

Inteligența artificială poate fi privită ca fiind o știință sui-generis al cărei obiect de studiu îl constituie modul de programare a calculatoarelor în vederea săvârșirii unor operațiuni pe care, deocamdată, oamenii le efectuează mai bine. Conform acestei definiții primare, putem gândi inteligența artificială ca fiind acel domeniu al informaticii care se ocupă de **automatizarea comportamentului inteligent**.

Domeniu al informaticii => opereaza cu:

- ***structurile de date*** folosite în reprezentarea cunoștințelor;
 - ***algoritmii*** necesari pentru aplicarea cunoștințelor;
 - ***limbajele și tehnicile de programare*** folosite la implementarea acestor algoritmi.
- problema definirii domeniului inteligenței artificiale devine, până la urmă, una a definirii conceptului însuși de **inteligentă** => legătura inteligenței artificiale cu alte domenii, cum ar fi, în primul rând, filozofia.

➤ **Domeniul inteligenței artificiale își propune:**

- *înțelegerea entităților inteligente;*
- *construirea unor asemenea entități.*

➤ **IA are drept scop apariția unor calculatoare care să aibă o inteligență de nivel uman sau chiar mai bună.**

Exista mai multe *definiții ale domeniului*. Acestea variază de-a lungul a *două mari dimensiuni*. Prima dimensiune este aceea a *procesului de gândire și a raționamentului*. Cea de-a doua adresează *comportamentul* (“behavior”). De asemenea, unele definiții măsoară *succesul* în termenii *performanței umane*, în timp ce altele îl măsoară relativ la un concept ideal de inteligență, pe care îl vom numi *rațiune*.

Aceste definiții pot fi grupate în patru mari categorii (care indică și patru țeluri majore urmărite în inteligența artificială):

- **sisteme care gândesc ca și ființele umane;**
- **sisteme care se comportă ca și ființele umane;**
- **sisteme care gândesc rațional;**
- **sisteme care se comportă rațional.**

Varietate de definiții ⇒

- **cercetători diferiți gândesc în mod diferit despre inteligența artificială;**
- **IA apare ca o știință inter-disciplinară, în care își aduc contribuția filozofi, psihologi, lingviști, matematicieni, informaticieni și ingineri.**

Prima lucrare recunoscută astăzi ca fiind de inteligență artificială aparține lui Warren McCulloch și Walter Pitts (1943). Aceștia s-au bazat pe *trei surse* și au conceput *un model de neuroni artificiali*. Cele *trei surse* utilizate au fost:

- cunoștințele despre fiziologia și funcțiile de bază ale neuronilor în creier;
- analiza formală a logicii propoziționale datorate lui Russel și Whitehead;
- teoria despre calcul a lui Turing.

O altă figură extrem de influentă în inteligența artificială este cea a lui John McCarthy, de la Princeton. După absolvire, McCarthy se mută la Dartmouth College, care va deveni locul oficial în care s-a născut domeniul.

În timpul workshop-ului organizat la Dartmouth, în vara anului 1956, se hotărăște și adoptarea, pentru noul domeniu, a numelui propus de McCarthy: cel de inteligență artificială.

SCURT ISTORIC

Un *an istoric* în evoluția domeniului este 1958, an în care McCarthy se mută de la Dartmouth la MIT (“Massachusetts Institute of Technology”). Aici el aduce trei contribuții vitale, toate în același an, 1958:

- definește limbajul de nivel înalt LISP, care urma să devină limbajul de programare dominant în inteligența artificială;
- introduce conceptul de *partajare* (“time-sharing”);
- publică articolul intitulat *Programs with Common Sense* (“Programe cu bun simț”) în care descrie un program ipotetic, numit “Advice Taker”, care poate fi privit ca reprezentând *primul sistem complet de inteligență artificială*.

ADVICE TAKER

- Programul era proiectat să folosească *cunoștințe* pentru a căuta soluții la probleme.
- Spre deosebire de programele de până atunci, încorpora *cunoștințe generale despre lume*.
- Era conceput în așa fel încât să poată accepta noi axiome pe parcurs \Rightarrow i se permitea să dobândească competență în noi domenii, fără a fi reprogramat.

Acest program încorpora *principiile de bază ale reprezentării cunoștințelor și ale raționamentului*, și anume:

- este necesar să dispunem de o reprezentare formală a lumii și a felului în care acțiunile unui agent afectează lumea;
- este necesar să putem manipula aceste reprezentări cu ajutorul unor procese deductive.

Discuțiile de început asupra domeniului s-au concentrat mai degrabă asupra *reprezentării problemelor* decât asupra *reprezentării cunoștințelor*. Accentul s-a pus pe formularea problemei care trebuie rezolvată și NU pe formularea resurselor care sunt disponibile programului.

- În perioada 1969-1979 apar și se dezvoltă sistemele expert (sau *sistemele bazate pe cunoștințe*).

Caracteristica majoră a sistemelor expert este aceea că ele se bazează pe cunoștințele unui expert uman în domeniul care este studiat; mai exact, pe cunoștințele expertului uman asupra *strategiilor de rezolvare a problemelor tipice unui domeniu*.

La baza sistemelor expert se află utilizarea în rezolvarea problemelor a unor mari cantități de cunoștințe specifice domeniului.

Tot în perioada 1969-1979, numărul mare de aplicații referitoare la *problemele lumii reale* a determinat creșterea cererii de *scheme de reprezentare a cunoștințelor*. Au fost astfel dezvoltate diferite *limbaje de reprezentare*. Unele dintre ele se bazează pe *logică*, cum este cazul limbajului Prolog, extrem de popular în Europa.

Limbajele programării logice sunt *limbaje declarative*. Un limbaj de programare declarativ scutește programatorul de a mai menționa procedura exactă pe care trebuie să o execute calculatorul pentru a realiza o funcție. Programatorii folosesc limbajul pentru a *descrie* o mulțime de *fapte* și de *relații* astfel încât utilizatorul să poată *interoga* apoi sistemul pentru a obține un anumit rezultat.

- Primul compilator de Prolog a fost creat în 1972 de către Philippe Roussel, la Universitatea din Marsilia.

În perioada 1980-1988 în inteligența artificială începe să se lucreze la nivel industrial.

