassert sine skip. Det er ingen usikkerhet i sammenheng med utfallet av en handling, for eksempel vil man enten treffe et skip eller ikke når man oppgir en posisjon til motstanderen i Battleship. Et annet eksempel er Kriegspiel, som er en delvis observerbar versjon av sjakk, der brikkene kan flyttes, men er usynlige for motstanderen. **For å finne løsningen må man bruke trotilstander (*belief state*) og strategier som velger bevegelser basert på perseptsekvenser som mottas**. Optimal spilling i delvis observerbare spill, krever at spilleren velger noen tilfeldige valg for å være uforutsigbar. Dette vil gjøre at motstanderen får mindre informasjon om deres lokasjon av brikker.

Stokastiske, delvis observerbare spill

I stokastiske, delvis observerbare spill vil manglende informasjon genereres tilfeldig. For eksempel i flere typer kortspill vil kort deles ut tilfeldig og kortene til motstanderen er ukjent. Den tilfeldige utdelingen av kortene er den stokastiske egenskapen ved spillet, mens de skjulte kortene til motstanderen er den delvis observerbare egenskapen ved spillet. For å finne løsningen kan man anse alle tilstander, løse disse som om de var fullt observerbare ved å bruke minimax søk og deretter velge bevegelsen som har best utfall blant alle tilstandene. Se side 183-184 for mer forklaring ☺

Topp moderne spill

En vanlig analogi er at spill er for AI det samme som grand prix racing er for bildesing. Topp moderne spillprogram er svært raske og høyt optimaliserte maskiner som bruker de siste fremskrittene innenfor teknologi, men de kan ikke brukes til mye annet enn å utføre spillet. De vil likevel inspirere og føre til en strøm med innovasjon som kan brukes til å løse andre problem. Det har blitt utviklet programmer som har slått eksperter innenfor deterministiske og fult observerbare spill som sjakk, dam og Othello. Mennesker har fortsatt overtaket i flere spill der informasjonen er ufullstendig, slik som poker, bridge og Kriegspiel (blindsjakk) eller spill som Go, der branching faktoren er svært stor og det er lite god heuristisk kunnskap.

Det er morsomt å arbeide med spill, men de kan også illustrere viktige poeng ved AI:

1. **Perfeksjon er uoppnåelig og må derfor tilnærmes** – det er blant annet umulig å utforske tilstandene i en «perfekt rekkefølge», slik at alpha-beta pruning kan halvere søketreet. Tiden på $O(b^{m/2})$ er en nedre teoretisk grense, som ikke er mulig å oppnå i praksis, men som man kan tilnærmes ved å bruke metoder som transposisjonstabell
2. **Det er lurt å resonnere om hvilke beregninger som skal utføres (= metaresonnering)** = bruken av alfa-beta pruning kan redusere antall tilstander som må utforskes, slik at søket kan bruke samme tid for å finne løsningen til et enda dypere tre
3. **Usikkerhet begrenser tildelingen av verdier til tilstander** = når usikkerhet blir en del av problemet må verdiene være mer nøyaktig, for ellers kan det hende at søkealgoritmen tar feil valg og klarer dermed ikke å finne den optimale løsningen.

Kapittel 6 – Constraint Satisfaction Problems

Vi har sett at problemer kan løses ved å søke i et rom av tilstander, som kan evalueres av problemspesifikk heuristikk og testes for å se om de er måltilstander. Fra synspunktet til søkealgoritmene blir tilstanden representert som en black-box som er atomisk og har ingen intern struktur. For å løse problemer mer effektivt, kan vi bruke en **faktorert representasjon** av tilstander, der hver tilstand gis et sett med variabler og tilhørende verdier. **Et problem vil være løst når hver variabel har en verdi som tilfredsstiller alle begrensningene på variabelen**. **Et problem som blir beskrevet slik, kalles et Constraint Satisfaction Problem (CSP)**. CSP søkealgoritmer bruker strukturen til tilstandene og generell heuristikk for å løse komplekse problem. **Ideen er at store deler av søkerommet vil elimineres med en gang ved å identifisere variabel/verdi kombinasjoner som bryter begrensningene**.

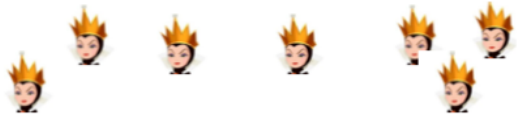
Definisjon av CSP

Et Constraint Satisfaction Problem (CSP) består av tre komponenter:

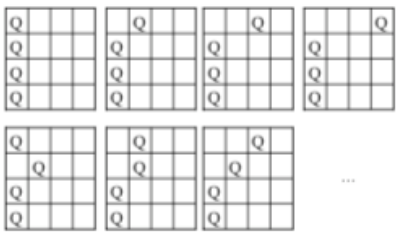
- $\boldsymbol{X}$ **= et sett med variabler** $\{X_1, \dots, X_n\}$
- $\boldsymbol{D}$ **= et sett med domener** $\{D_1, \dots, D_n\}$, en for hver variabel. Hvert domene gir et sett med lovlige verdier $\{v_1, \dots, v_k\}$ for variabelen X_i
- $\boldsymbol{C}$ **= et sett med begrensninger** mellom variabler. Hver begrensning består av paret $\langle scope, rel \rangle$, der $scope$ er en tuple av variablene som deltar i begrensningen og rel er relasjonen som definerer hvilke verdier variablene kan ha. Relasjonen kan være en liste over tuple-verdiene som tilfredsstiller begrensningen eller en abstrakt relasjon. For eksempel for X_1 og X_2 som begge har domene $\{A, B\}$, kan begrensningen som sier at disse ikke kan være like skrives som $\langle (X_1, X_2), [(A, B), (B, A)] \rangle$ eller $\langle (X_1, X_2), X_1 \neq X_2 \rangle$

Det grunnleggende problemet er å finne en verdi $\boldsymbol{d_i} \in \boldsymbol{D_i}$ for hver variabel $\boldsymbol{X_i}$, slik at alle begrensningene i $\boldsymbol{C}$ er tilfredsstilt. Hver tilstand i en CSP vil være en tildeling av verdier til noen eller alle variablene: $\{X_i = v_i, X_j = v_j, \dots\}$. En konsistent tildeling vil ikke bryte noen begrensninger, en fullstendig tildeling vil gi verdi til alle variablene og en delvis tildeling vil gi verdi til noen av variablene. **En løsning i CSP er en konsistent og fullstendig tildeling, dvs. alle variablene får verdier og ingen bryter begrensningene**

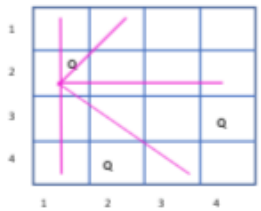
Eksempel – N-queen problemet (motivasjon)



N-queen problemet går ut på å plassere N dronninger på et $N \times N$ sjakkbrett, med begrensningen at de ikke skal kunne angripe hverandre (dvs. være på samme horisontale, vertikale eller diagonale linje). For 8-queen problemet vil det være 92 ulike løsninger. Når mennesker skal løse dette problemet vil de eksperimentere med ulike konfigurasjoner og bruke innsikt om problemet for å redusere antall konfigurasjoner som må undersøkes. Det vil være svært vanskelig for mennesker å løse 1000-queen problemet.



Derfor trenger vi datamaskiner for å løse slike problem. Datamaskiner er flinke til å gjøre små ting raskt, men for N-queen problemet vil det være N^N konfigurasjoner som må undersøkes, så dette vil ta for lang tid, selv for toppmoderne datamaskiner. I 4-queen problemet kan vi se at **med en gang vi plasserer noen dronninger, vil vi få et helt sett med konfigurasjoner som blir ugyldige.** Dette er ideen bak CSP.



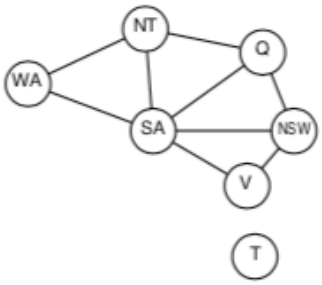
Eksempel – kartfarging

Vi ser på et kart over Australia, og har et problem der vi ønsker å farge hver region rød, grønn eller blå på en slik måte at ingen naboregioner har samme farge. For å formulere dette som en CSP, må vi definere de tre komponentene:

- $X = \{WA, NT, Q, NSW, V, SA, T\}$
- $D_i = \{red, green, blue\}$ for alle variablene
- $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, V \neq NSW, NSW \neq Q, Q \neq NT, NT \neq WA\}$

Begrensningene er at to land som grenser mot hverandre kan ikke være like. Siden det er ni land som grenser mot hverandre, vil vi ha ni begrensninger. Her har vi brukt $SA \neq WA$ som snarvei for $\langle (SA, WA), SA \neq WA \rangle$, der rel også kan skrives som en fullstendig liste av tuple-verdier: $[(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)]$. Dette problemet har flere mulige løsninger, for eksempel:

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}$$



Dette er en løsning fordi det er en konsistent og fullstendig tildeling av verdier, der alle variablene får en verdi og alle verdiene oppfyller betingelsene. **For å finne løsningen til en CSP, kan det hjelpe å lage en begrensningsgraf, der nodene er variabler i problemet og kantene vil være mellom variabler som deltar i en begrensning.** For eksempel vil kanten mellom WA og SA gi at disse ikke kan ha samme verdi,

Det er lurt å formulere problemer som en CSP, fordi det gir en naturlig representasjon av mange problemer og det kan gå raskere å løse CSP enn tilstandsrom-problem fordi CSP-løseren kan raskt eliminere store deler av søkerommet. For eksempel når vi har valgt $\{SA = blue\}$ i kartfarging-problemet, kan vi konkludere med at ingen av de fem naboregionene vil være blå. Dette vil redusere antall forsøk på tildeling av verdier fra $3^5 = 243$ til $2^5 = 32$, siden vi ikke trenger å teste om de fem regionene kan ha farge $blue$. Når CPS finner ut at en delvis tildeling ikke er løsningen, vil den se bort ifra videre finjusteringer av denne tildelingen. Vi kan se hvilke variabler som bryter begrensningen og dermed fokusere på variablene som er viktige. **Mange problemer som er uløselige for standard tilstandsromsøk kan løses raskt når de formuleres som en CPS**.

Eksempel – fabrikkplanlegging

Vi ser på problemet av å planlegge monteringen av en bil. Denne jobben består av en rekke oppgaver som kan modelleres som variabler, der hver variabel vil ha en verdi som gir starttidspunktet i antall minutter. Begrensningene kan være at en oppgave må utføres før enn annen (eks: hjul må festes før hjulkapsel), at det er et maks antall oppgaver som kan utføres samtidig og at en oppgave krever en bestemt mengde tid for å fullføres. Vi ser på en liten del av bilmonteringen, som består av 15 oppgaver som kan representeres som variabler: $X = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LF}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}$. Verdien til hver variabel vil være hvor lang tid det tar å utføre oppgaven.

Neste steg er å bestemme begrensningene. Vi begynner med **presedensbegrensninger, der $T_1 + d_1 \leq T_2$ betyr at oppgave T_1 må utføres før oppgave T_2 og den bruker d_1 tid på å utføres**. I vårt eksempel har vi presedensbegrensningene:

- $Axle_F + 10 \leq Wheel_{RF}$ og $Axle_F + 10 \leq Wheel_{LF}$

- $Axle_B + 10 \leq Wheel_{RB}$ og $Axle_B + 10 \leq Wheel_{LB}$
- $Wheel_{RF} + 1 \leq Nuts_{RF}$ og $NUTS_{RF} + 2 \leq CAP_{RF}$
- $Wheel_{LF} + 1 \leq Nuts_{LF}$ og $NUTS_{LF} + 2 \leq CAP_{LF}$
- $Wheel_{RB} + 1 \leq Nuts_{RB}$ og $NUTS_{RB} + 2 \leq CAP_{RB}$
- $Wheel_{LB} + 1 \leq Nuts_{LB}$ og $NUTS_{LB} + 2 \leq CAP_{LB}$

I tillegg har vi **disjunktiv begrensning, som gir at to oppgaver ikke kan overlappe**:

- $(Axle_F + 10 \leq Axle_B)$ or $(Axle_B + 10 \leq Axle_F)$

Vi må også sikre at inspeksjonen blir utført helt til slutt, ved å legge til $X + d_X \leq Inspect$ for alle variablene bortsett fra $Inspect$. For å sikre at hele monteringen er gjort på 30 minutter kan vi begrense domenet til alle variablene.

Dette er enkelt å finne løsningen til denne CPS. Det illustrerer hvordan CPS blir brukt i virkelig-verden problemer. Andre eksempler er problemer innenfor tildeling av roller (eks: hvilken lærer skal ha hvilken klasse), tidsplanlegging (hvilken klasse skal ha forelesning når), maskinvarekonfigurasjon, osv.

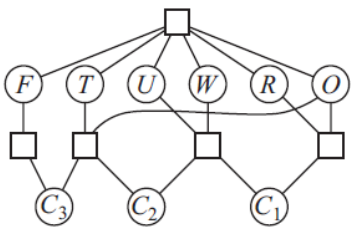
Variasjoner i CSP

Den enkleste formen for CSP har **diskrete**, **endelige domener** (eks: kartfarging, fabrikkplanlegging og n-queen). Et diskret domene kan også være uendelig, for eksempel et sett med heltall. I dette tilfellet er det ikke mulige å beskrive alle begrensningene ved å nummerere de tillatte kombinasjonene, så i stedet må man bruke et **begrensningsspråk** for å forstå begrensningene (eks: $T_1 + d_1 \leq T_2$). CPS med **kontinuerlige domener** er vanlige i den virkelige verden.

Det finnes også flere typer begrensninger:

- **Ensartet (*unary*) begrensning** = involverer én variabel, eks: $\langle (SA), SA \neq green \rangle$
- **Binære begrensning =** involverer et variabelpar, eks: $SA \neq WA$
- **Høyere-orden begrensninger**= involverer tre eller flere variabler, eks: $Between(X, Y, Z)$ som krever at verdien til Y er mellom X og Z
- **Global begrensning** = involverer et vilkårlig antall variabler (dvs. ikke nødvendigvis alle variablene i problemet). Eks: $Alldiff$ som krever at alle involverte variabler har ulike verdier (brukes i blant annet Sudoku)
- **Preferanse begrensninger** = indikerer hvilke løsninger som foretrekkes (motsatt av absolutte begrensninger, som løsninger må oppfylle). Det blir ofte representert som en kostnad for hver variabeltildeling (eks: professor foretrekker å forelese om morgenen, så morgenforelesning koster 1 poeng, mens kveldsforelesning koster 2 poeng). CPS med preferanse begrensninger kan løses med optimaliserende søkemetoder og kalles COP (Constraint Optimization Problem).

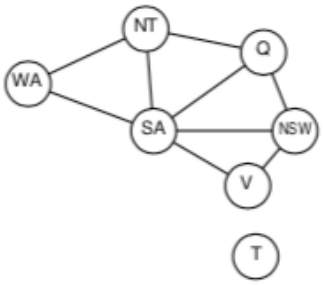
En binær CSP vil kun inneholde binære begrensninger, og den kan representeres som en begrensningsgraf. En **begrensningshypergraf** kan inneholde noder (sirkler) som viser binære begrensninger og hypernoder (firkanter) som viser n-ære begrensninger. **Alle begrensninger med endelig domene kan reduseres til et sett med binære begrensninger, ved å legge til hjelpevariabler i variabelsettet**. Dette vil gjøre at søkealgoritmen blir enklere. Fordeler ved å bruke globale begrensninger istedenfor et sett med binære begrensninger, er at det er lettere å skrive problembeskrivelsen og det er mulig å designe inferensalgoritmer for globale begrensninger, som ikke kan brukes på et sett med mer primitive begrensninger (mer senere).



Begrensningspropagering – inferens i CSPs

En standard søkealgoritme i tilstandsrommet kan bare søke, mens en algoritme i CSP kan:

- **Søke** = velge en ny variabeltildeling blant flere muligheter
- **Utføre begrensningspropagering** = en type slutning (*inference*) der begrensningene brukes for å redusere antall lovlige verdier for en variabel, som igjen kan redusere lovlige verdier for andre variabler. Dette kan utføres samtidig som søket eller før søket starter. Noen ganger vil det løse hele problemet, slik at søket ikke trengs.



Begrensningspropagering bruker **lokal konsistent tildeling**, der antall lovlige verdier for en variabel blir redusert ved å se på begrensningene som variabelen deltar i. Variablene behandles som noder i en graf, der buene er binære begrensninger. **Prosessen av å sikre lokal konsistent tildeling i hver del av grafen, vil føre til at inkonsistente verdier blir eliminert gjennom grafen**. Det er ulike typer lokal konsistens, som vi nå skal se videre på.

Node-konsistent

En variabel er node-konsistent, dersom alle verdiene i dens domene tilfredsstiller variabelens ensartet begrensninger. **Bruk av node-konsistent kan redusere domenet til variabelen ved å fjerne ulovlige verdier som bryter begrensninger**. I kartfaring eksempelet har SA domenet $\{red, green\ blue\}$. Hvis det viser seg at SA misliker $green$, kan vi gjøre den node-konsistent ved å eliminere $green$, slik at SA får redusert domene $\{red, blue\}$. **Et CSP nettverk vil være node-konsistent dersom alle variablene i nettverket er node-konsistente**. Alle ensartet begrensninger kan elimineres ved å kjøre node-konsistens.

Bue-konsistent

For buen (X_i, X_j), vil variabelen X_i være bue-konsistent mht. variabelen X_j, dersom det for alle verdier i domene D_i, eksisterer verdier i domene D_j som tilfredsstiller den binære begrensningen på buen. Altså, $\forall x \in D_i, \exists y \in D_j$ der (x, y) oppfyller begrensningen. Bruk av bue-konsistent vil eliminere verdier fra domenet til variabelen, dersom verdiene ikke oppfyller betingelsen (dvs. er ikke en del av løsningen). Et CSP nettverk er bue-konsistent hvis alle variablene er bue-konsistente med alle andre variabler. For eksempel kan vi se på begrensningen $Y = X^2$, der domenet til Y og X er settet av tal (dvs. 0, 1, 2, 3, ...). Denne begrensningen kan skrives som $\langle (X, Y), \{(0,0), (1,1), (2,4), (3,9)\}\rangle$. For å gjøre at X blir bue-konsistent mht. Y, kan vi redusere domenet til X til $\{0, 1, 2, 3\}$, mens for å gjøre at Y blir bue-konsistent mht. X, kan vi redusere domenet til Y til $\{0, 1, 4, 9\}$. Dermed vil hele CSP være bue-konsistent. For kartfaring-eksempelet vil det ikke ha noen effekt å bruke bue-konsistent, fordi begrensningene og domenet vil allerede oppfylle betingelsen. For eksempel for begrensningen $\langle (SA, WA), [(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)]\rangle$, ser vi at for enhver verdi i domenet til SA, vil det være gyldige verdier for WA (og motsatt).

Figuren viser **algoritmen AC-3 som brukes for å gjøre alle variablene bue-konsistente**. Algoritmen bruker en kø som fylles med alle buene som representerer de gitte binære betingelsene. Så lenge denne køen ikke er tom, vil algoritmen hente ut en tilfeldig bue (X_i, X_j) og kalle på funksjonen som vil sørge for at X_i blir bue-konsistent mht. X_j. Denne funksjonen vil gå igjennom alle verdiene i domenet til X_i og vil fjerne en verdi dersom den ikke oppfyller betingelsen. Dersom domenet blir redusert, vil AC-3 legge alle nabobuene til i køen på nytt, fordi

```
function AC-3( csp) returns false if an inconsistency is found and true otherwise
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← Remove-First(queue)
      if Remove-Inconsistent-Values(Xi, Xj) then
         for each Xk in Neighbors[Xi] do
            add (Xk, Xi) to queue
   return true

function Remove-Inconsistent-Values( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in Domain[Xi] do
      if no value y in Domain[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from Domain[Xi]; removed ← true
   return removed
```

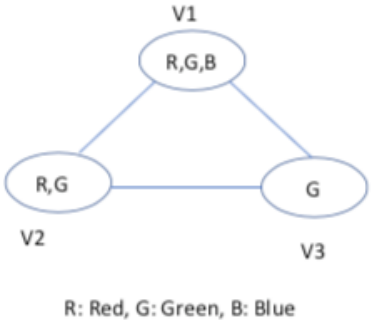
reduksjonen i domenet til X_i kan gjøre at domenet til nabonoder kan reduseres videre, selv om de allerede har blitt undersøkt. Når køen er tom, er det ikke mulig å redusere domenet videre, så algoritmen avslutter. **Hvis domenet blir redusert helt til det blir tomt, vil algoritmen redusere false, fordi da vil ikke CSP ha en konsistent løsning** (ikke på figur, se s. 209 i boka). **En bue-konsistent CSP er raskere å løse, siden variablene har mindre domene.**

Worst-case kjøretid for AC-3 er $\mathbf{O}(\boldsymbol{cd^3})$, der c er antall binære begrensninger og d er antall verdier i domenet til hver variabel. Dette skyldes at hver bue (begrensning) kan settes inn i køen maksimalt d ganger, siden det kan fjernes maksimalt d verdier fra domenet. For å sjekke konsistensen til en bue vil AC-3 bruke iden $\mathrm{O}(d^2)$ (dobbel for-løkke). Total tid blir derfor $\mathrm{O}(cd^3)$.

Eksempel – bue-konsistent

Vi har et CSP med tre variabler V1, V2 og V3, der verdiene innenfor noden på figuren viser initial domene. Det er tre binære begrensninger:

- $Diff\_color(V1, V3)$
- $Diff\_color(V1, V2)$
- $Diff\_color(V2, V3)$



R: Red, G: Green, B: Blue

AC-3 mottar CSP med tilhørende variabler, domener og begrensninger.
Den starter med å fylle køen med buene: V1-V2, V2-V1, V1-V3, V3-V1, V2-V3 og V3-V1. Legg merke til at hver binær begrensning blir representert som to buer. AC-3 vil deretter gå igjennom hver bue i køen og sjekker bue-konsistens. Loopene blir:

1. **V1-V2**: domenet til V1 er $\{R, G, B\}$, mens domenet til V2 er $\{R, G\}$. For $x = R$ hos V1, ser vi at det finnes en $y = G$ hos V2, slik at $V1 \neq V2$ og dermed blir ikke $x = R$ fjernet. For $x = G$, ser vi at $y = R$, slik at $x = G$ blir ikke fjernet. For $x = B$, ser vi at $y = R, G$, slik at $x = B$ blir ikke fjernet. Ingen reduksjon.
2. **V2-V1:** domenet til V2 er $\{R, G\}$, mens domenet til V1 er $\{R, G, B\}$. Ingen reduksjon.
3. **V1-V3:** domenet til V1 er $\{R, G, B\}$, mens domenet til V3 er $\{G\}$. Her ser vi at verdiene R og B i domenet til V1 vil oppfylle betingelsen, men for $x = G$, ser vi derimot at det ikke finnes noen verdi i domenet til $V3$ som gir at $V1 \neq V3$. Derfor må $x = G$ fjernes fra domenet til V1 og vi legger til bue V2-V1 og V3-V1
4. **V2-V3:** domenet til V2 er $\{R, G\}$, mens domenet til V3 er $\{G\}$. Her ser vi at verdien R i domenet til V2 vil oppfylle betingelsen, men for $x = G$, ser vi derimot at det ikke finnes noen verdi i domenet til $V3$ som gir at $V2 \neq V3$. Derfor må $x = G$ fjernes fra domenet til V2 og vi legger til bue V3-V2 og V1-V2
5. ...
6. **V1-V2:** domenet til V1 er $\{R, G\}$, mens domenet til V2 er $\{R\}$. Her ser vi at verdien G i domenet til V1 vil oppfylle betingelsen, men for $x = R$, ser vi derimot at det ikke finnes noen verdi i domenet til $V2$ som gir at $V1 \neq V2$. Derfor må $x = R$ fjernes fra domenet til V1 og vi legger til bue V3-V1 og V2-V1
7. ...

Denne prosessen gjentas helt til det ikke lenger skjer noen endring og køen med buer blir tom. Siden alle domenene reduseres til størrelse 1, vil dette være en løsning på CSP:

- V1 får domenet $\{B\}$
- V2 får domenet $\{R\}$
- V3 får domenet $\{G\}$

Hvis et av domenene blir tomt, vil det ikke vært noen løsning på problemet. Dette skyldes at løsningen må være en konsistent tildeling og må dermed være bue-konsistent. N**oen ganger vil begrensningspropagering være tilstrekkelig for å finne løsningen, mens andre ganger må man også utføre søk.**

Bane-konsistent

I noen nettverk er det ikke tilstrekkelig å se på bue-konsistens for å redusere domenet til variablene. I eksempelet med kartfarging av Australia, vil ikke bruk av bue-konsistent føre til at noen domener blir redusert, fordi variablene er allerede bue-konsistente. I dette tilfellet trenger vi en sterke form for konsistens, kalt **bane-konsistens,** som vil redusere domenene ved å se på tre variabler om gangen. **Et to-variabel sett $\{X_i, X_j\}$ er bane-konsistent mht. en tredje variabel X_m hvis det for alle tildelinger $\{X_i = a, X_j = b\}$ som er konsistente med begrensninger på $\{X_i, X_j\}$, finnes en tildeling for X_m som tilfredsstiller betingelsene på $\{X_i, X_m\}$ og $\{X_m, X_j\}$.** Dette kalles bane-konsistens fordi det kan ses på som en bane som går fra X_i til X_m til X_j. Bane-konsistent kan oppnås med PC-2 algoritmen, som fungerer på samme måte som AC-3, bortsett fra at den tester bane-konsistent istedenfor bue-konsistent.

Vi ser på kartfarging av Australia med to farger red og $blue$, der vi ønsker å gjøre settet $\{WA, SA\}$ bane-konsistent mht. NT. De mulige tildelingene av $\{WA, SA\}$ som oppfyller betingelsen at $WA \neq SA$ er $\{WA = red, SA = blue\}$ og $\{WA = blue, SA = red\}$. Gitt disse må vi se om det finnes en tildeling av NT som tilfredsstiller $WA \neq NT$ og $NT \neq SA$. For $\{WA = red, SA = blue\}$, ser vi at for at $WA \neq NT$ må $NT = blue$, men da vil ikke $NT \neq SA$ være mulig, så denne tildelingen må elimineres. Det samme gjelder $\{WA = blue, SA = red\}$. Dermed vil det ikke være noen gyldig tildeling av $\{WA, SA\}$, og CSP har ingen løsning.

K-konsistent

Sterkere former for konsistens blir definert som k-konsistens. **En CSP er k-konsistent hvis det for alle $k-1$ variabler med konsistent tildeling av verdier, finnes en konsistent verdi som kan tildeles en k'ende variabel (merk: konsistent betyr at tildelingen oppfyller betingelsene som variabelen(e) deltar i).** Merk at 2-konsistens er det samme som bue-konsistens. **En CSP er sterkt k-konsistent dersom den er k-konsistent, $(k-1)$-konsistent, $(k-2)$-konsistent, osv. helt ned til 1-konsistent.** For å løse en CSP med n noder som er n-konsistent, kan vi velge en konsistent verdi for X_1, og vi vil deretter kunne velge en verdi for X_2, siden grafen er 2-konsistent, og vi vil deretter kunne velge en verdi for X_3, siden grafen er 3-konsistent, osv. For hver variabel X_i trenger vi kun å søke gjennom d verdier i domenet til vi finner en verdi som er konsistent med $X_1, \dots, X_{i-1}$. Det er garantert at vi finner en løsning på tiden $O(n^2 d)$. Samtidig krever det eksponential tid og rom for å etablere n-konsistens, så det er kun vanlig å bruke 2- og 3-konsistens i praksis.

Globale begrensninger

Globale begrensninger forekommer ofte i virkelige problem, og de kan håndteres av spesielle algoritmer istedenfor de generelle metodene som er beskrevet over. Et eksempel er $Alldif$ begrensningen, som krever at alle de involverte variablene har ulike verdier. En enkel form for deteksjon av inkonsistens i dette tilfellet vil være å se på antall variabler og verdier; Hvis det er m variabler og de har n ulike verdier til stammen, kan ikke begrensningen være tilfredsstilt dersom $m > n$. En enkel algoritme for å bestemme om problemet er konsistent er å fjerne alle variabler med en verdi i domenet og fjern verdien fra domenet til alle andre variabler. **Hvis det er flere variabler enn distinkte verdier eller det produseres et tomt domene, vil CSP være inkonsistent og har dermed ingen løsning**. Det finnes også andre typer globale begrensninger som kan løses effektivt med spesielle algoritmer (eks: **ressursbegrensning**, s. 212).

Sudoku eksempel

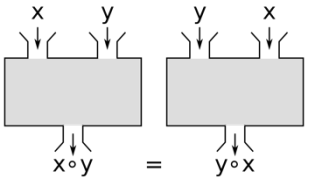
Sudoku kan behandles som en CSP, der hver firkant behandles som en variabel. Tomme firkanter har domene $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, mens fylte firkanter har domene lik den enkelte verdien. Det er 27 $Alldif$ begrensninger, en for hver kolonne, rad og boks. CPS løser kan

finne løsningen på alle Sudoku problem innen 0.1 sekunder. Alle strategiene som brukes for å løse Sudokuen kan brukes generelt på alle CSP. **Dette er fordelen ved CSP formalisme: for hvert nytt problem, trenger vi kun å definere problemet som begrensninger, og deretter kan de generelle mekanismene brukes.**

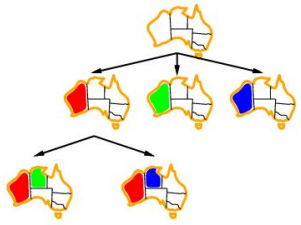
I noen tilfeller vil bruken av slutning (eks: bue-konsistens) gjøre at man finner løsningen til en CSP. For mange CSP er det ikke tilstrekkelig å bruke slutning over begrensninger for å finne løsningen. I dette tilfellet er det også nødvendig med søk i form av Backtracking eller lokal søk.

Backtracking søk for CSP

For å løse en CSP kan vi bruke en standard inkrementell søkealgoritme (eks: dybde-begrenset søk), der den initiale tilstanden er en tom tildeling, handlingen er en tildeling av verdier hos variablene og måltesten er om tildelingen er fullstendig. Problemet med denne fremgangsmåten, er at siden det er n variabler som hver har d mulige verdier, vil branching faktoren ved roten være nd, ved etterfølgeren er den $(n-1)d$, osv. Det lages et tre med $n!\,d^n$ bladnoder, selv om det er bare d^n mulige tildelinger.



Denne fremgangsmåten ignorerer en viktig egenskap som deles av alle CSP, kalt **kommutativitet**. Et problem er kommutativ dersom rekkefølgen til handlingene har ingen effekt på utfallet (se figur). **CSP er kommutativ fordi rekkefølgen til tildelingen av verdier hos variablene, har ingen effekt på det endelige utfallet (merk: kan påvirke effektiviteten)**. For eksempel vil $[WA = red, så\ NT = green]$ være det samme som $[NT = green, så\ WA = red]$. **Derfor trenger vi kun å se på tildelingen av verdi hos én variabel ved hver node i søketreet, slik at antall bladnoder blir d^n**. For kartfarging-eksempelet betyr det at ved roten kan vi velge mellom $WA = red$, $WA = blue$ og $WA = green$, og ikke mellom $SA = red$ og $WA = blue$. **Ved hvert nivå i treet vil vi tildele verdi til kun én variabel**.

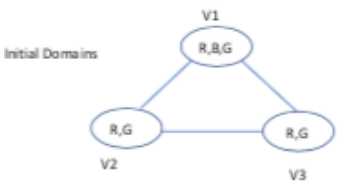


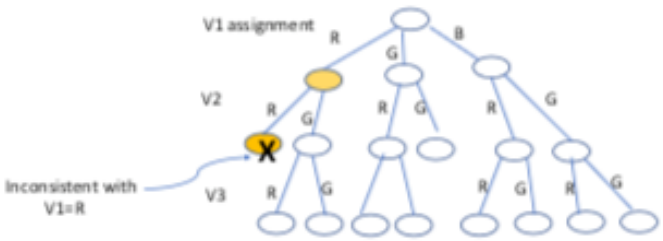
Backtracking search bruker dybde-først søk som velger verdi for én variabel om gangen og går tilbake dersom den møter en variabel som ikke har noen lovlige verdier. Algoritmen vil velge en ikke-tildelt variabel og vil deretter forsøke alle verdiene i domenet helt til den finner en løsning. Dette er en rekursiv algoritme og dersom den møter en inkonsistens (dvs. en node med ingen lovlige verdier gitt tildelingen i forgjengernodene), vil metoden returnere *failure* og hopper tilbake til forrige variabel og forsøker en ny verdi. Legg merke til at vi ikke gir noen informasjon om initial tilstand, handlinger, overgangsmodell eller måltest. **Backtracking søk er en grunnleggende, uinformert algoritme som brukes for å løse CSP.** Backtracking vil kun ta vare på én representasjon av en tilstand og vil i stedet endre denne representasjonen istedenfor å lage nye.

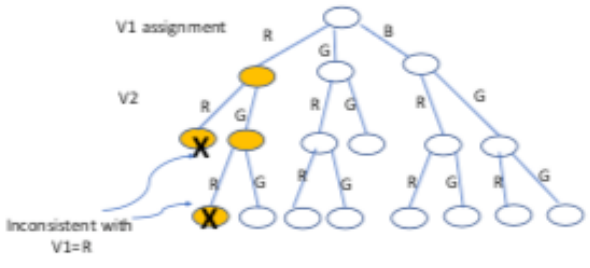
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Eksempel – Backtracking søk

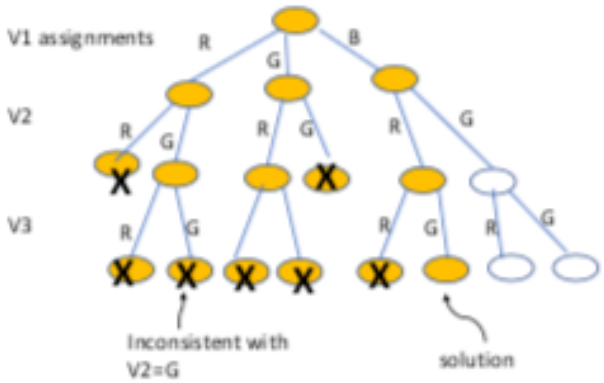
Vi ser igjen på eksempelet der tre variabler, V1, V2 og V3, skal fargelegges slik at de får ulike farge, gitt domenene på figuren. Når algoritmen tildeler verdi R til variabel V1, vil den møte en inkonsistens når den forsøker å tildele verdi R til variabel V2. Derfor vil den hoppe tilbake og prøve en ny verdi hos V1.







Når den tildeler verdi R til V1 og verdi G til V2, vil den også møte en inkonsistens når den forsøker å tildele verdi G til V3. Algoritmen fortsetter å møte inkonsistenser helt til den prøver å tildele verdi B til V1, verdi R til V2 og verdi G til V3. Dette er en konsistent løsning og algoritmen vil derfor avslutte. Det finnes også en annen løsning (V1 = B, V2 = G, V3 = R).



Backtracking forbedring

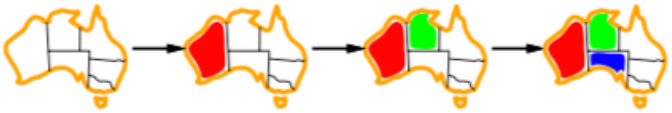
Tidligere har vi sett at et uinformert søk kan forbedres ved å bruke domenespesifikk heuristikk (eks: straight-line distance). I tilfelle med CSP og backtracking, kan generell heuristikk brukes for å forbedre effektiviteten til backtracking søk:

1. Hvilken variabel skal tildeles verdi neste gang, og i hvilken rekkefølge skal verdiene testes?
2. Hvilke slutninger skal utføres ved hvert steg?
3. Når søket ankommer en tildeling som bryter en begrensning, kan søket unngå å gjenta denne feilen?

Vi skal nå svare på disse spørsmålene.

Rekkefølge til variabler

Vi har sett at CSP er kommutativ, som vil si at rekkefølgen til tildelingen av verdier hos variablene ikke har noen effekt på den endelige, delvis tildelingen. Denne rekkefølgen vil derimot ha stor effekt på effektiviteten til backtracking søket. Hvilken rekkefølge som brukes vil påvirke hvordan funksjonen SELECT-UNASSIGNED-VARIABEL i algoritmen fungerer.



Den enkleste strategien for å velge neste variabel som skal tildeles verdi, er å bruke den statiske rekkefølgen $\{X_1, X_2, \dots\}$, men dette vil sjeldent resultere i det mest effektive søket. **En mer effektiv strategi er minimum-remaining-values (MRV) heuristikk, der neste variabel som velges er den med færrest lovlige verdier.** For eksempel i kartfarging-eksempelet kan vi ha rekkefølgen $\{WA, NT, Q, NSW, V, SA, T\}$. Etter å ha tildelt $WA = red$ og $NT = green$, vil det lønne seg å tildele $SA = blue$ istedenfor Q. Dette skyldes at SA har kun én mulig verdi (se figur), og ved å tildele $SA = blue$, vil verdien til resten av variablene være bestemt. **Denne metoden kalles også «fail-first», siden den velger variabelen som mest sannsynlig vil forårsake inkonsistens først og vil dermed prune søketreet. Hvis det finnes en variabel som ikke har noen lovlige verdier, vil MRV velge denne først og dermed detektere feilen med en gang, slik at man unngår unødvendig søk i andre variabler.**

MRV heuristikk vil ikke hjelpe til i begynnelsen av kartfargingen, siden alle regionene har like mange lovlige verdier. I dette tilfellet vil algoritmen bruke heuristikkgrad, der den velger variabelen som er involvert i størst antall begrensninger med andre ikke-tildelte variabler. Dette vil redusere fremtidig branching. Siden SA har grad 5, mens de andre har grad 2, 3 og 0, vil det være best å velge SA i begynnelsen. Dersom vi starter med SA kan vi finne løsningen uten noe backtracking, ved å velge enhver konsistent farge for hver variabel. Som regel vil man bruke MRV, med heuristiskgrad når det er uavgjort.

Rekkefølge til verdier

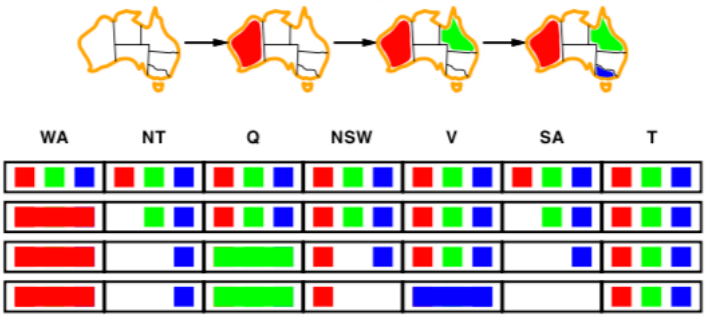
Når en variabel har blitt valgt, må algoritmen bestemme i hvilken rekkefølge verdiene skal undersøkes. I **least-constraining-value heuristikk, vil algoritmen velge verdien som fjerner færrest muligheter for nabovariablene i grafen. Dette vil øke sannsynligheten for at søket**

finner løsningen tidlig. For eksempel hvis vi har tildelingene $WA = red, NT = green$ og neste variabel er Q, vil $blue$ være et dårlig valg fordi den eliminerer den siste lovlige verdien for naboen SA. Least-constraining-value vil derfor foretrekke red over $blue$. Dette er kun nødvendig hvis vi ønsker å finne én løsning til problemet. Hvis vi forsøker å finne alle løsningene til problemet, spiller det ingen rolle hvilken rekkefølge man bruker. **Vi trenger kun én løsning, så derfor vil vi først se på verdiene som mest sannsynlig vil føre til en løsning.**

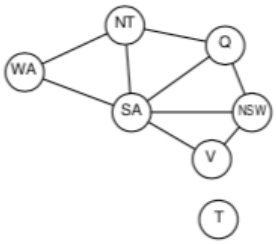
Flettende søk og slutninger

Ved å bestemme rekkefølgen til variablene og verdiene, kan vi redusere domenet til variabler før søket begynner. Underveis i søket vil slutning være et kraftig verktøy, siden hver gang en variabel tildeles en verdi, så kan domenet til nabovariabler reduseres. **Det er mulig å oppdage lite lovende noder vha slutning rundt konsistensen til grafen. I løpet av søket kan man bruke bue-konsistens for å redusere domenet til variabler, men dette er tidkrevende**.

En enklere form for slutning kalles **forward checking**. **Når variabel X blir tildelt en verdi, vil forward checking sikre at den er bue-konsistent, ved å eliminere verdier hos ikke-tildelte variabler koblet til X via begrensninger, dersom verdiene er inkonsistente med verdien som er valgt for X.** Det er kun nyttig å utføre forward checking dersom bue-konsistens ikke har blitt utført før søket. **Backtracking search kan prunes ved å bruke Forward checking, siden den fjerner verdier hos domenet til etterfølgernoder og begrenser dermed branchingen**. Det er enklere enn bue-konsistens, fordi det sjekker kun nabovariabler. Forward checking tar vare på lovlige verdier hos ikke-tildelte variabler, og det sjekker «fremtiden» istedenfor «fortiden», slik backtracking gjør. **Dersom ingen variabler har noen lovlige verdier vil søket terminere.** Forward checking kan kombineres med MRV for å gjøre søket enda mer effektivt, siden forward checking vil regne ut informasjonen som MRV heuristikken trenger.

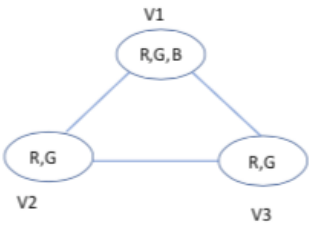


Figuren viser progresjonen for backtracking søk for kartfarging CSP. I starten vil domenet til alle variablene inneholder red, $green$ og $blue$. Ved andre linje ser vi at WA blir tildelt red, og forward checking gjør at red blir fjernet fra NT og SA, siden disse er nabovariabler til WA som deltar i ulikhetsbegrensning. Ved tredje linje ser vi at Q blir tildelt $green$, og forward checking gjør at $green$ blir fjernet fra NT, NSW og SA, siden disse er nabovariabler til Q. Ved fjerde linje ser vi at V blir tildelt $blue$, noe som fører til at domenet til SA blir tomt. Forward checking har dermed oppdaget at den delvis tildelingen $\{WA = red, Q = green, V = blue\}$ er inkonsistent med begrensningene til problemet, og algoritmen vil derfor backtracke med en gang. Dette eksempelet illustrerer hvordan **forward checking holder styr over gjenværende lovlige verdier for ikke-tildelte variabler og vil backtracke søket når den møter et tomt domenet.**



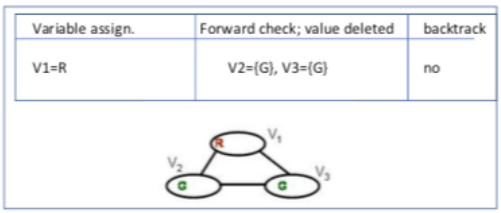
Forward checking kan ikke detektere alle inkonsistenser, fordi den vil kun gjøre nåværende variabel bue-konsistent og ser ikke lengre frem for å gjøre alle andre variabler bue-konsistente (dvs. den sikrer kun lokal bue-konsistens). For eksempel på figuren over kan vi se at når $WA = red$ og $Q = green$, vil både NT og SA tvinges til å være $blue$, noe som er en inkonsistens siden disse er naboer. **For å oppdage slike inkonsistenser må man**

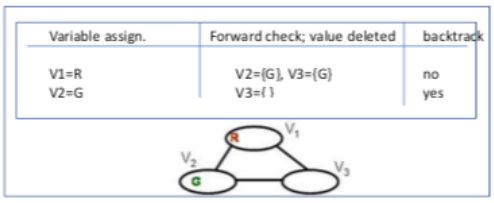
bruke bue-konsistens før søket eller etter hver tildeling (MAC = *Maintaining Arc Consistency*, AC-3 der køen består av utgående buer fra nylig tildelt variabel). Bue-konsistent vil oppdage feil tidligere enn forward checking, siden det sjekker om alle variablene i grafen er konsistente. Ulempen er at det er mer tidkrevende.

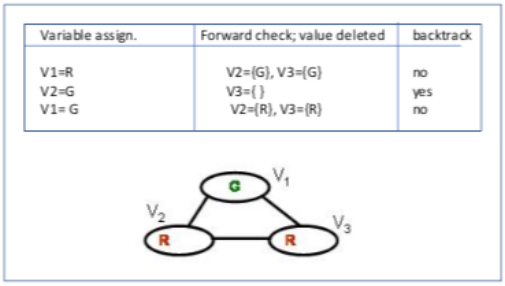


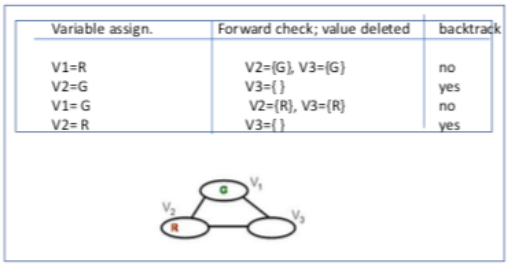
Figurene under viser et eksempel for backtracking med forward checking for CSP der farge blir tildelt tre variabler. Dette søket foregår i tre omganger:

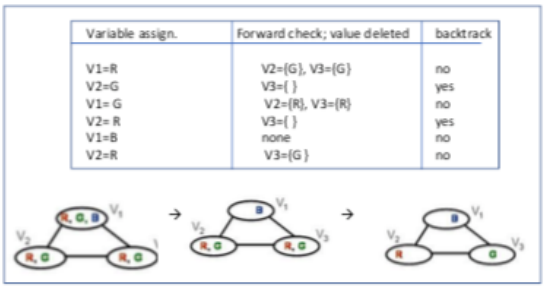
1. V1 blir tildelt R, som fjernes fra domenet til V2 og V3. Deretter blir V2 tildelt G, noe som gjør at G fjernes fra domenet til V3. Dette gjør at domenet til V3 blir tomt, og derfor vil søket backtracke til V1
2. V1 blir tildelt G, som fjernes fra domenet til V2 og V3. Deretter blir V2 tildelt R, noe som gjør at domenet til V blir tomt igjen. Søket vil derfor backtracke til V1
3. V1 blir tildelt B, noe som gjør at ingen verdier fjernes fra domenet til V2 og V3. Deretter blir V2 tildelt R og V3 blir tildelt G. Dermed er problemet løst.











Intelligent backtracking – ser bakover

Når en branch ved søket feiler, vil backtracking søket hoppe tilbake, men hvor den lander avhenger av implementasjonen av BACKTRACK-funksjonen. Den enkleste metoden er **kronologisk backtracking, der algoritmen går tilbake til forrige variabel og prøver en ny verdi**. Hvis vi har rekkefølgen Q, NSW, V, T, SA, WA, NT med delvis tildeling $\{Q = red, NSW = green, V = blue, T = red\}$, vil alle verdier hos SA bryte en begrensning. Derfor vil algoritmen gå tilbake til T og prøve en ny verdi, men dette vil ikke løse problemet for SA, siden disse ikke er naboer.

En bedre tilnærming er å backtracke til en variabel som kanskje kan fikse problemet, for eksempel en variabel som eliminerte en av verdiene hos SA. Konfliktsettet vil holde styr over settet av tildelinger som er i konflikt med en verdi hos SA, og i dette tilfellet vil det være $\{Q = red,\ NSW = green, V = blue\}$. **Backjumping metoden vil hoppe tilbake til den nyeste tildelingen i konfliktsettet**. Denne metoden vil derfor hoppe over T og heller forsøke en ny verdi for V. For denne metoden vil BACKTRACK akkumulere konfliktsettet, mens den leter etter en lovlig verdi å tildele variabelen. Hvis algoritmen bruker forward checking, vil backjumping være unødvendig, fordi søket vil aldri nå situasjonen der backjumping vil skje.

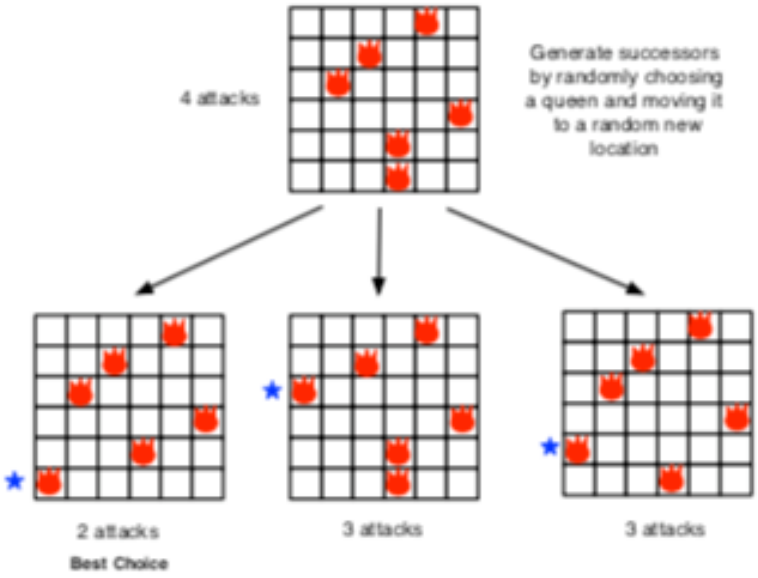
Backjumping vil detektere når domenet til en variabel er tomt, men i mange tilfeller vil konflikten skyldes tidligere tildelinger av variabler. En mer komplisert definisjon av konfliktsettet, er at det er settet av foregående variabler som sammen med etterfølgende variabler, gjør at det er ingen konsistent løsning. **Konflikt-rettet backjumping er basert på**

slike konfliktsett, og de vil hoppe direkte tilbake til kilden av problemet. Ved **Constraint learning** vil algoritmen finne variabelsettet med tilhørende verdier som forårsaker problemet (= *no-good*) og lagre dette ved å legge til en ny begrensning i CSP. Dermed vil søket unngå at samme problem oppstår igjen. No-goods kan brukes av forward checking eller backjumping, og det er en av de viktigste teknikkene i moderne CSP løsere.

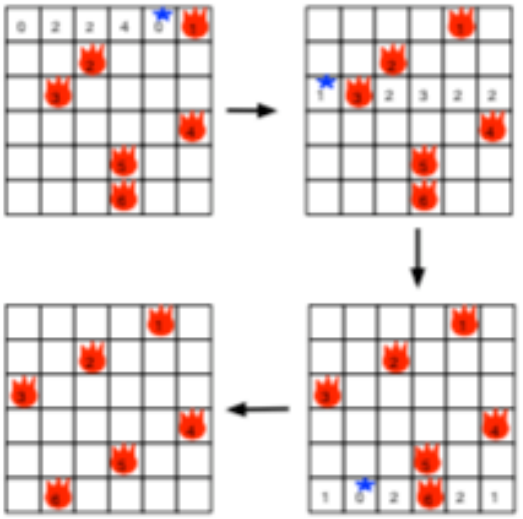
Lokal søk for CSP

Lokal søkealgoritmer kan brukes for å effektivt løse mange CSP. De genererer fullstendige tildelinger, ved at den initiale tilstanden tildeler en verdi til hver variabel og søket vil deretter evaluere om tildelingene bryter begrensninger og endre disse slik at antall begrensningsbrudd blir redusert. For eksempel i 8-queen problemet kan den initiale tilstanden være en tilfeldig konfigurasjon av dronningene i de åtte kolonnene, og hvert steg i søket vil flytte en enkel dronning til en ny posisjon i kolonnen. **Den initiale gjetningen vil som regel bryte flere begrensninger, så poenget til søket er å redusere dette.**

Alle lokale søkealgoritmer i kapittel 4 kan brukes for å løse CSP med fullstendig tildeling (dvs. alle variabler blir tildelt verdi før søket). For å bruke lokal søk må det være tillatt med tilstander som ikke tilfredsstiller begrensningene, for disse blir etterhvert endret ved at operatorer gir nye variabelverdier. **Seleksjonen av variabel kan gjøres på flere måter, for eksempel kan det gjøres tilfeldig ved å velge en av variablene som er i konflikt. Min-konflikter heuristikk vil velge verdien som resulterer i minimum antall konflikter med andre variabler**. Dette er overraskende effektivt for mange CSP, for eksempel for n-queen problemet vil kjøretiden være grovt uavhengig av problemstørrelsen (ser bort fra initial plassering). Selv million-queen problemet kan løses med gjennomsnittlig 50 steg etter initial tildeling. Min-konflikter fungerer også bra for vanskelige problemer. Dersom min-konflikter kombineres med hill-climbing, vil $h(n)$ være talt antall begrensinger som brytes. Landskapet til CSP med min-konflikter heuristikk vil ofte ha en rekke platåer, så det kan lønne seg å tillate sidelengs bevegelse eller bruke simulert annealing (s. 38).



Løser 6-queen problemet med dum lokal søk, som tilfeldig velger hvilken dronning i konflikt som skal få ny verdi og tilfeldig velger ny verdi.

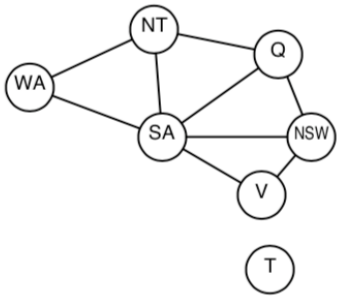


Løser 6-queen problemet med min-konflikter heuristikk som velger verdien som resulterer i minimum antall konflikter med andre dronninger

En annen fordel med lokal søk er at det er lettere å endre begrensninger eller legge til nye uten at resten av løsningen blir mye endret. Dette brukes i blant annet planleggingsproblem som er utsatt for små endringer (eks: planlegging av fly som er utsatt for værproblemer). Backtracking søk med nye begrensninger vil ofte kreve mye lengre tid og kan finne en løsning som har mange forskjeller fra nåværende plan.

Problemstrukturen

Strukturen til problemet, altså formen til begrensningsgrafen, kan brukes for å finne løsningen raskere. Når vi ser på begrensningsgrafen til kartfarging problemet, kan vi se at Tasmania ikke er koblet til hovedlandet. Farging av Tasmania og farging av hovedlandet er **uavhengige delproblem**, som betyr at de kan løses (dvs. farges) hver for seg for å finne løsningen til hele kartet. **Uavhengighet kan identifiseres som koblede komponenter i begrensningsgrafen, dvs. delgrafer som ikke er koblet til hverandre**. Hvis tildeling S_i er løsningen til CSP_i, så vil $\bigcup_i S_i$ være løsningen til $CSP = \bigcup_i CSP_i$. Dvs. vi kan finne løsningen til CSP ved å løse de uavhengige delproblemene. Dette kan ha stor effekt på kjøretiden til CSP løseren. En Boolean CSP med 80 variabler som deles inn i fire delproblem vil redusere worst-case kjøretid fra universets levetid til litt mindre enn ett sekund!



En annen måte å utnytte problemstrukturer er **trestruktur CSP. Hvis begrensningsgrafen er et tre, dvs. ingen sykluser, kan man finne løsningen på lineær tid i antall variabler, ved å bruke rettet bue-konsistens (DAC**). **En CSP med variabelorden $X_1, X_2, \ldots, X_n$ vil være DAC kun hvis alle X_i er bue-konsistent med hver X_j der $i < j$**. For å løse en trestruktur CSP kan du tilfeldig velge en variabel til å være roten og deretter ordne variablene, slik at hver variabel kommer etter sin foreldervariabel i treet (= topologisk sortering). Denne grafen kan lages på tiden $O(nd^2)$, siden den må plassere n noder og for hver plassering må den sammenligne d mulige domeneverdier for to variabler (avgjøre foreldre-barn forhold). Når grafen er laget kan søket gå nedover listen av variabler og velge enhver gjenværende verdi. Siden hver kobling er bue-konsistent, vet søket at hvert barn vil ha en gyldig verdi uansett hvilken verdi det velger for forelderen. Det er også mulig å endre mer generelle begrensningsgrafer til trær, ved å fjerne en node (eks: tilegne verdi til SA og fjerne alle inkonsistente verdier i domenene til de andre variablene). Se side 225 for mer detaljert beskrivelse (ikke nevnt i forelesning) ☺

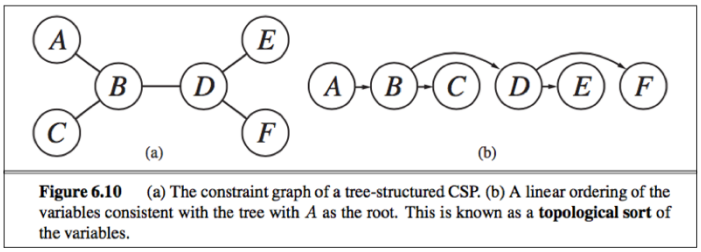
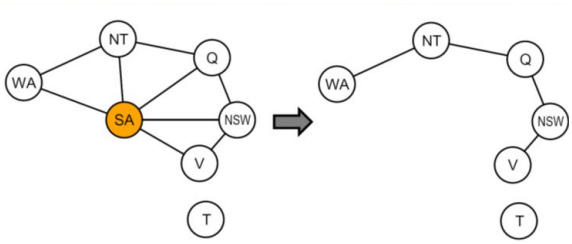


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

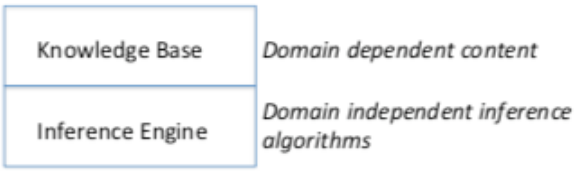


Kapittel 7 – Logiske agenter

Mennesker ser ut til å vite ting og dette hjelper dem å gjøre ting. Dette impliserer at menneskelig intelligens oppnås vha resonneringsprosesser som operer på en intern representasjon av kunnskap. I AI blir denne tilnærmingen til intelligens vist som **kunnskapsbaserte agenter**. De problemløsende agentene i kapittel 3 og 4 vet ting, men på en begrenset og lite fleksibel måte. I de neste kapitlene vil vi utvikle logikk som en type representasjon som støtter kunnskapsbaserte agenter, som kan kombinere og rekombinere informasjon for å støtte uttalelige muligheter.

Kunnskapsbaserte agenter

Den sentrale komponenten i en kunnskapsbasert agent er dens kunnskapsbase (KB). **En kunnskapsbase er et sett med setninger, der hver setning er utrykt i et kunnskapsrepresentasjon-språk og representerer en påstand om verden.** Siden den inneholder informasjon om verden vil KB være domeneavhengig. Dersom setningen ikke er utledet fra andre setninger, kalles det et **aksiom**. TELL operasjoner brukes for å legge til nye setninger til KB, mens ASK operasjoner brukes for å hente setninger fra KB. Begge operasjonene kan involvere inferens (slutninger), der nye setninger blir utledet fra gamle. **Inferensmotoren er et sett med prosedyrer som bruker representasjonsspråk for å inferere nye fakta fra kjente og svare på en rekke KB queries.** Dvs. inferensmotoren brukes for å legge til nye setninger til KB eller hente setninger fra KB. **Når man spør om en setning fra KB, bør svaret følge av det som har blitt fortalt KB tidligere**. Inferens prosessen bør altså ikke finne på ting underveis. Inferensmotoren er uavhengig av domene og kan brukes på flere ulike KBs.