Tot în această perioadă, în care inteligența artificială devine industrie, ea își propune noi țeluri ambițioase, cum ar fi acela al înțelegerii limbajului natural.

Începând din 1987 și până în prezent cercetătorii se bazează mult mai mult pe teoreme riguroase și mai puțin decât până acum pe intuiție. Spre exemplu, domeniul *recunoașterii limbajului* a ajuns să fie dominat de așa-numitele “hidden Markov models” (HMM-uri).

Subdomenii ale inteligenței artificiale

Principalele subdomenii ale inteligenței artificiale sunt considerate a fi următoarele:

- jocurile (bazate pe *căutarea* efectuată într-un spațiu de stări ale problemei);
- raționamentul automat și demonstrarea teoremelor (bazate pe rigoarea și generalitatea *logicii matematice*. Este cea mai veche ramură a inteligenței artificiale și cea care înregistrează cel mai mare succes. Cercetarea în domeniul demonstrării automate a teoremelor a dus la *formalizarea algoritmilor de căutare* și la dezvoltarea unor *limbaje de reprezentare formală*, cum ar fi calculul predicatelor și limbajul pentru programare logică Prolog).
- sistemele expert (care pun în evidență importanța *cunoștințelor specifice unui domeniu*).

- înțelegerea limbajului natural și modelarea semantică (Caracteristica de bază a oricărui sistem de înțelegere a limbajului natural o constituie reprezentarea sensului propozițiilor într-un anumit limbaj de reprezentare astfel încât aceasta să poată fi utilizată în prelucrări ulterioare).
- planificarea și robotica (Planificarea presupune existența unui robot capabil să execute anumite acțiuni atomice, cum ar fi deplasarea într-o cameră plină cu obstacole).
- învățarea automată (datorită căreia se realizează adaptarea la noi circumstanțe, precum și detectarea și extrapolarea unor șabloane - “patterns”. Învățarea se realizează, spre exemplu, prin intermediul așa-numitelor *rețele neurale* sau *neuronale*. O asemenea rețea reprezintă un tip de sistem de inteligență artificială modelat după neuronii -celulele nervoase- dintr-un sistem nervos biologic, în încercarea de a simula modul în care creierul prelucrează informațiile, învață sau își aduce aminte).

INVATAREA AUTOMATA

- Este subdomeniul IA care se ocupa de programe care invata din experienta.
- Invatarea statistica:
 - Intr-un scenariu tipic avem o masura de iesire care este cantitativa (pretul actiunilor) sau categorica (pacientul va face sau nu va face infarct) pe care vrem s-o prezicem pe baza unei multimi de caracteristici (cum ar fi dieta sau masuratorile clinice).
 - Avem o multime de date de antrenare corespunzator carora se observa rezultatul, precum si masuratorile caracteristicilor pentru o multime de obiecte (de ex. oameni).
 - Folosind aceste date construim un model de predictie – “learner” – care ne va permite sa prezicem rezultatul pentru obiecte noi, nevazute.
 - Acuratetea predictiei da calitatea learner-ului.
 - Aceasta problema de invatare este numita supervizata datorita prezentei variabilei rezultat, care ghideaza procesul de invatare.
 - In cazul unei probleme de invatare nesupervizate, sunt observate numai caracteristicile si nu exista masuratori ale rezultatului. Sarcina programatorului este, mai degraba, aceea de a descrie felul in care datele sunt organizate (în cluster-e; cluster = grup mic, multime).

Majoritatea tehnicilor din inteligența artificială folosesc pentru implementarea inteligenței

- *cunoștințe reprezentate în mod explicit*
- *algoritmi de căutare*
- *învățare automată*

- *cunoștințe reprezentate în mod explicit*
- *algoritmi de căutare*

Toate aceste subdomenii ale inteligenței artificiale au anumite trăsături în comun, și anume:

- O concentrare asupra problemelor care nu răspund la soluții algoritmice; din această cauză tehnica de rezolvare a problemelor specifică inteligenței artificiale este aceea de a se baza pe o *căutare euristică*.
- IA rezolvă probleme folosind și informație inexactă, care *lipsește* sau care *nu este complet definită* și utilizează formalisme de reprezentare ce constituie pentru programator o compensație față de aceste probleme.
- IA folosește raționamente asupra trăsăturilor calitative semnificative ale unei situații.
- IA folosește, în rezolvarea problemelor, mari cantități de cunoștințe specifice domeniului investigat.

Vom folosi abordarea în cadrul căreia:

- Reprezentarea cunoștințelor adresează problema reprezentării într-un limbaj formal, adică un limbaj adecvat pentru prelucrarea ulterioară de către un calculator, a întregii game de cunoștințe necesare comportamentului inteligent; în funcție de tehnica de reprezentare a cunoștințelor alese se folosește un anumit mecanism de inferență;
- căutarea este o tehnică de rezolvare a problemelor care explorează în mod sistematic un spațiu de stări ale problemei, adică de stadii succesive și alternative în procesul de rezolvare a acesteia. (Exemple de stări ale problemei pot fi configurațiile diferite ale tablei de șah în cadrul unui joc sau pașii intermediari într-un proces de raționament).

Tehnici ale IA

O tehnică a inteligenței artificiale este o metodă de exploatare și valorificare a cunoștințelor care, la rândul lor, ar trebui reprezentate într-un anumit mod, și anume conform următoarelor cerințe:

- Cunoștințele trebuie să înglobeze *generalizări*. Nu este necesară reprezentarea separată a fiecărei situații în parte. În schimb, situațiile care au în comun proprietăți importante pot fi grupate laolaltă. Dacă cunoștințele nu ar avea această proprietate, ar fi nevoie de foarte multă memorie și de multe operații de actualizare. Prin urmare, o mulțime de informații care nu posedă această proprietate nu va reprezenta cunoștințe, ci date.
- Cunoștințele trebuie să *poată fi înțelese* de către oamenii care le furnizează.