Figuren viser grunnlaget for et **kunnskapsbasert agentprogram**. I likhet med andre agenter vil den **ta persepter som input og returnerer en handling**. Den kunnskapsbaserte agenten vil opprettholde en kunnskapsbase (KB), som initialt kan inneholde noe bakgrunnskunnskap. Når agentprogrammet kalles, vil den gjøre tre ting:

```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

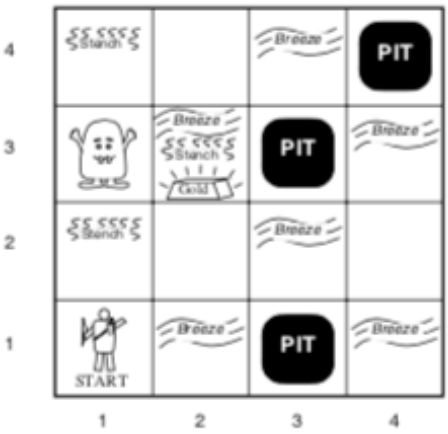
1. Den forteller (TELL) kunnskapsbasen hva den oppfatter
2. Den spør (ASK) kunnskapsbasen hvilken handling den bør utføre. Dette kan involvere kompleks resonnering om tilstanden til verden, utfall ved mulige handlinger, osv.
3. Den forteller (TELL) kunnskapsbasen hvilken handling som ble valg og utfører handlingen

Detaljene ved representasjonsspråket er gjemt bak de tre funksjonene MAKE-PERCEPT-SENTENCE, MAKE-ACTION-QUERY og MAKE-ACTION-SENTENCE. Disse markerer grensesnittet mellom sensorer, aktuatorer og resonneringssystem. Detaljene ved inferensmekanismene er gjemt bak TELL og ASK, som vi skal se på senere.

Det er TELL og ASK som skiller den kunnskapsbaserte agenten fra agentene i kapittel 2, som er program som regner ut handlinger. Den kunnskapsbaserte agenten kan motta beskrivelser på kunnskapsnivået, der agentens oppførsel kan bestemmes ved å spesifisere hva agenten vet og hvilke mål den har. For eksempel for den automatiserte taxien, kan målet være å ta passasjeren fra San Francisco til Marin County, og den vet at Golden Gate Bridge er den eneste koblingen mellom de to lokasjonene. Vi kan da forvente at den krysser Golden Gate Bridge, fordi den vet at dette vil oppnå målet.

En kunnskapsbasert agent kan bygges ved å fortelle den (TELL) hva den skal vite. Hvis man starter med en tom KB, kan agent designeren fortelle (TELL) setninger en om gangen helt til agenten vet hvordan den skal operere sine omgivelser. Dette kalles **deklarativ (forklarende) systembygging**. Det motsatte er prosedyretilnærmingen, der ønsket oppførsel blir gitt som programkode. Suksessfulle agenter vil som regel bruke begge deler.

WUMPUS verden



Wumpus verden er en omgivelse der kunnskapsbaserte agenter kan vise hva de er verdt. I denne verden befinner man seg i en hule med mange rom og i en av disse er wumpus-monsteret som spiser alle som entrer samme rom. Wumpus kan skytes av en agent, men agenten har kun én pil. Noen rom inneholder bunnløse groper (*pits*), som vil fange alle som entrer rommet (bortsett fra wumpus, som er for stor til å falle gjennom). Den eneste fordelen ved denne omgivelsen er muligheten for å finne en haug med gull. Følgende er en presis definisjon av oppgaveomgivelsene, som følger PEAS beskrivelsen (s. 9):

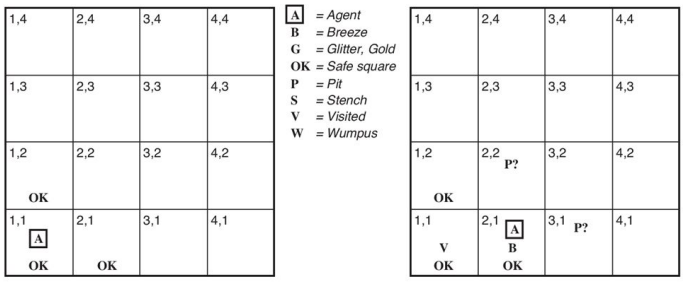
- **P.** ***Performance measure*** = +1000 for å klatre ut av hulen med gullet, -1000 for å falle i en *pit* eller bli spist av wumpus, -1 for hver handling og -10 for å bruke opp pilen. Spillet slutter enten ved at agenten dør eller ved at agenten klatrer ut av hulen.
- **E.** ***Environment*** = et 4x4 gitter med rom. Agenten vil alltid starte i rom [1, 1] og ser mot høyre. Lokasjonen til gullet, wumpus-monsteret og pits er tilfeldig, men de kan ikke være i startposisjonen.
- A. ***Actuators*** = agenten kan bevege seg med handlingene $Forward$, $TurnLeft$ og $TurnRight$ (snuoperasjonene er 90°). Hvis agenten forsøker å bevege seg fremover og krasjer i en vegg, vil ikke agenten bevege seg. Handlingen *Grab* brukes for plukke opp gullet, mens handlingen $shoot$ brukes for å skyte pilen i en rett linje i retningen agenten vender mot. Agenten har kun en pil, så det er kun første $shoot$ som har effekt. Handlingen $climb$ brukes for å klatre ut av hulen fra posisjon [1, 1].
- **S.** ***Sensors*** = agenten har fem sensorer:
  - **a.** I rommet som inneholder wumpus og i naborom (horisontalt og vertikalt) vil agenten oppfatte en stank (*stench*)
  - **b.** I naborom til *pits* vil agenten oppfatte en bris (*breeze*)
  - **c.** I rommet der gullet befinner seg vil agenten oppfatte en glitter
  - **d.** Når agenten går inn i en vegg vil den oppfatte en bump
  - **e.** Når wumpus blir drept vil agenten høre et skrik ($scream$)

  Disse blir gitt til agenten som en liste med fem symboler. For eksempel hvis det er en *stench* og en *breeze*, men ingen *glitter*, *bumb* eller *scream* vil agentprogrammet motta listen $[Stench, Breeze, None, None, None]$.

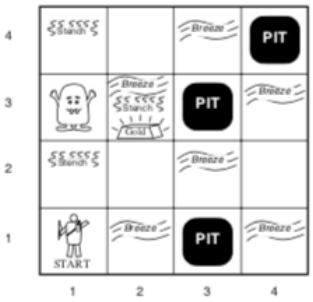
Vi kan karakterisere wumpus omgivelsen basert på dimensjonene i kapittel 2:

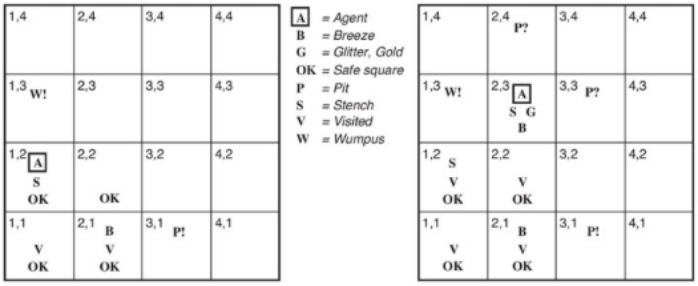
- **Deterministisk**: nåværende tilstand og handlingen som utføres av agenten vil bestemme neste tilstand
- **Statisk:** omgivelsen vil ikke endre seg mens agenten vurderer neste handling, siden wumpus og pits ikke flytter seg
- **Diskret**: det er et endelig nummer av distinkte tilstander
- **Single agent**: antar at wumpus er et naturlig fenomen (del av omgivelsen)
- **Delvis observerbar** = agenten har kun lokal persepsjon og kan dermed ikke sanse alle deler av omgivelsen fra én lokasjon
- **Sekvensiel**l: nåværende avgjørelse kan påvirke fremtidige avgjørelser (eks: oppdaget *breeze* kan gjøre at agenten snur)

Den største utfordringen for agenten er den initiale uvitenheten om konfigurasjonen til omgivelsen. For å overkomme dette må agenten bruke lokal resonnering. I de fleste tilfellene vil agenten klare å hente gullet, men i 21% av tilfellene er omgivelsene urettferdig, ved at gullet er i en pit eller er omringet av pits.



Vi ser på et eksempel der en kunnskapsbasert agent plasseres i omgivelsen på figuren. Vi bruker et uformelt kunnskapsrepresentasjon-språk som består av å skrive symboler i et gitter. Den initiale kunnskapsbasen til agenten består av reglene til omgivelsen, så den vet at den befinner seg i [1, 1] og at den er trygg (= 'A' i [1, 1] med 'OK'). Det første perseptet er [None, None, None, None, None], som gir agenten at lokasjonene [2, 1] og [1, 2] er trygge (= 'OK'). En forsiktig agent vil kun bevege seg inn i rom som den vet er trygg. I dette tilfellet velger agenten å bevege seg inn i [2, 1]. Her oppfatter agenten en *breeze* (= 'B'), noe som betyr at det må være en pit i et naborom. Det kan ikke være i [1, 1] pga. reglene til spillet, så det må være i [2, 2], [3, 1] eller begge (= 'P?' i gitter). Ved dette punktet er det kun ett rom som er trygt, så agenten vil snu og gå tilbake til [1, 1] og deretter til [1, 2]. Her oppfatter agenten en *stench* (= 'S'), noe som betyr at wumpus er i et naborom. Den kan ikke være i [1, 1] pga. reglene til spillet og den kan heller ikke være i [2, 2] fordi da ville agenten oppfattet stanken når den var i [2, 1]. Derfor kan agenten konkludere med at wumpus er i rom [1, 3] (= 'W!'). Siden agenten ikke oppfatter noe *breeze* i [1, 2] kan den også konkludere med at det er en pit i [3, 1] (= 'P!') og at rom [2, 2] er trygg (= 'OK'). Dette er relativt vanskelig inferens, siden det kombinerer kunnskap som er oppnådd ved ulike tidspunkt og ulike posisjoner. Agenten vil bevege seg inn i rom [2, 2] siden det er trygt. Her oppfatter den ingenting, så naborommene er trygge. Agenten beveger seg inn i rom [2, 3], der den oppfatter en *glitter*. Agenten vil derfor utføre *Grab* og vil deretter returnere til [1, 1] med gullet.





Legg merke til at når agenten konkluderer basert på tilgjengelig informasjon, er det garantert at konklusjonen er korrekt dersom informasjonen er korrekt. Dette er en grunnleggende egenskap ved logisk resonnering. Wumpus verden illustrerer hvordan agenten bruker kunnskapen i KB for å bestemme hvilke handlinger den skal utføre og hvordan KB blir oppdatert ettersom agenten mottar persepter. For at dette skal være mulig trenger vi et **kunnskapsrepresentasjon-språk, som uttrykker kunnskapen om verden på en måte som kan håndteres av datamaskiner.**

Logikk

Denne seksjonen vil introdusere de grunnleggende komponentene ved logisk representasjon og resonnering. Dette er viktige begrep som brukes i logikken.

Logisk representasjon

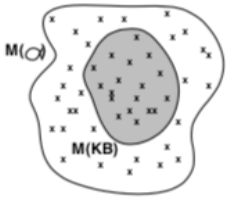
Setningene i kunnskapsbasen (KB) er uttrykt i et kunnskapsrepresentasjon-språk, som vil ha:

- **Syntaks** = setningene i KB er utrykket i henhold til syntaksen hos språket, og dette gir hva som er velformede setninger. For eksempel i aritmetikk vil "$x + y = 4$" være en velformet setning som følger syntaksen, mens "$x4y+=$ " er ikke det.

- **Semantikk** = setningene i KB vil ha en semantikk som bestemmer hva setningen betyr og definerer sannheten til en setning mht. hver mulige verden. For eksempel vil semantikken til aritmetikk bestemme at setningen "$x + y = 4$" er sann i en verden der $x = 2$ og $y = 2$, men ikke i en verden der $x = 1$ og $y = 1$.I logikken må hver setning være sann eller falsk (det er ingen mellomting). Vi kan erstatte «mulige verden» med modell, der «mulige verden» er en virkelig omgivelse som agenten kan befinne seg i (eks: $x + y = 4$, kan være at det sitter 2 kvinner og 2 menn ved bordet), mens modellen er en matematisk abstraksjon (eks: $x + y = 4$, vil være alle mulige tildelinger av reelle nummer til variablene x og y). **Hvis en setning $\boldsymbol{\alpha}$ er sann i modellen $\boldsymbol{m}$, sier vi at $\boldsymbol{m}$ tilfredsstiller $\boldsymbol{\alpha}$ eller at $\boldsymbol{m}$ er en modell av $\boldsymbol{\alpha}$. Notasjonen $\boldsymbol{M(\alpha)}$ er alle modellene av $\boldsymbol{\alpha}$ (dvs. alle modellene der setningen $\boldsymbol{\alpha}$ er sann)**. For eksempel vil $m = "x = 2, y = 3"$ og $\alpha = x + y = 4$ bety at $M(\alpha) = m$.

Logisk resonnering og utledning

Logisk resonnering involverer logiske følger (*entailment*) mellom setninger, der sannheten til en setning følger av sannheten til en annen setning. For å betegne at setning $\boldsymbol{\alpha}$ logisk følger av setning $\boldsymbol{KB}$, bruker vi notasjonen: $\boldsymbol{KB| = \alpha}$. Dette betyr at i alle modeller der $\boldsymbol{KB}$ er sann vil også $\boldsymbol{\alpha}$ være sann. Dette kan skrives som:



$$KB| = \alpha \quad \text{hvis og bare hvis} \quad M(KB) \subseteq M(\alpha)$$

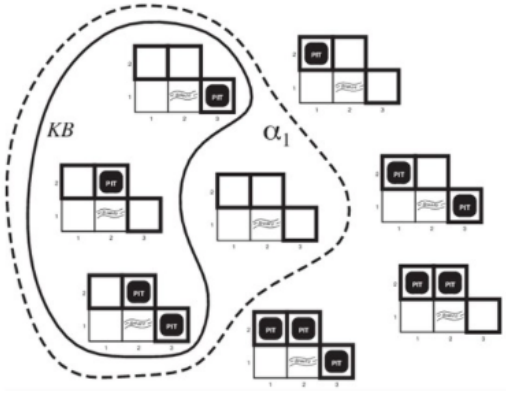
Legg merke til retningen til $\subseteq$, som skyldes at KB er en sterkere påstand enn α og vil derfor utelukke flere verdener. For eksempel kan $KB =$ "RBK vant", "Brann vant" og $\alpha =$ "RBK vant". Logiske følger er et forhold mellom setninger, som er basert på semantikk.

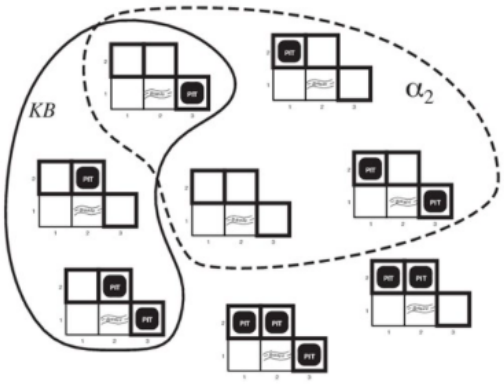
Bruk av logiske følger i inferens – modellsjekk

Logiske følger kan brukes av agenten for å bestemme om ulike påstander er sann eller ikke, gitt informasjonen den har i KB. **Dersom påstanden logisk følger av KB, kan agenten konkludere med at påstanden er sann. For at påstanden skal logisk følge av KB, må den være sann i alle modeller der KB er sann.** For å sjekke dette kan vi bruke en form for logisk inferens som kalles **modellsjekk**, **som ser på alle mulige modeller og sjekker om påstand $\boldsymbol{\alpha}$ er sann i alle modellene der $\boldsymbol{KB}$ er sann, dvs. $\boldsymbol{M(KB) \subseteq M(\alpha)}$**.

Vi bruker dette i analysen av wumpus verden. Vi ser på situasjonen på figuren til venstre, der KB inneholder reglene til spillet og informasjonen om at agenten ikke har oppfattet noe i [1, 1] og en *breeze* i [2, 1]. Agenten er interessert i om det er en pit i [1, 2], [2, 2] eller [3, 1]. Hver av disse kan inneholde en pit eller ikke, så det er $2^3 = 8$ mulige modeller (se figur til høyre). **KB er et sett med setninger og den vil være falsk i modeller som motsier det agenten vet**. I denne wumpus verdenen vil KB være falsk i enhver modell der [1, 2] inneholder en pit, siden det er ingen breeze i [1, 1]. Det er kun tre modeller der KB er sann. Vi ser på to konklusjoner: $\alpha_1 =$ "Det er ingen pit i $[1, 2]$" og $\alpha_2 =$ "Det er ingen pit i $[2, 2]$" . **Vi kan se at i alle modeller der KB er sann, vil også $\boldsymbol{\alpha_1}$ være sann, noe som betyr at $\boldsymbol{KB| = \alpha_1}$ og agenten vil konkludere med at det er ingen pit i [1, 2**]. Vi kan også se at i noen modeller der KB er sann, er α_2 falsk, noe som betyr at $KB| \neq \alpha_2$ og agenten kan derfor ikke konkludere med at det er ingen pit i [2, 2].

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 OK	2,2 P?	3,2	4,2
1,1 V OK	2,1 A B OK	3,1 P?	4,1





Inferensprosedyre – egenskaper

Modellsjekk er en form for **inferensprosedyre**, der logiske følger brukes for å utføre logisk inferens. For å forstå forskjellen mellom utledning og følger kan man bruke en analogi der KB er en høystakk og α er en nål. Logisk følge vil være at nålen er i høystakken, mens inferens vil finne nålen. **Hvis inferensprosedyre i kan utlede α fra KB, skriver vi:**

$$KB \vdash_i \alpha$$

Dette betyr at **α er utledet fra KB av i**. Det er to egenskaper som er viktige ved en inferensprosedyre:

1. **Soliditet (*soundess*)** = inferensprosedyren er solid, hvis den kun utleder en setning α fra andre setninger som α logisk følger av. Dvs. hvis $KB \vdash_i \alpha$, betyr det at $KB| = \alpha$. Dette betyr at inferensprosedyren ikke finner opp ting underveis. I analogien vil en solid inferensprosedyre kun finne nåler som eksisterer i høystakken.
2. **Kompletthet (*completeness*)** = inferensprosedyren er komplett, hvis den kan utlede alle setninger som følger av en annen setning. Dvs. hvis $KB| = \alpha$, betyr det at $KB \vdash_i \alpha$. I analogien vil en fullstendig inferensprosedyre finne alle nålene som eksisterer i høystakken.

Dette kan brukes for å lage en resonneringsprosess der konklusjonene er garantert å være sanne i en verden der premissene er sanne. **Hvis KB er sann i en reell verden, så vil enhver setning α som utledes fra KB av en solid inferensprosedyre også være sann i den reelle verden.**

Jording (*grounding*)

Jording (*grounding*) er koblingen mellom de logiske resonneringsprosessene og den virkelige omgivelsen som agenten eksisterer i. Det spesifiserer hvordan vi kan vite at KB er sann i den virkelige verden (husk: KB er syntaks i hodet til agenten). Et forenklet svar er at det er agentens sensorer som lager koblingen. For eksempel i wumpus verden vil agenten ha en luktesensor og når den merker en lukt, vil den lage en passende setning. Når denne setningen er i KB, vil det være sant i den virkelige verden. Betydningen og sannheten til setninger vil altså defineres av sanseprosessene og produksjonen av setningene. Generelle regler, slik som at stanken produseres av en wumpus, blir laget av setningsproduksjonen som kalles **læring**. Det kan være feil ved læringen, så derfor kan det hende at KB ikke er sann i den virkelige verden. Med en god læringsprosedyre vil dette minimeres.

Proposisjonslogikk

Proposisjonslogikk er en enkel, men kraft form for logikk. **En proposisjon er en forklarende påstand om verden som vil være sann eller falsk.** For eksempel vil «Norge er i Europa» være en sann proposisjon, mens «Stockholm er hovedstaden i Norge» er en falsk proposisjon. «Hva er navnet ditt?» er ikke en proposisjon, fordi det er ikke en forklarende påstand.

Syntaks

Syntaksen til proposisjonslogikk definerer de tillatte setningene. Den **atomiske setningen** består av et enkelt **proposisjonssymbol,** som representerer en proposisjon som kan være sann eller falsk. For eksempel kan $W_{1,3}$ = "wumpus er i [1, 3]". Proposisjonssymbolet kan også være konstansene False og True. **Sammensatte setninger** består av atomiske setninger som er koblet sammen av logiske konnektiver. Fem vanlige typer er negasjon (¬), konjunksjon (∧), disjunksjon (∨), implikasjon (⇒) og ekvivalens (⇔). Her vil negasjon ha høyest prioritet (dvs. $\neg A \vee B = (\neg A) \vee B$). Det er viktig å ta hensyn til parentesene.

Semantikk

Semantikken definerer reglene som brukes for å bestemme sannheten til en setning mht. en bestemt modell. **Modellen vil bestemme sannhetsverdien (*true* eller *false*) for alle proposisjonssymbolene**. For eksempel vil setningen bruker $P_{1,2}$, $P_{2,2}$ og $P_{3,1}$, kan en mulig modell være $\{P_{1,2} = false, P_{2,2} = false, P_{3,1} = true\}$. **Semantikken må spesifisere hvordan man regner ut sannhetsverdien til en setning, gitt en modell**. For atomiske setninger vil dette innebære:

- True er sann i alle modeller, og False er falsk i alle modeller
- Sannhetsverdien til alle andre proposisjonssymboler må spesifisere i modellen

For sammensatte setninger, er det fem regler:

- $\neg P$ er sann, hvis P er falsk i modellen
- $P \wedge Q$ er sann, hvis P og Q er falsk i modellen
- $P \vee Q$ er sann, hvis P eller Q er falsk i modellen
- $P \Rightarrow Q$ er sann, så lenge Q ikke er falsk når P er sann i modellen
- $P \Leftrightarrow Q$ er sann, så lenge både P og Q er falsk eller sann i modellen

Disse reglene kan utrykkes i sannhetstabeller.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

En enkel kunnskapsbase

Nå som vi har definert semantikken til proposisjonslogikken, kan vi lage en kunnskapsbase for wumpus verden. For hver $[x, y]$ lokasjon, bruker vi følgende symboler:

- $P_{x,y}$ er sann hvis det er en pit i $[x, y]$
- $W_{x,y}$ er sann hvis det er en wumpus i $[x, y]$, død eller i live
- $B_{x,y}$ er sann hvis agenten oppfatter en breeze i $[x, y]$
- $S_{x,y}$ er sann hvis agenten oppfatter en stench i $[x, y]$

Disse kan brukes for å utlede $\neg P_{1,2}$, altså at det er ingen pit i [1, 2], slik det ble gjort på side 69. Vi merker setningene, slik at vi kan referere til dem:

R_1: $\neg P_{1,1}$ (det er ingen pit i [1, 1])
R_2: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ (breeze oppfattes i [1, 1] kun hvis det er en pit i [1, 2] eller [2, 1])
R_2: $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$ (breeze oppfattes i [2, 1] kun hvis det er en pit i [1, 1], [2, 2], [3, 1])
R_4: $\neg B_{1,1}$ (det oppfattes ingen breeze i [1, 1])
R_5: $B_{2,1}$ (det oppfattes breeze i [2, 1])

Dette er en blanding av regler i wumpus-verden og persepter som agenten mottar.
Innholdet i KB er: $KB = R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5$.

En enkel inferensprosedyre – modellsjekk

For å sjekke om $KB| = \alpha$ gitt en kunnskapsbase som inneholder proposisjonslogikk, kan vi bruke en **modellsjekk**:

1. Bestem setningene i KB gitt regler og oppfattede persepter frem til øyeblikket agenten ønsker å sjekke påstand α. $KB = R_1 \wedge R_2 \wedge \ldots \wedge R_n$
2. Finn alle relevante proposisjonssymboler fra KB og α
3. Finn alle mulige modeller (= $2^{\#\, proposisjonssymboler}$) og lag en sannhetstabell for hver modell med verdier for proposisjonssymbolene og setningene i KB
4. Bestem modellene der KB er sann ($KB = R_1 \wedge R_2 \wedge \ldots \wedge R_n$)

5. Bestem modellene der α er sann, og se om α er sann i alle modellene der KB er sann. Hvis det er tilfellet vil $KB| = \alpha$ og agenten kan konkludere med at påstanden α er sann

Generelt eksempel

World	A	B	C	A ∧ B	A ∧B → C
0	0	0	0	0	1
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	0	1
4	1	0	0	0	1
5	1	0	1	0	1
6	1	1	0	1	0
7	1	1	1	1	1

Vi har en kunnskapsbase som består av setningene:

- $R_1: A \wedge B \Rightarrow C$
- $R_2: A \wedge B$

Vi ønsker å se om påstanden $\alpha: C$ følger av denne KB, altså om $(R_1 \wedge R_2)| = \alpha$. I dette tilfellet vil de relevant proposisjonssymbolene være A, B og C, så det er $2^3 = 8$ ulike modeller (hver av symbolene kan være *true* eller *false*). Sannhetstabellen viser de ulike modellene. KB vil være sann i modellene der $R_1 \wedge R_2$ er sann, mens α vil være sann i modellene der C er sann. På figuren kan vi se at KB er sann i modellene $\{7\}$, mens α er sann i modellene $\{1, 3, 5, 7\}$. Siden α er sann i alle modellene der KB er sann ($\{7\} \subseteq \{1, 3, 5, 7\}$) vil $KB| = \alpha$, og vi kan si at proposisjonen α er sann.

Wumpus-verden

Vi ser på wumpus-verden frem til situasjonen der agenten har oppfattet en *breeze* i [2, 1] og må bestemme hva den skal gjøre. Agenten ønsker å sjekke om $KB| = \alpha$, der $\alpha = \neg P_{1,2}$, altså om det følger av informasjonen i KB at det ikke er noen pit i [1, 2]. Setningene i KB er gitt på forrige side (R_1-R_5). På side 69 så vi på inferensprosedyren for proposisjonssymbolene $P_{1,2}$, $P_{2,2}$ og $P_{3,1}$. **For at resonneringen skal være mer presis må vi inkludere alle relevante proposisjoner gitt KB og α**, og i dette tilfellet vil det være $B_{1,1}, B_{1,2}, P_{1,1}, P_{1,2}, P_{2,2}, P_{2,1}$ og $P_{3,1}$. Dermed får vi $2^7 = 128$ mulige modeller. Sannhetstabellen viser de noen av de ulike modellene. KB vil være sann i modellene der $R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5$ er sann, mens α vil være sann i modellene der $\neg P_{1,2}$ er sann (dvs. $P_{1,2}$ er falsk). På figuren kan vi se at α er sann i alle modellene der KB er sann. Derfor vil $KB| = \alpha$, og agenten kan konkludere med at det ikke er noen pit i [1, 2].

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	false	false	false	false	false	false	true	true	true	true	false	false
false	false	false	false	false	false	true	true	true	false	true	false	false
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	true	false	false	false	false	false	true	true	false	true	true	false
false	true	false	false	false	false	true	true	true	true	true	true	true
false	true	false	false	false	true	false	true	true	true	true	true	true
false	true	false	false	false	true	true	true	true	true	true	true	true
false	true	false	false	true	false	false	true	false	false	true	true	false
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	true	true	true	true	true	true	false	true	true	false	true	false

Denne inferensprosedyren er **solid** fordi den bruker de logiske følgene og den er **komplett** fordi den fungerer for enhver KB og α, og vil alltid terminere (det er kun endelig mange modeller som kan undersøkes). Hvis KB og α inneholder n symbolder, vil metoden sjekke sannhetstabellen inneholde 2^n modeller som må sjekkes. **Tidskompleksiteten er derfor $\mathbf{O(2^n)}$.** Opplistingen er dybde-først, så romkompleksiteten vil være $O(n)$. Vi skal senere se at det finnes algoritmer som er mer effektive. Alle kjente inferensalgoritmer for proposisjonslogikk har likevel worst-case kompleksitet som er eksponentiell til størrelsen til input, siden det er et co-NO-komplett problem.

Inferensmetoder brukes for å sjekke om $KB| = \alpha$, og i proposisjonslogikk har vi følgende metoder:

- **Modellsjekking/opplisting**
- **Inferens algoritmer:**
  - **Resolusjon og bevis ved motsigelse**
  - **Forward og Backward chaining**

Vi skal nå se nærmere på inferensalgoritmene, og vi begynner med resolusjon og bevis ved motsigelse.

Teorembevis

Teorembevis kan brukes for å vise følger (eks: $KB| = \alpha$), og det innebærer bruken av inferensregler på setningene i kunnskapsbasen for å lage bevis på den ønskede setningen uten å lage modeller. Teorembevis kan være mer effektivt enn modellsjekking, dersom det er mange modeller og lengden til beviset er kort.

Før vi kan se på teorembevis algoritmer, må vi definere noen begrep:

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Longrightarrow \beta) &\equiv (\neg\beta \Longrightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Longrightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Longrightarrow \beta) \wedge (\beta \Longrightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

- **Logisk ekvivalens ($\alpha \equiv \beta$)** = to setninger er logisk ekvivalente dersom de er sann i samme modeller, dvs. de har lik sannhetstabell. For eksempel vil $(P \vee Q) \equiv (Q \vee P)$. Figuren viser viktige ekvivalenser. Merk: hvis $\alpha \equiv \beta$ vil $\alpha| = \beta$ og $\beta| = \alpha$
- **Gyldig/tautologi** = en setning er gyldig hvis den er sann i alle modeller. For eksempel vil $P \vee \neg P$ være gyldig. Dette er koblet til inferens via **deduksjonsteoremet**: «$KB| = \alpha$ kun hvis $KB \Rightarrow \alpha$ er gyldig. Legg merke til at dette er grunnlaget i modellsjekking, som ser om α er sann i alle modeller der KB er sann.
- **Satisfiable (SAT)** = en setning er satisfiable hvis den er sann i minst en av modellene. Dette kan sjekkes ved å liste opp modellene helt til man finner en modell der setningen er sann (NP-komplett problem). Dette er koblet til inferens ved at $KB| = \alpha$ kun hvis $(KB \wedge \neg\alpha)$ er unsatisfiable. **Dette tilsvarer bevis ved motsigelse, siden man antar $\neg\alpha$ og viser at dette fører til en motsigelse på et kjent aksiom KB.**
- **Monotoni** =når en setning legges til kunnskapsbasen (KB), vil den logiske følgen fortsatt gjelde. Altså, hvis $KB| = \alpha$ så vil $KB \wedge \beta| = \alpha$. En setning som legges til kan hjelpe agenten til å trekke andre konklusjoner, men det kan ikke ugyldiggjøre en konklusjon som allerede er utledet.

Inferens og bevis

Inferensregler kan brukes for å utlede et bevis, og det er kjeder av konklusjoner som fører til et ønsket mål. Figurene under viser noen kjente inferensregler:

- Modus ponens (method of affirming)

$$\frac{A \rightarrow B \quad A}{B}$$

- Modus Tollens (method of denying)

$$\frac{A \rightarrow B \quad \neg B}{\neg A}$$

- Hypothetical Syllogism

$$\frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C}$$

- Disjunctive syllogism (Unit resolution)

$$\frac{A \vee B \quad \neg A}{B}$$

- And-elimination

$$\frac{A_1 \wedge A_2 \wedge \ldots\ldots \wedge A_n}{A_i}$$

- And-introduction

$$\frac{A_1, A_2, \ldots\ldots, A_n}{A_1 \wedge A_2 \wedge \ldots\ldots \wedge A_n}$$

- Or-introduction

$$\frac{A_i}{A_1 \vee A_2 \vee \ldots A_i \ldots \vee A_n}$$

- Elimination of double negation

$$\frac{\neg\neg A}{A}$$

- Implication Creation

$$\frac{A}{B \Longrightarrow A}$$

- Implication Distribution

$$\frac{A \Longrightarrow (B \Longrightarrow C)}{(A \Longrightarrow B) \Longrightarrow (A \Longrightarrow C)}$$

For eksempel ser vi at **Modus ponens** gir at dersom $A \rightarrow B$ og A er sann, vil også B være sann. Et annet eksempel er **And-eliminering** som gir at dersom $A \wedge B$ er sann, vil også A være sann. **Alle de logiske ekvivalentene kan brukes som inferensregler.**

Vi skal nå se hvordan inferensregler kan brukes i wumpus-verden, for å vise $\neg P_{1,2}$ gitt setningene $R_1 - R_5$ i KB (s. 70). Vi begynner med å bruke bibetingelse eliminering på R_2:

$$R_6: \quad B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1}) \wedge (P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}$$

Deretter bruker vi And-eliminering på R_6 og kontraposisjon på R_7:

$$R_7: \quad (P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}$$
$$R_8: \quad \neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})$$

Modus ponens på R_8 og $R_4(\neg B_{1,1})$ gir:

$$R_9: \quad \neg(P_{1,2} \vee P_{2,1})$$

De Morgens regel gir dermed konklusjonen:

$$R_{10}: \quad \neg P_{1,2} \wedge \neg P_{2,1}$$

Det er altså ingen pit i [1, 2] eller [2, 1].

Dette er den såkalte inferensregel-tilnærmingen, der man bruker inferensregler som matcher setningene i KB, helt til man har bevist α. Søkealgoritmene i kapittel 3 kan brukes for å finne sekvensen av steg som utgjør et slikt bevis, dersom problemet defineres som følger:

- **Initial tilstand** = den initiale kunnskapsbasen
- **Handlinger** = bruken av øverste del av inferensreglene på setninger som matcher
- **Overgangsmodell** = nederste del av inferensreglene som utføres
- **Måltest** = en setning som inneholder setningen vi forsøker å bevise

Dette er en alternativ tilnærming til modellsjekking. Fordelen ved å søket etter beviset er at den kan ignorere irrelevante proposisjoner, uansett hvor mange det er.. For eksempel kan vi se på beviset over at den ikke bruker $B_{2,1}, P_{1,1}, P_{2,2}$ eller $P_{3,1}$. Disse kan ignoreres fordi målproposisjonen $P_{1,2}$ blir kun nevnt i R_2. De andre proposisjonssymbolene i R_2 er kun nevnt i R_4, så derfor trenger ikke søket å se på R_1, R_3 eller R_5. Dette gjelder selv om vi legger til millioner av setninger til KB. **Modellsjekk må derimot ta hensyn til alle proposisjonssymbolene** og vil kreve $O(2^n) \rightarrow \infty$ tid! I tillegg er det en solid inferensmetode. **Problemet med inferensregel-tilnærmingen er at ved ulike steg kan det hende at flere regler kan brukes**. Dette løses med resolusjon, som vi nå skal se nærmere på.

Resolusjonsbevis

Inferensalgoritmene som bruker inferensreglene for å finne beviset vil være komplett derom målet kan nås, men hvis de tilgjengelige inferensreglene er utilstrekkelige, vil ikke målet kunne nås. For eksempel for beviset over, hvis algoritmen ikke har tilgang til bibetingelse eliminering, vil den ikke klare å finne beviset. **Resolusjon er en inferensregel som alltid vil gi komplette inferensalgoritmer når den kombineres med en fullstendig søkealgoritme (dvs. søket finner løsningen hvis den eksisterer).**

$$\frac{l_1 \vee \ldots \vee l_{i-1} \vee l_i \vee l_{i+1} \ldots \vee l_n, \qquad m_1 \vee \ldots \vee m_{j-1} \vee m_j \vee m_{j+1} \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \ldots \vee l_n \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \ldots \vee m_n}$$

Dette er en inferensregel. Legg merke til kommaet mellom de to disjunksjonene på toppen

Dette er resolusjon inferensregelen. Her er l_i og m_j komplementære literaler (dvs. $l_i = \neg m_j$), så disse fjernes fra disjunksjonene som slås sammen. Hvis KB inneholder setninger som er disjunksjoner av flere proposisjonssymboler og inneholder negerte versjoner av samme proposisjonssymbol (eks: $R_1: A \vee B$ og $R_2: D \vee \neg B$), vil proposisjonssymbolet fjernes fra begge disjunksjonene (eks: $R_3: A \vee D$).

Vi ser på wumpus-verden, der agenten har returnert fra [2, 1] til [1, 1] og går deretter til [1, 2], der den oppfatter en *stench*, men ingen *breeze*. Vi legger til følgende setninger til KB:

$$R_{11}: \neg B_{1,2}$$
$$R_{12}: B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3})$$

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic
  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
    for each C_i, C_j in clauses do
      resolvents ← PL-RESOLVE(C_i, C_j)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

R_1 i KB gir at $\neg P_{1,1}$. Vi kan bruke samme fremgangsmåte som på forrige side for å finne at det er ingen pit i [1, 3] og [2, 2]:

$$R_{13}: \ \neg P_{2,2}$$
$$R_{14}: \ \neg P_{1,3}$$

Ved å bruke bibetingelse eliminering på R_3, etterfulgt av Modus Ponens med R_5, vil vi få følgende proposisjon:

$$R_{15}: \ P_{1,1} \vee P_{2,2} \vee P_{3,1}$$

Nå kommer første bruk av resolusjonsregelen, siden $\neg P_{2,2}$ i R_{13} vil oppløse (resolvere) $P_{2,2}$ i R_{15}, slik at vi får resolvent:

$$R_{16}: \ P_{1,1} \vee P_{3,1}$$

Her har vi brukt at R_{15} sier at det er en pit i [1, 1], [2, 2] eller [3, 1] og R_{13} sier at det ikke er noen pit i [2, 2], noe som må bety at den er i [1, 1] eller [3, 1]. På samme måte vil **$\neg P_{1,1}$ i R_1 oppløse (resolvere) $P_{1,1}$ i R_{16}, slik at vi får resolvent:**

$$R_{17}: \ P_{3,1}$$

Her har vi brukt at R_{16} sier at det er en pit i [1, 1] eller [3, 1] og R_1 sier at det ikke er noen pit i [1, 1], noe som må bety at det er en pit i [3, 1]. De to siste inferensstegene er eksempler på bruk av resolusjon inferensregelen. **Det er også viktig at den resulterende disjunksjonen inneholder kun en kopi av hver proposisjonssymbol = faktorisering**. For eksempel hvis vi oppløser $(A \vee B)$ med $(A \vee \neg B)$ vil vi få $(A \vee A)$ som reduseres til bare A. **Resolusjon er solid og komplett for proposisjonslogikk.** Dvs. den vil kun utlede setninger som følger av andre setninger og gitt enhver setning α og β i proposisjonslogikk, kan resolusjon brukes for å bestemme om $\alpha| = \beta$.

Merk forskjellen mellom klausul og CNF: en klausul er en disjunksjon av literaler, mens CNF er en konjunksjon av klausuler. Ved definite og hornklausuler er det begrensninger på antall positive literaler i disjunksjonen

Conjunction normal form (CNF)

En setning er i CNF dersom den er utrykket som en konjunksjon av disjunktive klausuler. En disjunktiv klausul er en disjunksjon av literaler, som er proposisjonssymbol eller negerte proposisjonssymbol. For å kunne brukes resolusjon på setninger, må de være gitt i CNF. Alle setninger i proposisjonslogikk er logisk ekvivalent med konjunksjoner av klausuler, så derfor kan de omformes til CNF. Vi ser på et eksempel der setningen $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ omformes til CNF:

1. Erstatt $A \Leftrightarrow B$ med $A \Rightarrow B \wedge B \Rightarrow A$:
$$B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1}) \wedge (P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}$$
2. Erstatt $A \Rightarrow B$ med $\neg A \vee B$:
$$\left(\neg B_{1,1} \vee (P_{1,2} \vee P_{2,1})\right) \wedge \left(\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}\right)$$
3. CNF krever at $\neg$ kun er ved proposisjonssymboler, så disse flyttes innover med de Morgan:
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge \left((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}\right)$$
4. Distribusjon av $\vee$ over $\wedge$ gir:
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$$

Denne setningen er nå i CNF. Den er vanskeligere å lese, men kan brukes som input til resolusjonsregelen. Da vil resolusjonen brukes på klausulene.

Resolusjonsalgoritme

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
    for each C_i, C_j in clauses do
      resolvents ← PL-RESOLVE(C_i, C_j)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

Inferensprosedyrer som er basert på resolusjon bruker bevis ved motsigelse, som vil si at de viser at $KB| = \alpha$ ved å vise at $(KB \wedge \neg\alpha)$ er unsatisfiable. Algoritmen vil først omforme $(KB \wedge \neg\alpha)$ til CNF, og de resulterende klausulene blir deretter sendt som input til resolusjonen. Hvert klausulpar som inneholder komplementære literaler, blir oppløst og produserer en ny klausul, som legges til settet hvis den ikke allerede er tilstede. Prosessen gjentas helt til man når ett av følgende resultat:

Legg merke til at α defineres som en query. For eksempel kan $\alpha = P_{3,1}$ formuleres som «Er det en pit i [3, 1]?

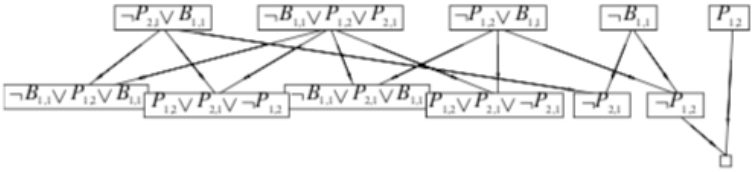
- **Ingen flere klausuler kan legges til, noe som betyr at α ikke følger av KB**
- **To klausuler oppløses til en tom klausul, noe som betyr at $KB| = \alpha$**

En tom klausul er ekvivalent med *false*, siden en disjunksjon kun er *true* dersom en av disjunktene er *true*. Dette vil gjøre at $(KB \wedge \neg\alpha)$ blir ekvivalent med *false*, siden det er en konjunksjon av klausulene, så når en klausul alltid er *false* vil også konjunksjonen alltid være *false*. Dermed vil $(KB \wedge \neg\alpha)$ være unsatisfiable, noe som betyr at $KB| = \alpha$.

Vi ser på wumpus-verden, der agenten er i lokasjon [1, 1]. Den oppfatter ingen *breeze*, så derfor kan det ikke være noen pits i nabolokasjonene. Den relevante kunnskapsbasen er:

$$KB = R_2 \wedge R_4 = \left(B_{1,1} \Leftrightarrow \left(P_{1,2} \vee P_{2,1}\right)\right) \wedge \neg B_{1,1}$$

Og vi ønsker å vise $\alpha = \neg P_{1,2}$. Når vi omformer $(KB \wedge \neg\alpha)$ til CNF, får vi klausulene på toppen av figuren. Den andre raden viser klausulene man får ved å oppløse (resolvere) par i den første raden. Deretter når $P_{1,2}$ blir resolvert med $\neg P_{1,2}$, får vi den tomme klausulen, vist som en liten firkant nederst til høyre. Dermed har vi vist at $KB| = \alpha$, og agenten kan konkludere med at det er ingen pit i [1, 2]. Legg merke til at mange resolusjonssteg er unødvendige, for eksempel vil $\neg B_{1,1} \vee P_{1,2} \vee B_{1,1}$ være ekvivalent med *true*. Dette skyldes at resolusjonsregelen vil kun fjerne ett par med komplementære literaler, så klausuler som inneholder flere komplementære literaler kan derfor droppes.



Denne resolusjonsalgoritmen er solid og komplett (se bevis s. 255). Problemet er at den bruker eksponentiell tid og rom.

Horn- og bestemte klausuler

Kompletthetten til resolusjon gjør at det blir en veldig viktig inferensmetode, men det er ikke alltid at man trenger hele kraften til resolusjonen. **En mer begrenset og effektiv algoritme kan oppnås ved å ha ulike begrensninger på setningene som KB inneholder**. To typer er:

- **Definite klausul** = en disjunksjon av literaler der nøyaktig én er positiv. For eksempel vil $(\neg A \vee \neg B \vee C)$ være en bestemt klausul, mens $(\neg A \vee B \vee C)$ er ikke det
- **Hornklausul** = en disjunksjon av literaler der maksimalt én er positiv. Alle definite klausuler vil være hornklausuler. Disse er lukket under resolusjon, så hvis de løser opp to hornklausuler, vil du få en hornklausul

Det er tre grunner til at det er ønsket med KB som kun inneholder slike klausuler:

1. **De kan skrives som en implikasjon med positive literaler, noe som er lettere å forstå (blir på formen $\mathbf{premiss} \rightarrow \mathbf{konklusjon}$).** For eksempel kan $\left(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1}\right)$ skrives som $\left(L_{1,1} \wedge Breeze\right) \Rightarrow B_{1,1}$, slik at det er lettere å forstå at hvis man er i [1, 1] og det er en *breeze* der, vil agenten oppfatte en $breeze$ i lokasjon [1, 1].

2. **Inferens med hornklausuler kan gjøres med forward- eller backward-chaining algoritme, som er enkle å følge for mennesker**.
3. **Hornklausuler kan brukes for å bestemme logisk følger på en tid som er lineær med størrelsen til kunnskapsbasen**

Forward og backward chaining

Forward og backward chaining brukes for å bestemme om et proposisjonssymbol q (query) følger av en kunnskapsbase med definite klausuler. De er basert på modus ponens, som gir at hvis premissene er kjent kan konklusjonen utledes ($\alpha, \alpha \Rightarrow \beta$ gir β). Fordelen ved disse er at de bruker lineær tid etter størrelsen til KB.

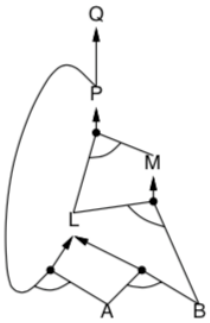
Forward chaining

Forward-chaining er en form for data-dreven resonnering, der fokuset i begynnelsen er på kjent data. Algoritmen begynner med kjent fakta i KB, i form av positive literaler. Hvis alle premissene til en implikasjon er kjent, vil konklusjonen legges til settet med kjente fakta. For eksempel hvis KB inneholder $(A \wedge B) \Rightarrow C$ og A og B er kjent, kan C legges til KB. Denne prosessen gjentas helt til query q blir lagt til eller det ikke kan utføres videre inferens (dvs. fastpunkt er nådd). **Denne algoritmen kjører på lineær tid, noe som er en stor fordel!**

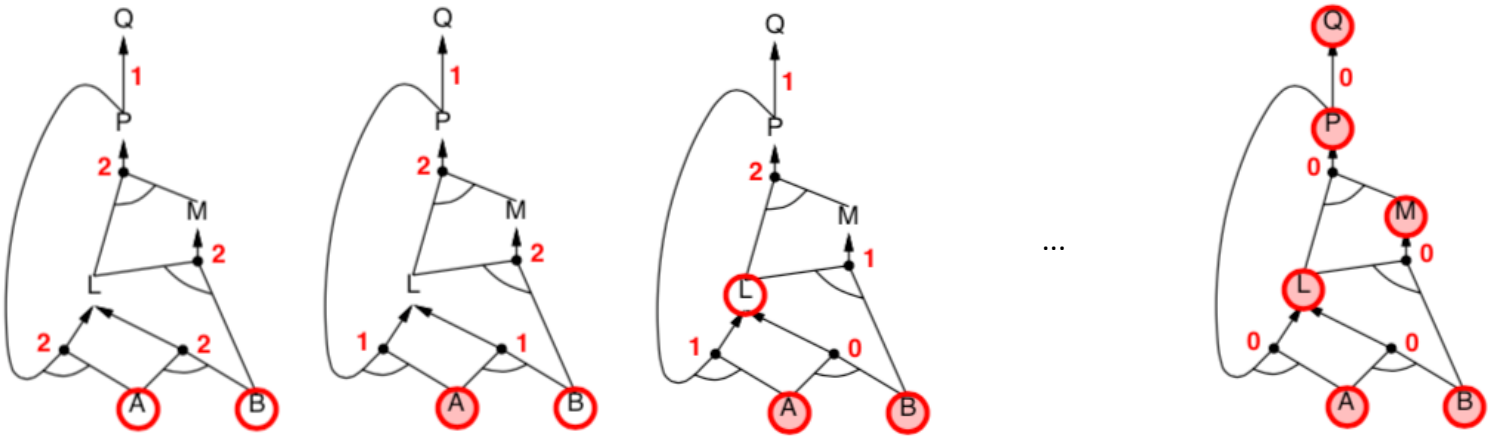
```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional Horn clauses
          q, the query, a proposition symbol
  local variables: count, a table, indexed by clause, initially # of premises
                   inferred, a table, indexed by symbol, initially false
                   agenda, a list of symbols, initially the symbols known in KB