- Cunoștințele trebuie să poată fi ușor modificate pentru a se corecta erori și pentru a reflecta atât schimbările din lumea înconjurătoare, cât și schimbările din percepția și imaginea noastră despre aceasta.
 - Cunoștințele trebuie să poată fi folosite în foarte multe situații, chiar dacă uneori duc lipsă de acuratețe și nu sunt complete.
 - Cunoștințele trebuie să poată fi folosite astfel încât să ajute la îngustarea gamei de posibilități care trebuie luate în considerație.
- Scopul tehnicilor inteligenței artificiale este acela de a putea susține *o folosire eficientă a cunoștințelor.*

Un exemplu de problemă tipică în inteligența artificială **(QUESTION ANSWERING)**

Un exemplu de problemă tipică tratată de inteligența artificială îl constituie problema furnizării răspunsului la întrebări (“Question Answering”).

- **Problema:** Se citește de la tastatură text într-un limbaj natural (spre exemplu în limba engleză) și apoi se răspunde unor întrebări referitoare la acel text, formulate tot în engleză.
- **Observație:** În cazul unei astfel de probleme este mai greu de precizat care este răspunsul corect, așa cum se va vedea în următorul exemplu:

Exemplul nr. 1

Presupunem că textul reprezentând input-ul este dat de unica propoziție

“Russia massed troops on the Czech border.”

Corespunzător unui astfel de input este posibilă existența ambelor dialoguri care urmează. Aceste dialoguri au reprezentat efectiv output-ul programului numit POLITICS (Carbonell, 1980):

Dialogul nr. 1

Q: Why did Russia do this?

A: Because Russia thought that it could take political control of Czechoslovakia by sending troops.

Q: What should the United States do?

A: The United States should intervene militarily.

Dialogul nr. 2

Q: Why did Russia do this?

A: Because Russia wanted to increase its political influence over Czechoslovakia.

Q: What should the United States do?

A: The United States should denounce the Russian action in the United Nations.

În programul POLITICS răspunsurile au fost construite luând în considerație atât textul reprezentând input-ul, cât și un model separat referitor la concepțiile și acțiunile diverselor entități politice, inclusiv cele ale fostei Uniuni Sovietice. Între cele două dialoguri acest model a fost schimbat. Atunci când modelul se schimbă și răspunsurile sistemului se schimbă. Astfel, primul dialog a fost produs de programul POLITICS atunci când i s-a dat acestuia un model care corespunde concepțiilor unui *conservator* american. Cel de-al doilea a fost produs atunci când i s-a dat un model care corespunde concepțiilor unui *liberal* american. Prin urmare, în acest caz este foarte greu de spus ce înseamnă un *răspuns corect*. Acest lucru depinde de model.

- Corectitudinea răspunsului *depinde de model*.
- O procedură eficientă de “răspunsuri la întrebări” trebuie să se bazeze în mod solid pe cunoștințe și pe exploatarea computațională a acelor cunoștințe.
- Însuși scopul tehnicilor inteligenței artificiale este acela de a putea susține o folosire eficientă a cunoștințelor.

Concluzii

IA \equiv acel domeniu al informaticii care se ocupă de automatizarea comportamentului inteligent.

Scopul urmarit de domeniul IA

Scopul stiintific al IA este acela al determinarii de teorii cu privire la reprezentarea cunostintelor, invatare, sisteme bazate pe reguli si cautare, teorii care sa explice diverse tipuri de inteligenta.

Scopul ingineresc al IA este acela de a conferi masinii abilitatea de a rezolva problem ale lumii reale.

Principalele tehnici folosite in IA cu acest scop sunt: reprezentarea cunostintelor, invatarea automata, sistemele bazate pe reguli si cautarea in spatiul de stari.

Tehnici folosite in IA: reprezentarea, invatarea, regulile, cautarea

Reprezentarea

Toate sistemele IA au o importanta trasatura (caracteristica) de reprezentare a cunostintelor. Sistemele bazate pe reguli, sistemele bazate pe cadre si retelele semantice folosesc o secventa de *reguli if-then*, in timp ce retelele neuronale artificiale folosesc conexiuni impreuna cu ponderi ale conexiunilor.

Invatarea

Toate sistemele IA au capacitatea de a invata. Folosind aceasta capacitate, ele achizitioneaza automat cunostinte din mediul inconjurator. De exemplu, pot achizitiona regulile pentru un sistem expert bazat pe reguli sau determina ponderile adecvate ale conexiunilor intr-o retea neuronală artificială.

Reguli

Regulile unui sistem IA pot fi implicite sau explicite. Atunci cand sunt explicite, regulile sunt create de catre un inginer specializat in ingineria cunostintelor (de pilda regulile pentru un sistem expert). Reguli implicite pot sa apara, de pilda, sub forma ponderilor conexiunilor intr-o retea neuronală.

Cautarea

Cautarea poate sa apara in multiple forme, de pilda sub forma cautarii succesiunii de stari care conduce cel mai repede la o solutie, ori sub forma cautarii unei multimi optime de ponderi ale conexiunilor intr-un ANN – prin minimizarea functiei de fitness. (ANN \equiv Artificial Neural Network).

N.B. Acest curs introductiv are doua parti distincte: una dintre ele se refera la cautare si reprezentarea cunostintelor, cealalta la invatarea automata.

TEHNICI DE CĂUTARE

Căutarea este o traversare sistematică a unui spațiu de soluții posibile ale unei probleme.

Un spațiu de căutare este de obicei un graf (sau, mai exact, un arbore) în care un nod desemnează o soluție parțială, iar o muchie reprezintă un pas în construirea unei soluții. Scopul căutării poate fi acela de

- a găsi un drum în graf de la o situație inițială la una finală;
- a ajunge într-un nod care reprezintă situația finală.

Programul

- **Programul reprezinta un agent inteligent.**
- **Agenții cu care vom lucra vor adopta un *scop* și vor urmări *satisfacerea* lui.**

Rezolvarea problemelor prin intermediul căutării

- În procesul de rezolvare a problemelor, formularea scopului, bazată pe situația curentă, reprezintă primul pas.
- Vom considera un scop ca fiind o mulțime de stări ale universului, și anume acele stări în care scopul este satisfăcut.
- Acțiunile pot fi privite ca generând tranziții între stări ale universului.
- Agentul va trebui să afle care acțiuni îl vor conduce la o stare în care scopul este satisfăcut. Înainte de a face asta el trebuie să decidă ce tipuri de acțiuni și de stări să ia în considerație.
- Procesul decizional cu privire la acțiunile și stările ce trebuie luate în considerație reprezintă formularea problemei. Formularea problemei urmează după formularea scopului.