  while agenda is not empty do
      p ← POP(agenda)
      if p = q then return true
      if inferred[p] = false then
          inferred[p] = true
          for each clause c in KB where p is in c.PREMISE do
              decrement count[c]
              if count[c] = 0 then add c.CONCLUSION to agenda
  return false
```

For å forstå algoritmen vil vi se på et eksempel. Til venstre på figuren kan vi se en enkel KB med hornklausuler, der A og B er kjente fakta. Til høyre på figuren kan vi se samme KB tegnet som en **AND-OR graf**. Her vil banene representere premissene som samles i en konklusjon. Hvis det er en bue mellom banene, må begge banene være kjent for at konklusjonen skal bli kjent. Hvis det ikke er noen bue, holder det at en av premissene er kjent. **De kjente faktaene blir satt i bunnen og inferensen forplanter seg oppover grafen så langt som mulig. Dersom den når proposisjonssymbolet (query) på toppen, er det vist at symbolet følger av KB**.

$P \Longrightarrow Q$ (Query)
$L \wedge M \Longrightarrow P$
$B \wedge L \Longrightarrow M$
$A \wedge P \Longrightarrow L$
$A \wedge B \Longrightarrow L$
A
B



For å holde styr over hvilke konklusjoner som er lagt til, kan det være lurt å markere konklusjonene med antall premisser som foreløpig er oppfylt. Når dette antallet blir 0, vil konklusjonen legges til KB (se figurene under).

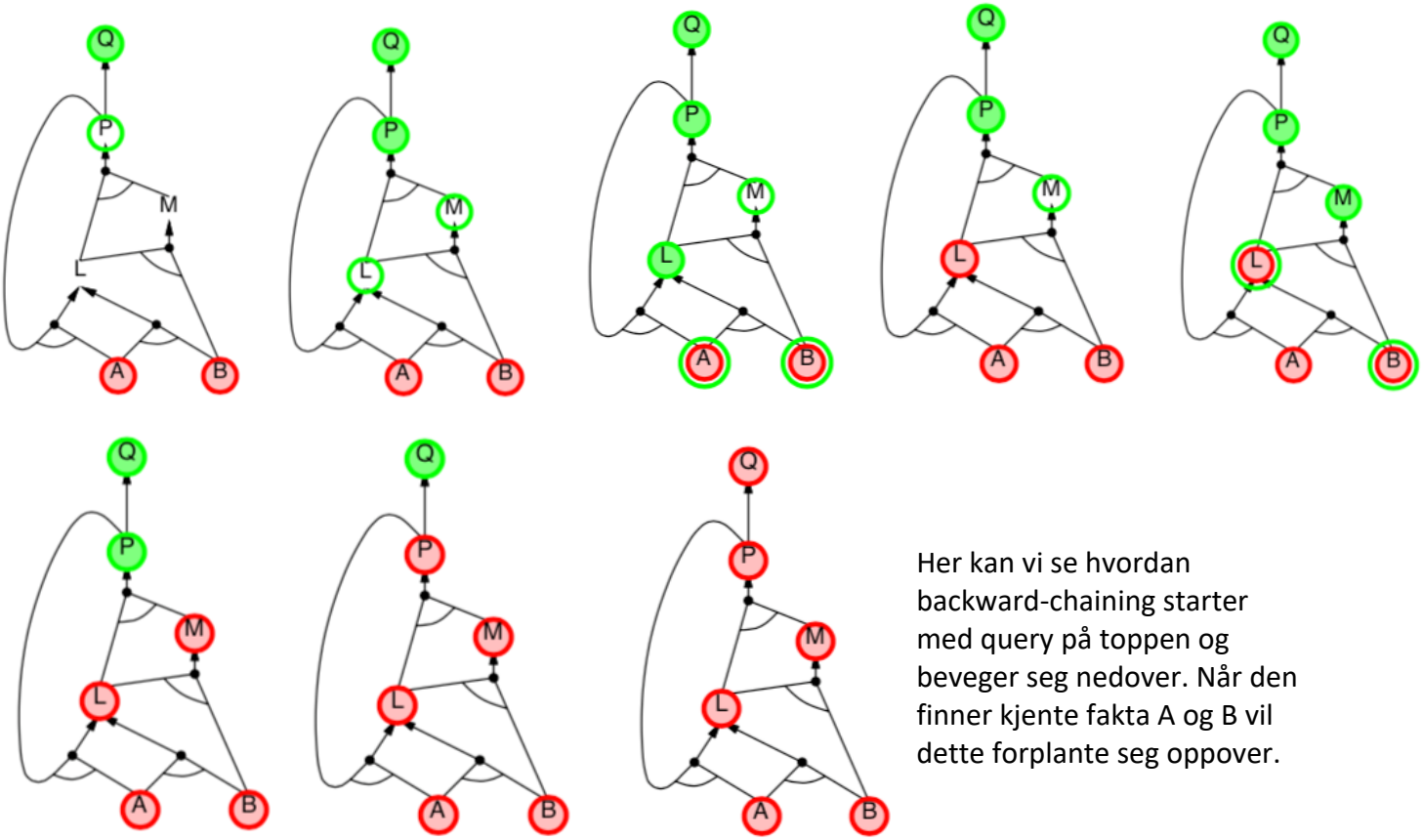


Forward chaining er solid, siden den er basert på modus ponens inferensregel, og den er komplett, siden det vil utlede alle atomiske setninger som følger av KB. Det brukes av agenten for å utlede konklusjoner fra innkommende persepter (ofte uten utgangspunkt i en bestemt query).

Backward chaining

Forward-chaining er en form for mål-dreven resonnering, der fokuset er å finne svaret på en spesifikk query. Algoritmen begynner med query og arbeider seg bakover. Hvis q er kjent, er det ikke nødvendig å utføre noe arbeid. Hvis ikke vil algoritmen finne implikasjonene i KB, der q er konklusjonen. q er sann dersom alle premissene i en av disse kan vises å være sann (ved backward chaining). **Algoritmen bruker en mål-stack for å unngå loops og ekstra arbeid, ved at den sjekker om nye delmål allerede har blitt vist å være sann eller falsk.**

Figurene under viser et eksempel. Backward chaining utføres med en AND-OR graf, der man begynner ved query i toppen og inferensen forplanter seg nedover helt til den når kjente fakta.



Her kan vi se hvordan backward-chaining starter med query på toppen og beveger seg nedover. Når den finner kjente fakta A og B vil dette forplante seg oppover.

Forward-chaining vs. backward-chaining

Forward-chaining (FC) ser på alle faktaene, så det kan derfor hende at den utfører mye arbeid som irrelevant til query (vil alltid bruke lineær tid i størrelsen til KB). FC blir brukt i automatisk, ubevisst prosessering, slik som objekt gjenkjenning og rutineavgjørelser. **Backward-chaining ser kun på fakta som er relevant til query, og vil dermed unngå å utføre irrelevant arbeid. Dette gjør at backward-chaining ofte vil bruke mindre tid enn lineær i størrelsen til KB**. BC blir brukt i problemløsing, der man forsøker å svare på et spørsmål. Eksempelet over illustrerer ikke denne forskjellen, fordi det inkluderer kun fakta som er relevant til query.

Denne seksjonen er veldig komplisert og blir ikke gjennomgått i forelesning, så den er kun dekket med et overblikk ☺

Effektiv modellsjekking (ikke i forelesning)

Denne seksjonen ser på to effektive algoritmer som brukes for å sjekke om setninger er satisfiable (dvs. løse SAT-problemet), slik at de kan teste følgen $\alpha| = \beta$, ved å se om $\alpha \wedge \neg\beta$ er unsatisfiable. De to algoritmene er:

1. **Fullstendig backtracking algoritme** = en blanding av backtracking (CSP) og modellsjekk, som tar inn setninger i CNF og vil rekursivt utføre en dybde først

opplisting av mulige modeller. Den tillater tidlig terminering, hvis den detekterer at en setning må være sann eller falsk (kan unngå søk av subtrær i søkerommet). Algoritmen bruker ulike typer heuristikk og triks fra CSP (eks: heuristikkgrad) for å gjøre søket mer effektivt (s. 260-261 i boka)

2. **Lokal søk algoritmer** = hill-climbing og simulert annealing kan brukes for å løse SAT-problemet dersom evalueringsfunksjonen er antall klausuler som er unsatisfiable (vi ønsker å finne tildeling som gjør at alle klausuler blir satisfiable). Søket vil begynne i en tilstand der alle symbolene er tildelt en sannhetsverdi, og den vil deretter flippe en verdi om gangen helt til alle klausuler er satisfiable (hvis det er mulig, dvs. finnes en løsning). Disse algoritmene vil være solide, men ikke komplette.

Noen SAT-problemer er vanskeligere å løse enn andre. Lett problemer kan løses av enhver algoritme, men siden SAT-problemet er NP-komplett vil noen problemer kreve eksponentiell kjøretid. Problemer med få klausuler sammenlignet med antall variabler, kalles underbegrenset og de er som regel lette å løse. Problemer med mange klausuler sammenlignet med antall variabler, vil ha flere begrensninger på variablene og kalles overbegrenset. Slike problemer har som regel ingen løsning.

Agenter basert på proposisjonslogikk

Vi bruker det vi har lært frem til nå for å lage wumpus-verden agenter som bruker proposisjonslogikk for å finne gullet. Dette blir ikke nevnt i forelesning, så bruk det som motivasjon for hvordan proposisjonslogikk kan brukes av en logisk agent for å løse et problem.

Nåværende tilstand til verden

En logisk agent bruker en kunnskapsbase fylt med setninger om verden, for å utlede hva de skal gjøre. KB er fylt med aksiomer, som er generell kunnskap om hvordan verden fungerer og perseptsetninger som agenten oppnår ved å erfare verden. Vi begynner med å samle aksiomene. Agenten vet at det ikke er noen pit eller wumpus i startlokasjonen, dvs. $\neg P_{1,1}$ og $\neg W_{1,1}$. For hver lokasjon vet den at den vil oppdage en *breeze* hvis det er en pit i en nabolokasjon og en *stench* hvis wumpus er i en nabolokasjon. Dette gir en stor samling setninger på formen: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ og $W_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$. Agenten vet også at det er en wumpus: $W_{1,1} \vee W_{1,2} \vee \ldots \vee W_{4,4}$ og at det ikke er flere, slik at for to lokasjon er vil en av de være wumpus-fri: $\neg W_{1,1} \vee \neg W_{1,2}$, $\neg W_{1,2} \vee \neg W_{1,3}$, osv.

Deretter ser vi på perseptene til agenten. **For perseptene blir tidssteget lagt til for å indikere hvilket tidspunkt perseptet ble oppfattet, slik at man unngår motsigelser**. For eksempel hvis agenten oppfatter en stank ved tidssteg 4, vil den legge til $stench^4$ i KB og dette vil ikke motsi $\neg stench^3$ som den oppfattet tidligere. Det samme gjelder *breeze, bump, glitter* og *scream* perseptene. **Dette kan også brukes for andre aspekter som endres over tid (kalles flytende aspekter), for eksempel vil $\boldsymbol{L_{1,1}^0}$ gi at agenten befinner seg i lokasjon [1, 1] ved tiden 0**. Dette kan brukes for å koble perseptene til lokasjonene der de ble oppfattet, for eksempel:

$$L_{x,y}^t \Rightarrow (Breeze^t \Leftrightarrow B_{x,y})$$
$$L_{x,y}^t \Rightarrow (Stench^t \Leftrightarrow W_{x,y})$$

For at agenten skal kunne bestemme hvordan de flytende aspektene ved verden endrer seg ettersom den utfører handlinger, trenger den en overgangsmodell for verden. Dette krever at handlingene kan uttrykkes som proposisjonssymboler. Disse blir også gitt med tidssteg. For eksempel vil $Forward^0$ bety at agenten beveger seg fremover ved tiden 0. For å beskrive hvordan verden endrer seg, kan vi skrive **effekt aksiomer** som gir utfallet ved at en handling utføres. For eksempel vil $L_{1,1}^0 \wedge FacingEast^0 \wedge Forward^0 \Rightarrow (L_{2,1}^1 \wedge \neg L_{1,1}^1)$ beskrive

situasjonen der agenten forlater [1, 1] og entrer [2, 1], ved å bevege seg fremover. Vi vil lage en slik setning for hver mulige tidssteg, i hver av de 16 lokasjonene og for hver av de fire orienteringene. Vi må også ha lignende setninger for *Grab, Shoot, Climb, TurnLeft* og *TurnRight*.

Hvis agenten bestemmer seg for å utføre $Forward^0$ og legger til dette i KB, vil den kunne bruke det effektive aksiomet på forrige side, for å utlede at den nå befinner seg i [2, 1]. Altså, vil $ASK(KB, L_{2,1}^1) = true$. Problemet er at aksiomet ikke gir hva som er uendret som resultat av handlingen. Derfor vil for eksempel $ASK(KB, HaveArrow^1) = false$, altså agenten vet ikke om den fortsatt har pilen. Dette kalles **rammeproblemet**, og det kan løses ved å legge til **rammeaksiomer,** som spesifiserer hvilke proposisjoner som forblir de samme. For eksempel:

$$Forward^t \Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1})$$
$$Forward^t \Rightarrow (WumpusAlice^t \Leftrightarrow WumpusAlive^{t+1})$$

Disse rammeaksiomene vil gjøre at agenten vet at den fortsatt har pilen og at wumpus fortsatt er i livet etter den har beveget seg fremover. Dette vil likevel være en ineffektiv løsning, siden det trengs $O(nm)$ rammeaksiomer dersom det er m ulike handlinger og n flytende aspekter ved verden. Dette kalles **representasjonelt rammeproblem**, og det kan blant annet løses ved å fokusere på å lage aksiomer om flytende aspekter istedenfor handlinger. Dvs. vi bytter ut effekt aksiomer med såkalte **suksessor-state aksiomer,** der sannhetsverdien til et flytende aspekt ved tiden $t + 1$ bestemmes av det flytende aspektet ved tiden t og handlingene som foregår ved denne tiden. For eksempel:

$$HaveArrow^{t+1} \Leftrightarrow (Havearrow^t \wedge \neg Shoot^t)$$

Dersom agenten har et fullstendig sett med suksessor-state aksiomer og de andre aksiomene som gir spillereglene (eks: $\neg P_{1,1}$9, vil den kunne svare på alle ASK-operasjonene om nåværende tilstand til verden. Agenten vil vite hvor den er, om den fortsatt har pilen igjen, om wumpus er i livet, osv.

Hybrid agent

Evnen til å utlede ulike aspekter ved tilstanden til verden kan kombineres med betingelse-handling reglene og problemløsende-algoritmer, for å produsere en **hybridagent** i wumpus verden. **En hybrid agent vil bruke den logiske delen og kunnskapsbasen for å finne ut hvilke lokasjoner som er trygge og som ikke har blitt besøkt enda**. Den problemløsende delen vil lage en plan basert på prioriteten til mål. Hvis den oppfatter glitter, vil den plukke opp gullet og finne en trygg rute tilbake til lokasjon [1, 1]. Hvis ikke vil den **planlegge en trygg rute til nærmeste trygge lokasjon som enda ikke er besøkt, vha A* søk**. Hvis det ikke er noen trygge lokasjoner, må agenten forsøke å skyte wumpus ved å skyte pilen på en mulig wumpus lokasjon. Agenten bruker den logiske delen for å avgjøre hvor wumpus kan befinne seg. **For å finne gullet vil altså agenten bruke både logiske og problemløsende egenskaper.** Agenten kan også bruke logisk inferens for å lage planer, ved å bruke SAT løser som finner modeller som gir fremtidige handlingssekvenser som vil nå målet (= SATPLANER).

Logisk tilstandsestimering

En ulempe med dette agentprogrammet er at kallene til ASK involverer stadig større beregningskostnader ettersom tiden går. Dette skylde at inferensene må gå stadig lengre tilbake i tid og involverer stadig flere proposisjonssymboler. For å løse dette problemet kan man **cache resultatet til inferensen**, slik at inferensprosesser ved neste tidssteg kan bygge på resultatet til tidligere steg istedenfor å starte på nytt. Man kan også bruke **trotilstander** for å representere tidligere historikk og oppdatere disse ettersom nye persepter ankommer (= t**ilstandsestimering**).

Kapittel 8 – Første-orden logikk

I kapittel 7 så vi hvordan en kunnskapsbasert agent kan representere verden vha. proposisjonslogikk og bruke dette for å utlede hvilke handlinger den skal utføre. For komplekse omgivelser vil proposisjonslogikk bli for puslete for å representere kunnskapen på en presis måte. Dette kapitlet introduserer **første-orden logikk** som er mer uttrykksfullt og kan derfor representere en større del av vår kunnskap om verden. Det omfatter eller danner grunnlaget for mange andre representasjonsspråk.

Representasjonsspråk

Programmeringsspråk (eks: Java, C++) er den største klassen med formelle språk som brukes til vanlig. Slike språk bruker datastrukturer i program for å representere fakta, for eksempel kan en 4×4 array brukes for å representere innholdet i wumpus-verden.
Programmeringsspråk mangler likevel en generell mekanisme for å utlede fakta fra andre fakta. Hver oppdatering av en datastruktur blir gjort av en domenespesifikk prosedyre der detaljene stammer fra programmererens kunnskap om domenet. Dette er i kontrast til den deklarative egenskapen til proposisjonslogikk, der kunnskap og inferens er separat, og inferens er uavhengig av domenet. En annen ulempe ved programmeringsspråk er at det er lite uttrykksfullt, for eksempel mangler det en lett måte å si «Det er en pit i [2, 2] og [3, 1]». Dette skyldes at programmene kan bare lagre en enkel verdi for hver variabel.

Fordeler med proposisjonslogikk som representasjonsspråk er:

1. **Det er et deklarativt språk,** siden semantikken er basert på sannhetsrelasjonen mellom setninger og mulige verdener. Dette betyr at proposisjonslogikk gir hva som skal gjøres og ikke hvordan det skal gjøres
2. **Det kan håndtere delvis informasjon** vha negasjon og disjunksjon
3. **Det er kompositorisk**, som vil si at meningen til en setning er bygd opp av meningene til setningens deler

Proposisjonslogikk har likevel begrensninger som gjør at det ikke klarer å presist beskrive omgivelser med mange objekter. For eksempel i wumpus verden må vi skrive mange separate regler på formen «$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$» for å representere at «nabolokasjoner til pits er *breezy*». **Noe kunnskap er vanskelig eller umulig å kode i proposisjonslogikk fordi den representerer påstander om verden uten å reflektere rundt strukturen eller eksplisitt modellere enhetene.** Påstander om objektgrupper, lignende objekter og relasjoner må listes opp. For eksempel hvis vi ønsker å uttrykke at «alle studenter liker ferie», vil det kreve «Per liker ferie» ∧ «Mari liker ferie ∧ «Ola liker ferie»... Dette gjør at KB blir stor.

Naturlig språk (eks: engelsk og norsk) er svært uttrykksfulle. Meningen til en setning kan avhenge av setningen og konteksten der setningen blir uttrykt. Det kan også avhenge av hvilket språk det er (eks: kjønn til ord kan påvirke hva man tenker når man hører ordet). Den samme kunnskapen kan representere på flere ulike måter.

Første-orden logikk = kombinasjon av formelt og naturlig språk

For å bygge en mer uttrykksfull logikk, kan vi låne representasjonsideer fra naturlig språk og bruke grunnlaget ved proposisjonslogikken, altså at det er deklarativ, kompositorisk, kontekst-uavhengig og entydig. Når vi ser på syntaksen til naturlig språk, ser vi at den ofte bruker substantiv for å referere til objekter og verb for å referere til relasjoner blant objekter eller egenskaper. Noen av relasjonene er funksjoner, der det er én verdi for et gitt input. Eksempler er:

- Objekter: personer, hus, tall, farger, osv.

- Relasjoner og egenskaper: større enn, bror til, rød, rund, osv.
- Funksjoner: far til, begynnelse av.., osv.

Nesten alle påstander kan defineres basert på disse. For eksempel for påstanden «Ond kong John styrte England i 1200», vil objektene være John, England og 1200, relasjonene er styrte og egenskaper er ond og konge. **Første-orden logikk er bygget rundt objekter og relasjoner, og det kan derfor brukes for å uttrykke fakta om noen eller alle objektene i den virkelige verden.** For eksempel kan det brukes for å representere generelle regler, slik som «Nabolokasjoner til wumpus lukter».

Vi ser på påstanden: «Hvis lokasjonen er *breezy*, må det være en pit i en nabolokasjon». Proposisjonslogikk krever 16 setninger på formen «$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$» for å representere denne setningen, mens første-orden logikk (FOL) lar oss uttrykke dette med én setning. Påstanden inneholder to objekter (pit og lokasjon), der pit har egenskapen av å være *breezy* og det er en naborelasjon mellom en *breezy* lokasjon og en pit. **I FOL kan denne påstanden gis som: $\forall \mathbf{lokasjon}, \mathbf{nabo}(\mathbf{lokasjon}, \mathbf{pit}) \Rightarrow \mathbf{breezy}(\mathbf{lokasjon})$.**

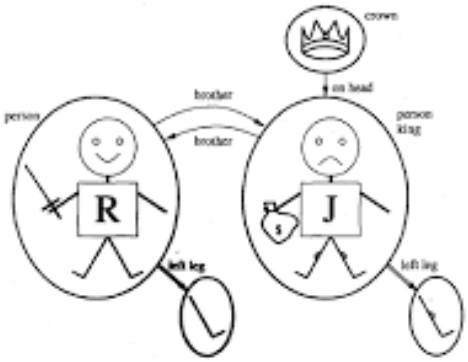
Hovedforskjellen mellom proposisjonslogikk og første-orden logikk (FOL) er at FOL er mer uttrykksfullt, noe som skyldes :

1. **De har ulike antagelser om egenskaper til den virkelige verden**. Proposisjonslogikk antar at det er fakta om verden som vil være sann eller falsk, mens FOL antar at verden består av objekter der bestemte relasjoner vil gjelde eller ikke
2. **FOL bruker variabler for å referere til vilkårlige objekter og kan erstattes av et spesifikt objekt**
3. **FOL bruker kvantifiseringer i påstand om objektgrupper, slik at man slipper å representere hvert objekt separat**

Syntaks og semantikk for første-orden logikk

Syntaksen til et språk er grammatikken som gir hvilke symboler som er tillatt og regler for å kombinere dem. **Semantikken er hvordan setningene relaterer til verden, altså hva de betyr.** Inferensprosedyren brukes for å utlede nye setninger fra de som befinner seg i KB. Vi skal nå se nærmere på disse begrepene for første-orden logikk, men begynner med hvordan det bruker modeller.

Modeller for første-orden logikk



Modeller vil koble setningene til elementer i en mulig verden, slik at man kan bestemme sannheten til setningen. I proposisjonslogikk vil modeller koble proposisjonssymboler til forhåndsbestemte sannhetsverdier. **Modellene til første-orden objekter inneholder objekter, og domenet til en modell er settet av objekter det inneholder**. **Domenet kan ikke være tomt, fordi alle modeller må inneholde minst ett objekt**. Figuren viser en modell med fem objekter: Richard løvehjertet, Kong John, deres venstre føtter og en krone. Her kan vi se at **modellen også vil vise relasjoner mellom to objekter (= binære relasjoner) og egenskaper (= ensartet relasjoner)**. Binære relasjoner kan formelt representeres som tupler, for eksempel kan brorrelasjonen representeres av tuplene $\{\langle Richard, John\rangle, \langle John, Richard\rangle\}$, mens kronen på hodet til Kong John kan representeres av tuplen: $\langle Krone, John\rangle$. Eksempler på egenskaper er at John er konge eller at begge er personer. Noen typer relasjoner vil være funksjoner, ved at et gitt objekt vil være relatert til nøyaktig ett objekt på denne måten. For eksempel vil hver person ha én venstre fot, så modellen har en ensartet «venstre-fot»-funksjon med følgende kartlegging: $\langle Richard\rangle \rightarrow$ Richards venstre fot og $\langle John\rangle \rightarrow$ Johns venstre fot.

Syntaks for første-orden logikk

Syntaksen til et representasjonsspråk er grammatikken som gir hvilke symboler som er tillatt og regler for hvordan de kan kombineres. Vi skal nå se på elementene ved syntaksen til FOL.

Symboler

De grunnleggende elementene i syntaksen til først-orden logikk er symbolene som representerer objekter, relasjoner og funksjoner. Disse symbolene deles inn i:

- **Konstanter** = representerer objekter (eks: $NTNU$ og $KingHarald$)
- **Predikater** = representerer relasjoner, ved å assosiere et sett med objekter med en sannhetsverdi (eks: $HairColor(Per, Brun) = false$ og $Smart(Einstein) = true$)
- **Funksjoner** = representerer funksjoner, ved å kartlegge et eller flere objekter til et unikt objekt i konseptualiseringen av verden (eks: $Mother(Mathilde) \rightarrow Marit$)

Som vi kan se av eksemplene over, blir predikater og funksjoner gitt med «input» som bestemmer antall argumenter. **Modellen vil også inkludere en interpretasjon (tolkning), som gir hvilke objekter, relasjoner og funksjoner som tilhører de ulike symbolene**, for eksempel:

- $Richard$ refererer til Richard løvehjerte, mens $John$ refererer til Kong John
- $Brother$ referer til brorrelasjonen, $OnHead$ refererer til å-hode relasjonen mellom kronen og Kong John. $Person$, $King$ og $Crown$ referer til settet av objekter som er personer, konger og kroner
- $LeftLeg$ refererer til venstre-fot funksjonen

Det er mange mulige tolkninger, for eksempel kan $Richard$ referere til kronen, mens *John* refererer til kong Johns venstre fot. Man trenger ikke å gi navn til alle objektene, for eksempel vil ikke tolkningen gi noe navn til føttene. Et objekt kan også ha flere navn, for eksempel kan både $Richard$ og $John$ referere til kronen. I likhet med proposisjonslogikk blir logisk følge, gyldighet, osv. definert mht. **alle mulige modeller**, som kan inneholde ulikt antall objekter (fra en til uendelig) og har ulike kartlegging mellom konstantsymbolene og objektene. **Siden det er ubegrenset antall mulige modeller, kan man ikke sjekke følger ved opplisting av alle mulige modeller (modellsjekk i proposisjonslogikk)**.

Termer

En term er et logisk uttrykk som refererer til et objekt. Konstantsymboler er derfor termer, men det er ikke alltid nyttig å ha distinkte symboler for hvert objekt. For eksempel bruker vi funksjonen $LeftLeg(John)$ for å referere til den venstre foten til John, istedenfor å navngi denne foten med et konstantsymbol. **Dette viser hvordan en funksjon vil være en term, og den vil kun referere til objektet** (dvs. ikke returnere en verdi, gi en fakta, osv.). En term vil altså bruke funksjoner for å referere til objekter, slik at man slipper å gi et konstantsymbol for hvert objekt. For termen $\text{funksjon}(t_1, \dots, t_n)$ vil argumentene referere til objekter i domenet og hele termen refererer til objektet som er verdien til funksjonen som påføres objektene. For eksempel hvis LeftLeg refererer til funksjonen $\langle John \rangle \rightarrow \text{Johns venstre fot}$ og $John$ refererer til Kong John, så vil termen $LeftLeg(John)$ refererer til venstre fot hos Kong John. **Termer kan være konstanter, funksjoner eller variabler. En variabel kan altså brukes for referere til et objekt i modellen**.

Atomiske setninger

Atomiske setninger som gir fakta kan lages ved å kombinere termer som refererer til objekter og predikatsymboler som refererer til relasjoner. For eksempel vil $Brother(Richard, John)$ gi under interpretasjonen over at Richard løvehjerte er broren til Kong John. Et annet eksempel er $Married\big(Father(Richard), Mother(John)\big)$. **En atomisk setning vil være sann i en gitt modell dersom relasjonen som predikatet refererer til gjelder for objektene som termene refererer til**.

Kompleks setninger – logiske konnektiviteter

Logiske konnektiviteter brukes for å lage komplekse setninger som består av en eller flere atomiske setninger. For eksempel $\neg Brother(LeftLeg(Richard), John)$ og $Brother(Richard, John) \wedge Brother(John, Richard)$.

Kvantifiseringer

Kvantifiseringer brukes for å uttrykke egenskaper ved grupper av objekter, istedenfor å liste opp objektene. Ulike typer kvantifiseringer i første-orden logikk er:

- **Universell kvantifisering (∀)** = brukes for å uttrykke at egenskapen ved en setning gjelder for alle objektene i modellen: ∀⟨variabler⟩ ⟨Setning⟩. For eksempel vil $\forall x\ King(x) \Rightarrow Person(x)$ bety at alle konger er personer (mer nøyaktig: for alle x, hvis x er en konge, så vil x være en person). **∀x P vil være sann i en modell, dersom P er sann hvis x er hver av de mulige objektene i modellen**. I vårt eksempel vil $x = \{Richard\ løvehjerte, Kong\ John, Richards\ venstrefot, Johns\ venstrefot, kronen\}$ Disse gir ulike instanser av $P(x)$, og $\forall x\ P(x)$ er sann hvis konjunksjonen av instansene er sann:

$$King(Richard) \Rightarrow Person(Richard) \wedge$$
$$King(John) \Rightarrow Person(John) \wedge$$
$$King(Richards\ venstre\ fot) \Rightarrow Person(Richards\ venstre\ fot) \wedge$$
$$King(Johns\ venstre\ fot) \Rightarrow Person(Johns\ venstre\ fot) \wedge$$
$$King(kronen) \Rightarrow Person(kronen)$$

P	Q	$P \Rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

I vår modell er det kun Kong John som er konge, og siden han er en person vil andre instans være sann. De andre instansene vil også være sann, fordi premisset er falskt og implikasjonen er derfor sann uansett verdien til konklusjonen (se sannhetstabell). **Å hevde den universale kvantifiseringen vil kun involvere å hevde konklusjonen til objektene der premisset er sant, og det sier ingenting om objektene der premisset er falskt.** Derfor er det naturlig å bruke implikasjon (⇒) som hovedkonnektivitet, og ikke konjunksjon (∧) i setningen $P(x)$.

- **Eksistensiell kvantifisering (∃)** = brukes for å uttrykke at det eksisterer noen objekter i modellen der en bestemt egenskap gjelder: ∃⟨variabler⟩ ⟨Setning⟩. For eksempel vil $P(x)$: $\exists x\ Crown(x) \wedge OnHead(x, John)$ bety at kong John har en krone på hodet sitt (mer nøyaktig: det eksisterer en x, slik at x er en krone og som sitter på hodet til kong John). **∃x P vil være sann i en modell, dersom P er sann for minst ett objekt x i modellen.** Dette krever at disjunksjonen av instansene til $P(x)$ er sann, altså at følgende er sann:

$$Crown(Richard) \wedge OnHead(Richard, John) \vee$$
$$Crown(John) \wedge OnHead(John, John) \vee$$
$$Crown(Richards\ venstre\ fot) \wedge OnHead(Richards\ venstre\ fot, John) \vee$$
$$Crown(Johns\ venstre\ fot) \wedge OnHead(Johns\ venstre\ fot, John) \vee$$
$$Crown(kronen) \wedge OnHead(kronen, John)$$

I vår modell er det kun kronen som vil være en krone og som sitter på hodet til kong John, slik at siste instans er sann. Derfor vil $\exists x\ P(x)$ være sann. **I den universale kvantifiseringen er det naturlig å bruke ⇒ som hovedkonnektivitet, mens i den eksistensielle kvalifiseringen er det naturlig å bruke** ∧. Dette skyldes at implikasjon vil gi en svak påstand, siden den er sann også når premissene er falske.

Obs: det er en vanlig feil å bruke implikasjon (⇒) og konjunksjon (∧) feil i uttrykk med kvantifiseringer. **Tommelfingerregelen at vi bruker ⇒/⇔ som hovedkonnektivitet for ∀ og ∧ som hovedkonnektivitet for ∃.** Hovedgrunnen til dette er at implikasjoner vil være sann når premissene er falske, noe som vil påvirke hva setningen betyr.

For å uttrykke mer komplekse setninger kan vi bruke nøstede kvantifiseringer, for eksempel $\forall x\ \forall y\ Brother(x, y) \Rightarrow Sibling(x, y)$. **Dersom det er samme kvantifisering, kan**

de skrives sammen, for eksempel: $\forall x, y\ Brother(x, y) \Rightarrow Sibling(x, y)$. **Dersom det er ulike kvantifiseringer, vil rekkefølgen være svært viktig**. For eksempel vil $\forall x\ \exists y\ Loves(x, y)$ bety at alle elsker noen, mens $\exists y\ \forall x\ Loves(x, y)$ betyr at det finnes noen som elskes av alle. **Det anbefales å bruke ulike variabler i de nøstede utrykkene, for å unngå forvirring**. For eksempel er det bedre å bruke $\forall x(Crown(x) \vee (\exists z\ Brother(Richard, z)$ istedenfor $\forall x(Crown(x) \vee (\exists x\ Brother(Richard, x))$. **Hvis like variabler er brukt, er regelen at variabelen tilhører den innerste kvantifiseringen.**

Koblinger mellom ∀ og ∃

De to kvantifiseringene er koblet sammen via negasjon. For å hevde at alle liker iskrem, kan man hevde at det ikke eksisterer noen som ikke liker iskrem:

$$\forall x\ Liker(x, Iskrem) \equiv \neg\exists x\ \neg Liker(x, Iskrem)$$

∀ er egentlig en konjunksjon over alle objektene, mens ∃ er en disjunksjon, så derfor vil de følge De Morgans regler:

De Morgan's Rule	Generalized De Morgan's Rule
$P \wedge Q \equiv \neg(\neg P \vee \neg Q)$	$\forall x\ P \equiv \neg\exists x(\neg P)$
$P \vee Q \equiv \neg(\neg P \wedge \neg Q)$	$\exists x\ P \equiv \neg\forall x(\neg P)$
$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$	$\neg\forall x\ P \equiv \exists x(\neg P)$
$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$	$\neg\exists x\ P \equiv \forall x(\neg P)$

Hvis du bringer negasjon inn i en disjunksjon eller konjunksjon, skal disse byttes om. For nøstede kvantifiseringer vil negasjonen gradvis flyttes innover, for eksempel:

$$\neg\big(\forall x \exists y\ P(x, y)\big) \equiv \exists x \neg\exists y\ P(x, y) \equiv \exists x \forall y\ \neg P(x, y)$$

Ekvivalens

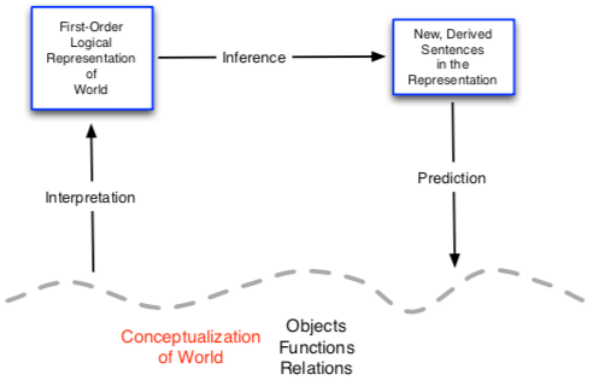
Ekvivalenssymbol brukes for å lage atomiske setninger som gir at to termer refererer til samme objekt. For eksempel vil $Father(John) = Henry$ gi at objektet som refereres til av $Father(John)$ er det samme objektet som $Henry$ refererer til. For å bestemme sannheten til en ekvivalens, må vi se om de to termene refererer til samme objekt. Det kan også brukes med negasjon for å gi at to termer refererer til ulike objekter, for eksempel:

$$\exists x, y\ \ Brother(x, Richard) \wedge Brother(y, John) \wedge \neg(x = y)$$

Der siste ledd er nødvendig for å spesifisere at x og y ikke er samme objekt. Man kan bruke notasjonen $x \neq y$.

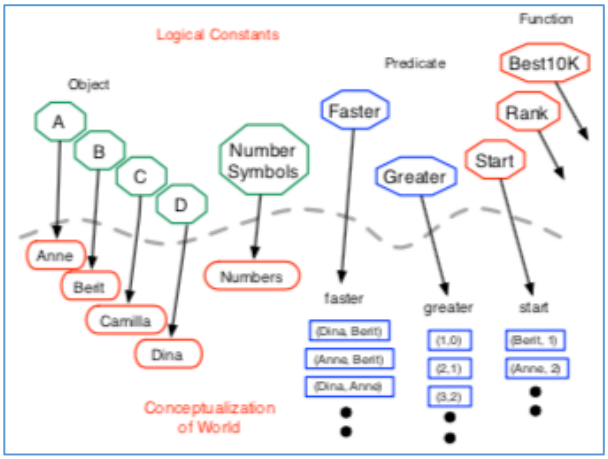
Semantikk ved første-orden logikk

Semantikken til et representasjonsspråk er hvordan setningene relaterer til verden, altså hva de betyr. **I første-orden logikk blir en interpretasjon (tolkning) I brukt for å definere kartleggingen av symboler (eks: konstanter, predikater) til objektene, relasjonene og funksjonene i modellen**. Vi sier at den kartlegger symbolene til diskursens domene (D), som er settet av objektene i verden vi representerer og refererer til (dvs. konseptualiseringen av verden). Interpretasjonen vil koble konstantsymboler til objekter i D, predikatsymboler til relasjoner som er egenskaper på D og funksjonssymboler til funksjoner på D.

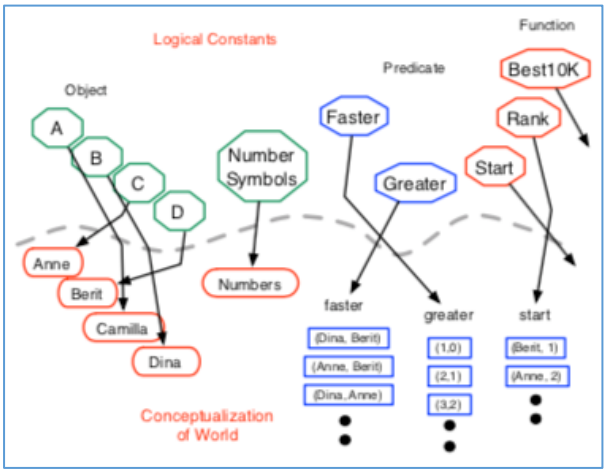


Konseptene innenfor semantikken får følgende betydning for FOL:

- **Modell** = modellen hos en setning er en interpretasjon som gjør at setningen blir sann
- **Gyldighet** = setningen er sann ved alle interpretasjoner
- **Satisfiable** = setningen er sann ved minst én interpretasjon
- **Unsatisfiable** = setningen er sann ved ingen interpretasjoner
- **Følger** = setningen er sann ved alle interpretasjoner der KB er sann

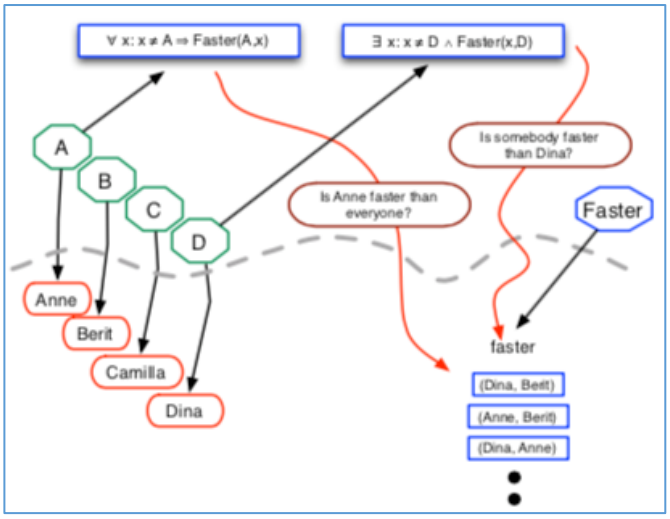


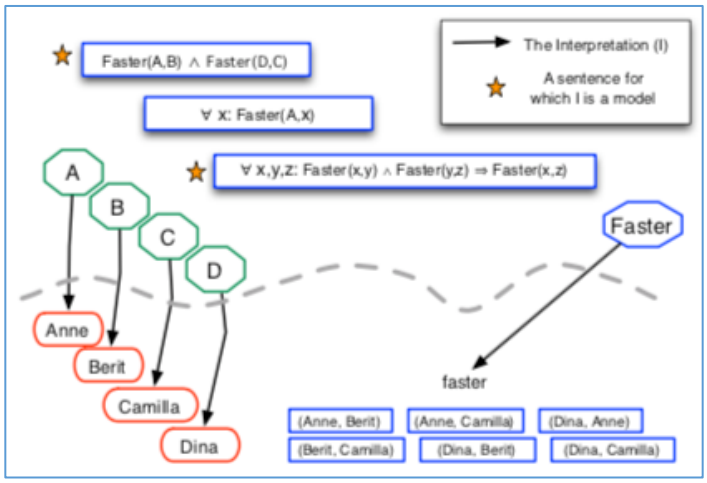
Figuren til venstre viser et eksempel på en interpretasjon som kartlegger fra konstant-, funksjon- og predikatsymboler til representasjoner i konseptualiseringen av verden. For eksempel ser vi at konstantsymbolet A kartlegges til objektet $Anne$ i konseptualiseringen, og perseptsymbolet $Faster$ kartlegges til relasjonene $(Dina, Berit), (Anna, Berit), (Dina, Anne)$, osv.

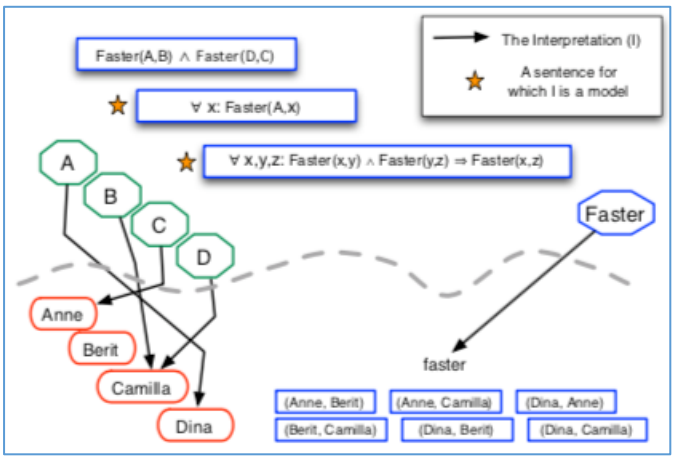


Figuren vil høyre viser en annen interpretasjon som har en annen kartlegging av symbolene. Her vil konstantsymbolet A kartlegges til objektet $Camilla$ i konseptualiseringen,

Figurene under viser hvordan evalueringen av setninger vil avhenge av interpretasjonen.







På figuren over til høyre ser vi at interpretasjonen vil være en model for setningen $Faster(A, B) \wedge Faster(D, C)$ fordi denne setningen er sann ved denne kartleggingen. Interpretasjonen til venstre vil ikke være en modell for denne setningen, fordi her er kartleggingen slik at setningen blir falsk.

En alternativ semantikk

Ved **database-semantikk** vil alle konstantsymboler referere til distinkte objekter (= unik-navn antagelse), alle atomiske setninger er falske hvis det ikke er kjent om de er sanne (= lukket-verden antagelse) og hver modell inneholder ikke flere elementer enn det som er gitt av konstantsymbolene (= lukket domene). Denne semantikken har noen fordeler, for eksempel vil $Brother(John, Richard) \wedge Brother(Geoffry, Richard)$ gi at Richard har to brødre. Det vil ikke være tilfellet for FOL semantikk, fordi den vil ikke vite om $John$ og $Geoffry$ refererer til samme eller ulike objekter og den vet ikke om Richard har flere brødre. Database-semantikken er populær i databasesystemer.

Bruk av første-orden logikk

Vi skal se hvordan første-orden logikk blir brukt i noen enkle domener, dvs. deler av verden som vi ønsker å uttrykke kunnskap om.

TELL og ASK i første-orden logikk

TELL-operasjoner brukes for å legge til setninger til kunnskapsbasen, og disse setningene kalles påstander. For eksempel kan vi påstå at John er kongen, at Richard er en person og at alle konger er personer:

$$TELL(KB, King(John))$$
$$TELL(KB, Person(Richard))$$
$$TELL(KB, \forall x\ King(x) \Rightarrow Person(x))$$

ASK-operasjoner brukes for å spørre kunnskapsbasen om noe (query). For eksempel vil følgene returnere true:

$$ASK(KB, King(John))$$

Enhver query som logisk følger av KB vil få et bekreftende svar, for eksempel vil $ASK(KB, Person(John))$ returnere true på grunn av første og siste TELL-operasjon. For $ASK(KB, \exists x\ Person(x))$ vil det ikke gis hvilke verdier av x som gir true, men heller at det finnes en x som gir true. For å returnere verdiene kan vi bruke $ASKVARS(KB, Person(x))$, som vil returnere $\{x/John\}$ og $\{x/Richard\}$.

Slektskapsdomenet

Vi ser på domenet for slektskap, der objektene er mennesker, egenskaper (ensartet relasjon) er $Male$ og $Female$, binære relasjoner er $Parent, Sibling, Brother, Sister$, osv. Vi bruker funksjoner for $Mother$ og $Father$, siden alle personer har nøyaktig en av hver. Eksempler på setninger for dette domenet er:

- Mor er kvinnelig forelder: $\forall m, c\ Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m, c)$
- Ektemann er mannlig partner: $\forall w, h\ Husband(h, w) \Leftrightarrow Male(h) \wedge Partner(h, w)$
- Mann og kvinne er ulike kategorier: $\forall x\ Male(x) \Leftrightarrow \neg Female(x)$
- Foreldre og barn er inverse relasjoner: $\forall p, c\ Parent(p, c) \Leftrightarrow Child(c, p)$
- Besteforeldre er foreldre til foreldrene: $\forall g, c\ Grandparent(g, c) \Leftrightarrow \exists p\ Parent\ (g, p) \wedge Parent(p, c)$
- En søsken er et annet barn til forelderen: $\forall x, y\ Sibling(x, y) \Leftrightarrow x \neq y \wedge \exists p\ Parent(p, x) \wedge Parent(p, y)$
- Osv.

Hver av disse setningene kan ses på som et aksiom for slektskapsdomenet. De gir grunnleggende informasjon som kan brukes for å utlede konklusjoner. Disse aksiomene er **definisjoner**, siden de har formen $\forall x, y\ P(x, y) \Leftrightarrow \cdots$. Aksiomene over definerer $Mother$

funksjonen og predikatene $Husband, Male, Parent, Grandparent$ og $Sibling$ mht. andre predikater. Noen logiske setninger om domener vil være **teorem**, for eksempel at søskenskap er symmetrisk: $\forall x, y\ Sibling(x, y) \Leftrightarrow Sibling(y, x)$. Dette teoremet vil følge av aksiomene som definerer søskenskap. Alle aksiomene vil ikke være definisjoner, for eksempel kan det hende vi ikke vet nok om predikatet for å lage en fullstendig definisjon (eks: $Person$). I dette tilfellet vil $\forall x\ Person(x) \Leftrightarrow \cdots$ brukes for å spesifisere enkelte egenskaper ved predikatet. Aksiomer kan også være enkle fakta, for eksempel $Male(Jim)$ og $Partner(Jim, Laura)$. Disse gjør at spesifikke spørsmål kan bli svart.

Wumpus verden

Vi skal nå se hvordan wumpus verden kan beskrives vha første-orden logikk.

Input og output for agenten

Input til Wumpus agenten er en perseptvektor med fem elementer (s. 66). Den korresponderende setningen som lagres i KB må inkludere setningen og tiden den ble oppfattet (dvs. tidssteg = hindrer misforståelser). En perseptsetning kan være:

$$Percept([Stench, Breeze, Glitter, None, None], 5)$$

Her vil $Percept$ være en binær predikat, mens argumentene er konstanter. Output til wumpus agenten vil være handlingene, og de kan representeres som logiske termer: $Turn(Right)$, $Turn(Left)$, $Forward$, $Shoot$, $Grab$ og $Climb$. For å bestemme hvilken handling som er best vil programmet utføre:

$$ASKVARS(\exists a\ BestAction(a, 5))$$

Denne vil returnere for eksempel $\{a/Grab\}$, slik at programmet kan returnere at handlingen $Grab$ skal utføres. Perseptdataen vil gi bestemte fakta om nåværende tilstand, for eksempel vil:

$$\forall t, s, g, m, c\ Percept([s, Breeze, g, m, c], t) \Rightarrow Breeze(t)$$
$$\forall t, s, b, m, c\ Percept([s, b, Glitter, m, c], t) \Rightarrow Glitter(t)$$
$$\ldots$$

Legg merke til kvantifiseringen av tiden, som gjør at vi slipper å lage en setning for hvert tidssteg, slik vi må i proposisjonslogikk. Kvantifiserte implikasjonssetninger kan brukes for å lage enkel refleksoppførsel som kobler persepter direkte til handlinger, for eksempel:

$$\forall t\ Glitter(t) \Rightarrow BestAction(Grab, t)$$

Omgivelsene

For å representere lokasjonene, pits og wumpus i omgivelsen kan vi bruke objekter, men vi skal se at for lokasjonene og pits finnes det bedre alternativ. For å representere lokasjonene kan vi bruke termer, der raden og kolonnen er gitt som heltall (eks: [1, 2]). Dette gjør at det blir enklere å definere nabolokasjoner:

$$\forall x, y, a, b\ Adjacent([x, y], [a, b]) \Leftrightarrow (x = a \wedge (y = b - 1 \vee\ y = b + 1) \vee$$
$$(y = b \wedge (x = a - 1 \vee x = a + 1)$$

Pits blir representert som egenskapen Pit, som er sann for lokasjoner som inneholder pits. Dette er enklere enn å navngi alle pits. Siden det er nøyaktig én wumpus kan denne representeres som konstantsymbol $Wumpus$.

Lokasjonen til agenten endres over tid, så vi bruker $At(Agent, s, t)$ for å si at agenten er ved lokasjon s ved tiden t. Lokasjonen til wumpus er konstant: $\forall t\ At(wumpus, [2,2], t)$. Vi kan si at objekter bare kan være ved en lokasjon om gangen:

$$\forall x, s_1, s_2, t\ At(x, s_1, t) \wedge At(x, s_2, t) \Rightarrow s_1 = s_2$$

Agenten kan bruke persepter for å bestemme egenskaper ved lokasjoner. For eksempel hvis agenten er i en lokasjon og oppfatter en *breeze* vil denne lokasjonen være *breezy*:

$$\forall s, t\ At(Agent, s, t) \wedge Breeze(t) \Rightarrow Breezy(s)$$

Det er nyttig å vite hvilken lokasjon som er *breezy* og hvilken som stanker, fordi agenten vil bruke dette for å utlede hvor det er pits og hvor wumpus befinner seg. Dette brukes for å finne ut hvilke lokasjoner som er trygge å bevege seg inn i. For å gi at nabolokasjoner kan inneholde pits, vil proposisjonslogikk kreve en aksiom per nabolokasjon, mens første-orden logikk trenger kun ett aksiom:

$$\forall s\ Breezy(s) \Leftrightarrow \exists r\ Adjacent(r, s) \wedge Pit(r)$$

Kvantifiseringen over tid og lokasjon gjør altså at man kan beskrive påstander med færre setninger i første-orden logikk, enn det som er mulig i proposisjonslogikk.

Kunnskapsingeniør i første-orden logikk

Kunnskapsingeniør er en prosess der man lager en kunnskapsbase ved å undersøke et bestemt domene, lære konseptene som er viktig og lager en formell representasjon av objekter og relasjoner i domenet. Denne tilnærmingen vil utvikle KB med spesielt formål der domenet er avgrenset og typer queries er kjent på forhånd. Generelt-formål KB blir diskutert i kapittel 12. Dette er gitt med et overblikk, siden det ikke nevnes i forelesning ☺

Kunnskapsingeniør

For å lage en kunnskapsbase bruker man følgende steg:

1. **Identifiser oppgaven** = avgrens rekkevidden til spørsmål som KB skal støtte og type fakta som skal være tilgjengelig. Bestem hvilken kunnskap som må være representert for å koble probleminstanser til svar.
2. **Samle relevant kunnskap** = forstå omfanget til KB og hvordan domenet fungerer
3. **Bestem vokabular for predikater, funksjoner og konstanter** = oversett viktige domenenivå konsepter til logisknivå navn (eks: Pits representeres som egenskaper)
4. **Kod generell kunnskap om domenet** = skriv aksiomer for alle termene, slik at deres betydning blir bestemt. Dette trengs for å støtte inferensprosedyren.
5. **Kod en beskrivelse av den spesifikke probleminstansen** = skriv enkle atomiske setninger om konseptinstanser.
6. **Send queries til inferensprosedyren og få svar** = inferensprosedyren vil operere på aksiomene og de problemspesifikke faktaene for å utlede fakta vi er interessert i.
7. **Debugging av KB** = svarene til queries er sjeldent riktig ved første forsøk, så dette må fikses. Svarene er riktig for kunnskapsbasen slik den er skrevet, men ikke i forhold til det brukeren forventer. For eksempel hvis det mangler et aksiom, vil det ikke gis svar på bestemte queries.

Kapittel 9 – Inferens i første-orden logikk

I kapittel 7 så vi hvordan solid og komplett inferens kan oppnås for proposisjonslogikk. I dette kapittelet skal vi se hvordan dette kan utvides for å oppnå algoritmer som kan svare på spørsmål gitt i første-orden logikk. I dette kapittelet blir begrepet proposisjon og proposisjon

Proposisjon vs. første-orden logikk

Noen enkle inferensregler kan brukes for å fjerne kvantifiseringen fra setninger. Disse reglene gjør at første-orden inferens kan utføres ved å omforme KB til proposisjonslogikk og dermed bruke proposisjonell inferens, slik vi så i kapittel 7. Senere skal vi se at det finnes inferensmetoder som direkte manipulerer første-orden setninger.

Proposisjonalisering – universell instansiering (UI)

Vi begynner med universale kvantifiseringer. Kunnskapsbasen inneholder følgende aksiom:

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil\ (x)$$

For å omforme dette til proposisjonslogikk, må vi bli kvitt kvantifiseringen og variablene. **Regelen for Universell instansiering (UI) gir at vi kan utlede setninger ved å erstatte variablene med *ground* termer som ikke inneholder variabler. Dette gjør at vi kan fjerne ∀.** Dersom $SUBT(\theta, \alpha)$ er resultatet av å bruke substitusjon θ på setningen α vil definisjonen av universell instansiering (UI) være:

$$\frac{\forall x\ \ \alpha}{SUBST(\{x/g\},\ \ \alpha)}$$

For alle variabler x i setningen α vil x byttes ut med ground term g. For setningen over får vi følgende setninger:

$$King(John) \wedge Greedy(John) \Rightarrow Evil\ (John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil\ (Richard)$$
$$King\big(Father(John)\big) \wedge Greedy\big(Father(John)\big) \Rightarrow Evil\ \big(Father(John)\big)$$
$$\dots$$

Her er substitusjonene $\{x/John\}$, $\{x/Richard\}$ og $\{x/Father(John)\}$.

Proposisjonalisering – eksistensiell instansiering (EI)

Vi ser deretter på eksistensiell kvantifiseringer. Kunnskapsbasen inneholder følgende aksiom:

$$\exists x\ Crown(x) \wedge OnHead(x, John)$$

Her må vi også fjerne kvantifiseringen og variablene for å omforme til proposisjonslogikk. **Ved eksistensiell instansiering (EI), blir variabelen erstattet med en ny og unik konstant, så lenge variabelen og konstanten ikke brukes andre plasser i KB.** Definisjonen av eksistensiell instansiering er:

$$\frac{\exists x\ \ \alpha}{SUBST(\{x/k\},\ \ \alpha)}$$

Variablene x i setningen α byttes ut med et konstantsymbol k som ikke fremkommer andre plasser i kunnskapsbasen. For setningen over får vi følgende setning:

$$Crown(C_1) \wedge OnHead(C_1, John), \quad C_1 \notin KB$$

Den nye konstanten kalles **Skolem konstant**. Eksistensiell instansiering er et spesielt tilfelle i en mer generell prosess som kalles **skolemisering** (mer senere).

Universell instansiering kan brukes mange ganger for å produsere ulike konsekvenser, mens eksistensiell instansiering kan bare brukes én gang. Når $\exists x\ Kill(x, Victim)$ er erstattet med $Kill(Murder, Victim)$ trenger vi ikke den opprinnelige setningen lenger, så denne fjernes fra KB. **Den nye kunnskapsbasen vil ikke være ekvivalent med den gamle, men den er inferensekvivalent, noe som betyr at den er satisfiable der den originale KB er satisfiable.**

Proposisjonalisering = reduksjon til proposisjonsinferens

Universell og eksistensiell instansiering gjør det mulig å redusere første-orden inferens til proposisjonsinferens. **En eksistensiell kvantifisert setning kan erstattes av en instansiering, mens en universell kvantifisert setning kan erstattes med settet av alle mulige instansieringer.** For eksempel hvis KB inneholder følgende setninger:

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil\ (x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John)$$

Vil bruken av UI på første setning bruke alle mulige ground-term substitusjoner fra vokabularet til KB. I dette tilfellet $\{x/John\}$ og $\{x/Richard\}$. Vi får:

$$King(John) \wedge Greedy(John) \Rightarrow Evil\ (John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil\ (Richard)$$

Den universelle kvantifiseringen kan dermed droppes. Dersom vi ser på de atomiske setningene, $King(John), Greedy(John)$ og $Brother(Richard, John)$ som proposisjonssymboler, vil kunnskapsbasen nå være i proposisjonslogikk. Dermed kan vi bruke proposisjonsinferensen fra kapittel 7 for å finne konklusjonen: $Evil(John)$.

Denne teknikken kalles **proposisjonalisering**, og vi skal senere se at den kan generaliseres slik at alle første-orden KB og query kan proposisjonaliseres slik at følger er bevart.
Problemer ved proposisjonalisering er:

1. **Når kunnskapsbasen inneholder funksjoner, kan det produseres uendelig mange ground-term substitusjoner**. For eksempel hvis KB inneholder $Father$, kan det lages uendelige mange termer slik som $Father\left(Father\left(Father(John)\right)\right)$. Løsningen til dette er å bruke ulike dybder, helt til vi får laget et proposisjonelt bevis til den logiske følgen (eks: først konstanter ($Richard$ og $John$), deretter dybde 1 ($Father(Richard)$ og $Father(John)$), deretter dybde 2, osv. Dette kalles Herbrands teorem.
2. **Proposisjonalisering vil generere mange irrelevante setninger**. For eksempel for KB over vil proposisjonalisering produsere fakta som $Greedy(Richard)$ som er irrelevant for å utlede query $Evil(John)$. For p predikater som er k-ære og n konstaner vil det være pn^k instansieringer (eks: binær predikat gir $k = 2$).

Første-orden inferens via proposisjonalisering er komplett, altså vil den kunne vise alle setninger som følger av KB. Den vil likevel ikke kunne vise at en setning ikke følger av KB, fordi da vil beviset generere stadig dypere nøstede termer, og algoritmen vil ikke vite om den sitter fast i en håpløs loop eller om beviset vil komme i neste iterasjon. **Følger for første orden-logikk er halvveis-avgjørbart, siden det kan bekrefte alle setninger som følger av KB, men ikke avkrefte setninger som ikke følger av KB**.

Unifikasjon og løfting

Proposisjonalisering er ineffektivt, siden det genererer mange irrelevante setninger. For eksempel hvis man har query $Evil(John)$? og kunnskapsbasen på forrige side, er det unødvendig å generere setninger slik som $King(Richard) \wedge Greedy(Richard) \Rightarrow Evil\ (Richard)$. Utledning av $Evil(John)$ er åpenbar fra setningene:

$$King(John) \wedge Greedy(John) \Rightarrow Evil\ (John)$$
$$King(John)$$
$$Greedy(John)$$

Vi skal se hvordan dette kan gjøres åpenbart for datamaskiner.

Første-orden inferensregel – løftet versjon av modus ponens

Vi skal se hvordan man kan utlede at John er ond ved å bruke $\{x/John\}$ for å løse query $Evil(x)$. For å bruke regelen av grådige konger er onde, finn en x, slik at x er en konge og x er grådig, for å utlede (*infer*) at x er ond. **Hvis det finnes en substitusjon $\boldsymbol{\theta}$ som gjør at hver konjunkt i premisset til implikasjonen blir identisk med setninger som er i KB, kan vi hevde konklusjonen til implikasjonen etter å ha brukt $\boldsymbol{\theta}$.** I vårt eksempel vil $\theta = \{x/John\}$.

Inferenssteget kan gjøre mer. For eksempel hvis vi vet at alle er grådige, dvs. $\forall y\ Greedy(y)$, vil problemet være at kunnskapsbasen inneholder $Greedy(y)$, mens implikasjonen inneholder $Greedy(John)$. Vi vil fortsatt kunne konkludere med at $Evil(John)$, fordi John er en konge og John er grådig (siden alle er grådige). For at dette skal fungere må vi finne en substitusjon for variablene i implikasjonen og for variablene i setningene som allerede er i kunnskapsbasen. I vårt tilfelle vil bruken av substitusjonen $\{x/John, y/John\}$ gjøre at $King(x)$ og $Greedy(x)$ i implikasjonen og $King(John)$ og $Greedy(y)$ i kunnskapsbasen blir identiske. Dermed kan vi utlede konklusjonen til implikasjonen.

Denne inferensprosedyren bruker en **generalisert modus ponens**. For atomiske setninger p_i, p_i' og q, der det er en substitusjon θ, slik at $SUBST(\theta, p_i') = SUBST(\theta, p_i)$, for alle i, vil:

$$\frac{p_1', p_2', \ldots, p_n', \quad (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{SUBST(\theta, q)}$$

Dvs. hvis variablene i konjunktene til premisset blir erstattet med $\boldsymbol{\theta}$ vil de bli identiske med de atomiske setningene i KB, slik at konklusjonen $\boldsymbol{q}$ vil gjelde. I vårt eksempel vil:

$$\frac{King(John), Greedy(y) \quad King(x) \wedge Greedy(x) \rightarrow Evil(x)}{Evil(John)}$$

Dersom KB inneholder flere atomiske setninger og en implikasjon der premissene inneholder substitusjoner av de atomiske setningene, kan vi utføre substitusjonen for å vise at konklusjonen vil gjelde. Se på premissene til implikasjonen; kan variablene substitueres slik at de blir lik atomiske setninger som allerede er i KB?

Den generaliserte modus ponens er solid ($SUBST(\theta, q)$ følger av modus ponens, s. 326). **Det er en løftet versjon av modus ponens, siden den løfter modus ponens fra ground (variabel-fri) proposisjonslogikk til første-orden logikk. Fordelen ved løftet inferensregler sammenlignet med proposisjonalisering, er det vil kun utføre substitusjoner som kreves for å tillate den bestemte inferensen.**

Unifikasjon

Løftet inferensregler krever at man finner substitusjoner som gjør at ulike logiske uttrykk blir identiske. Denne prosessen kalles **unifikasjon** og det er en essensiell del av alle første-orden inferensalgoritmer. **UNIFY algoritmen tar inn to setninger og returnerer en *unifier* (forener) hvis den eksisterer:**

$$UNIFY(p, q) = \theta, \quad \text{der } SUBST(\theta, p) = SUBST(\theta, q)$$

***UNIFY* vil altså gi substitusjonen θ som kan erstatte variablene i p og q setningene med verdier som gjør at disse setningene blir like (dersom det er mulig). Ved unifikasjon kan variablene erstattes med en konstant, en annen variabel og en funksjon som ikke inneholder variabelen.** Vi ser på et eksempel der vi har query $ASKVARS\big(Knows(John, x)\big)$ som vil returnere objektene som John kjenner. For å finne svaret må vi finne alle setningene i KB som kan forenes med $Knows(John, x)$. Her er resultatet for fire ulike setninger som kan være i KB:

$$UNIFY\big(Knows(John, x), Knows(John, Jane)\big) = \{x/Jane\}$$
$$UNIFY\big(Knows(John, x), Knows(y, Bill)\big) = \{x/Bill, y/John\}$$
$$UNIFY\big(Knows(John, x), Knows(y, Mother(y))\big) = \{y/John, x/Mother(John)\}$$
$$UNIFY\big(Knows(John, x), Knows(x, Elisabeth)\big) = fail$$

Legg merke til at den siste unifikasjonen over feiler fordi x kan ikke være John og Elisabeth samtidig. $Knows(x, Elisabeth)$ betyr at alle kjenner Elisabeth, så vi bør kunne utlede at John kjenner Elisabeth. Problemet skyldes at de to setningene bruker samme variabelnavn, x. **Dette problemet kan løses ved å standardisere fra-hverandre en av de to setningene som blir forenet, som vil si å endre navnet til variablene for å unngå navn-krasj**. For eksempel kan vi se på:

$$UNIFY\big(Knows(John, x), Knows(x_{17}, Elisabeth)\big) = \{x/Elisabeth, x_{17}/John\}$$

Dette illustrerer hvorfor det er viktig at variabler i setninger har ulike navn!

I noen tilfeller kan *UNIFY*-algoritmen gi flere *unifiers*. For eksempel vil $UNIFY\big(Knows(John, x), Knows(y, z)\big)$ kunne returnere $\{y/John, x/z\}$ eller $\{y/John, x/John, z/John\}$. Den første gir $Knows(John, z)$, mens den andre gir $Knows(John, John)$. Det andre resultatet kan oppnås ved å bruke substitusjonen $\{z/John\}$. Vi sier at første unifier er mer **generell** enn den andre, siden den gir færre restriksjoner på verdiene til variablene. **Alle forente uttrykkspar vil ha en mest-generell unifier (MGU)**. Legg merke til at $\{x/John\}$ er ekvivalent med $\{y/John\}$, og $\{x/John, y/x\}$ er ekvivalent med $\{x/John, y/John\}$.

Figuren viser algoritmen som rekursivt regner ut MGUs. Den undersøker de to uttrykkene samtidig og bygger opp en *unifier* underveis. Hvis to korresponderende punkter ikke matcher, vil algoritmen feile. **Unifikasjonen vil også feile dersom den forsøker å forene en variabel med en kompleks term som inneholder variabelen**. For eksempel $S(x)$ kan ikke forenes med $S\big(S(x)\big)$.

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)
  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure

function UNIFY-VAR(var, x, θ) returns a substitution
  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

If we have failed or succeeded, then fail or succeed.

Lagring og henting

$TELL$- og ASK-funksjonene brukes for å legge til nye setninger eller hente eksisterende fra kunnskapsbasen, og grunnlaget for disse er $STORE$- og $FETCH$-funksjonene. **$\boldsymbol{STORE(s)}$ vil lagre setning $\boldsymbol{s}$ i kunnskapsbasen, mens $\boldsymbol{FETCH(q)}$ vil returnere alle *unifiers* som gjør at query $\boldsymbol{q}$ blir forenet med en setning i KB**. Problemet av å finne alle faktaene som forenes med $Knows(John, x)$ er en instans av $FETCH$. Den enkleste måten å implementere $STORE$ og $FETCH$ er å holde alle faktaene i en lang liste og deretter forene hver query med alle elementer i denne listen. Denne prosessen er ineffektiv, men den fungerer. For å gjøre prosessen mer effektiv kan man bruke **predikat indeksering,** som indekserer faktaene i kunnskapsbasen for å sikre at unifikasjonen kun blir forsøkt på setningene som har en større sjanse for å forenes. For mer detaljer se side 328-329 (ikke nevnt i forelesning) ☺

Forward chaining

Ideen ved forward chaining er å starte med de atomiske setningene i kunnskapsbasen og deretter bruke Modus Ponens i fremover retning, slik at det legges til nye atomiske setninger helt til det ikke kan utføres videre inferens. Vi skal se hvordan dette gjøres for første-orden logikk.

Første-orden definite klausuler

Første-orden definite klausuler ligner definite klausuler for proposisjonslogikk, som er disjunksjoner av literaler, der nøyaktig én er positiv. **En første-orden klausul er enten en positiv atomisk setning eller en implikasjon der premisset er en konjunksjon av positive literaler og konklusjonen er en enkel positiv literal. Første-orden literaler kan inkludere variabler, og disse antas å være universelt kvantifisert (∀ blir som regel utelatt).** Eksempler er:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(y)$$

Gitt et problem i naturlig språk, må vi først oversette dette til setninger i første-orden logikk. Deretter må vi omforme disse setningene til første-orden definite klausuler ved å fjerne ∀ og eliminere ∃ vha eksistensiell instansiering. Vi ser på et eksempel, der problemet er som følger: «Loven sier at det er kriminelt for en Amerikaner å selge våpen til fiendtlige nasjoner. Landet Nono, som er en fiende av Amerika, har noen missiler, og alle disse missilene er kjøpt fra Colonel West, som er Amerikansk». Vi ønsker å vise at West er en kriminell. Tabellen under viser hvordan dette kan oversettes til første-orden klausuler:

	Naturlig språk	Første-orden setning	Første-orden definite klausul
S1	«Det er kriminelt for en amerikaner å selve våpen til en fiendtlig nasjon»:	$\forall x, y, z\ American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$	$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$
S2	«Nono… har noen missiler»:	$\exists x\ Missile(x) \wedge Owns(Nono, x)$	$Owns(Nono, M_1)$ $Missile(M_1)$
S3	«Alle disse missilene er kjøpt fra Colonel West»	$\forall x\ Owns(Nono, x) \wedge Missile(x) \Rightarrow Sells(West, x, Nono)$	$Owns(Nono, x) \wedge Missile(x) \Rightarrow Sells(West, x, Nono)$
S4	Missiler er våpen	$\forall x\ Missile(x) \Rightarrow Weapon(x)$	$Missile(x) \Rightarrow Weapon(x)$
S5	En fiende av Amerika er «hostile»:	$\forall x\ Enemy(x, America) \Rightarrow Hostile(x)$	$Enemy(x, America) \Rightarrow Hostile(x)$
S6	«West, som er Amerikansk»:	$American(West)$	$American(West)$
S7	«Landet Nono, som er en fiende av Amerika»:	$Enemy(Nono, America)$	$Enemy(Nono, America)$

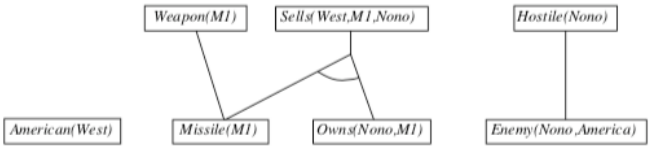
Enkel forward-chaining algoritme

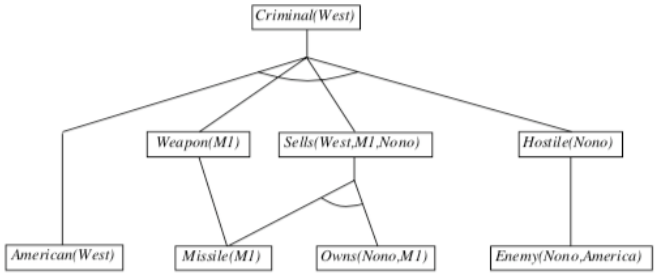
```
function FOL-FC-ASK(KB, α) returns a substitution or false
  repeat until new is empty
    new ← ∅
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⟹ q) ← STANDARDIZE-APART(r)
      for each θ such that (p1 ∧ ... ∧ pn)θ = (p'1 ∧ ... ∧ p'n)θ
                for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of a sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

Figuren viser en enkel algoritme for forward-chaining. Den starter fra kjent fakta, og for alle implikasjonene der premissene er oppfylt vil den legge til konklusjonen i KB. Prosessen gjentas helt til query blir svart (dvs. den er en konklusjon som utledes) eller til det ikke blir lagt til flere fakta. Legg merke til at faktaen ikke vil være ny dersom det kun er en gammel fakta der variablene har fått nye navn. For eksempel vil $Likes(x, IceCream)$ og $Likes(y, IceCream)$ være identiske, fordi de betyr det samme selv om variabelnavnet er byttet ut.

Vi bruker eksempelet på forrige side for å vise hvordan algoritmen fungerer. Implikasjonene er S1, S3, S4 og S5, og query er $Criminal(West)$? Algoritmen vil bruke to iterasjoner for å finne svaret:

1. For implikasjonene S3, S4 og S5 er premissene gitt av de andre setningene i KB:
   a. S3 er tilfredsstilt av $\theta = \{x/M_1\}$, så $Sells(West, M_1, Nono\}$ legges til KB
   b. S4 er tilfredsstilt av $\theta = \{x/M_1\}$, så $Weapon(M_!)$ legges til KB
   c. S5 er tilfredsstilt av $\theta = \{x/Nono\}$, så $Hostile(Nono)$ legges til KB
2. For implikasjonen S1 er premissene gitt av de andre setningene i KB:
   a. S1 er tilfredsstilt av $\theta = \{x/West, y/M_1, z/Nono\}$, så $Criminal(West)$ legges til KB. Dermed har algoritmen svart på query.





AND-OR **bevistreet** blir laget ved at algoritmen starter med de kjente faktaen i funnen og inferensen forplanter seg oppover så langt som mulig. Ved figuren til høyre er det ikke mulig å utføre videre inferens (utledning), fordi KB vil ved dette tidspunktet inneholde alle setninger som kan konkluderes ved forward chaining. En slik kunnskapsbase kalles et **fastpunkt** for inferensprosessen

Algoritmen er solid, fordi hver inferens er basert på generalisert modus ponens, som er solid. **Den er komplett** for definite klausuler, altså vil den svare på alle queries dersom de følger av KB. Hvis klausulene inneholder funksjoner kan algoritmen generere uendelig mange nye fakta, noe som løses ved å bruke ulike dybder (= Herbrands teorem). Hvis query ikke har noe svar kan algoritmen entre en uendelig lopp. For første-orden logikk vil følger med definite klausuler være halvveis-avgjørbart.

Effektiv forward chaining

For å gjøre forward chaining mer effektiv, har man følgende tiltak:

1. **Matching av regler mot kjente fakta** = en kilde til ineffektivitet i algoritmen er at den vil finne alle mulige *unifiers* som vil forene premissene med fakta i KB. Dette kalles mønster matching og kan være svært dyrt. For eksempel for $Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$ kan det hende at KB inneholder mange objekter som er eid av Nono og veldig få Missiler, slik at det er best å finne alle

missilene først og deretter sjekke om de er eid av Nono. Dette kalles **konjunksjon rekkefølgeproblemet,** der man løser konjunktene i en rekkefølge som minimerer den totale kostnaden. Å finne denne rekkefølgen er et NP-hardt problem, men man kan bruke gode heuristikker (eks: minimal-remaining-value). Kobling mellom forward chaining og CSP: hver konjunkt kan ses på som en begrensning på variablene (eks: $Missile(x)$ = ensartet begrensning på x), mens CSP med endelig domene kan behandles som én definite klausul og enkelte ground fakta. Matching er NP-hardt, men som regel er det få og enkle regler, overflødige regler kan elimineres, osv.

2. **Inkrementell forward chaining** = en kilde til ineffektivitet i algoritmen er overflødig matching av premisser mot kjente fakta i KB. For eksempel vil $Missile(x) \Rightarrow Weapon(x)$ matches med $Missile(M_1)$ på nytt i implementasjon 2, men det skjer ingenting siden $Weapon(M_1)$ allerede er lagt til KB. Dette kan unngås ved å se at alle nye fakta som utledes ved iterasjon t må være utledet fra minst en ny fakta som ble utledet i iterasjon $t-1$ (hvis ikke ville den ha blitt utledet ved iterasjon $t-1$). Overflødig matching kan dermed unngås ved å kun sjekke implikasjoner der premissene ble utledet i forrige iterasjon. Dette gir en mer effektiv forward chaining.
3. **Irrelevante fakta** = i likhet med forward chaining i utsagnslogikk vil den generere mange fakta som er irrelevante for målet. Dette kan løses ved å bruke backward-chaining eller begrense forward-chaining til et subsett av reglene.

Backward chaining

Ideen ved backward chaining er å starte med målet og deretter bruke Modus Ponens i bakover retning helt til det finner fakta som støtter beviset.

Backward-chaining algoritme

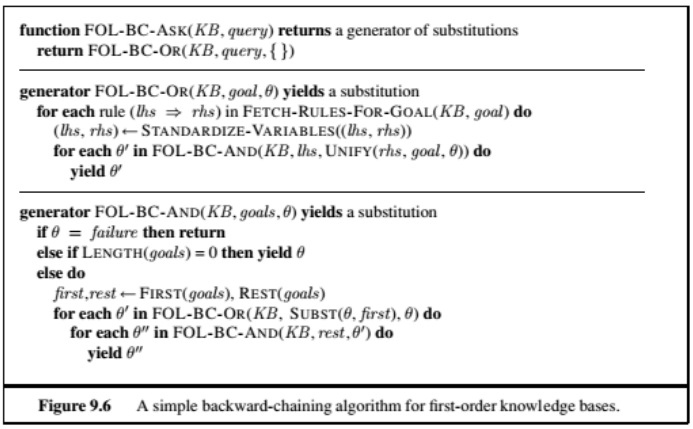


Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

Figuren viser en **backward-chaining algoritme for definite klausuler, som vil svare på en query $goal$ dersom KB inneholder en klausul på formen $left\_side \Rightarrow goal$, der $left\_side$ er konjunksjon.** En atomisk fakta $American(West)$ er en klausul der $left\_side$ er en tom konjunksjon. En query som inneholder variabler kan vises på flere måter. For eksempel hvis query er $Person(x)$ kan det svares med substitusjonen $\{x/John\}$ og $\{x/Richard\}$. Backward-chaining algoritmen er derfor en generator, som vil si en funksjon som returnerer flere ganger, der hver gang gir et mulig resultat.

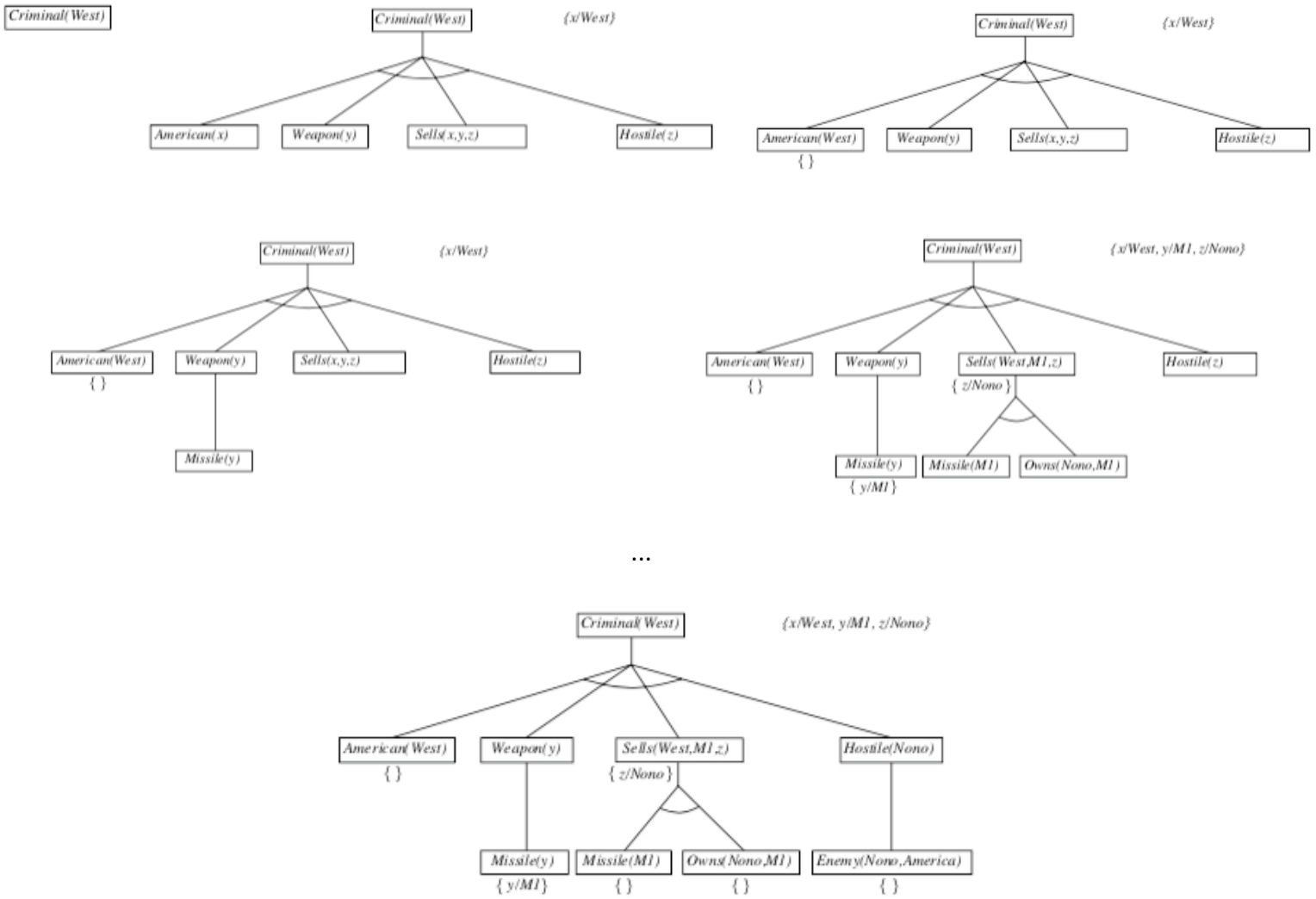
Backward chaining er en form for AND/OR-søk. OR-delen av søket vil hente alle implikasjonsklausulene som har konklusjon som kanskje kan forenes med målet og disse blir standardisert for å hindre overlapp av variabler med målet. AND-delen av søket vil vise alle konjunktene i premisset til implikasjonen der konklusjonen kan forenes med målet. Algoritmen vil vise hver del rekursivt og holder styr over den akkumulerte substitusjonen underveis. Når algoritmen finner en substitusjon, for eksempel $\{z/Nono\}$, vil den bruke denne på alle etterfølgende konjunksjoner.

Vi ser igjen på eksempelet på side 93 og bruker backward chaining for å finne løsning på query $Criminal(West)$? Algoritmen vil utføre følgende rekursive iterasjoner:

1. **$goal$ er $Criminal(West)$:** OR-delen finner at konklusjonen til S1 kan forenes ved å bruke $\theta = \{x/West\}$. AND-delen finner at konjunksjonene som må vises er $American(West)$, $Weapon(y)$, $Sells(West, y, z)$ og $Hostile(z)$. Siden $American(West)$ er i KB, er denne allerede vist.

2. ***goal* er $\boldsymbol{Weapon(y)}$:** OR-delen finner at konklusjonen til S4 kan forenes ved å bruke $\theta = \{y/x\}$. AND-delen finner at konjunksjonen som må vises er $Missile(x)$. Siden $Missile(M_1)$ er i KB, er denne vist med substitusjon $\theta = \{y/M_1\}$. Denne substitusjonen overføres til resten av konjunksjonene.
3. ***goal* er $\boldsymbol{Sells(West, M_1, z)}$**: OR-delen finner at konklusjonen til S3 kan forenes ved å bruke $\theta = \{z/Nono\}$. AND-delen finner at konjunksjonen som må vises er $Owns(Nono, M_1)$ og $Missile(M_1)$. Siden disse er i KB, er de allerede vist. Substitusjonen overføres til resten av konjunksjonene
4. ***goal* er $\boldsymbol{Hostile(Nono)}$**: OR-delen finner at konklusjonen til S5 kan forenes. AND-delen finner at konjunksjonen som må vises er $Enemy(Nono, America)$. Siden denne er i KB er den allerede vist

Dermed er alle konjunksjonene i steg 1 vist, og algoritmen har dermed vist query. Figurene under viser hvordan bevistreet blir laget for dette eksempelet.



...

Backward chaining er en **dybde-først søkealgoritmen**, så den romkompleksiteten er lineær med størrelsen til beviset. **Dette betyr også at backward chaining vil være utsatt for overflødige baner og uendelig looper, noe som kan løses med memoisering (= caching av løsninger til delmål og gjenbruk av disse).** Backward chaining blir brukt i logisk programmering, som er den mest brukte formen for automatisk resonnering (dvs. kunnskap uttrykkes i et formelt språk og systemet løser problemer ved å kjøre inferensprosedyrer på kunnskapen). Backward chaining brukes for å utføre program innenfor logisk programmering.

Resolusjon

Vi skal se hvordan resolusjon kan utvides for å gjelde første-orden logikk.

Merk forskjellen mellom klausul og CNF: en klausul er en disjunksjon av literaler, mens CNF er en konjunksjon av klausuler. Ved definite og hornklausuler er det begrensninger på antall positive literaler i disjunksjonen

Første-orden logikk i CNF

Første-orden resolusjon krever at setningene er i CNF (Conjunctive Normal Form), altså en konjunksjon av klausuler, der hver klausul er en disjunksjon av literaler. Literalene kan inneholde variabler som antas å være universelt kvantifisert. For eksempel vil setningen:

$$\forall x\ American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x)$$

Bli følgende i CNF:

$$\neg American(x) \lor \neg Weapon(y) \lor \neg Sells(x, y, z) \lor \neg Hostile(z) \lor Criminal(x)$$

Alle setninger i første-orden logikk kan omformes til inferensekvivalent CNF setninger (dvs. CNF setningen er satisfiable hvis den originale setningen er satisfiable). Prosessen av å omdanne setninger til CNF ligner metoden for proposisjonslogikk, bortsett fra at vi må i tillegg fjerne kvantifiseringene. Vi ser på et eksempel der vi omformer setningen «alle som elsker alle dyr, er elsket av noen» til CNF form. Setningen er $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y\ Loves(y, x)]$.

1. **Eliminer bibetingelser og implikasjoner** = bruker inferensregler for å fjerne ⇔ og ⇒:
   $$\forall x\ [\neg\forall y\ \neg Animal(y) \lor Loves(x, y)] \lor [\exists y\ Loves(y, x)]$$

2. **Flytt ¬ innover** = bruker de Morgans lover for å flytte ¬ innover:
   $$\forall x\ [\exists y\ Animal(y) \land \neg Loves(x, y)] \lor [\exists y\ Loves(y, x)]$$

3. **Standardiser variabler** = for setninger på formen $\exists x P(x) \lor \exists x Q(x)$, der samme variabelnavn brukes, vil vi endre navnet på en av variablene for å hindre overlapp:
   $$\forall x\ [\exists y\ Animal(y) \land \neg Loves(x, y)] \lor [\exists z\ Loves(z, x)]$$

4. **Skolemisering** = eksistensielle kvantifiseringer fjernes ved eliminering. I det enkle tilfellet vil dette være eksistensiell instansiering der $\exists x\ P(x)$ erstattes med $P(A)$ der A er en ny konstant (= Skolem konstant). Hvis den eksistensielle kvantifiseringen er innenfor omfanget til en universell kvantifisering, må vi i stedet bruke en Skolem funksjon der argumentet er variabelen til den universelle kvantifiseringen. I vårt tilfelle er Skolem enhetene avhengig av en universell kvantifisert variabel x, så derfor må vi erstatte variablene hos de eksistensielle kvantifiseringene med Skolem funksjoner:
   $$\forall x\ \big[Animal\big(F(x)\big) \land \neg Loves\big(x, F(x)\big)\big] \lor Loves(G(x), x)$$

5. **Fjern universell kvantifisering** = fjern ∀:
   $$\big[Animal\big(F(x)\big) \land \neg Loves\big(x, F(x)\big)\big] \lor Loves(G(x), x)$$

6. **Distribuer ∨ over ∧** = flytt ∨ inn i ∧-parenteser:
   $$\big[Animal\big(F(x)\big) \lor Loves(G(x), x)\big] \land \big[\neg Loves\big(x, F(x)\big) \lor Loves(G(x), x)\big]$$

Denne setningen er nå i CNF og består av to klausuler. Her vil $F(x)$ refererer til dyrene som kanskje er uelsket av x, mens $G(x)$ refererer til noen som kanskje elsker x.

Resolusjonsregelen for første-orden logikk

Resolusjonsregelen for første-orden logikk er en løftet versjon av resolusjonsregelen for proposisjonslogikk. To klausuler som er standardisert fra hverandre (dvs. inneholder ikke like variabler) kan resolveres hvis de inneholder komplementære literaler. I proposisjonslogikk er literalene komplementære hvis de er negasjonen av hverandre, mens i **første-orden logikk er literalene komplementære hvis en kan forenes med negasjonen til den andre**. Dvs. hvis det finnes en substitusjon av variablene som gjør at den ene literalen blir identisk med negasjonen til den andre literalen. Resolusjonsregelen blir:

$$\frac{l_1 \vee \ldots \vee l_{i-1} \vee l_i \vee l_{i+1} \ldots \vee l_n, \qquad m_1 \vee \ldots \vee m_{j-1} \vee m_j \vee m_{j+1} \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \ldots \vee l_n \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \ldots \vee m_n}$$

der $UNIFY(l_i, \neg m_i) = \theta$. Dvs. hvis vi bruker substitusjonen θ på l_i vil vi få et uttrykk som er identisk med $\neg m_i$. For eksempel kan vi se på følgende to klausuler:

$$\big[Animal\big(F(x)\big) \vee Loves(G(x), x)\big]$$
$$[\neg Loves(u, v) \vee \neg Kills(u, v)]$$

Disse kan resolveres med $\theta = \{u/G(x), v/x\}$, slik at vi får resolvement:

$$\big[Animal\big(F(x)\big) \vee \neg Kills(u, v)\big]$$

Dette kalles **binær resolusjon**, siden vi oppløser to literaler. Inferensprosedyren vil også involvere **faktorisering, der overflødige literaler blir fjernet**. I proposisjonslogikk involverer dette å representere to identiske literaler vha én literal, mens i første-orden logikk involverer dette å representere to literaler som en hvis de kan forenes. $UNIFY$ må i så fall brukes på hele klausulen. **Kombinasjonen av binær resolusjon og faktorisering er komplett**.

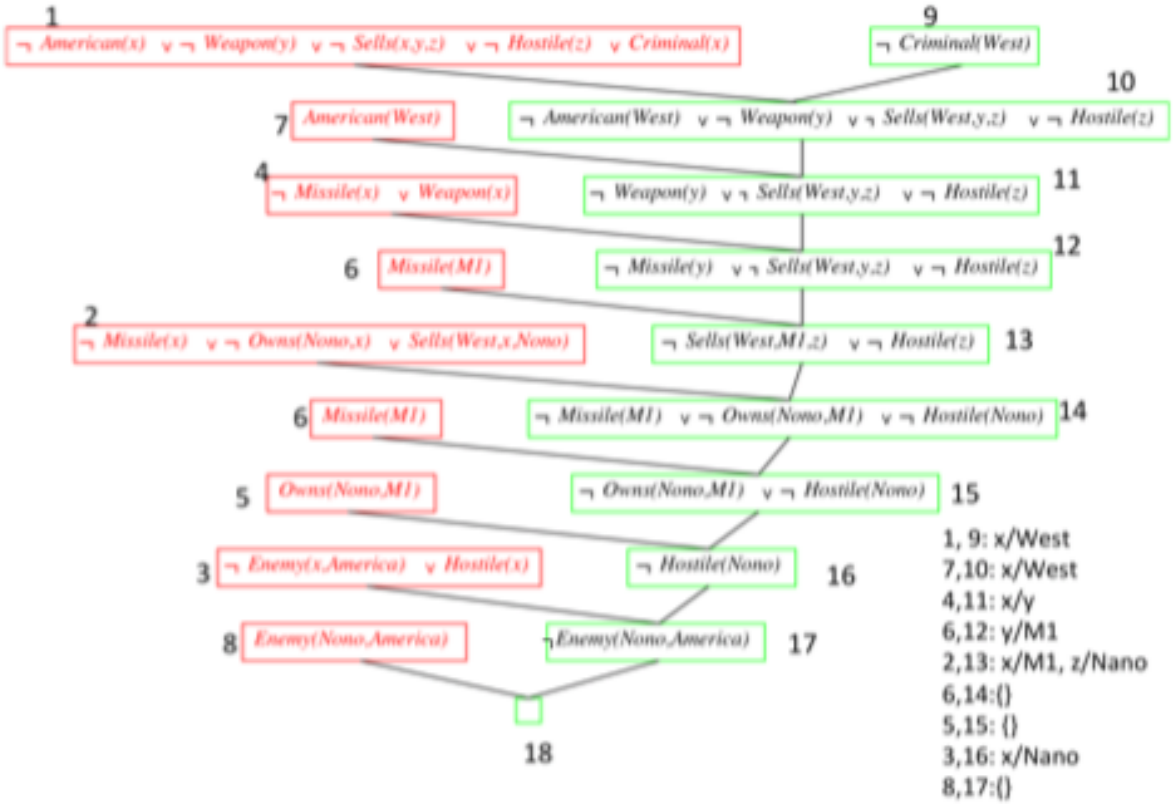
Eksempler

Resolusjon vil bevise $KB| = \alpha$ ved å vise at $(KB \wedge \neg\alpha)$ er unsatisfiable, noe som er tilfellet dersom den tomme klausulen kan utledes. Algoritmen for resolusjon med første-orden logikk er lik algoritmen for proposisjonslogikk (s. 75).

Vi ser på hvordan resolusjon kan brukes for å løse kriminell-eksempelet. I dette tilfellet er $\alpha = Criminal(West)$,. Setningene i CNF er:

$$\neg\alpha = \neg Criminal(West)$$
$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x)$$
$$\neg Owns(Nono, x) \vee \neg Missile(x) \vee Sells(West, x, Nono)$$
$$\neg Missile(x) \vee Weapon(x)$$
$$\neg Enemy(x, America) \vee Hostile(x)$$
$$Owns(Nono, M_1)$$
$$Missile(M_1)$$
$$American(West)$$
$$Enemy(Nono, America)$$

Figuren på neste side viser resolusjonsbeviset. Legg merke til strukturen som er en «ryggrad», som begynner med målklausulen ($\neg Criminal(West)$) og løser opp med CNF-klausuler fra KB helt til man oppnår en tom klausul. For hvert steg ser vi at de to setningene inneholder komplementære literaler som gjør at de løser opp hverandre. Legg merke til at dette kan kreve substitusjon av variabler, som er markert med tall. Standardiseringen vil gjøre at disse variablene ikke overlapper i algoritmen.



I dette tilfellet («ryggrad»-struktur) kan vi se at vi har valgt å resolvere literalen til venstre ved hver iterasjon. Dette tilsvarer backward-chaining der *goal* vil være literalen som resolveres. **Backward chaining er et spesielt tilfelle med resolusjon med en bestemt kontrollstrategi som bestemmer hvilken resolusjon som skal utføres neste gang**.