- Un agent care va avea la dispoziție mai multe opțiuni imediate va decide ce să facă examinând mai întâi diferite secvențe de acțiuni posibile, care conduc la stări de valori necunoscute, urmând ca, în urma acestei examinări, să o aleagă pe cea mai bună. Procesul de examinare a unei astfel de succesiuni de acțiuni se numește *căutare*.
- Un algorithm de căutare primește ca *input* o *problemă* și întoarce ca *output* o *soluție* sub forma unei succesiuni de acțiuni.
- Odată cu găsirea unei soluții, acțiunile recomandate de aceasta pot fi duse la îndeplinire. Aceasta este *faza de execuție*. Prin urmare:
 - ✓ agentul *formulează, caută și execută*.

- **După formularea unui scop și a unei probleme de rezolvat, agentul cheamă o procedură de căutare pentru a o rezolva. El folosește apoi soluția pentru a-l ghida în acțiunile sale, executând ceea ce îi recomandă soluția ca fiind următoarea acțiune de îndeplinit și apoi înlătură acest pas din succesiunea de acțiuni. Odată ce soluția a fost executată, agentul va găsi un nou scop.**

Probleme și soluții corect definite

➤ Probleme cu o singură stare

Elementele de bază ale definirii unei probleme sunt *stările* și *acțiunile*. Pentru a descrie stările și acțiunile, din punct de vedere formal, este nevoie de următoarele elemente:

- Starea inițială în care agentul știe că se află.
- Mulțimea acțiunilor posibile disponibile agentului. Termenul de operator este folosit pentru a desemna descrierea unei acțiuni, prin specificarea stării în care se va ajunge ca urmare a îndeplinirii acțiunii respective, atunci când ne aflăm într-o anumită stare. (O formulare alternativă folosește o *funcție succesori* S . Fiind dată o anumită stare x , $S(x)$ întoarce mulțimea stărilor în care se poate ajunge din x , printr-o unică acțiune).

- **Spațiul de stări** al unei probleme reprezintă mulțimea tuturor stărilor în care se poate ajunge plecând din starea inițială, prin intermediul oricărei secvențe de acțiuni.
- Un **drum** în spațiul de stări este orice secvență de acțiuni care conduce de la o stare la alta.
- **Testul scop** este testul pe care un agent îl poate aplica unei singure descrieri de stare pentru a determina dacă ea este o **stare de tip scop**, adică o stare în care scopul este atins (sau realizat). Uneori există o mulțime explicită de stări scop posibile și testul efectuat nu face decât să verifice dacă s-a ajuns în una dintre ele. Alteori, scopul este specificat printr-o proprietate abstractă și nu prin enumerarea unei mulțimi de stări. De exemplu, în șah, scopul este să se ajungă la o stare numită “șah mat”, în care regele adversarului poate fi capturat la următoarea mutare, orice ar face adversarul. S-ar putea întâmpla ca o soluție să fie preferabilă alteia, chiar dacă amândouă ating scopul. Spre exemplu, pot fi preferate drumuri cu mai puține acțiuni sau cu acțiuni mai puțin costisitoare.

- *Funcția de cost a unui drum* este o funcție care atribuie un cost unui drum. Ea este adeseori notată prin g . Vom considera costul unui drum ca fiind suma costurilor acțiunilor individuale care compun drumul.

Împreună starea inițială, mulțimea operatorilor, testul scop și funcția de cost a unui drum definesc o problemă.

➤ Probleme cu stări multiple

Pentru definirea unei astfel de probleme trebuie specificate:

- o mulțime de stări inițiale;
 - o mulțime de operatori care indică, în cazul fiecărei acțiuni, mulțimea stărilor în care se ajunge plecând de la orice stare dată;
 - un test scop (la fel ca la problema cu o singură stare);
 - funcția de cost a unui drum (la fel ca la problema cu o singură stare).
-
- ✓ Un operator se aplică unei mulțimi de stări prin reunirea rezultatelor aplicării operatorului fiecărei stări din mulțime.
 - ✓ Aici un drum leagă *mulțimi de stări*, iar o soluție este un drum care conduce la o *mulțime de stări*, dintre care *toate sunt stări scop*.
-
- Spațiul de stări este aici înlocuit de spațiul mulțimii de stări.

Un exemplu: Problema misionarilor si a canibalilor

Definiție formală a problemei:

- Stări: o stare constă dintr-o secvență ordonată de trei numere reprezentând numărul de misionari, de canibali și de bărci, care se află pe malul râului. Starea de pornire (inițială) este (3,3,1).
- Operatori: din fiecare stare, posibili operatorii trebuie să ia fie un misionar, fie un canibal, fie doi misionari, fie doi canibali, fie câte unul din fiecare și să îi transporte cu barca. Prin urmare, există *cel mult cinci operatori*, deși majorității stărilor le corespund mai puțini operatori, întrucât trebuie evitate stările interzise. (*Observație*: Dacă am fi ales să distingem între indivizi, în loc de cinci operatori ar fi existat 27).
- Testul scop: să se ajungă în starea (0,0,0).
- Costul drumului: este dat de numărul de traversări.

Căutarea soluțiilor și generarea secvențelor de acțiuni

- Rezolvarea unei probleme începe cu *starea inițială*.
- Primul pas este acela de a testa dacă starea inițială este o *stare scop*.
- Dacă nu, se iau în considerație și alte stări. Acest lucru se realizează aplicând *operatorii* asupra stării curente și, în consecință, generând o mulțime de stări. Procesul poartă denumirea de *extinderea stării*.
- Atunci când se generează mai multe posibilități, trebuie făcută o alegere relativ la cea care va fi luată în considerație în continuare, aceasta fiind esența căutării.