Vi skal nå se på et mer komplisert eksempel som involverer bruken av skolemisering og klausuler som ikke er definite. Problemet er som følger «Alle som elsker alle dyr er elsket av noen. Alle som dreper et dyr er elsket av ingen. Jack elsker alle dyr. Enten Jack eller Curosity drepte katten, som het Tuna. Drepte Curosity katten?». Først må vi uttrykke setningene:

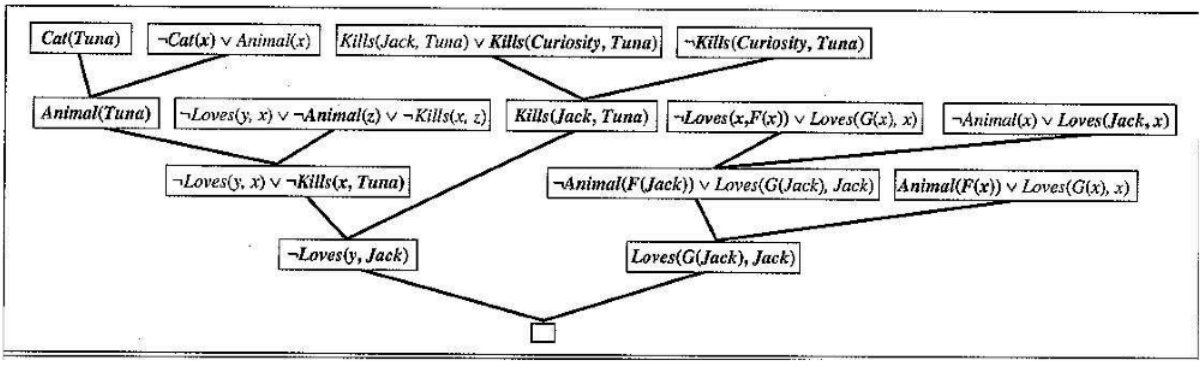
A. $\forall x \left(\forall y\ Animal(y) \Rightarrow Loves(x,y)\right) \Rightarrow \left(\exists y\ Loves(y,x)\right)$
B. $\forall x \left(\exists z\ Animal(z) \wedge Kills(x,z)\right) \Rightarrow \left(\forall y\ \neg Loves(y,x)\right)$
C. $\forall x\ Animal(x) \Rightarrow Loves(Jack,x)$
D. $Kills(Jack,Tuna) \vee Kills(Curosity,Tuna)$
E. $Cat(Curosity)$
F. $\forall x\ Cat(x) \Rightarrow Animal(x)$
G. $\neg Kills(Curosity,Tuna)$

Her vil G representere $\neg\alpha$. Neste steg er å omforme disse til CNF:

A. $Animal\big(F(x)\big) \vee Loves(G(x),x)$
   $\neg Loves\big(x,F(x)\big) \vee Loves(G(x),x)$
B. $\neg Animal(z) \vee \neg Kills(x,z) \vee \neg Loves(y,x)$
C. $\neg Animal(x) \vee Loves(Jack,x)$
D. $Kills(Jack,Tuna) \vee Kills(Curosity,Tuna)$
E. $Cat(Curosity)$
F. $\neg Cat(x) \vee Animal(x)$
G. $\neg Kills(Curosity,Tuna)$

Figuren viser resolusjonsbeviset for dette tilfellet. Beviset kan forklares slik: Anta at Curosity ikke drepte Tuna. Vi vet at enten Jack eller Curosity gjorde det, så derfor må Jack ha gjort det. Tuna er en katt og katter er dyr, så Tuna er en katt Siden alle som dreper dyr er elsket av

ingen, vet vi at ingen elsker Jack. Samtidig så elsker Jack alle dyr, så noen elsker han. Vi har en motsigelse. Derfor må Curosity ha drept katten.



Resolusjonsstrategier

Noen strategier for å finne bevis mer effektivt er:

- **Enhetspreferanse** = prioriter resolusjoner der en av setningene er en enkel literal (= enhetsklausul). Vi prøver å produsere en tom klausul, så det kan være lurt å bruke inferenser med kortere klausuler
- **Støttesett** = vil eliminere noen mulige resolusjoner ved å kreve at hvert resolusjonssteg involverer minst ett element fra et spesielt sett med klausuler, kalt støttesettet. Hvis dette settet er lite sammenlignet med hele KB, kan søkerommet dramatisk reduseres
- **Input resolusjon** = alle resolusjonssteg kombinerer en av inputsetningene fra KB eller query med en annen setning. Gir «ryggmarg»-struktur.
- **Subsumsjon** = eliminerer alle setninger som gis av en eksisterende setning i KB. For eksempel hvis $P(x)$ er i KB, vil den eliminere $P(A)$. Dette holder KB liten og vil dermed redusere søkerommet

Kapittel 10 – Klassisk planlegging

Planlegging er utarbeiding av en handlingsplan for å oppnå mål, og det er en essensiell del av AI. Vi har tidligere sett på to eksempler på planleggende agenter, den problemløsende agenten i kapittel 3 og hybridagenten i kapittel 7. Dette kapittelet introduserer en representasjon for planlegging av problemer som kan skaleres opp til problemer som ikke kan håndteres av tilnærmingene vi har sett på tidligere. Det ser på omgivelser som er fullt observerbare, deterministiske, statiske og singel-agent.

Hva er planlegging? (F)

Et menneske kan handle uten eksplisitt planlegging dersom det er et øyeblikkelig formål, oppførselen er godt trent inn eller handlingsforløpet er frivillig. **Planlegging er nødvendig dersom man er i en ny situasjon, når oppgavene er komplekse, når omgivelsene medfører stor risiko eller høye kostnader og når man må samarbeide med andre**. Det er kun når det er nødvendig at mennesker vil planlegge.

Grunnleggende terminologi (F)

For planlegging og handling i AI har vi følgende terminologi:

- **Planleggingsproblem** = et problem som kan løses av et AI planleggingssystem
- **Planleggingsdomene** = verden der planleggingen foregår
- **Tilstand** = et punkt i søkerommet til planleggingsdomenet. Et planleggingsproblem blir karakterisert av en initial tilstand og en måltilstand. Den initiale tilstanden er tilstanden til verden rett før planen utføres, mens måltilstanden er den ønskede tilstanden til verden som vi ønsker å nå etter å ha utført planen

En plan er en samling av handlinger for å utføre en oppgave. Representasjonen beskriver tilstander, handlinger, mål og planer, mens metodene beskriver planleggende algoritmer og mekanismer. Så langt i kompendiet har vi ikke fokusert på representasjonen og behandlingen av **endring**, som er et viktig konsept i planlegging.

Søk vs. planlegging (F)

	Search	Planning
State	Data structures	Logical sentences
Action	Code	Preconditions and outcomes
Goal	Goal test	Logical sentence (conjunction)
Plan	Sequence from initial state	Constraints on actions

Et planleggingsproblem ligner et søkeproblem ved at det har tilstander, operatorer og mål, men problemrepresentasjonen er mer strukturer (se tabell). **I komplekse søkeproblem for planlegging vil det som regel være svært stor branching faktor og det kan være vanskelig å finne gode heuristiske funksjoner**.

Representasjon – STRIPS

Den problemløsende agenten (kapittel 3) kan finne handlingssekvenser som resulterer i en måltilstand, men den krever atomiske tilstander og god domene-spesifikk heuristikk for å fungere. Den logiske agenten (kapittel 7) kan bruke generelt-formål heuristikk, men den avhenger av ground (variabel-fri) proposisjonell inferens som kan føre til mange setninger hvis det er mange handlinger og tilstander.

I planlegging blir det brukt en **faktorert representasjon**, der tilstanden til verden representeres som en samling av variabler. **STRIPS representasjon blir brukt for å definere søkeproblemet** (STRIPS = Stanford Research Institute Problem Solver). Dette språket er basert på database semantikk, der lukket-verden antagelsen gir at alt som ikke blir eksplisitt

I boka bruker de PDDL (Planning Domain Definition Language) som er utledet fra STRIPS og er litt mindre begrenset (tillater negative literaler). I forelesning blir STRIPS brukt, så derfor brukes dette i kompendiet.

uttrykt antas å være falskt og ulike navn betyr at det er distinkte elementer. Definisjonen av problemet blir:

1. **Tilstander** = hver tilstand er representert som en konjunksjon av flytende aspekter (endres over tid) som ikke inneholder funksjoner eller variabler. Eks: $At(P_1, JFK) \wedge Plane(P_1) \wedge Airport(JFK)$. Tilstandene har ikke negasjon (lukket-verden)
2. **Handlinger** = hver handling blir beskrevet med et handlingsskjema som definerer $ACTION(s)$ og $Result(s, a)$ funksjonene som man trenger for problemløsende søk. For å unngå rammeproblemet, vil STRIPS spesifisere resultatet av handlingen mht. det som endres og alt som ikke nevnes er uendret. Handlingsskjemaet består av:
    a. **Navn**: identifiserer handlingen og gir variablene som brukes. Variablene antas å være universelt kvantifisert (variabler kan initialiseres med enhver verdi). Eks: $Action\big(Fly(p, from, to)\big)$ kan bli $Action\big(Fly(P_1, SFO, JFK)\big)$
    b. **Prebetingelser**: bestemmer hvilken tilstand handlingen kan utføres i og består av en konjunksjon av positive literaler. Eks: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
    c. **Effekter**: definerer resultatet ved å utføre handlingen og består av en konjunksjon av literaler. Det representeres som en Add-liste som gir proposisjonene som blir sanne etter handlingen, og en Delete-liste som gir proposisjonene som blir falske etter handlingen. Eks: $At(p, to) \wedge \neg At(p, from)$

    En handling a kan utføres i tilstand s, dersom prebetingelsene til a følger av s (i formell notasjon: $a \in ACTIONS(s) \Leftrightarrow s| = PRECOND(a)$). Vi sier at handlingen vil være anvendelig ($applicable$) i tilstanden s hvis prebetingelsene er tilfredsstilt av s. Hvis handlingen inneholder variabler kan den ha flere instanser for en tilstand (eks: $Fly(P_1, SFO, JFK)$ og $Fly(P_2, JFK, SFO)$. Et krav i handlingsskjemaet er at alle variablene i effekter må også være i prebetingelsene. STRIKT vil ikke eksplisitt gi tiden, fordi prebetingelsene vil alltid gjelde tid t, mens effekten gjelder tid $t + 1$
3. **Mål** = konjunksjon av literaler som kan inneholde eksistensielt kvantifiserte variabler. Eks: $At(p, SFO) \wedge Plane(p)$ betyr at målet er å ta ethvert fly fra SFO-flyplassen

Planleggingsdomenet vil være et sett med handlingsskjemaer, mens problemet i domenet vil i tillegg inkludere initial tilstand og mål. Problemet er løst når vi kan finne en handlingssekvens som ender i en tilstand som målet følger av. For eksempel vil tilstanden $At(P_1, SFO) \wedge Plane(P_1)$ være en løsning for problemet definert over, siden $At(p, SFO) \wedge Plane(p)$ følger av denne tilstanden.

STRIPS er et veldig begrenset språk, som legger til rette for effektiv inferens siden alle handlingene er addisjoner og sletting av proposisjoner, og de vil være i kunnskapsbasen. Ulempen er at det antas at kun et lite antall proposisjoner vil endres ved hver handling og det er et begrenset språk, så det kan ikke brukes i alle domener.

Eksempel – lufttransport

$Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK)$
$\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)$
$\wedge Airport(JFK) \wedge Airport(SFO))$
$Goal(At(C_1, JFK) \wedge At(C_2, SFO))$
$Action(Load(c, p, a),$
PRECOND: $At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $\neg At(c, a) \wedge In(c, p))$
$Action(Unload(c, p, a),$
PRECOND: $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $At(c, a) \wedge \neg In(c, p))$
$Action(Fly(p, from, to),$
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\neg At(p, from) \wedge At(p, to))$

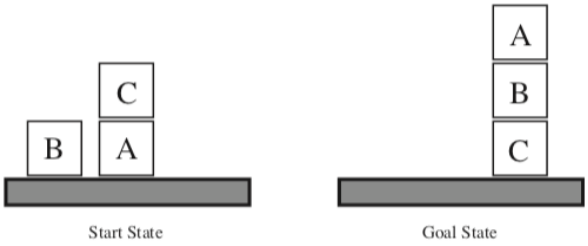
Figuren viser et lufttransport-problem som involverer å laste og lossing av last og transport ved å fly lasten fra en plass til en annen. Problemet defineres med tre handlinger: $Load$, $Unload$ og Fly. Handlingene påvirker to predikater: $In(c, p)$ betyr at lasten c (*cargo*) er i fly p (*plane*), mens $At(x, a)$ betyr at objekt x (last eller fly) er ved flyplass a (*airport*). Legg merke til at problemet er nøye med å passe på at At-predikatet blir håndtert riktig. Når flyet flyr fra en plass til en annen, vil lasten i flyet bli med. Dette blir gitt ved å si at lasting innebærer at lasten slutter å være ved en plass og

lossing innebærer at lasten starter å være ved en plass. Gitt den initiale tilstanden, målet og handlingene vil følgende plan være løsningen til problemet:

$$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$$
$$Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P_2, SFO)]$$

Det kan være lurt å legge til ulikheter i prebetingelsene for å hindre handlinger slik som $Fly(P_1, JFK, JFK)$. Dette problemet kan ignoreres, fordi de vil som regel utlignes i effektene.

Eksempel – blokkverden



Blokkverden er en av de mest kjente planleggingsdomenene, og dette domenet består av et sett med kubeformet blokker som er plassert på et bord. Blokkene kan stables, men det er kun plass til én blokk i bredden. En robotarm kan plukke opp blokken og flytte den til en annen posisjon, som enten er på bordet eller på toppen av en annen blokk. Armen kan bare plukke opp en blokk om gangen, så den kan ikke plukke en blokk som har en annen på toppen. Et mål kan være å få blokk A på blokk B som er på blokk C (se figur).

$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A)$
$\wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C))$
$Goal(On(A, B) \wedge On(B, C))$
$Action(Move(b, x, y),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$
$(b \neq x) \wedge (b \neq y) \wedge (x \neq y),$
EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$
$Action(MoveToTable(b, x),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x),$
EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$

Figuren til venstre viser dette blokkverden-problemet. Her blir blokkene representert som konstanter og bordet representeres som konstanten $Table$. Vi bruker $On(b, x)$ for å indikere at blokk b er på x, der x enten er en annen blokk eller bordet. $Move(b, x, y)$ er handlingen av å flytte blokk b fra x til y. En prebetingelse vil være at det ikke er noen andre blokker på blokk b, noe vi representerer som $Clear(b)$ som er sann når ingenting er på b (merk: i første-orden logikk ville vi ha løst dette som $\exists x \neg On(x, b)$, men STRIPS tillater ikke eksistensiell kvantifisering)

Legg merke til handlingen $MoveToTable$. Denne trengs fordi handlingen $Move$ kan ikke brukes når x eller y er bordet, fordi da vil det gi $Clear(Table)$ i prebetingelse eller effekt, noe som aldri vil være tilfellet siden det alltid er blokker på bordet. Derfor introduserer vil handlingen $MoveToTable$ som brukes for å flytte blokk b fra x til bordet. I tillegg lar vi $Clear(x)$ bety at det er en klar plass på x som kan holde en blokk, slik at $Clear(Table)$ alltid er sann. Problemdefinisjonen har også håndter følgende problem:

1. **Planleggingssystemet vet ikke når den skal bruke $MoveToTable$ istedenfor $Move$** = løst ved å introdusere $Block(b)$ og $Block(y)$ i prebetingelsen, slik at $Move$ blir kun utført når blokken flyttes til toppen av en annen blokk
2. **Det kan utføres unødvendige handlinger som $Move(b, c, c)$** = løst ved å introdusere ulikhetskrav i prebetingelsene, slik at blokken må flyttes mellom ulike plasser

Kompleksiteten ved klassisk planlegging

For worst-case probleminstanser vil planlegging være NP-hardt, men agenter vil som regel ikke bli bedt om å lage planer for slike instanser. I stedet blir agentene ofte bedt om å planlegge i et spesifikt domene (eks: blokkverden), som kan være mye enklere enn i teoretisk worst-case tilfellet. Optimal planlegging er som regel vanskelig, men sub-optimal planlegging er noen ganger lett. For å lage gode planer, trenger vi god søkeheuristikk. Fordelen ved klassisk planleggingsformalisme er at den legger til rette for utviklingen av god heuristikk som er uavhengig av domenet.

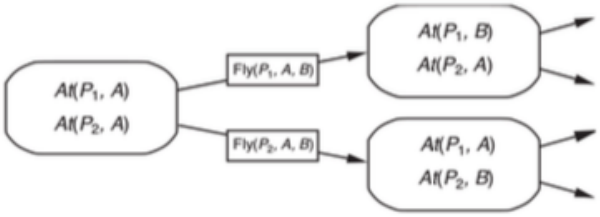
Metoder – tilnærminger til planlegging

For å lage planer bruker vi som regel metoder som bruker logisk inferens for å finne løsning. Det er **to grunnleggende tilnærminger til planlegging**:

- **Tilstandsrom planlegging (søk) =** arbeider ved nivået for tilstander og handlinger. Den finner en plan ved å søke gjennom tilstandsrommet. Dette er mest likt konstruktiv søk
- **Planrom planlegging** = arbeider ved nivået for planer. Den finner en plan ved å søke gjennom rommet av planer, der den starter med en delvis ukorrekt plan og påfører endringer for å korrigere den. Dette er mest likt iterativ forbedring.

Tilstandsrom planlegging (søk)

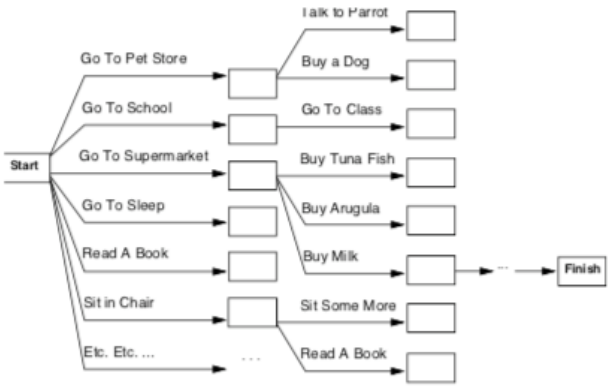
Vi har sett hvordan planleggingsproblemet kan defineres som et søkeproblem, der vi søker fra initial tilstand etter en måltilstand gjennom tilstandsrommet. Vi kan også søke bakover fra måltilstanden etter en initialtilstand.



Forward (progresjon) tilstandsromsøk

Siden planleggingsproblemet kan defineres som et søkeproblem, kan det løses med heuristiske søkealgoritmer og lokalt søk. **Ved forward tilstandsrom søk starter søkealgoritmen med den initiale tilstanden og bruker handlingene for å søke mot en måltilstand**. **Progresjonsplanleggere vil resonnere fra starttilstanden, og de forsøker å finne handlinger som fører til måltilstanden ved å matche prebetingelser med tilstanden.** Søket har følgende steg:

1. **Bestem alle anvendbare handlinger** i starttilstanden ved å matche tilstanden med prebetingelsene til handlingene
2. **Ground handlingene** ved å erstatte alle variabler med konstanter
3. **Velg en handling** som skal gjennomføres
4. **Bestem det nye innholdet til kunnskapsbasen**, basert på effekten til handlingen
5. **Gjenta helt til måltilstanden nås**



Figuren viser et eksempel for supermarked-domenet. For eksempel hvis målet er å skaffe en drill, vil vi ha følgende problemdefinisjon:

- Initial tilstand: $At(Home) \wedge Sells(HardwareStore, Drill)$
- Måltilstand: $At(Home) \wedge Have(Drill)$
- Handling:
  - $Action(Go(x, y),$
    $PRECOND: At(x)$
    $Effect: At(y) \wedge \neg At(x))$
  - $Action(Buy(x, y),$
    $PRECOND: At(x) \wedge Sells(x, y)$
    $Effect: Have(y))$

Progresjonsplanleggeren vil først bestemme at eneste anvendbar handling i starttilstanden er $Go(x, y)$, så den erstatter variablene med passende konstanter: $Go(Home, Pet\ Store)$, $Go(Home, School)$, $Go(Home, HardwareStore)$, osv. Legg merke til at forward tilstandsrom søk vil se på alle mulige handlinger, og deretter velge å utføre en av disse. I dette tilfellet vil den velge å utføre $Go(Home, HardwareStore)$ som gjør at effekten $At(HardwareStore) \wedge \neg At(Home)$ blir lagt til KB. Dette vil igjen gjøre at andre handlinger blir anvendbare, for eksempel $Buy(HardwareStore, Hammer)$, $Buy(HardwareStore, Drill)$, osv. I dette tilfellet

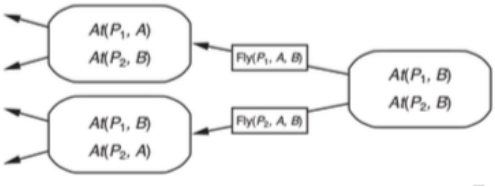
vil den velge å utføre $Buy(HardwareStore, Drill)$ som gjør at effekten $Have(Drill)$ blir lagt til KB. Agenten gjentar prosessen helt til den når målet. **Legg merke til at dette involverer mange handlinger som kan utføres**.

Forward tilstandsromsøk er ineffektivt fordi:

- **Det må utforske irrelevante handlinger** i forsøket på å finne en som fører til målet
- **Det kan bruke lang tid på å finne en relativ lett løsning hvis søkerommet er stort**, noe som ofte er tilfellet siden planleggingsproblem som regel har stor branching faktor

Forward søk er likevel mulig på grunn av god domenespesifikk heuristikk.

Backward (regresjon) relevant-tilstand søk



Ved backward tilstandsrom søk starter søkealgoritmen med måltilstanden og bruker handlingene bakover for å søke mot en initial tilstand. Regresjonsplanleggere vil resonnere fra måltilstanden, og de forsøker å finne handlinger som fører til initial tilstand ved å matche effektene med måltilstanden. Det kalles **relevant-tilstand søk**, fordi det ser kun på handlinger som er relevant for måltilstanden eller nåværende tilstand. Søket har følgende steg:

1. **Velg en handling der effekten tilfredsstiller hele eller deler av målet som er gitt som en konjunksjon**
2. **Lag et nytt målsett** ved å fjerne målbetingelsene som tilfredsstilles av handlingen og legge til prebetingelsene til handlingen. Målbetingelsene som ikke er tilfredsstilt blir bevart
3. **Gjenta helt til man når hele eller deler av initial tilstand**

Vi ser igjen på supermarked-domenet. I dette tilfellet er målet å kjøpe en drill og melk, så problemet kan defineres som følger:

- Initial tilstand: $At(Home) \wedge Sells(HardwareStore, Drill)$
- Måltilstand: $At(Home) \wedge Have(Drill)$
- Handling:
  - $Action(Go(x, y),$
    $PRECOND: At(x)$
    $Effect: At(y) \wedge \neg At(x))$
  - $Action(Buy(x, y),$
    $PRECOND: At(x) \wedge Sells(x, y)$
    $Effect: Have(y))$

Regresjonsplanleggeren begynner med måltilstanden $At(Home) \wedge Have(Drill)$, og den vil utføre følgende iterasjoner:

1. Velger handling $Go(HardwareStore, Home)$, siden denne har effekt $At(Home) \wedge \neg At(HardwareStore)$ som tilfredsstiller deler av målet. Prebetingelsen til denne handlingen er $At(HardwareStore)$, så det nye målsettet blir $Have(Drill) \wedge At(HardwareStore)$
2. Velger handling $Buy(HardwareStore, Drill)$, siden denne har effekt $Have(Drill)$ som tilfredsstiller deler av målet. Prebetingelsen til denne handlingen er $At(HardwareStore) \wedge Sells(HardwareStore, Drill)$, så det nye målsettet blir $At(HardwareStore) \wedge Sells(HardwareStore, Drill)$
3. Velger handling $Go(Home, HardwareStore)$, siden denne har effekt $At(HardwareStore) \wedge \neg At(Home)$ som tilfredsstiller deler av målet. Prebetingelsen til denne handlingen er $At(Home)$, så det nye målsettet blir $At(Home) \wedge Sells(HardwareStore, Drill)$. Siden dette er lik den initiale tilstanden, er søket ferdig.

Rekkefølgen til hvilke handlinger som utføres vil ha betydning! Denne prosessen involverer substitusjon, eks: $\theta = \{x/HardwareStore\}$. Legg merke til at søket skiller mellom negert og ikke-negert literal! Det er kun når literalen til effekten er lik literalen i målsettet at literalen kan fjernes fra målsettet. **Hvis effekten til handlingen inneholder negasjonen av en literal i målet, vil ikke planleggeren velge denne effekten**. For eksempel hvis målet er $A \wedge B \wedge C$ og en handling har effekten $A \wedge B \wedge \neg C$, vil ikke denne handlingen velges! Planleggeren vil altså kun velge handlingen dersom den har en effekt som har literaler som er identiske med literaler i målet og ikke negerte literaler hos målet.

Backward søk vil kun fungere hvis algoritmen vet hvordan den kan regne ut forgjengerne til en tilstand eller sett av tilstander. For eksempel vil det være vanskelig å søke bakover for en løsning på n-queen problemet siden det ikke er enkelt å beskrive tilstandene som er et trekk unna måltilstanden. Egenskapene til STRIPS gjør det enkelt å søke bakover, siden effekten gir tilstandene ved tiden $t + 1$ og prebetingelsene gir tilstandene ved tiden t.

Backward søk vil reduserer branching faktoren sammenlignet med forward søk for de fleste problemdomenene. **Ulempen er at det er vanskeligere å lage god heuristikk for backward søk, noe som gjør at er vanligere å bruke forward søk.**

Heuristikk for planlegging

Både forward og backward søk krever god heuristikk for å være effektiv. Den heuristiske funksjonen estimerer avstanden fra en tilstand til målet, og dersom vi kan finne en **admissible heuristikk** (dvs. en som aldri overestimerer), kan vi bruke A* søk for å finne optimale løsninger. Løsningen til det relakserte problemet vil være en admissible heuristikk for det originale problemet.

Planlegging bruker en faktorert representasjon av tilstander og handlingsskjema, som gjør det **mulig å definere god heuristikk som er uavhengig av domene**. Søkeproblemet kan ses på som en graf der nodene er tilstander og kantene er handlinger. I så fall vil problemet være å finne en bane som kobler sammen den initiale noden med målnoden. Det er to måter man kan relaksere dette problemet:

1. **Legge til kanter til grafen** = for å relaksere handlingene i planleggingsproblemet kan vi for eksempel bruke **ignorer-prebetingelse heuristikk**, der alle prebetingelsene hos handlingene blir ignorert. Dette gjør at hver handling blir anvendbar i hver tilstand og heuristikken vil være minimum antall handlinger som trengs for at unionen av deres effekter skal tilfredsstille målet. Noen handlinger kan angre effekten til andre handlinger, men dette blir ofte ignorert. Andre alternativer er å ignorere deler av prebetingelsene eller se bort fra negative literaler i effektene slik at handlinger ikke angrer effekten til hverandre (= **ignorere DEL-listen heuristikk**). Å finne optimale løsninger på disse relakserte problemene er NP-hardt (de reduserer ikke antall tilstander), men man kan finne tilnærminger
2. **Gruppere noder sammen** = det dannes en abstraksjon av tilstandsrommet med færre tilstander, slik at det er enklere å søke etter måltilstanden. Den enkleste formen er å ignorere noen flytende aspekter. For eksempel i supermarked-domenet hvis vi vet at vi skal kjøpe noe, kan vi kun se bort i fra tilstandene som involverer skolen. Dette vil redusere antall tilstander, så løsningen til dette tilstandsrommet vil være kortere enn løsningen til det originale rommet (= admissible). En annen mulighet er å dekomponere problemet inn i deler som løses separat og deretter kombinere løsningene (kan gi ikke-admissible heuristikk).

Vi skal nå se at **planleggergrafer kan gi et bedre heuristisk estimat**.

Planrom planlegging

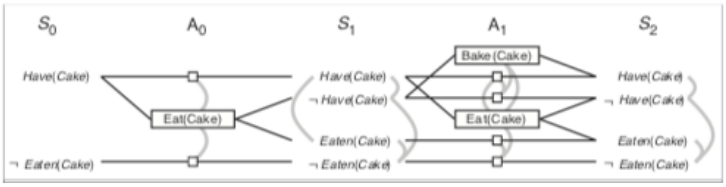
Planrom planlegging finner en plan ved å søke gjennom rommet av planer, der den starter med en delvis ukorrekt plan og påfører endringer for å korrigere den. Vi skal nå se på planleggergrafen som kan brukes som heuristikk i tilstandsrom planlegging eller for å finne planen direkte vha. GRAPHPLAN algoritmen.

Planleggergraf

Et planleggingsproblem ser om vi kan nå en måltilstand fra den initiale tilstanden. Hvis vi har et tre med alle mulige handlinger mellom forgjengere og etterfølgere, kan vi svare på planleggingsproblemet ved å indeksere treet. Dette treet vil ha eksponentiell størrelse, så denne tilnærmingen er upraktisk. **En planleggingsgraf er en polynomial tilnærming til dette treet som kan raskt lages (tar polynomial tid å lage).** Planleggingsgrafen kan ikke gi et nøyaktig svar på om måltilstanden kan nås fra en tilstand, men den kan estimere hvor mange steg som trengs for å nå måltilstanden. **Estimatet er alltid korrekt når det gir at målet ikke kan nås og det vil aldri overestimere antall steg, så det er en admissible heuristikk.**

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake))$

Figure 10.7 The "have cake and eat cake too" problem.



En planleggergraf er en rettet graf som er ordnet inn i nivåer (se figur). Første nivå S_0 representerer initial tilstand og består av en node per literal i den initiale tilstanden. Andre nivå A_0 består av noder for hver ground handling som kan anvendes i S_0. Deretter følger alternerende nivåer med S_i (proposisjoner) og A_i (handlinger):

- **Nivå A_i** = består av de anvendbare handlingene som er koblet til prebetingelser i S_i og effekter i S_{i+1}. Det består også av **vedvarende handlinger (no-op)** som representerer at literaler kan skyldes at ingen handling negerte den (gis som tomme bokser). Nivået vil også vise konflikter mellom handlinger, som kalles **gjensidig utelukkelse.** Handlinger som er koblet med grå linjer betyr at de ikke kan utføres samtidig. For eksempel vil $Eat(Cake)$ og $Bake(Cake)$ være gjensidig utelukkende.
- **Nivå S_i** = består av alle ground literaler som kan resultere ved at handlingene i A_{i-1} utføres (merk: inkluderer no-op). Nivået vil også vise konflikter mellom literaler, som kalles **gjensidig utelukkelse**. Literaler som er koblet med grå linjer, betyr at de ikke kan forekomme samtidig. For eksempel vil $Have(Cake)$ og $Eaten(Cake)$ være gjensidig utelukkende.

Det vil være tre typer kanter mellom nivåene:

1. **Prebetingelse** = kant fra P til A hvis P er en prebetingelse hos A
2. **Add** = kant fra A til P hvis A har effekt P
3. **Delete** = kant fra A til $\neg P$ hvis A har effekt $\neg P$

Alterneringen mellom tilstands- og handlingsnivå vil fortsette helt til to etterfølgende nivåer er identisk. Ved dette nivået sier vi at grafen har **flatet ut**. På figuren ser vi at dette vil være tilfellet ved S_2 (siden den er identisk med S_1). Resultatet er en planleggergraf der hver nivå A_i består av alle handlinger som er anvendbare i S_i og begrensningene som gir hvilke handlinger som ikke kan utføres ved samme nivå. Hver nivå S_i vil inneholde alle literalene som kan resultere fra alle mulige valg av handlinger i A_{i-1} og begrensningene som gir hvilke literaler som ikke kan oppnås samtidig. **Prosessen av å lage planleggergrafen involverer ikke å velge mellom handlingene!**

Handlinger er gjensidig utelukkende hvis de har:

1. **Inkonsistent effekt** = effekten til en handling negerer effekten til den andre. Eks: $Eat(Cake)$ vil ha effekt $\neg Have(Cake)$ som er inkonsistent med effekten til no-op $Have(Cake)$
2. **Inferens** = effekten til en handlingen negerer en prebetingelse til den andre. Eks: $Eat(Cake)$ vil ha effekt $\neg Have(Cake)$ som infererer med prebetingelsen til $Have(Cake)$
3. **Konkurrerende behov** = handlingene har gjensidig utelukkende prebetingelser. Eks: $Eat(Cake)$ og $Bake(Cake)$ er gjensidig utelukkende fordi de konkurrerer om verdien til $Have(Cake)$

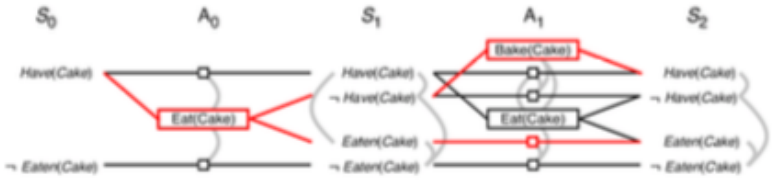
Literaler vil være gjensidig utelukkende hvis en er negasjonen av den andre eller hvis handlingene som kan oppnå literalene er gjensidig utelukkende.

Når planleggergrafen er laget vil den være en kilde til informasjon om problemet. **Problemet er uløselig dersom en literal ved måltilstanden ikke er tilstede i grafen. Kostnaden av å nå enhver målliteral fra en tilstand kan estimeres som nivået der literalen fremkommer. Dette kalles nivåkostanden hos literalen. Disse estimatene er admissible**, men de vil ikke alltid være nøyaktige siden heuristikken teller nivåer og flere handlinger kan utføres ved hvert nivå. Kan løses ved å bruke serie-planleggergraf som kun tillater en handling per nivå (oppnås ved å legge til gjensidig utelukkelse mellom alle nivåene). **For å estimere kostnaden til konjunksjonen av målliteraler, kan vi se på den største nivåkostanden blant literalene (= maksnivå heuristikk), summen av nivåkostnadene (= nivåsum heuristikk) eller nivåkostnaden der alle målliteralene fremkommer uten å være parvis gjensidig utelukkende (= settnivå heuristikk)**. Planleggergrafen kan ses som et relaksert problem og løsningen i planleggergrafen vil derfor være en admissible heuristikk.

GRAPHPLAN algoritmen

```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for tl = 0 to ∞ do
    if goals all non-mutex in S_t of graph then
      solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution ≠ failure then return solution
    if graph and nogoods have both leveled off then return failure
    graph ← EXPAND-GRAPH(graph, problem)
```

GRAPHPLAN algoritmen vil hente en plan direkte fra planleggergrafen. En **gyldig plan** er en delgraf av planleggergrafen der alle prebetingelsene til målet er tilfredsstilt og ikke gjensidig utelukkende i siste nivå, handlinger ved samme nivå er ikke gjensidig utelukkende og alle prebetingelsene til handlingene er tilfredsstilt av planen. Hvis en gyldig plan eksisterer vil den være en del av planleggingsgrafen, så derfor vil GRAPHPLAN forsøke å finne denne ved å søke gjennom grafen. **GRAPHPLAN algoritmen vil begynne med å lage planleggergrafen ved å gjentatt legge til et nivå i grafen helt til alle målliteralene fremkommer uten å være gjensidig utelukkende**. **Deretter vil den søke etter den gyldige planen som løser problemet vha CSP søk eller backward søk**. **Hvis den feiler vil den utvide grafen med et nivå og prøver på nytt. Dette gjentas helt til den finner løsningen eller grafen og alle no-goods (målliteral som ikke ble nådd av søket) er flatet ut.** Det vil ta polynomial tid å lage planleggergrafen. I worst-case vil søket være uløselig, så søket krever god heuristikk for å kunne gjennomføres effektivt.



Figuren viser den gyldige planen for spise-kake-problemet der den initiale tilstanden er $Have(Cake)$ og måltilstanden er $Have(Cake) \wedge Eaten(Cake)$. GRAPHPLAN vil lage planleggergrafen nivå for nivå. Den stopper ved nivå S_2, siden dette nivået inneholder alle målliteralene og de er ikke gjensidig utelukkende. Deretter kan den bruke backward søk for å finne den gyldige planen. Det er

viktig at dette søket ikke bryter noen gjensidige utelukkelser. Dette søket er basert på følgende problemdefinisjon:

- Initial tilstand: $Have(Cake)$
- Måltilstand: $Have(Cake) \wedge Eaten(Cake)$
- Handling:
  - $Eat(Cake,$
    $PRECOND: Have(Cake)$
    $Effect: \neg Have(Cake) \wedge Eaten(Cake))$
  - $Action(Bake(Cake),$
    $PRECOND: \neg Have(Cake)$
    $Effect: Have(Cake))$

Regresjonsplanleggeren begynner med måltilstanden $Have(Cake) \wedge Eaten(Cake)$, og gir:

4. Velger handling $Bake(Cake)$, siden denne har effekt $Have(Cake)$ som tilfredsstiller deler av målet. Prebetingelsen til denne handlingen er $\neg Have(Cake)$, så det nye målsettet blir: $\neg Have(Cake) \wedge Eaten(Cake)$
5. Velger handling$Eat(Cake)$, siden denne har effekt : $\neg Have(Cake) \wedge Eaten(Cake)$ som tilfredsstiller målet. Prebetingelsen til denne handlingen er $Have(Cake)$, så det nye målsettet blir: $Have(Cake)$. Siden dette er den initiale tilstanden har søket funnet en gyldig plan

Legg merke til at søket ikke vil velge handling $Eat(Cake)$ først, siden denne inneholder $\neg Have(Cake)$ som er en negasjon av en literal i målsettet. **Den gyldige planen er $\boldsymbol{Eat(Cake)}$ i $\boldsymbol{A_0}$ etterfulgt av $\boldsymbol{Bake(Cake)}$ i $\boldsymbol{A_1}$.**

Terminering av GRAPHPLAN

GRAPHPLAN vil terminere ved at den enten returnerer en gyldig plan eller *failure* hvis det ikke er noen løsning. Algoritmen vil returnere *failure* dersom både grafen og alle no-goods er flatet ut. Det er ikke tilstrekkelig å stoppe utvidingen av planleggergrafen dersom kun grafen flater ut, fordi dette kan skje før løsningen kan hentes ut dersom problemet involverer mye repetitiv handling (eks: flytte n laster fra A til B når hvert fly kan ta bare en last om gangen). Hvis søket ikke klarer å finne en løsning, må det ha vært minst ett sett med mållliteraler som ikke kunne nås og ble markert som no-goods. **Hvis det er færre no-goods i neste nivå, bør algoritmen fortsette fordi det betyr at den nærmer seg løsningen**. **Hvis både grafen og no-goods flater ut (dvs. ingen flere mållliteraler nås) kan algoritmen terminere, fordi videre endringer vil ikke føre til en løsning.**

Grafen og no-goods vil alltid flate ut, noe som skyldes at:

- **Antall literaler vil alltid øke**, siden alle literalene fra tidligere nivåer blir videreført av vedvarende handlinger (no-ops)
- **Antall handlinger vil alltid øke**, siden alle handlingene fra tidligere nivåer blir videreført som følge av at literalene blir videreført
- **Gjensidig utelukkelse vil alltid reduseres**. For literaler skyldes dette at antall måter å oppnå literalene øker, mens for handlingene skyldes dette at antall gjensidig utelukkelser mellom literalene reduseres.
- **Antall no-goods vil reduseres**, fordi antall mållliteraler er distinkt og dersom mållliteralene er oppnåelig ved ett nivå, vil det være oppnåelig ved senere nivå.

Siden antall handlinger og literaler øker og det er et endelig antall av disse, må det komme et nivå som har samme antall handlinger og literaler som forrige nivå. Siden antall gjensidig utelukkelser og no-goods reduseres og de kan ikke være mindre enn null, må det komme et nivå som har samme antall gjensidig utelukkelser og no-goods som forrige nivå. Ved dette nivået vil grafen og no-goods være flatet ut, og algoritmen vil returnere *failure*.

Andre klassiske planleggingsmetoder

De mest populære tilnærmingene for planlegging er:

- Oversettelse til Boolean satisfiable (SAT) problem
- Forward tilstandsrom søk med god heuristikk
- Søk vha planleggergraf

De to siste har vi allerede sett på, så vi skal nå se nærmere på den første og i tillegg noen andre tilnærminger for klassisk planlegging.

Oversettelse til boolean satisfiable (SAT) problem

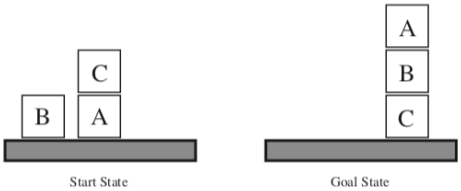
Et problem som er gitt i STRIPS representasjon kan oversettes til proposisjonslogikk, slik at det kan løses som et SAT problem. Dette innebærer proposisjonalisering av handlinger og mål, definering av initial tilstand og produksjon av ulike typer aksiomer

Situasjonsberegning

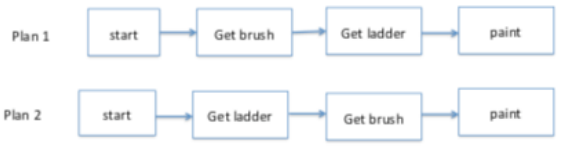
Første-orden logikk kan kombineres med situasjoner for å gi en representasjon som kalles situasjonsbergening. Ved situasjonsberegning tar man hensyn til at fakta gjelder i situasjoner og ikke uendelig lenge. Noe som er sant i en situasjon, trenger ikke nødvendigvis å være sant i en annen situasjon. **I situasjonsberegning blir situasjoner brukt for å ta hensyn til tidsaspektet ved fakta.** Situasjoner blir behandlet som objekter og kan dermed refereres til, for eksempel kan vi skrive $On(A, B, s_1)$, for å gi at A vil være på B i situasjon s_1. En $result$-funksjon vil returnere en ny situasjon, for eksempel vil $result\big(move(A, X, Y, s_0)\big)$ returnere den nye situasjonen s_1 som resulterer av å flytte entitet A fra X til Y i situasjon s_0.