- *Alegerea referitoare la care dintre stări trebuie extinsă prima este determinată de strategia de căutare.*
- *Procesul de căutare construiește un arbore de căutare, a cărui rădăcină este un nod de căutare corespunzând stării inițiale. La fiecare pas, algoritmul de căutare alege un nod-frunză pentru a-l extinde.*

➤ Observație:

Este important să facem *distincția între spațiul stărilor și arborele de căutare*. Spre exemplu, într-o problemă de căutare a unui drum pe o hartă, pot exista doar 20 de stări în spațiul stărilor, câte una pentru fiecare oraș. Dar există un număr infinit de drumuri în acest spațiu de stări. Prin urmare, arborele de căutare are un număr infinit de noduri. Evident, un bun algoritm de căutare trebuie să evite urmarea unor asemenea drumuri.

➤ Observație:

Este importantă distincția între noduri și stări:

- Un nod este o structură de date folosită pentru a reprezenta arborele de căutare corespunzător unei anumite realizări a unei probleme, generată de un anumit algoritm.
- O stare reprezintă o configurație a lumii înconjurătoare.

De aceea, nodurile au adâncimi și părinți, iar stările nu le au. Mai mult, este posibil ca două noduri diferite să conțină aceeași stare, dacă acea stare este generată prin intermediul a două secvențe de acțiuni diferite.

Reprezentarea nodurilor în program

Există numeroase moduri de a reprezenta nodurile. În general, se consideră că un nod este o structură de date cu cinci componente:

- starea din spațiul de stări căreia îi corespunde nodul;
- nodul din arborele de căutare care a generat acest nod (nodul părinte);
- operatorul care a fost aplicat pentru a se genera nodul;
- numărul de noduri aflate pe drumul de la rădăcină la acest nod (adâncimea nodului);
- costul drumului de la starea inițială la acest nod.

Reprezentarea colecției de noduri care așteaptă pentru a fi extinse

Această colecție de noduri poartă denumirea de *frontieră*. Cea mai simplă reprezentare ar fi aceea a unei mulțimi de noduri, iar strategia de căutare ar fi o funcție care selectează, din această mulțime, următorul nod ce trebuie extins. Deși din punct de vedere conceptual această cale este una directă, din punct de vedere computațional ea poate fi foarte scumpă, pentru că funcția strategie ar trebui să se “uite” la fiecare element al mulțimii pentru a-l alege pe cel mai bun. De aceea, *vom presupune că această colecție de noduri este implementată ca o coadă.*

Evaluarea strategiilor de căutare

Strategiile de căutare se evaluează conform următoarelor patru criterii:

- **Completitudine**: dacă, atunci când o soluție există, strategia dată garantează găsirea acesteia;
- **Complexitate a timpului**: durata de timp pentru găsirea unei soluții;
- **Complexitate a spațiului**: necesitățile de memorie pentru efectuarea căutării;
- **Optimalitate**: atunci când există mai multe soluții, strategia dată să o găsească pe cea mai de calitate dintre ele.

Căutarea neinformată

Termenul de căutare neinformată desemnează faptul că o strategie de acest tip nu deține nici o informație despre numărul de pași sau despre costul drumului de la starea curentă la scop. Tot ceea ce se poate face este să se distingă o stare-scop de o stare care nu este scop. Căutarea neinformată se mai numește și căutarea oarbă.

Căutarea informată

Să considerăm, de pildă, problema găsirii unui drum de la Arad la București, având în față o hartă. De la starea inițială, Arad, există trei acțiuni care conduc la trei noi stări: Sibiu, Timișoara și Zerind. O căutare neinformată nu are nici o preferință între cele trei variante. Un agent mai inteligent va observa însă că scopul, București, se află la sud-est de Arad și că numai *Sibiu* este în această direcție, care reprezintă, probabil, cea mai bună alegere. Strategiile care folosesc asemenea considerații se numesc strategii de căutare informată sau strategii de căutare euristică.

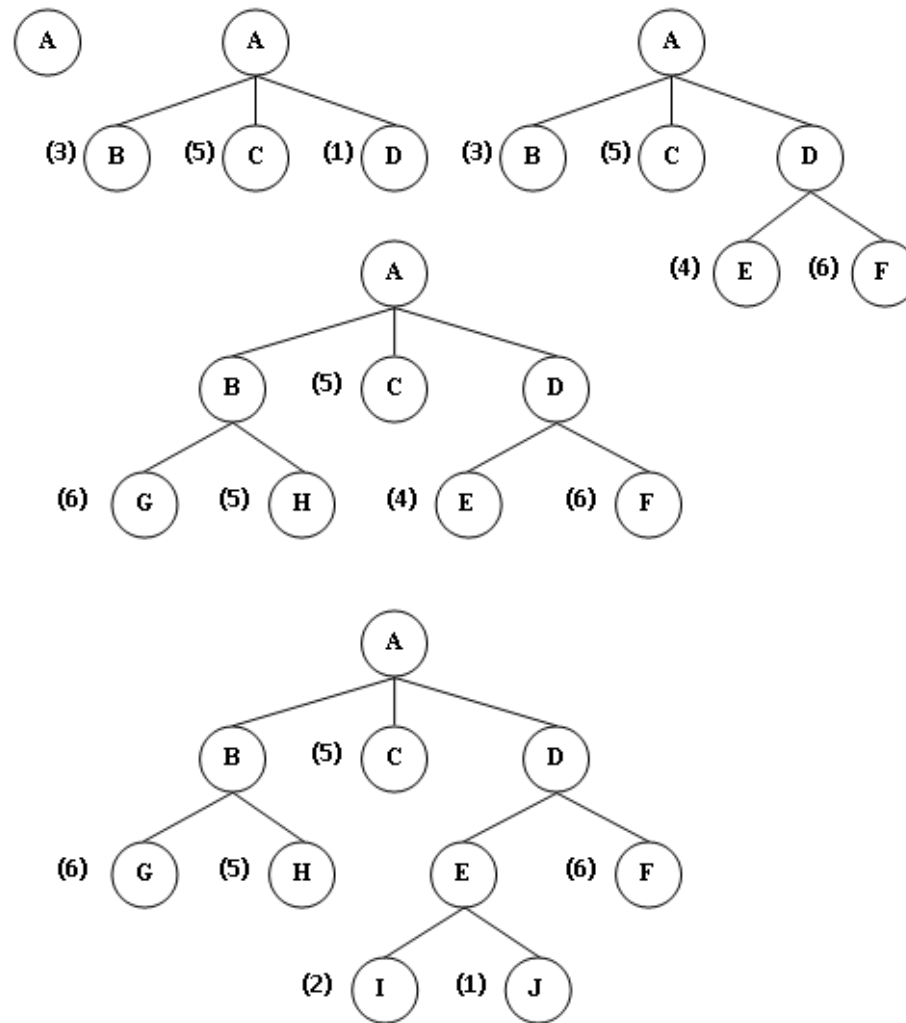
Căutarea informată

- Căutarea informată se mai numește și căutare euristică.
- Euristica este o metodă de studiu și de cercetare bazată pe descoperirea de fapte noi. În acest tip de căutare vom folosi informația despre spațiul de stări. Se folosesc cunoștințe specifice problemei și se rezolvă probleme de optim.