I planlegging med situasjonsberegning kan logikk brukes for representasjon og resonnering (eks: resolusjon), men logikk er generelt en treg resonneringsmekanisme. I tillegg vil ikke-trivielle domener kreve et enormt antall rammeaksiomer. Som vi så på side 79 vil rammeproblemet være at aksiomer ikke gir hva som er uendret som resultat av en handling. For å løse dette problemet kan man lage rammeaksiom som spesifiserer hvilke proposisjoner som forblir de samme. I mange tilfeller er det ikke nødvendig med FOL som er veldig uttrykksfullt, fordi enklere planleggingsmekanismer kan heller bruke mer strukturerte og effektive logikkbaserte språk (eks: STRIPS). **Logikk kan brukes for å representere tilstander og handlinger, mens forward- og backward chaining kan brukes for å finne en plan**. **Planlegging vil da involvere å søke over et sett med tilstander.**

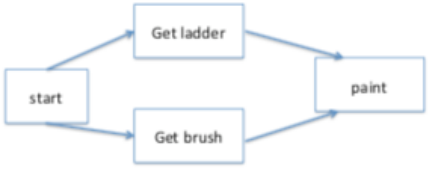
Partial order planning – total vs. delvis rekkefølge



Lineære planleggere bruker total rekkefølge og vil som regel dele målet inn i delmål. For eksempel for blokkverden kan målet: «plasser A på toppen av B som er på toppen av C» deles inn i delmålene: (1) få A på toppen av B og (2) få B på toppen av C. Figuren viser starttilstanden. Hvis vi starter med å oppnå delmål 1, kan vi ikke oppnå delmål 2, uten å angre det vi gjorde for å oppnå delmål 1. Hvis vi starter med å oppnå delmål 2, kan vi ikke oppnå delmål 1, uten å angre det vi gjorde for å oppnå delmål 2. Uansett hvilken rekkefølge vi gjør det i, vil interaksjonen mellom delmål forårsake rot. Dette kalles Sussman Anomaly, og det viser begrensningene ved ikke-flettet planlegging metoder. Vi skiller mellom:



- **Total rekkefølge** = planen er alltid en fast ordning av sekvensene. Dette brukes av progresjon- og regresjonsplanlegging, slik at de ikke kan ta nytte av problem dekomponeringen
- **Delvis rekkefølge** = planen kan være uordnet. Dette brukes av partial order planning som bruker minst-forpliktelse strategi som lar den utsette valget i løpet av søket



Partial Order Planning (POP)

Partial Order Planning (POP) er en form for planlegging basert på delvis rekkefølge. **Den søker i planrommet og bruker minste forpliktelse når det er mulig, noe som vil si at den vil kun gjøre valg som er relevante for å løse nåværende del av problemet**. To handlinger kan plasseres i en plan uten noe forpliktelse om hvilken som skal utføres først. I tilstandsromsøk vil søkerommet være et sett med tilstander av verden og handlinger forårsaker overganger mellom disse tilstandene. I dette tilfellet vil planen være en bane gjennom tilstandsrommet. **POP er basert på søk i planrommet der søkerommet er et sett med delvis ordnet planer og planoperatorer forårsaker overganger** (legger til handlinger og kausal koblinger, gi rekkefølge, osv.). I dette tilfellet er målet en lovlig plan. **POP prosessen er som følger:**

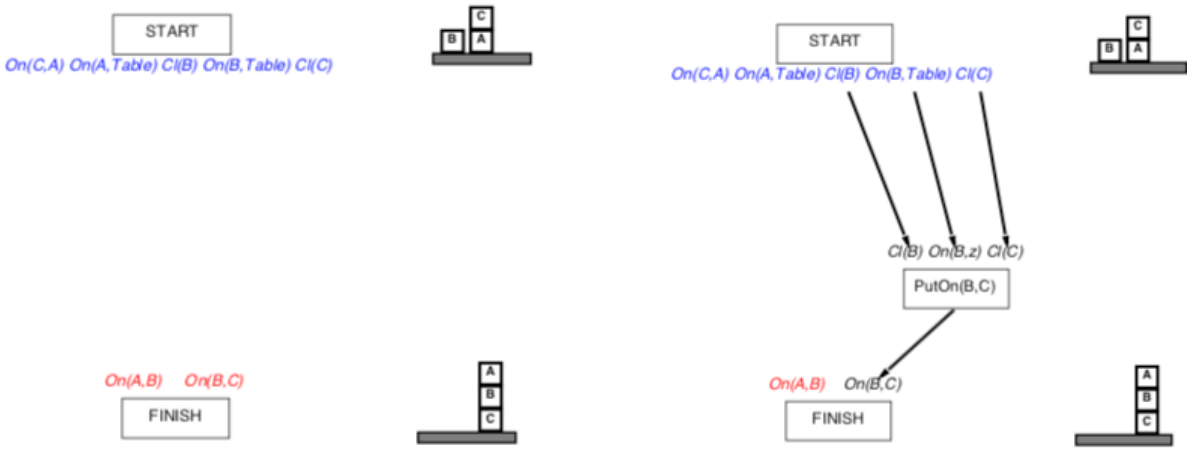
1. **Start med en tom plan** som består av et **startsteg** som har den initial tilstanden som effekt og **sluttsteg** som har måltilstanden som prebetingelse
2. Fortsett ved å:
    a. **Legg til handlinger som oppnår prebetingelser**
    b. **Legg til kausale koblinger fra en eksisterende handling for å oppnå prebetingelser**
    c. **Ordne handlingen mht. en annen for å fjerne mulige konflikter**

Dette vil gjøre at man **går gradvis fra en ufullstendig og vag plan til en komplett og korrekt plan**. Hvis man når en tilstand som er uoppnåelig eller en konflikt som ikke kan løses, kan man **backtracke**. Planen vil være komplett hvis alle prebetingelsene til sluttsteget er oppnådd (dvs. målet er tilfredsstilt). Prebetingelsen vil være oppnådd dersom den er effekten til et tidligere steg og ingen andre mellomliggende trinn opphever effekten.

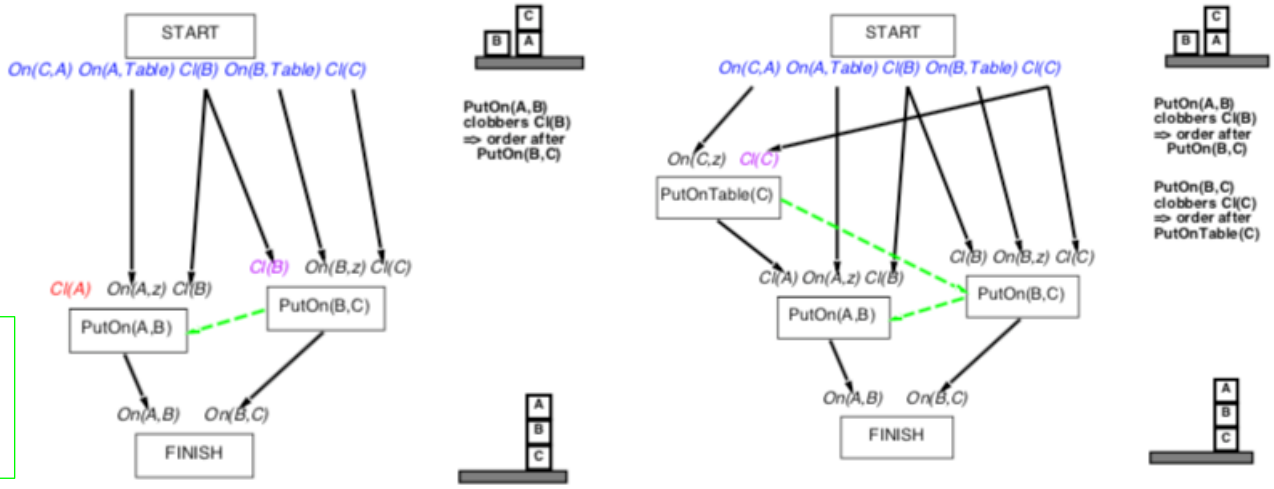
Vi ser igjen på blokkverden-problemet, som har følgende definisjon (merk litt annen enn den tidligere definisjonen):

- Initial tilstand: $On(C, A) \wedge On(A, Table) \wedge Clear(B) \wedge On(B, Table) \wedge Clear(C)$
- Måltilstand: $On(A, B) \wedge On(B, C)$
- Handling:
    - $Action(PutOn(x, y),$
      $PRECOND: Clear(x) \wedge On(x, z) \wedge Clear(y)$
      $Effect: \neg On(x, z) \wedge \neg Clear(y) \wedge Clear(z) \wedge On(x, y))$
    - $Action(PutOnTable(x),$
      $PRECOND: Clear(x) \wedge On(x, z)$
      $Effect: \neg On(x, z) \wedge Clear(z) \wedge On(x, Table))$

Det vil også være flere ulikhetsbegrensninger. Figurene under viser POP prosessen. For hver tilstand vil effekten skrives under boksen, mens prebetingelsen skrives over. Figuren til venstre viser den tomme planen som består av startsteget og sluttsteget. Figuren til høyre viser neste steg i prosessen der det legges til en handling med effekt som tilfredsstiller deler av prebetingelsen til sluttsteget og det kan lages en kausal kobling fra effekten til startsteget til prebetingelsen til handlingen.



Figuren til venstre viser neste steg i prosessen der det legges til en handling som oppnår resten av prebetingelsen til sluttsteget og det lages kausal kobling fra startsteget til prebetingelsen til handlingen. Siden $PutOn(A,B)$ vil gi $\neg Clear(B)$, vil den være i konflikt med $PutOn(B,C)$ som krever $Clear(B)$. For å løse denne konflikten vil $PutOn(A,B)$ ordnes sist i rekkefølgen av handlinger som utføres (siden det er effekten til denne som «ødelegger»). Figuren til høyre viser siste steg der det legges til en handling som oppnår prebetingelsen til $PutOn(A,B)$ og det lages kausal kobling fra startsteget til prebetingelsen til handlingen. Siden $PutOn(B,C)$ vil gi $\neg Clear(C)$, vil den være i konflikt med $PutOnTable(C)$ som krever $Clear(C)$. For å løse denne konflikten vil $PutOn(B,C)$ ordnes sist i rekkefølgen av handlinger som utføres (siden det er effekten til denne som «ødelegger»).



Merk: den grønne linjen markerer begrensninger på rekkefølgen av handlinger.

Fordeler med POP er at planstegene kan utføres uordnet, den kan håndtere samtidige planer, minst-forpliktelse kan føre til kortere søketider, det er komplett og solid og det vil som regel produsere den optimale planen. Ulempen er at komplekse planoperatorer kan føre til høye kostnader for å generere handlinger og samtidige handlinger fører til et større søkerom.

Planrom planlegging

En planleggergraf er en spesiell datastruktur som kan brukes for å gi et bedre heuristisk estimat. Denne heuristikken kan brukes av alle søkealgoritmene vi har sett på frem til nå eller av **algoritmen GRAPHPLAN** som kan søke etter løsningen i rommet som dannes av planleggergrafen.

Analyse av planleggingstilnærminger

Planlegging kombinerer to hovedområder innenfor AI, nemlig søk og logikk. En planlegger kan ses på som et program som søker etter en løsning eller forsøker å vise at det eksisterer en løsning. Planleggere blir stadig mer brukt i industrielle områder, men foreløpig har vi **liten forståelse for hvilke teknikker som fungerer best på ulike typer problemer**. Det kommer stadig nye teknikker som dominerer de eksisterende metodene. **Planlegging handler om å kontrollere en kombinatorisk eksplosjon.** Hvis det er n proposisjoner i et domene, vil det være 2^b tilstander. Planlegging er PSPACE-hardt, så teknikkene er avhengig av ulike «triks» for være effektive. Forward søk bruker god heuristikk, GRAPHPLAN bruker gjensidig utelukkelse (mutex) for å vise vanskelige interaksjoner, osv. Problemet har serialiserbare delmål dersom det eksisterer en rekkefølge for utføring av delmål som gjør at man slipper å angre effekten av delmålene. Dette vil gi en effektiv planlegging, siden det blir ingen backtracking.

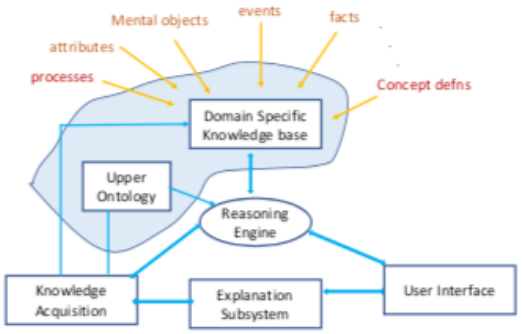
Planlegging og handling i den virkelige verden (F)

Planleggere som brukes i den virkelige verden kan være mer komplekse og involvere:

- **Begrensning på varighet og ressurser** = dette kan for eksempel være begrenset antall ansatte og begrenset tid.
- **Svært store tilstandsrom** = handlinger med lavnivå beskrivelse gir svært lange planer, men er alltid utførbare. Handlinger med høy-nivå beskrivelse gir forståelige planer, men kan være urealistiske.. Hierarki og abstraksjon kan gjøre planleggingen mer effektiv. Abstraksjon innebærer å se bort fra irrelevant informasjon. **Hierarchical Task Network (HTN)** vil dele handlingene inn i primitive handlinger som kan utføres direkte og høy-nivå handlinger som utgjør handlingssekvenser. Høy-nivå handlingene kan omformes til primitive handlinger eller andre høy-nivå handlinger (hvordan defineres av domeneeksperter og læres via erfaring). I dette tilfellet finner man planen ved å gjentatt omforme høy-nivå handlinger til man når primitive handlinger som kan utføres. Backtracking kan brukes hvis det blir nødvendig.
- **Usikkerhet** = skyldes ukorrekt og ufullstendig informasjon. I den virkelige verden vil ting som regel ikke være slik man forventer. Det kan være ufullstendig informasjon i form av ukjente prebetingelser eller disjunkte effekter, og ukorrekt informasjon i form av feil nåværende tilstand. Kvalifikasjonsproblemet gir at man vil aldri bli ferdig med å liste opp alle nødvendige prebetingelser og mulige betinget utfall ved handlinger. En løsningen er å bruke beredskapsplanlegging, der planen inkluderer observasjonshandlinger som oppnår informasjon og delplaner lages for hvert mulige utfall av observasjonene. Dette er kostbart fordi det planlegger for mange usannsynlige tilfeller. En annen løsning er re-planlegging, der man antar normale tilstander og sjekker progresjonen underveis i utføringen. Hvis det er nødvendig kan man starte planleggingen på nytt.

Kapittel 12 – Kunnskapsrepresentasjon

«Kunnskap er å vite at en tomat er en frukt, mens visdom er å ikke bruke tomat i en fruktsalat». I tidligere kapitler har vi sett på kunnskapsbaserte agenter, der vi har definert syntaksen, semantikken og inferens for proposisjonell og første-orden logikk. **Kunnskapsbaserte system** bruker fakta om verden, mentale objekter, attributter, prosesser, hendelser og konseptdefinisjoner for å lage en domenespesifikk kunnskapsbase. Denne kunnskapsbasen blir også oppdatert via **kunnskapstilegnelse**, som er en del av jobben til en kunnskapsingeniør som bestemmer innholdet og organiseringen av kunnskapen som KB trenger. Ved kunnskapstilegnelse vil denne kunnskapen hentes inn i KB via manuelt arbeid (samle kunnskap fra menneskelige eksperter) eller fra databaser, tekster og data. Kunnskapen i kunnskapsbasen blir igjen brukt av en resonneringsmaskin for å bestemme hvordan systemet skal oppføre seg

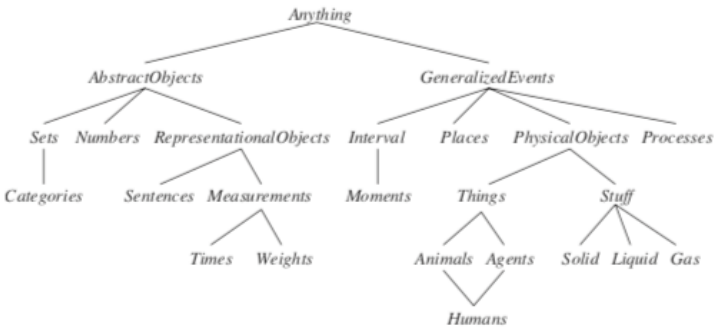


Dette kapittelet ser på hvilken type innhold man plasserer i kunnskapsbasen til en agent, altså hvordan man representerer fakta om verden.

Ontologisk engineering

I «leke»-domener vil ikke representasjonen være så viktig, fordi de fleste valgene vil fungere. **Komplekse domener krever mer generelle og fleksible representasjoner** (eks: shoppe på Internett, kjøre bil i trafikk). Dette kapittelet ser på hvordan vi kan lage representasjoner som er konsentrert på generelle konsepter som forekommer i mange ulike domener. Eksempler på generelle konsepter er hendelser, tid, fysiske objekter og tro. **Ontologisk engineering er representasjonen av slike abstrakte konsepter.** Det arbeider på en større skala enn kunnskapsingeniør.

Det er vanskelig å representere alt i verden, så i stedet kan man definere et generelt rammeverk for konsepter, kalt **øvre ontologi**, der ny kunnskap kan fylles inn senere. Det kalles øvre ontologi fordi det kan tegnes inn i en graf, der **de generelle konseptene er plassert på toppen og de mer spesifikke konseptene plasseres under**. Hver kobling indikerer at det lavere konseptet er en spesialisering av det øvre konseptet.



Dette kapitlet bruker første-orden logikk for å diskutere innholdet og organiseringen av kunnskap, selv om noen aspekter ved den virkelige verden er vanskelig å fange i FOL. Den største utfordringen er at de fleste generaliseringene har unntak eller er kun gyldig til en viss grad. For eksempel vil en nyttig regel være «Tomater er røde», men i noen tilfeller er tomater grønne, gule, osv. **Evnen til å håndtere unntak og usikkerhet er svært viktig, men det er ikke hovedfokuset ved å forstå generell ontologi.**

Hvis vi ser på ontologien til wumpus verden (s. 66), kan vi se at den representerer tid, men har en enkel struktur der ingenting skjer med mindre agenten handler og alle endringer er umiddelbare. En mer generell ontologi, som passer bedre til den virkelige verden, vil tillatte

samtidige endringer over tid, ulike typer pits med forskjellige egenskaper, flere dyr enn wumpus med ulike oppførsler, osv. Dette illustrerer hvordan man kan utføre endringer i en spesielt-formål ontologi for å bevege seg mot en mer generell ontologi. To hovedegenskaper ved generelt-formål ontologier som skiller dem fra spesielt-formål ontologier er:

1. **En generelt-formål ontologi bør kunne brukes i nesten alle spesielt-formål ontologier, ved å legge til noen domenespesifikke aksiomer**
2. **I tilstrekkelig utfordrende domener må ulike kunnskapsområder forenes, fordi resonnering og problemløsing kan involvere flere områder samtidig**. For eksempel et repareringssystem for roboter må kunne resonnere om kretser og tid, så setningene som beskriver tid må derfor kunne kombineres med de som beskriver romlig utforming av kretser

Kategorier og objekter

En viktig del av kunnskapsrepresentasjon er organiseringen av objekter inn i kategorier basert på egenskapene til objektene. Interaksjoner med verden foregår på nivået med individuelle objekter, mens resonneringen foregår på nivået med kategorier. For eksempel vil en shopper som regel ha mål om å kjøpe en basketball, og ikke en spesifikk basketball slik som BB_9. **Når et objekt har blitt plassert i en kategori basert på oppfattet egenskaper, kan informasjon om kategorien fra KB brukes for å lage prediksjoner om objektet**. For eksempel hvis objektet har rund form, rødt kjøtt, svarte frø, grønt og gult skall og ligger i fruktavdelingen, kan man utlede at objektet er en vannmelon og dermed bruke informasjonen om vannmelon for å utlede at det kan brukes i en fruktsalat. **Kategorier brukes for å organisere og forenkle kunnskapsbasen via arv.** Hvis vi sier at alle instanser i kategorien $Food$ er spiselig og vi gir at $Fruit$ er en subklasse av $Food$ og at $Apple$ er en subklasse av $Fruit$, kan vi utlede at alle epler er spiselige. Vi sier at eplene arver spiselig-egenskapen fra $Food$ kategorien. **Disse subklasse relasjonene vil organisere kategoriene inn i en taksonomi eller taksonomisk hierarki.**

I FOL kan kategorier representeres som predikater eller objekter. For eksempel kan basketballkategorien representeres som predikatet $Basketball(b)$ eller objektet $Basketballs$. FOL gjør det lett å gi fakta om kategorier, for eksempel:

- **Et objekt er medlem i en kategori**: $BB_9 \in Basketballs$ som betyr $Member(BB_{12}, Basketballs)$
- **En kategori er en subklasse av en annen kategori**: $Basketballs \subset Balls$ som betyr $Subcategory(Basketballs, Balls)$
- **Alle medlemmer i en kategori har noen egenskaper**: $(x \in Basketballs) \Rightarrow Bouncy(x)$
- **Noen egenskaper kan brukes for å gjenkjenne medlemmer av en kategori**: $Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5" \wedge x \in Balls \Rightarrow x \in Basketballs$
- **Kategorien har noen egenskaper:** $Dogs \in DomesticatedSpecies$

Merk: Vi bruker $Egenskap(x)$ og $x \in Kategori$

Siden $Dogs$ er en kategori og den er medlem av $DomesticatedSpecies$, så må den siste være en **kategori av kategorier**.

Kategoriene kan ha flere relasjoner i tillegg til subklasse- og medlemsrelasjonene. **To eller flere kategorier er disjunkte, dersom de ikke har noen felles medlemmer**. Et eksempel er kategoriene $Males$ og $Females$. **En uttømmende dekomponering (*exhaustive*) vil si at alle medlemmene i en kategori er dekket av et sett av kategorier**. For eksempel vil alle nordamerikanere være amerikaner, kanadier og/eller meksikaner (kan ha to statsborgerskap)**.** **En partisjon er en disjunkt, uttømmende dekomponering**. For eksempel vet vi at et dyr som ikke er hannkjønn må være hunnkjønn, så $Males$ og $Females$ er en partisjon av $Animals$. Dette kan uttrykkes formelt som:

$$Disjoint(\{Female, Male\})$$

$$ExhaustiveDecomposition(\{Americans, Canedians, Mexicans\}, NorthAmericans)$$
$$Partition(\{Males, Females\}, Animals)$$

Disse predikatene er definert som følger:

$$Disjoint(s) \Leftrightarrow (\forall c_1, c_2\ c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow Intersection(c_1, c_2) = \{\})$$
$$ExhaustiveDecomposition(s, c) \Leftrightarrow (\forall i\ i \in c\ \Leftrightarrow \exists c_2\ c_2 \in s \wedge i \in c_2)$$
$$Partition(s, c) \Leftrightarrow Disjoint(s) \wedge ExhaustiveDecomposition(s, c)$$

Kategorier kan defineres ved å gi nødvendige betingelser for medlemskap, for eksempel $x \in Bachelors \Leftrightarrow Unmarried(x) \wedge x \in Adults \wedge x \in Males$. **Mange kategorier har likevel ikke klare definisjoner**. For eksempel kan tomater være grønne, røde, gule, svarte, osv., men de er som regel runde og røde. En løsning på dette kan være å bruke kategorien $Typical(Tomatoes)$, slik at man får definisjonen:

$$x \in Typical(Tomatoes) \Rightarrow Red(x) \wedge Spherical(x)$$

Dette lar oss skrive ned nyttige fakta om kategorier uten å gi eksakte definisjoner.

Fysisk komposisjon

Vi bruker den generelle $PartOf$ relasjonen for å si at en ting er en del av en annen. Objekter kan grupperes i $PartOf$ hierarki, som ligner subklasse-hierarkiet:

$$PartOf(Oslo, Norway)$$
$$PartOf(Norway, Europe)$$
$$PartOf(Europe, Earth)$$
$$\dots$$

$PartOf$-relasjonen er transitiv og refleksiv, noe som betyr at:

$$PartOf(x, x)$$
$$PartOf(x, y) \wedge PartOf(y, z) \Rightarrow\ PartOf(x, z)$$

Sammensatte objekter vil ofte karakteriseres av strukturelle relasjoner blant deler. For eksempel vil biped ha to føtter som er koblet til en kropp:

$$Biped(a) \Rightarrow \exists l_1, l_2, b\ \ Leg(l_1) \wedge Leg(l_2) \wedge Leg(l_3) \wedge Body(b)$$
$$\wedge PartOf(l_1, a) \wedge PartOf(l_2, a) \wedge PartOf(b, a)$$
$$\wedge Attached(l_1, b) \wedge Attached(l_2, b)$$
$$\wedge l_1 \neq l_2 \wedge [\forall l_3\ Leg(l_3) \wedge PartOf(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)]$$

For at et objekt skal kunne være medlem av $Biped$ må det oppfylle alle betingelsene gitt av de strukturelle relasjonene. Sammensatte objekter kan også defineres som objekter som består av flere deler, uten noe bestemt struktur mellom delene. Dette kan representeres med $BunchOf$-relasjon, for eksempel for kan det sammensatte eplesekk-objektet representeres som:

$$BunchOf(\{Apple_1, Apple_2, Apple_3\})$$

Målinger

Målinger er verdier som tildeles ulike egenskaper ved objektene, for eksempel høyde, vekt, lengde, kostnad, osv. Kvantitative målingene kan representeres som **enhetsfunksjoner** der verdien gis som argument. For eksempel kan lengde defineres som $Length(L_1) =$

$Inches(1.5) = Centimeters(3.81)$. For å bytte mellom målingsenheter kan vi se på $Centimeters(2.54 \times d) = Inches(d)$, der vi bruker at 1 inch = 2.54 cm. Andre eksempler er:

$$Diameter(BB_{12}) = Inches(9.5)$$
$$ListPrise(BB_{12}) = \$(19)$$
$$d \in Days \Rightarrow Duration(d) = Hours(24)$$

Noen målinger har ingen skala, for eksempel skjønnhet, vanskelighetsgrad, osv. For slike målinger er det ikke mulig å gi numeriske verdier, men de kan ordnes i en rekkefølge. Selv om målinger ikke er nummer, kan de sammenlignes med symbolet $>$. For eksempel kan et emne ved NTNU være vanskeligere enn et annet: $Difficulty(TDT4136) > Difficulty(TDT4160)$.

Objekter – ting og sånt

Den virkelige verden består av primitive objekter (eks: atomer) og sammensatte objekter som er bygd fra disse. Ved å resonnere på nivået med store objekter (eks: epler og biler) unngår vi kompleksiteten som oppstår når man håndterer det enorme antallet primitive objekter individuelt. **En del av den virkelige verden vil likevel ikke kunne karakteriseres som individuelle objekter, og disse kalles sånt (*stuff*)**. For eksempel hvis man ser på smør, vil man ikke kunne si at det er et antall «smør-objekter» uten å gå ned på et svært lite nivå. Hvis man ser på en blyant kan man si at det er et «blyant-objekt», så dette er en **ting**. **I lingvistikk brukes tellesubstantiv for ting (eks: blyanter, bøker), mens massesubstantiv brukes for sånt (eks: vann, smør, melk).** I ontologien kan dette representeres på ulike måter. En løsning er å representere sånt som en kategori, for eksempel $Butter$. Slike kategorier vil ha **intrinsike egenskaper** som hører til substansen hos objektene (eks: farge, smeltepunkt, fettinnhold, osv.). De vil ikke ha **ekstrinsike egenskaper** som vekt, form, lengde, osv. Når du deler et objekt i to vil den bevare de intrinsike egenskapene, mens de ekstrinsike egenskapene blir endret. **Et sånt-objekt vil kun ha intrinsike egenskaper, mens en ting-objekt vil i tillegg ha noen ekstrinsike egenskaper.**

Hendelsesberegning

Situasjonsberegning brukes for å vise handlinger og deres effekter i en verden der handlingene er diskrete, umiddelbare og skjer en om gangen. Det kan ikke brukes for å beskrive kontinuerlige handlinger (dvs. skjer mer underveis) eller handlinger som utføres samtidig. For å håndtere dette introduserer vi **hendelsesberegning, som er basert på tidspunkt istedenfor situasjoner**. **Hendelsesberegning ser på det som skjer i løpet av handlingen**, og det bruker spesifikke definisjoner av objekt og hendelser. $At(Knut, NTNU)$ er et objekt som kun refererer til faktaen om at Knut er på NTNU og gir ikke om det er sant. For å bestemme om en fakta er sant ved et tidspunkt t bruker vi predikatet T som følger: $T(At(Knut, NTNU), t)$.

Hendelser er instanser av hendelseskategorier. For eksempel kan hendelsen E_1 av at Per flyr fra Værnes til Oslo beskrives som: $E_1 \in Flyings(Per, Værnes, Oslo)$. Vi kan deretter bruke $Happens(E_1, i)$ for å gi at hendelse E_1 ble utført over tidsintervallet i. Tidsintervall blir representert som $(start, end)$ par med tider. Det fullstendige settet med predikater i hendelsesberegning er:

- $T(f, t)$ = fakta f er sann ved tiden t
- $Happens(e, i)$ = hendelse e skjer ved tidsintervall i
- $Initiates(e, f, t)$ = hendelse e gjør at fakta f starter å gjelde ved tiden t
- $Terminates(e, f, t)$ = hendelse e gjør at fakta f slutter å gjelde ved tiden t

- $Clippes(f, i)$ = fakta f slutter å være sann i intervallet i
- $Restores(f, i)$ = fakta f starter å være sann i intervallet i

Hendelsen $Start$ vil beskrive den initiale tilstanden ved å si hvilke fakta som blir initialisert og hvilke som blir terminert ved starttidspunktet. Vi definerer T ved å si at en fakta vil gjelde ved et tidspunkt dersom den ble initialisert og har ikke blitt gjort falsk (*clipped*). Faktaen vil ikke gjelde hvis den ble terminert eller har ikke blitt gjort sann (*restored*). Det er nyttig å utvide predikatet T, slik at det gjelder over tidsintervaller og ikke bare tidspunkt; en fakta vil gjelde over et intervall dersom det gjelder ved hvert punkt i intervallet:

$$T\big(f, (t_1, t_2)\big) \Leftrightarrow [\forall t\ (t_1 \leq t < t_2) \Rightarrow T(f, t)]$$

Fakta og handlinger blir definert med domenespesifikke aksiomer. For eksempel vil følgende aksiom gi at eneste måte wumpus-verden agenten kan få en pil er ved starten:

$$Initiates(e, HaveArrow(a), t) \Leftrightarrow e = start$$

Denne måten å definere hendelser, gjør at vi kan legge til informasjon om hendelsene. For eksempel hvis flyturen til Per var humpete kan dette beskrives som $Bumpy(E_1)$. I en ontologi der hendelsene blir beskrevet som binære relasjoner, er det ingen måte å beskrive dette. **Hendelsesberegning kan i tillegg utvides for å representere samtidige utførte hendelser, kontinuerlige hendelser, osv.**

Prosesser

Diskrete hendelser har en endelig struktur, så hvis de deles opp vil det ikke være samme hendelse**. Prosesser** er hendelser som kan deles opp og det vil fortsatt være samme hendelse. En prosess som foregår over et intervall vil også foregå over et subintervall.

Tidsintervall

Hendelsesberegning gjør at vi kan snakke om tid og tidsintervaller. To ulike typer tidsintervall er øyeblikk (0 varighet) og utvidet intervaller. Vi bruker en tidsskala og lar øyeblikk representere punkter på denne skalaen. Tidsskalaen er vilkårlig, for eksempel kan den måles i sekunder og øyeblikket midnatt, 01.01.1990 kan ha tiden 0. Funksjonene $Begin$ og End vil gi de første og siste øyeblikkene ved intervallet, mens funksjonen $Time$ vil gi punktet på tidsskalaen for et øyeblikk. Funksjonen $Duration$ gir forskjellen i sluttid og starttid. To intervaller vil møtes (*meet*) hvis sluttiden til en er lik starttiden til den andre. Figuren viser det fullstendige settet av relasjoner mellom tidsintervaller.

```
Meet(i,j)      ⇔ End(i) = Begin(j)
Before(i,j)    ⇔ End(i) < Begin(j)
After(j,i)     ⇔ Before(i,j)
During(i,j)    ⇔ Begin(j) < Begin(i) < End(i)
Overlap(i,j)   ⇔ Begin(i) < Begin(j) < End(i)
Begins(i,j)    ⇔ Begin(i) = Begin(j)
Finishes(i,j)  ⇔ End(i) = End(j)
Equals(i,j)    ⇔ Begin(i) = Begin(j) ∧ End(i)
```

Fakta og objekter

Fysiske objekter kan ses som generaliserte hendelser, siden de er en del av romtiden. For eksempel kan USA ses på som en hendelse som begynte i 1776 som en union av 14 stater. For å beskrive endringene i egenskapene til USA kan vi bruke tilstandsfakta, for eksempel $Population(USA)$. Objektet $President(USA)$ vil bestå av ulike mennesker ved ulike tidspunkt. For å si at George Washington var president i 1790, bruker vi:

$$T(Equals(President(USA), GeorgeWashington), AD1790)$$

Mentale hendelser og objekter

Kunnskapsbaserte agenter kan ha trotilstander og utlede nye trotilstander, men de har ingen kunnskap *om* trotilstandene eller *om* utledingen. Kunnskap om sin egen kunnskap og resonneringsprosess er nyttig for å kontrollere utledningen. For eksempel hvis spørsmålet er «Hva er kvadratroten av 1764?» vil man kunne svare på spørsmålet hvis man tenker hardt, mens hvis spørsmålet er «Sitter Erna Solberg nå?» vil det ikke hjelpe å tenke hardere. Det er også viktig å ha kunnskap om kunnskapen til andre agenter, for eksempel at Erna Solberg vil vite om hun sitter eller ikke, så derfor kan vi finne svaret ved å spørre henne.

Vi trenger en modell av de mentale objektene som er i hodet til noen og de mentale prosessene som manipulerer disse objektene. Denne modellen er gitt i modal logikk. For å illustrere behovet for denne logikken, ser vi på de ulike proposisjonelle holdninger agentene kan ha mot mentale objekter, for eksempel tro, vite, ønske, osv. Disse holdningene oppfører seg ikke som normale predikater, for eksempel kan følgende gi at Lois vet at Superman kan fly:

$$Knows\big(Lois, CanFly(Superman)\big)$$

Vi vil normalt tenke på $CanFly(Superman)$ som en atomisk setning, men her blir det brukt som en term. Problemet er at hvis «Superman er Clark Kent» er sant, vil inferensreglene konkludere med at «Lois vet at Clark kan fly», noe som ikke er tilfellet. Dette kalles referensiell transparent, og det ser kun på objektet som termen refererer til og ikke på selve termen. For holdninger som «tro» og «vite» ønsker vi at termen som brukes skal være viktig. Dette kan løses ved å bruke modal logikk.

Vanlig logikk er basert på sannhetsmodaliteten som lar oss uttrykke «P er sann». **Modal logikk har en spesiell operator som kan brukes for å si at en agent har kunnskap om noe.** $K_A P$ brukes for å uttrykke at agent A vet setning P. Eks: Bond vet at noen er en spion = $\exists x\ K_{Bond} Spy(x)$. Modellen hos modal logikk involverer alle mulige verdener som er koblet sammen av hva agentene vet om verden. **Agenter kan bruke denne modellen for å resonnere rundt sin egen kunnskap og kunnskapen til andre agenter.** Se side 451-453 for mer grundig forklaring ☺

Resonneringssystem for kategorier

Kategorier er grunnleggende byggesteiner i kunnskapsrepresentasjon Det er to ulike typer system som brukes for å organisere og resonnere med kategorier:

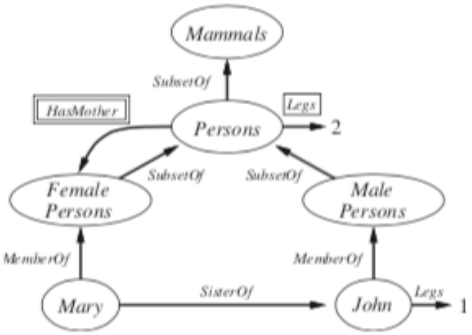
- **Semantiske nettverk** = gir grafiske virkemidler som visualiserer en kunnskapsbase og effektive algoritmer for å utlede egenskaper ved objektet basert på hvilken kategori det er medlem av
- **Deskriptiv logikk** = gir et formelt språk for å lage og kombinere kategoridefinisjoner og effektive algoritmer for å bestemme sub- og superklasse-relasjoner mellom kategorier

Vi skal se nærmere på disse to.

Semantiske nettverk

Hovedideen ved semantiske nettverk er at kunnskap vil være store deler som er koblet sammen istedenfor store samlinger av små deler. Betydningen til konsepter vil komme av hvordan de er koblet til andre konsepter. For å bestemme egenskapene til objekter brukes det effektive inferensalgoritmer som ser på arv. Det finnes flere varianter av semantiske nettverk, men alle representerer individuelle objekter, kategorier for objekter og relasjoner blant objekter. Som regel vil objekt- eller kategorinavn plasseres i ovaler eller bokser, og de

kobles sammen av navngitte koblinger. For eksempel på figuren kan vi se at det er en $MemborOf$ kobling mellom $Mary$ og $FemalePerson$, som korresponderer til den logiske påstanden $Mary \in FemalePerson$. På lignende måte vil $SisterOf$ koblingen mellom $Mary$ og $John$ korrespondere til påstanden $SisterOf(Mary, John)$. Kategorier kan kobles sammen vha $SubsetOf$-koblinger



Det er viktig å passe på hva som er objekter og hva som er kategorier. Vi vet at personer har kvinnelige personer som mødre, men dette kan ikke vises som en kobling mellom $Persons$ og $FemalePersons$ fordi disse er kategorier og har dermed ikke mødre. I stedet kan vi bruke en spesiell notasjon som vises med dobbelt boks. Denne koblingen gir:

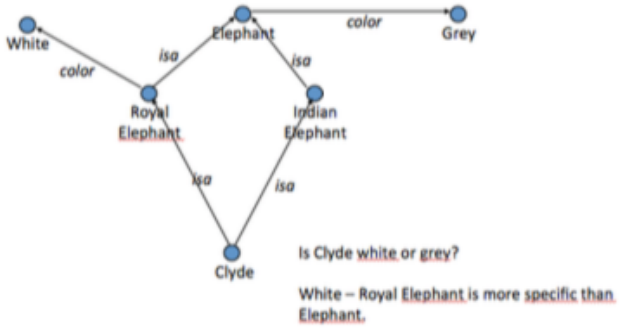
$$\forall x \ x \in Persons \Rightarrow [\forall y \ HasMother(x, y) \Rightarrow y \in FemalePersons]$$

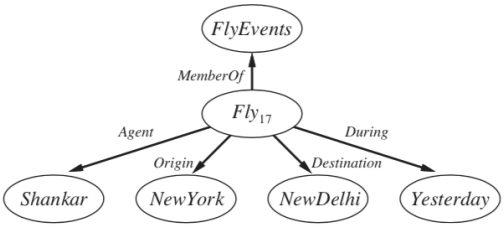
Denne notasjonen brukes for å gi at det er medlemmer av kategoriene som har disse egenskapene og ikke kategoriene. For å gi at alle medlemmer i en kategori har en egenskap, kan vi bruke en boks som er festet til en pil som går ut av kategorien. For eksempel kan dette brukes for å gi at personer har to føtter:

$$\forall x \ x \in Persons \Rightarrow Legs(x, 2)$$

På figuren kan vi se at dette er gitt som *Legs*-boksen som går ut av *Persons*. Tenk at en boks med én linje brukes for å vise egenskapen til alle medlemmene av én kategori, mens en boks med to linjer brukes for å vise egenskapen til alle medlemmene av to kategorier.

Semantisk nettverk gjør det enkelt for en inferensalgoritme å resonnere rundt arv. For eksempel kan algoritmen utlede at $Mary$ har to føtter, ved å følge $MemberOf$ koblingen fra $Mary$ til kategorien hun hører til, og deretter følge $SubsetOf$ koblingene oppover hierarkiet til den finner kategorien som har en Leg-kobling, som i dette tilfellet er $Persons$ kategorien. **Effektiviteten og enkelheten til denne inferensmekanismen har gjort at semantiske nettverk er populære**. **Arv blir mer komplisert ved multiple arv, der et objekt kan høre til flere kategorier eller en kategori kan være et subsett av flere kategorier.** I slike tilfeller kan inferensalgoritmen finne to eller flere svar, og modellpreferansen brukes for å bestemme blant disse (eks: religiøs tro kan settes før politisk tro). **I tillegg vil inferensalgoritmen aldri gå fra en spesifikk kategori til en mer generell kategori,** så i tilfellet på figuren vil svaret være *white*.





***n*-ære relasjoner kan ikke representeres direkte i et semantisk nettverk, men vi kan oppnå effekten ved å representere proposisjonen som en hendelse som hører til en hendelseskategori.** For eksempel for $Fly(Shnakar, NewYork, NewDelhi, Yesterday)$ får vi det semantiske nettverket på figuren til venstre.