Căutarea de tip best-first

- Procesul de căutare nu se desfășoară în mod uniform plecând de la nodul inițial. El înaintează în mod preferențial de-a lungul unor noduri pe care informația euristică, specifică problemei, le indică ca aflându-se pe drumul cel mai bun către un scop. Un asemenea proces de căutare se numește căutare euristică sau căutare de tip best-first.
- Principiile pe care se bazează căutarea de tip best-first sunt următoarele:
 1. Se presupune existența unei funcții euristice de evaluare, \hat{f} , cu rolul de a ne ajuta să decidem care nod ar trebui extins la pasul următor. Se va adopta *convenția* că valori mici ale lui \hat{f} indică nodurile cele mai bune. Această funcție se bazează pe informație specifică domeniului pentru care s-a formulat problema. Este o funcție de descriere a stărilor, cu valori reale.
 2. Se extinde nodul cu cea mai mică valoare a lui $\hat{f}(n)$. *În cele ce urmează, se va presupune că extinderea unui nod va produce toți succesorii acelui nod.*

Figura următoare ilustrează începutul unei căutări de tip best-first:



Aici există inițial un singur nod, A, astfel încât acesta va fi extins.

Pentru a nu fi induși în eroare de o euristică extrem de optimistă, este necesar să înclinăm căutarea în favoarea posibilității de a ne întoarce înapoi, cu scopul de a explora drumuri găsite mai devreme. De aceea, vom adăuga lui \hat{f} *un factor de adâncime*,

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n),$$

unde:

- $\hat{g}(n)$ este o *estimație a adâncimii lui n în graf*, adică reprezintă lungimea celui mai scurt drum de la nodul de start la n ;
- $\hat{h}(n)$ este o *evaluare euristică a nodului n* .

Prezentăm un algoritm de căutare generală bazat pe grafuri. Algoritmul include versiuni ale căutării de tip best-first ca reprezentând cazuri particulare.

Algoritm de căutare general bazat pe grafuri

Acest algoritm, pe care îl vom numi GraphSearch, este unul general, care permite orice tip de ordonare preferată de utilizator - euristică sau neinformată. Iată o *primă variantă* a definiției sale:

GraphSearch

1. Creează un arbore de căutare, T_r , care constă numai din nodul de start n_0 . Plasează pe n_0 într-o listă ordonată numită OPEN.
2. Creează o listă numită CLOSED, care inițial este vidă.
3. Dacă lista OPEN este vidă, EXIT cu eșec.
4. Selectează primul nod din OPEN, înlătură-l din lista OPEN și include-l în lista CLOSED. Numește acest nod n .
5. Dacă n este un nod scop, algoritmul se încheie cu succes, iar soluția este cea obținută prin urmarea în sens invers a unui drum de-a lungul arcelor din arborele T_r , de la n la n_0 . (Arcele sunt create la pasul 6).
6. Extinde nodul n , generând o mulțime, M , de succesori. Include M ca succesori ai lui n în T_r , prin crearea de arce de la n la fiecare membru al mulțimii M .
7. Reordonează lista OPEN, fie în concordanță cu un plan arbitrar, fie în mod euristic.
8. Mergi la pasul 3.

Observație: Acest algoritm poate fi folosit pentru a efectua căutări de tip best-first, breadth-first sau depth-first. În cazul algoritmului *breadth-first* noile noduri sunt puse la sfârșitul listei OPEN (organizată ca o coadă), iar nodurile nu sunt reordonate. În cazul căutării de tip *depth-first* noile noduri sunt plasate la începutul listei OPEN (organizată ca o stivă). În cazul căutării de tip *best-first*, numită și căutare euristică, lista OPEN este reordonată în funcție de meritele euristice ale nodurilor.

Algoritmul A*

Vom particulariza algoritmul GraphSearch la un algoritm de căutare best-first care reordonează, la pasul 7, nodurile listei OPEN în funcție de *valorile crescătoare ale funcției \hat{f}* . Această versiune a algoritmului GraphSearch se va numi Algoritmul A*.

Pentru a specifica familia funcțiilor \hat{f} care vor fi folosite, introducem următoarele notații:

- $h(n)$ = costul *efectiv* al drumului de cost minim dintre nodul n și un nod-scop, luând în considerație toate nodurile-scop posibile și toate drumurile posibile de la n la ele;
- $g(n)$ = costul unui drum de cost minim de la nodul de start n_0 la nodul n .

Atunci, $f(n) = g(n) + h(n)$ este costul unui drum de cost minim de la n_0 la un nod-scop, drum ales dintre toate drumurile care trebuie să treacă prin nodul n .

Observație: $f(n_0) = h(n_0)$ reprezintă costul unui drum de cost minim nerestricționat, de la nodul n_0 la un nod-scop.

Pentru fiecare nod n , fie $\hat{h}(n)$, numit *factor heuristic*, o estimație a lui $h(n)$ și fie $\hat{g}(n)$, numit *factor de adâncime*, costul drumului de cost minim până la n găsit de A^* până la pasul curent. Algoritmul A^* va folosi funcția $\hat{f} = \hat{g} + \hat{h}$.