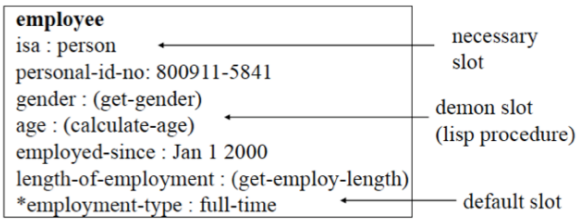
Noen typer universell kvantifisering kan representeres med inverse koblinger og enkle og doble bokser, men det er fortsatt mye som mangler for å nå nivået til første-orden logikk. For eksempel vil ikke det semantiske nettverket gi negasjon, disjunksjon, eksistensiell kvantifisering, osv. **Det er mulig å utvide notasjonen slik at den blir ekvivalent med første-**

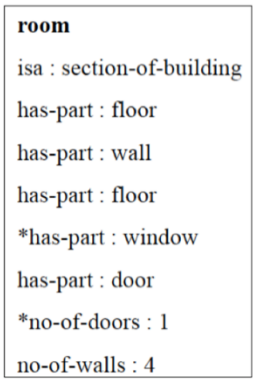
orden logikk, men da mister man en av hovedfordelene ved semantiske nettverk, nemlig at det er enkelt og transparent for inferensprosessene.

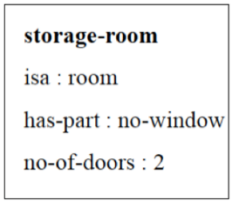
Semantiske nettverk bruker ***procedural attachement*** for å fylle inn gap. Når det mottas en query om en bestemt relasjon vil det føre til at det kalles på en prosedyre som er spesifikt designet for denne relasjonen. **En av de viktigste egenskapene ved semantiske nettverk er deres evne til å representere standardverdier for kategorier**. For eksempel kan vi se på figuren på forrige side at John har 1 fot, selv om han er en person og alle personer har to føtter. I en logisk KB vil dette være en motsigelse, men i semantiske nettverk vil påstanden om at alle mennesker har to føtter bare være en standardverdi. En person antas å ha to føtter med mindre det oppgis mer spesifikk informasjon. Dette følges av inferensalgoritmen, siden den vil stoppe med en gang den finner en verdi. Standarden blir overskrevet av en mer spesifikk verdi.

Rammebaserte representasjoner (F)

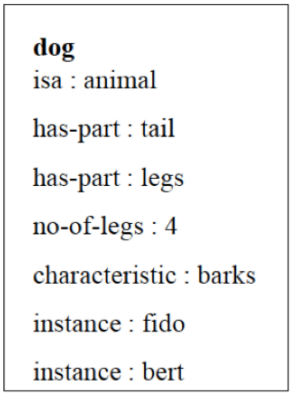
Rammebaserte representasjoner brukes for å representere noder i det semantiske nettverket. En ramme består av et antall spalter (eks: alder, høyde, osv.). Hver spalte lagrer informasjon i form av en variabel (egenskap) som har en verdi, og når agenten møter nye situasjoner kan spaltene fylles inn. Når spalter blir fylt inn kan det igjen trigge nye handlinger som kan fylle andre spalter. Det er arv av egenskaper mellom rammer. Dette ligner objekter i objektorientert programmering. På figuren kan vi se at isa er en nødvendig spalte som gir hva rammen representerer. Spalter med standardverdier representeres med *. Demon-spalter gir variabler som får verdi av prosedyrer som kan aktiveres. Figurene under viser eksempler på noen rammer:

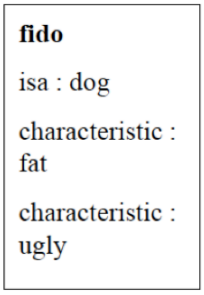






here **storage-room** inherits from room, but overrides the default values of has-part : window and no-of-doors : 1





fido inherits from 'dog'

38

f

Deskriptiv logikk

Syntaksen til FOL er designet for å gjøre det enklere å si ting om objekter. Deskriptiv logikk er på samme måte designet for å gjøre det enklere å beskrive definisjoner og egenskaper ved kategorier. De viktigste inferensoppgavene innenfor deskriptiv logikk er **subsumsjon** (sjekke om en kategori er et subsett av en annen ved å sammenligne deres definisjoner) og **klassifisering** (sjekke om et objekt hører til en kategori). Noen systemer inkluderer i tillegg sjekk av **konsistensen** til kategoridefinisjonen, dvs. om medlemskapskriteriene er satisfiable (dvs. kan tilfredsstilles**). Et eksempel på deskriptiv logikk er CLASSIC språket, der syntaksen er gitt på figuren.** For eksempel for å skrive at bachelors er ugifte, voksne menn, får vi:

$$Bachelor = And(Unmarried, Adult, Male)$$

I første-orden logikk vil dette tilsvare:

$$\forall x \; Bachelor(x) \Leftrightarrow Unmarried(x) \wedge Adult(x) \wedge Male(x)$$

Deskriptiv logikk har operatorer som kan brukes på predikater, noe som ikke er mulig i FOL. **Alle beskrivelser i CLASSIC kan oversettes til FOL, og noen beskrivelser er enklere å gi i CLASSIC**. For eksempel for å beskrive settet av menn med minst tre sønner som alle er arbeidsledig og gift med doktorer, og har maks to døtre som er alle professorer i fysikk eller matte, kan vi bruke:

$$And(Man, AtLeast(3, Son), AtMost(2, Daughter),$$
$$All(Son(And(Unemployed, Married, All(Spouse, Doctor))),$$
$$All(Daughter, And(Professor, Fills(Department, Physics, Math))))$$

Denne beskrivelsen vil være utfordrende å gi i FOL, siden den krever mye kvantifisering. **En av de viktigste fordelene ved deskriptiv logikk er at det gir sporbar inferens**. Et problem løses ved å beskrive det og deretter spørre om det er subsumert av en av kategoriene i løsningen. Subsumsjonstest kan utføres i polynomial tid i størrelsen til beskrivelsen (motsetning til FOL der det ofte er umulig å forutsi løsningstid). Dette gi derimot innebære at vanskelige problem kan ikke oppgis eller krever eksponentiell store beskrivelser. Sporbarheten vil likevel vise hvilke konstruksjoner som forårsaker problem og dermed gjøre det lettere å forstå hvordan ulike representasjoner oppfører seg. Deskriptiv logikk vil for eksempel ofte mangle negasjon og disjunksjon. Begge disse kan gjøre at FOL systemer må gå igjennom eksponentielle analyser for å sikre kompletthet. I CLASSIC er det kun tillatt å bruke disjunksjon i $Fills$ og $OneOf$. Disjunksjon kan føre til et eksponentialt antall alternative ruter der en kategori kan subsumere en annen.

Resonnering med standardinformasjon

I forrige seksjon så vi hvordan påstanden «Mennesker har to føtter» hadde standardstatus. Denne standarden kan overskrives av mer spesifikk informasjon. Arvmekanismene i semantiske nettverk implementerer dette på en enkel og naturlig måte. Denne seksjonen ser mer generelt på standarder og forsøker å forstå semantikken til standarder.

Circumscription og standardlogikk

Logikk har monotoniegenskapen som gir at alle setninger som følger av KB vil være bevart etter at nye setninger blir lagt til (dvs. hvis $KB| = \alpha$, så vil $KB \wedge \beta| = \alpha$). Dette gjør at settet av setninger som følger av KB kan bare øke. I dette kapittelet har vi sett et unntak til denne egenskapen, der en egenskap som arves av alle medlemmer i en kategori i et semantisk nettverk kan overskrives av en mer spesifikk informasjon i subklasse kategorien.

Ved ikke-monotoni logikk vil settet av trotilstander ikke vokse over tid ettersom nye bevis ankommer. Resonneringen innebærer en standard konklusjon som vil utføres dersom det ikke er noen grunn til å tvile (eks: konkludere at bilen har fire hjul, selv om man ser bare tre). Hvis nye bevis ankommer kan konklusjonen trekkes tilbake (eks: ser eier bære det fjerde hjulet). Ikke-monotone logikker har annen definisjon av sannhet og følger for å fange slik oppførsel. To typer ikke-monotone logikker er:

- **Circumscription** = en kraftigere og mer presis versjon av lukket-verden-antagelsen. Det blir spesifisert predikater som antas å være så falske som mulige, dvs. falske for alle objekter bortsett fra dem det er gitt at de er sanne. For eksempel kan vi påstå standard regel at alle fugler kan fly: $Bird(x) \wedge \neg Abnormal(x) \Rightarrow Flies(x)$. Her vil $Abnormal$ være circumscribed (avgrenset), slik at man kan anta at $\neg Abnormal(x)$ er sann hvis ikke det er kjent at $Abnormal(x)$ er sann. Dette gjør at man kan trekke konklusjonen $Flies(Tweety)$ fra premisset $Bird(Tweety)$ så lenge det ikke er oppgitt at $Abnormal(Tweety)$ er sann. Alternativt kan vi bruke inferensregler for å utlede $Abnormal$: $Bird(x) \wedge \neg Flies(x) \Rightarrow Abnormal(x)$

- **Standardlogikk** = standardregler brukes for å lage betinget og ikke-monotone konklusjoner. For eksempel vil $Bird(x)$: $Flies(x)/Flies(x)$, bety at hvis $Bird(x)$ er sann og $Flies(x)$ er konsistent med KB, kan $Flies(x)$ konkluderes som standard. Standardregler vil generelt ha formen P: $J_1, \dots, J_n/C$, der P er forutsetning, C er konklusjonen og J_i er begrunnelser. Dersom man ikke kan vise at P og $J_1, \dots, J_n$ er falsk, kan man trekke konklusjonen. Enhver variabel som er i J_i eller C, må også være i P.

Sannhetsvedlikeholdssystem

Mange av utledningene vil ha standardstatus istedenfor å være helt sikre. Utledet fakta kan derfor være feil og må trekkes tilbake i møte med ny informasjon (eks: bilen har ikke fire hjul, siden eier bærer ett av hjulene). Denne prosessen kalles trosrevisjon. Anta at KB inneholder en setning P og vi ønsker å utføre $TELL(KB, \neg P)$. For å unngå at det blir en motsigelse i KB, må vi først utføre $RETRACT(KB, P)$. Problemet er at KB kan inneholde andre setninger som følger fra P. For eksempel kan implikasjonen $P \Rightarrow Q$ ha blitt brukt for å legge til Q i KB. Det vil heller ikke være riktig å fjerne alle setninger som følger av P, for det kan hende setningene følger av andre setninger enn P (eks: $R \Rightarrow Q$).
Sannhetsvedlikeholdssystem er designet for å håndtere disse komplikasjonene.
Det finnes ulike typer sannhetsvedlikeholdssystem (TMS). **En enkel tilnærming er å holde styr over rekkefølgen setningene ble lagt til i KB, ved å nummerere dem fra P_1 til P_n. Når systemet mottar $RETRACT(KB, P_i)$ vil systemet gå tilbake til tilstanden rett før P_i ble lagt til, slik at P_i og alle setningene som følger av P_i blir fjernet**. Dette er enkelt og garanterer at KB er konsistent, men det krever at $n - i$ setninger blir vurdert på nytt. Dette er upraktisk i systemet med store KB. **En annen tilnærming er JTMS, der hver setning merkes med setningene den ble utledet av (= justifikasjon).** For eksempel hvis KB inneholder $P \Rightarrow Q$ og $TELL(P)$ utføres, vil det føre til at Q legges til i KB med justifikasjon $\{P, P \Rightarrow Q\}$. Dette gjør at trosrevisjonen blir effektiv, fordi $RETRACT(P)$ vil føre til at alle setninger som kun utledes av P også vil fjernes. Dette gjør at Q vil fjernes dersom den har justifikasjon $\{P, P \Rightarrow Q\}$ eller $\{P, P \vee R \Rightarrow Q\}$, men den vil ikke fjernes dersom den har justifikasjon $\{R, P \vee R \Rightarrow Q\}$. Tiden som trengs for å fjerne P vil derfor kun avhenge av antall setninger som er utledet fra P og ikke alle setninger lagt til etter P. JTMS vil markere setninger som *in* eller *out* istedenfor å fjerne de fra KB. Dersom setningen blir lagt til på nytt vil dette gjøre at den slipper å utføre inferensprosessen på nytt.

TMS kan også brukes for å gjøre analysen av flere hypotetiske situasjoner raskere. For eksempel i valg av lokasjoner for OL kan første hypotese være $Site(Swimming, Pitesti)$, $Site(Athletics, Bucharest)$ og $Sites(Equestrian, Arad)$. Det må utføres en god del resonnering for å finne de logiske konsekvensene ved dette valget. Hvis vi ønsker å se på $Site(Athletics, Sibiu)$ i stedet, vil TMS gjøre at vi slipper å starte på nytt, siden det lar oss fjerne $Site(Athletics, Bucharest)$ og sette inn $Site(Athletics, Sibiu)$. TMS vil håndtere de nødvendige revisjonene og inferenskjeder som ble laget for Bucharest kan brukes på nytt for Sibiu, så lenge konklusjonene er like.

Beregningskompleksiteten til TMS er NP-hard, men dersom de brukes riktig kan de gi en betraktelig økning i logiske systemers evne til å håndtere komplekse omgivelser og hypoteser.

Multiagent system og spillteori

Denne delen av kompendiet er basert på forelesningsnotatene og er ikke dekket av boka. **Agentdesign blir utført på et mikronivå**, der vi ser på hvordan vi kan bygge agenter som kan utføre uavhengige, autonome handlinger for å suksessfullt utføre oppgaver som vi delegerer til dem. **Samfunnsdesign blir utført på et makronivå**, deri vi ser på type og egenskaper ved interaksjoner mellom agenter, hvordan agenter oppfører seg i samfunn og hvordan de samarbeider, konkurrerer og koordinerer med andre agenter.

Spillteori – en introduksjon

Spillteori ser på hvilke valg agenten tar i multiagent omgivelser, der utfallet til en handling utført av agenten vil avhenge av utfallet til handlingene som utføres av andre agenter. Dette fører til at agenter må utføre strategiske avgjørelser. **Strategisk resonnering** vil si at avgjørelsen agenten tar vil avhenge av hva den tror de andre agentene vil gjøre. Agenten er usikker om avgjørelsene til de andre deltagende agentene. Den vil prøve å forutsi deres neste avgjørelse og regne ut hvordan dette vil påvirke den. Basert på dette vil agenten utføre sin avgjørelse. Ulike typer spill er:

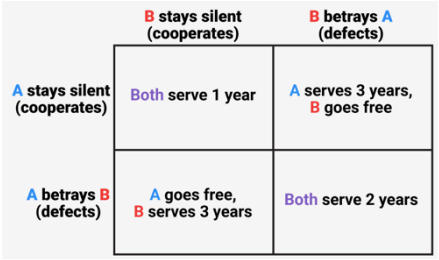
- **Samtidige/strategisk normalform spill**
- **Sekvensielle (omfattende form) spill**
- **Gjentagelsesspill**

Vi skal nå se nærmere på disse

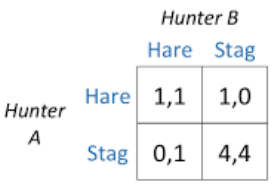
Samtidige/strategiske normalform-spill

I samtidig/strategiske normalform-spill vil agenten velge strategi en gang ved starten av spillet og alle agenter utfører handlingen samtidig (= one-shot). Hovedproblemet er å forutsi hvordan de andre agentene vil spille. **Slike spill blir representert som en payoff matrise**. Eksempler på samtidige/strategiske normalform-spill er:

- **Prisoners dilemma** = to menn er samlet siktet for en forbrytelse og blir holdt i separate celler. Det er tre mulige utfall: (1) den ene tilstår og den andre gjør ikke det, slik at den som tilstår blir frigjort og den andre må sitte i fengsel i tre år, (2) begge tilstår, slik at begge må sitte i fengsel i to år og (3) ingen tilstår, så begge må sitte i fengsel i ett år. Begge er rasjonelle og begge vet at den andre er rasjonell. Det er ingen kommunikasjon mellom partene, så de kan ikke bli enige

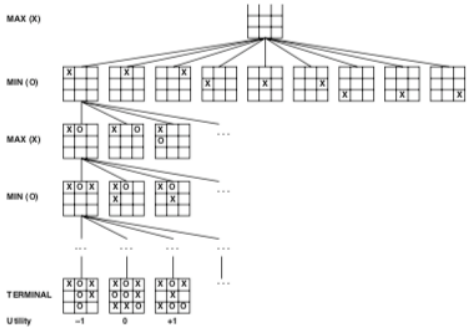


- **Hjortejakt** = beskriver en konflikt mellom sikkerhet og sosialt samarbeid, noe som kalles et tillitsdilemma. To individer drar ut på jakt. Hver av dem kan velge å jakte en hjort eller en hare. Hver spiller må velge en handling uten å vite hva den andre velger. Hvis et individ jakter en hjort, må han samarbeide med partnere for å lykkes. Hvis et individ jakter en hare, kan han lykkes alene, men haren er verdt mindre enn en hjort. Dette er en viktig analogi for sosialt samarbeid: hver agent vil gjøre det den andre gjør, som kan være annerledes enn det de sier de vil gjøre.



- **Kjønnskampen** = Jonathan og Sofie er gift og de er på kontoret fredagskveld og prøver å finne ut hva de skal gjøre etter jobb. De kan ikke komme i kontakt med hverandre, men ønsker å møtes og dra på kino. Jonathan har mest lyst til å se en actionfilm, mens Sofia har mest lyst til å se en romantiskfilm. Det viktigste er at de gjør noe sammen, så kvelden er «bortkastet» hvis de ikke ser samme film.

Sekvensielle (omfattende form) spill



I sekvensielle spill vil agenten kunne bestemme eller endre strategi ved hvert punkt i spillet. Eksempler er sjakk, tre på rad og Go. **Slike spill blir representert som et tre**. Figuren viser hvordan tre på rad kan representeres som et tre.

Gjentagelsesspill

I gjentakelsesspill vil agenten huske tidligere oppførsel hos andre agenter og velger strategi basert på dette. Hovedutfordringen er utviklingen av samarbeid, læring og motstandermodellering. Eksempler er Prisoners dilemma som blir spilt gjentatte ganger

Antagelser i spillteori

Viktige antagelser er:

- **Agentene er rasjonelle** = de har et veldefinert mål blant settet av utfall og velger handlinger som fører til at de oppnår dette målet
- **Agentene har felles kunnskap** = de kjenner reglene til spillet og de vet at andre agenter er rasjonelle og at de andre agentene vet at de er rasjonelle

Dette er viktig for å kunne lage en strategi, fordi det betyr at de andre agentene ikke vil oppføre seg tilfeldig.

Strategiske omgivelser og spill

agent j

agent i	action1	action2
action1	1 1	4 1
action2	1 4	4 4

Det er tre hovedkomponenter i strategiske omgivelser:

- **Spillere (agenter)**
- **Strategier og handlinger**
- **Payoff (*utility*)**

For å representere payoff hos ulike spillere kan man bruke en matrise (se figur).

For eksempel for et samtidige/strategiske normalform spill vil de tre komponentene være:

- **Spillere (agenter)** = $N > 1$
- **Strategier og handlinger** = hver agent i velger en handling a_i fra sitt eget sett av handlinger A_i. Vektoren $a = (a_1, \dots, a_n)$ består av de individuelle handlingene til agentene og kalles felleshandlingen (*joint action*). Settet A består av alle felleshandlingene
- **Payoff (*utility*)** = hver agent i har sin egen utilityfunksjon $u_i(a)$ som måler hvor god felleshandlingen er for dem. Hver agent kan gi ulike preferanser til ulike felleshandlinger. En payoff matrise gir utility for hver felleshandling hos hver agent.

Tilstand transformeringsfunksjon

Det er to agenter, $Agent = \{i, j\}$ **Alle mulige utfall i systemet er $\Omega = \{\omega_1, \omega_2, \dots\}$, der ω representerer et utfall som skyldes en samling handlinger, en for hver agent.** For å finne de mulige utfallene må vi se på **tilstandstransformeringsfunksjonen**:

$$\tau: A_i \times A_j \to \Omega \quad \text{som betyr} \quad \tau(a_i, a_j) = \omega$$

Denne gir at utfallet ω vil avhenge av kombinasjonen av handlingene som utføres av agent i og j. **Utilityfunksjonen beskriver hvilke utfall agenten foretrekker**: $u: \Omega \to R$.
Utilityfunksjoner kan derfor brukes for å ordne preferansene over utfall:

$$\omega \geq_i \omega' \quad \text{hvis} \quad u_i(\omega) > u_i(\omega')$$

Der u_i er utilityfunksjonen til utfall ω for agent i. Utility (payoff) kan altså brukes for å bestemme hvilket utfall agenten foretrekker og dermed hvilke handlinger den bør utføre.

Anta at hver agent har kun to mulige handlinger som de kan utføre: C og D. Utfallet vil kunne avhenge av handlingene til begge, ingen eller en av agentene. Tabellen viser transformasjonsfunksjonene i de tre ulike omgivelsene.

Avhenger av begge	Avhenger av ingen	Avhenger av én (her: j)
$\tau(D,D) = \omega_1$ $\tau(D,C) = \omega_2$ $\tau(C,D) = \omega_3$ $\tau(C,C) = \omega_4$	$\tau(D,D) = \omega_1$ $\tau(D,C) = \omega_1$ $\tau(C,D) = \omega_1$ $\tau(C,C) = \omega_1$	$\tau(D,D) = \omega_2$ $\tau(D,C) = \omega_1$ $\tau(C,D) = \omega_2$ $\tau(C,C) = \omega_1$

Dersom vi har tilfellet der begge agentene vil påvirke utfallet og $Action = \{C, D\}$, vil det være fire mulige utfall som kan produseres av systemet:

$$\tau(D,D) = \omega_1$$
$$\tau(D,C) = \omega_2$$
$$\tau(C,D) = \omega_3$$
$$\tau(C,C) = \omega_4$$

Dersom agentene har følgende utilityfunksjon:

- Agent i: $u_i(\omega_1) = 1$, $u_i(\omega_2) = 1$, $u_i(\omega_3) = 4$ og $u_i(\omega_4) = 4$
- Agent j: $u_j(\omega_1) = 1$, $u_j(\omega_2) = 4$, $u_j(\omega_3) = 1$ og $u_j(\omega_4) = 4$

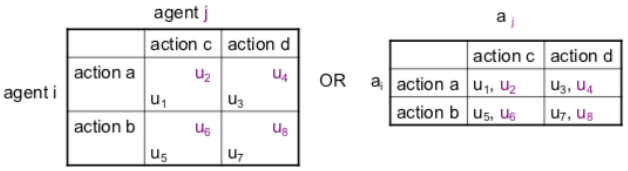
Så vil preferansene til agentene kunne ordnes som følger:

- Agent i: $\omega_4 \succeq_i \omega_3 \succ_i \omega_2 \succeq_i \omega_1$
- Agent j: $\omega_4 \succeq_i \omega_2 \succ_i \omega_3 \succeq_i \omega_1$

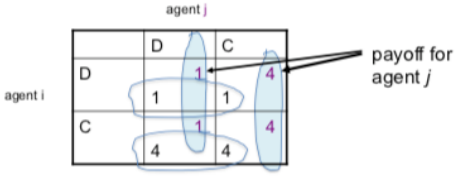
Hvis i er en rasjonell agent, vil den velge å utføre handling C siden dette gir bedre utilityfunksjon og foretrukket utfall (ω_4 eller ω_3).

Payoff-matrise

Payoff-matrise brukes for å vise utilityfunksjonen for hver agent og for hver felleshandling. Payoff u_1 er utility for agent i når agent i velger handling a og agent j velger handling c. Payoff u_2 er utility for agent j når agent i velger handling a og agent j velger handling c. Figuren viser to måter å tegne payoff-matrisen



Payoff-matrisen kan brukes for å se hvilken handling en rasjonell agent vil velge. Figuren til venstre viser payoff-matrisen for eksempelet over, og her kan vi se at agent i vil velge handling C. Agent i foretrekker alle utfallene som kommer av C over alle utfallene som kommer av D, uansett hva agent j gjør. Dette er et eksempel der det er **ingen strategisk tenking**, siden det ikke spiller noe rolle hva den andre agenten gjør.



Eksempel – Prisoners Dilemma (PD)

Vi kan bruke en Payoff matrise for å finne løsningen på Prisoners Dilemma (se figur). Her vil *defect* være at fangen tilstår, mens *cooperate* er at fangen ikke tilstår. Legg merke til at tallet i matrisen ikke er antall år i fengsel, men representerer hvor godt utfallet er for agenten. Øvre venstre boks er når begge tilstår slik at begge får 2 år i fengsel, mens nedre høyre boks er når ingen tilstår, slik at begge får 1 år i fengsel. **Jo kortere tid i fengsel, desto bedre (= høyere payoff).**

agent j

agent i	defect	cooperate
defect	2, 2	4, 1
cooperate	1, 4	3, 3

Kanonisk PD payoff-matrise

Figuren viser en PD payoff-matrise der $T > R > P > S$.

	Coop	Defect
Coop	R,R	S,T
Defect	T,S	P,P

Løsningskonsepter

En løsning på spillet er en prediksjon av utfallet til spillet der man bruker antagelsen om at alle agenter er rasjonelle og strategiske. Vi skal se på hvordan ulike typer spillteori kan lage slike prediksjoner.

Løsningskonsept – Strengt Dominerende Strategi (SDS)

Dersom vi ser på Prisoners Dilemma kan vi se at preferansene kan ordnes som følger:

- Agent i: $\boldsymbol{D}, D > \boldsymbol{C}, D$ og $\boldsymbol{D}, C > \boldsymbol{C}, C$ (sammenligner rader)
- Agent j: $D, \boldsymbol{D} > D, \boldsymbol{C}$ og $C, \boldsymbol{D} > C, \boldsymbol{C}$ (sammenligner kolonner)

	agent j	
agent i	defect	cooperate
defect	2, 2	4, 1
cooperate	1, 4	3, 3

For begge agentene kan vi se at $Defect$ er den **strenge dominerende handlingen, siden den gir høyest payoff for alle handlingene.** Derfor vil (D, D) være den **strenge dominante strategilikevekten, der alle agentene velger den strenge dominerende handlingen**. En dominerende handling må være unik, og når den eksisterer vil den velges av en rasjonell agent. A er settet av alle felleshandlinger (*joint actions*), a_{-i} er felleshandlinger tatt av alle spillere bortsett fra i og A_{-i} er settet av alle felleshandlinger bortsett fra handlingene til i (dvs. $a_{-i} \in A_{-i}$). **Gitt et spill i strategisk normalform, så vil en handling $\boldsymbol{a_i} \in \boldsymbol{A_i}$ for spiller $\boldsymbol{i}$ være strengt dominerende over handling $\boldsymbol{b_i} \in \boldsymbol{A_i}$ dersom $\boldsymbol{u_i(a_i, a_{-i}) > u_i(b_j, a_{-i})}$ for alle $\boldsymbol{a_{-i}} \in \boldsymbol{A_{-i}}$. Dvs. handling $\boldsymbol{a_i}$ gir større payoff enn $\boldsymbol{b_i}$.**

For payoff-matrisen på figuren, får vi følgende preferanser:

- Agent i: $\boldsymbol{b}, a = \boldsymbol{b}, b > \boldsymbol{a}, b = \boldsymbol{a}, a$ (sammenligner rader)
- Agent j: $b, \boldsymbol{b} = a, \boldsymbol{b} > b, \boldsymbol{a} = a, \boldsymbol{a}$ (sammenligner kolonner)

	agent j	
agent i	action a	action b
action a	1, 1	1, 4
action b	4, 1	4, 4

Her kan vi se at handling b er strengt dominerende for agent i og handling b er strengt dominerende for agent j. Den strengt dominerende strategien blir dermed (b, b).

Weakly Dominerende Strategi (WDS)

Dersom $\boldsymbol{u_i(a_i, a_{-i}) \geq u_i(b_j, a_{-i})}$ vil $\boldsymbol{a_i}$ svakt dominere $\boldsymbol{b_i}$. Ved WDS vil alle agenter velge den svake dominerende handlingen. For eksempel for payoff-matrisen på figuren, er det ingen sterk dominerende strategi for hver agent. Her kan vi se at b er en svak dominerende handling for både agent i og j, så derfor vil (b, b) være en svakt dominerende strategi.

	agent j	
agent i	action a	action b
action a	2, 1	0, 2
action b	2, 3	4, 3

	agent j		
agent i	L	M	R
U	1,0	1,2	0,1
D	0,3	0,1	2,0

I noen tilfeller er det ingen dominerende strategi i det hele tatt, for eksempel for payoff-matrisen på figuren til venstre.

Løsningskonsept – Iterativ eliminering av dominerende handlinger (IEDS)

Dersom det er ingen streng eller svak dominerende strategi likevekt, kan vi bruke IEDS for å finne løsningen (dvs. prediksjon av utfallet til spillet). Iterativ eliminering av dominerende handlinger krever en felles kunnskap i form av at agentene vet at de andre agentene er rasjonelle og de kjenner utilityfunksjonen hos hverandre. IEDS ser på handlinger som dominerer andre handlinger. For et spill i strategisk form der spiller i har handlingene $a_i, b_i \in A_i$, vil a_i være **strengt dominert** av b_i dersom $u_i(a_i, a_{-i}) < u_i(b_j, a_{-i})$. **En rasjonell agent vil aldri velge en suboptimal handling, som vil si at handlingen er dominert av en**

annen. **IEDS utnytter dette ved å iterativt eliminere handlinger som er strengt dominert av andre. Dette gjentas helt til det ikke er igjen flere handlinger som er strengt dominert av andre. Rekkefølgen for elimineringen har ingenting å si**. For eksempelet på figuren får vi følgende iterasjoner:

1. Vi ser på handlingene til agent j, og siden handling R er strengt dominert av handling M (dvs. handling M gir større payoff for agent j) vil handling R elimineres. Agent i er rasjonell og vet at agent j ikke vil velge R.
2. Vi ser på handlingene til agent i, og siden handling D er strengt dominert av handling U (dvs. handling U gir større payoff for agent i) vil handling D elimineres. Agent j er rasjonell og vet at agent i ikke vil velge D.
3. Vi ser på handlingene til agent j, og siden handling L er strengt dominert av handling M (dvs. handling M gir større payoff for agent j) vil handling L elimineres. Agent i er rasjonell og vet at agent j ikke vil velge L

agent j

agent i	L	M	R
U	1,0	1,2	0,1
D	0,3	0,1	2,0

agent j

agent i	L	M	R
U	1,0	1,2	0,1
D	0,3	0,1	2,0

agent j

agent i	L	M	R
U	1,0	1,2	0,1
D	0,3	0,1	2,0

Legg merke til hvordan vi alternerer mellom agentene. Det kan ses som om agenten velger bort handlingen til den andre agenten, fordi den vet at den andre agenten er rasjonell og vil derfor aldri velge en handling som er dominert av en annen. **Løsningen er $(\mathbf{1}, \mathbf{2})$.**

Iterativ eliminering av svakt dominerte handlinger (IEWDS)

Vi sier at a_i **er svakt dominert** av b_i dersom $u_i(a_i, a_{-i}) \leq u_i(b_j, a_{-i})$ for alle $a_{-1} \in A_{-i}$ og $u_i(a_i, a_{-i}) < u_i(b_j, a_{-i})$ for noen $a_{-1} \in A_{-i}$. Dersom det ikke er noen sterkt dominerte handlinger igjen, vil IEWDS eliminere svakt dominerte handlinger**. Et problem ved IEWDS er at ulik rekkefølge på elimineringen gi ulike utfall.** For eksempel for payoff-matrisen på figuren vil vi få ulike løsninger basert på om vi starter med å eliminere M eller U. Dette gjør at elimineringen kan gi flere løsninger.

	L	R
U	$3,1$	$2,0$
M	$4,0$	$1,1$
D	$4,4$	$2,4$

Vi trenger et kraftigere løsningskonsept, og dette gis av Nash likevekt.

Løsningskonsept – Nash likevekt

To viktige begrep for sammenligning og valg av løsninger er:

1. **Pareto optimalitet** = en løsning der det er ingen andre utfall der noen agenter kan øke deres payoff, uten å redusere payoff til andre agenter. Altså, løsningen S^P er Pareto optimal dersom det finnes ingen annen løsning S' som gir at $\exists\ agent\ i\colon\ u_i(S') > u_i(S^P)$ og $\forall\ agent\ j\colon\ u_j(S') \geq u_j(S^P)$.
   For eksempel for payoff-matrisen på figuren vil Pareto optimal være $(A, A), (A, B)$ og (B, A) fordi for disse er det ingen andre utfall der en agent kan gjøre det bedre, uten at den andre agenten vil gjøre det dårligere

Agent1	A	B
A	9,9	7, 10
B	10, 7	8, 8

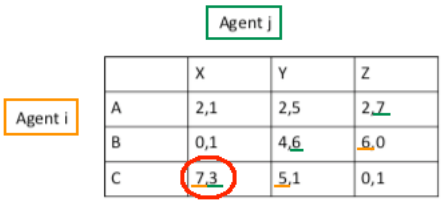
2. **Maksimalisering av sosial velferd** = den sosiale velferden til en løsning er summen av utility hos alle agenter for denne strategiske profilen. Hvis en løsning maksimerer sosial velferd (= sosial optimum), vil ikke de tilgjengelige utilities kastes bort. Hvis løsning er en sosial optimum vil den også være Pareto optimal (ikke motsatt).

Nash likevekt er et sett med strategier (= strategiprofil), en for hver spiller, der ingen spiller ønsker å endre sin strategi gitt hva de andre spillerne gjør. Hver strategi vil være beste respons til strategien til de andre agentene. Beste respons hos agent i er gitt av:

$$BR_i(a_{-i}) = \{a_i \in A_i : u_i(a_i, a_{-i}) \geq u_i(b_i, a_{-i}) \quad \text{for alle } b_i \in A_i\}$$

Hvis det er to spillere kan vi bruke payoff-matrisen for å finne Nash likevekten. Gitt at den ene agenten velger en handling vil den andre agenten velge sin respons som vil maksimere payoff. For å finne Nash likevekten må vi derfor gå igjennom alle handlingene til den ene agenten og markere den beste responsen for den andre agenten gitt disse handlingene. Deretter bytter vi agenter. **Nash likevekten vil være handlingene der de beste responsene krysser**.

For eksempel for payoff-matrisen på figuren vil vi begynne med å se på hvordan agent j responderer til handlingene til agent i:



- Når agent i velger handling A vil beste respons for agent j være handling Z
- Når agent i velger handling B vil beste respons for agent j være handling Y
- Når agent i velger handling C vil beste respons for agent j være handling X.

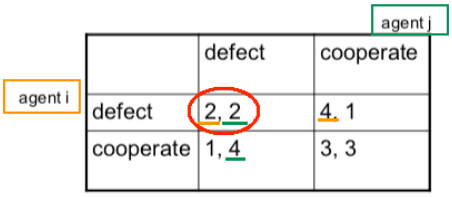
Deretter ser vi på hvordan agent i responderer til agent j:

- Når agent j velger handling X vil beste respons for agent i være handling C
- Når agent j velger handling Y vil beste respons for agent i være handling C
- Når agent j velger handling Z vil beste respons for agent i være handling B

Disse er markert med streker på figuren og siden det er (C, X) er den eneste som er beste respons for begge agentene, vil dette være Nash likevekt.

Prisoners Dilemma

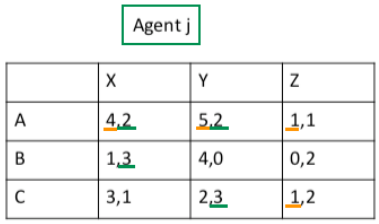
For Prisoners Dilemma med payoff-matrise på figuren, vil Nash likevekt være $(defect, defect)$, siden dette er krysningspunktet for de beste responsene til begge agentene. Hvis agent i endrer strategi til $cooperate$ vil den gjøre det dårligere, mens hvis agent j endrer strategi til $cooperate$ vil den gjøre det dårligere. $defect$ er derfor beste respons for alle strategiene.

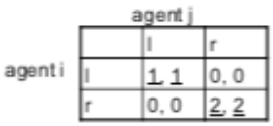


I dette tilfellet vil (D, C), (C, D) og (C, C) være Pareto optimale, siden disse utfallene ikke kan endres uten at minst én av agentene gjør det dårligere. Det er (C, C) som vil maksimere sosial velferd. Siden løsningen er (D, D) vil dette bety at Nash likevekten er ikke Pareto optimal eller sosial maksimerende. Intuisjon sier at dette ikke er beste utfall. Det vil være best om de begge ikke tilstår, slik at begge får payoff 3. Dette er det grunnleggende problemet ved multiagent interaksjoner.

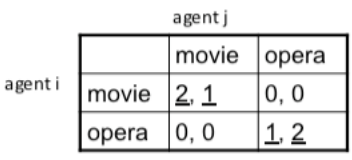
Multiple Nash likevekter – kjønnskamp og hjortejakt

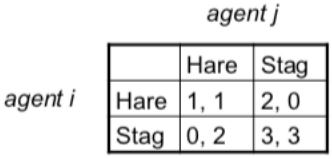
En payoff-matrise kan ha flere Nash likevekter. For eksempel for payoff-matrisen på figuren til høyre vil Nash likevekt være (A, X) og (A, Y). Payoff-matrisen til venstre har også flere Nash likevekter. Legg merke til at (r, r) er en sosial optimum. Dette kalles rent koordineringsspill.





Kjønnskampen er et eksempel på en multiagent interaksjon med flere Nash likevekter. De er enige om at det er best å samarbeide, men de er «uenige» i hva som er det beste utfallet.





Det samme gjelder hjortejakten som vil ha Nash likevekt (H, H) og (S, S). Her vil (S, S) gi mest payoff, mens (H, H) er mest trygg. Så dette vil være en balanse mellom høyre belønning og større risiko. Det er best for samfunnet at man jakter hjort, men det krever at man stoler på hverandre (= koordinering)

Det er delte meninger i hva som er det rasjonelle valget i hjortejakten. (S, S) er payoff-dominant, mens (H, H) er risikodominant. Alternativer er å endre omgivelsene, for eksempel ved å si at dersom begge agentene jakter på harer, vil de bli gitt til hverandre. Hvis en fanger en hare, mens den andre forsøker å fange en hjort, vil de to agentene dele haren.

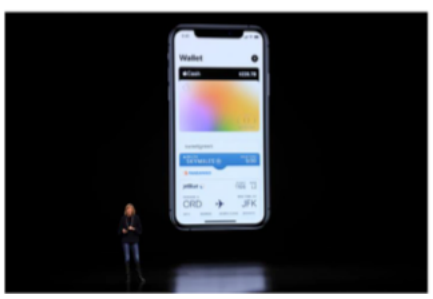
Ingen Nash likevekter

I zero-sum interaksjoner vil det ikke være noen Nash likevekt, fordi utilities vil summere til 0, slik at agentene har ingen preferanse. Ved slike interaksjoner vil preferansen til agentene være diametralt motsatt, slik at vi får strenge konkurransescenarier. Et eksempel på en zero-sum spill er matching av mynt og krone. I dette spillet er det to agenter som enten vil velge mynt eller krone samtidig. Hvis de har ulike valg vil agent i tape, mens hvis de har like valg vil agent i vinne. I dette tilfellet er det ingen mulighet for samarbeid eller koordinering. Hver spiller vil gjøre sitt beste for å vinne.

		agent j	
		H	T
agent i	H	1, -1	-1, 1
	T	-1, 1	1, -1

AI og etikk

Denne delen av kompendiet er en liten introduksjon til etikkens rolle i AI. Det er basert på forelesningsnotatene.



Når man arbeider med AI er det viktig å huske to ting:

1. **AI har begrensninger**
2. **AI har påvirkningskraft**

Etikk er de moralske prinsippene som styrer oppførsel eller handlinger hos et individ eller en gruppe. Det er reglene eller avgjørelsesbanene som hjelper til med å bestemme hva som er bra eller riktig. **Dersom AI er dataen og algoritmene, vil etikken være omgivelsene som AI arbeider i**. AI er en mektig kraft som omformer daglige praksiser, personlige og profesjonelle interaksjoner og omgivelser. For menneskehetens velvære er det avgjørende at denne makten blir brukt som en positiv kraft. **Etikk har en essensiell rolle i denne prosessen, ved at det vil sikre at AI reguleres slik at man kan utnytte dens kraft, samtidig som man reduserer risikoene**. Noen sentrale konsepter er personvern, rettferdighet, sikkerhet, transparens, nøyaktighet, ansvarlighet, osv. Teknologi er verken god eller dårlig, men den er heller ikke nøytral.

Noen utfordringer ved AI:

- AI sliter med det ekstraordinære
- Det er nesten umulig å bygge sikre AI applikasjoner
- Dataen som brukes av AI system korresponderer ikke med virkeligheten
- Det er en mangel av mangfold i AI forskning og industri
- AI systemer produserer skjulte kostnader