➤ Observație:

În definiția Algoritmului A* de până acum nu s-a ținut cont de următoarea problemă: ce se întâmplă dacă graful implicit în care se efectuează căutarea nu este un arbore? Cu alte cuvinte, există mai mult decât o unică secvență de acțiuni care pot conduce la aceeași stare a lumii plecând din starea inițială. (Există situații în care fiecare dintre succesorii nodului n îl are pe n ca succesor i.e. acțiunile sunt reversibile). Pentru a rezolva astfel de cazuri, pasul 6 al algoritmului GraphSearch trebuie înlocuit cu următorul pas 6' :

6'. Extinde nodul n , generând o mulțime, M , de succesori care nu sunt deja *părinți* ai lui n în T_r . Instalează M ca succesori ai lui n în T_r prin crearea de arce de la n la fiecare membru al lui M .

Pentru a rezolva problema ciclurilor mai lungi, se înlocuiește pasul 6 prin următorul pas 6'':

6''. Extinde nodul n , generând o mulțime, M , de succesori care nu sunt deja *strămoși* ai lui n în T_r . Instalează M ca succesori ai lui n în T_r prin crearea de arce de la n la fiecare membru al lui M .

➤ Observație:

Pentru a verifica existența acestor cicluri mai lungi, trebuie văzut dacă structura de date care etichetează fiecare succesor al nodului n este egală cu structura de date care etichetează pe oricare dintre strămoșii nodului n . Pentru structuri de date complexe, acest pas poate mări complexitatea algoritmului. Pasul 6 modificat în pasul 6'' face însă ca algoritmul să nu se mai învârtă în cerc, în căutarea unui drum la scop.

➤ **Observație:**

Există încă posibilitatea de a vizita aceeași stare a lumii via drumuri diferite. Dacă două noduri din T_r sunt etichetate cu aceeași structură de date, vor avea sub ele subarbori identici. Prin urmare, algoritmul va duplica anumite eforturi de căutare.

- Pentru a preveni duplicarea efortului de căutare atunci când nu s-au impus condiții suplimentare asupra lui f , sunt necesare niște modificări în algoritmul A^* , și anume: deoarece căutarea poate ajunge la același nod de-a lungul unor drumuri diferite, algoritmul A^* generează un *graf de căutare*, notat cu G . G este structura de noduri și de arce generată de A^* pe măsură ce algoritmul extinde nodul inițial, succesorii lui ș.a.m.d.. A^* menține și un arbore de căutare, T_r .
- T_r , un subgraf al lui G , este arborele cu cele mai bune drumuri (de cost minim) produse până la pasul curent, drumuri până la toate nodurile din graful de căutare. Prin urmare, unele drumuri pot fi în graful de căutare, dar nu și în arborele de căutare.
- Graful de căutare este menținut deoarece căutări ulterioare pot găsi drumuri mai scurte, care folosesc anumite arce din graful de căutare anterior ce nu se aflau și în arborele de căutare anterior.

Dăm, în continuare, versiunea algoritmului A^* care menține graful de căutare. În practică, această versiune este folosită mai rar deoarece, de obicei, se pot impune condiții asupra lui \hat{f} care garantează faptul că, atunci când algoritmul A^* extinde un nod, el a găsit deja drumul de cost minim până la acel nod.

Algoritmul A*

1. Creează un graf de căutare G , constând numai din nodul inițial n_0 . Plasează n_0 într-o listă numită OPEN.
2. Creează o listă numită CLOSED, care inițial este vidă.
3. Dacă lista OPEN este vidă, EXIT cu eșec.
4. Selectează primul nod din lista OPEN, înlătură-l din OPEN și plasează-l în lista CLOSED. Numește acest nod n .
5. Dacă n este un nod scop, oprește execuția cu succes. Returnează soluția obținută urmând un drum de-a lungul pointerilor de la n la n_0 în G . (Pointerii definesc un arbore de căutare și sunt stabiliți la pasul 7).
6. Extinde nodul n , generând o mulțime, M , de succesori ai lui care nu sunt deja strămoși ai lui n în G . Instalează acești membri ai lui M ca succesori ai lui n în G .
7. Stabilește un pointer către n de la fiecare dintre membrii lui M care nu se găseau deja în G (adică nu se aflau deja nici în OPEN, nici în CLOSED). Adaugă acești membri ai lui M listei OPEN. Pentru fiecare membru, m , al lui M , care se afla deja în OPEN sau în CLOSED, redirecționează pointerul său către n , dacă cel mai bun drum la m găsit până în acel moment trece prin n . Pentru fiecare membru al lui M care se află deja în lista CLOSED, redirecționează pointerii fiecăruia dintre descendenții săi din G astfel încât aceștia să țină seama înapoi de-a lungul celor mai bune drumuri până la acești descendenți, găsite până în acel moment.
8. Reordonează lista OPEN în ordinea valorilor crescătoare ale funcției \hat{f} . (Eventuale legături între valori minimale ale lui \hat{f} sunt rezolvate în favoarea nodului din arborele de căutare aflat la cea mai mare adâncime).
9. Mergi la pasul 3.

Observație:

La pasul 7 sunt redirecționați pointeri de la un nod dacă procesul de căutare descoperă un drum la acel nod care are costul mai mic decât acela indicat de pointerii existenți. Redirecționarea pointerilor descendenților nodurilor care deja se află în lista CLOSED economisește efortul de căutare, dar poate duce la o cantitate exponențială de calcule. De aceea, această parte a pasului 7 de obicei nu este implementată. Unii dintre acești pointeri vor fi până la urmă redirecționați oricum, pe măsură ce căutarea progresează. (A se vedea slide-urile cu implementarea algoritmului).

Admisibilitatea Algoritmului A*

Există anumite condiții asupra grafurilor și a lui \hat{h} care garantează că algoritmul A*, aplicat acestor grafuri, găsește întotdeauna drumuri de cost minim. Condițiile asupra *grafurilor* sunt:

1. Orice nod al grafului, dacă admite succesori, are un număr finit de succesori.
2. Toate arcele din graf au costuri mai mari decât o cantitate pozitivă, ϵ .

Condiția asupra lui \hat{h} este:

3. Pentru toate nodurile n din graful de căutare, $\hat{h}(n) \leq h(n)$. Cu alte cuvinte, \hat{h} nu supraestimează niciodată valoarea efectivă h . O asemenea funcție \hat{h} este uneori numită un *estimator optimist*.

➤ Observație:

Este relativ ușor să se găsească, în probleme, o funcție \hat{h} care satisface această *condiție a limitei de jos*. De exemplu, în probleme de găsire a drumurilor în cadrul unor grafuri ale căror noduri sunt orașe, distanța de tip linie dreaptă de la un oraș n la un oraș-scop constituie o limită inferioară asupra distanței reprezentând un drum optim de la nodul n la nodul-scop.

❖ Cu cele trei condiții formulate anterior, algoritmul A^* garantează găsirea unui drum optim la un scop, în cazul în care există un drum la scop.

➤ Teoremă

Atunci când sunt îndeplinite condițiile asupra grafurilor și asupra lui \hat{h} enunțate anterior și cu condiția să existe un drum de cost finit de la nodul inițial, n_0 , la un nod-scop, algoritmul A^* garantează găsirea unui drum de cost minim la un scop.

➤ Definiție

Orice algoritm care garantează găsirea unui drum optim la scop este un algoritm admisibil.

➤ Atunci când cele trei condiții ale Teoremei sunt îndeplinite, A* este un algoritm admisibil. Prin extensie, vom spune că orice funcție \hat{h} care nu supraestimează pe h este admisibilă. (Estimația \hat{h} trebuie să aibă valori cât mai apropiate de cele ale lui h - pentru a nu mări efortul de căutare - dar fără a supraestima pe h).

➤ În cele ce urmează, atunci când ne vom referi la Algoritmul A*, vom presupune că cele trei condiții ale Teoremei sunt verificate.

➤ **A*** este un algoritm complet, admisibil și optim!

Complexitatea Algoritmului A*

S-a arătat că o creștere exponențială va interveni, în afara cazului în care eroarea în funcția euristică nu crește mai repede decât logaritmul costului efectiv al drumului. Cu alte cuvinte, *condiția pentru o creștere subexponențială* este:

$$|h(n) - h^*(n)| \leq O(\log h^*(n)),$$

unde $h^*(n)$ este *adevăratul* cost de a ajunge de la n la scop.

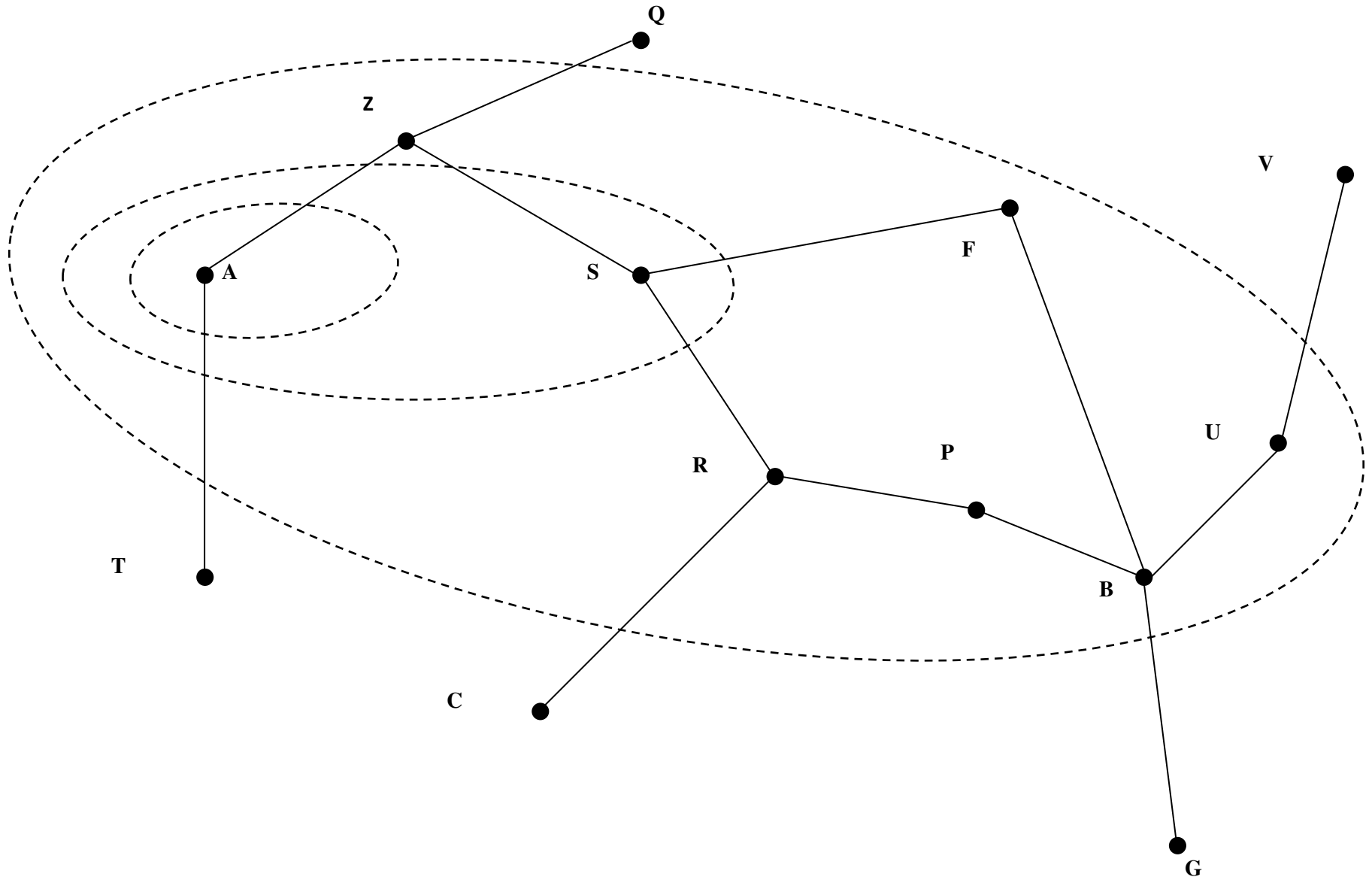
În afară de timpul mare calculator, algoritmul A* consumă și mult spațiu de memorie deoarece *păstrează în memorie toate nodurile generate*.

Algoritmi de căutare mai noi, de tip “memory-bounded” (cu limitare a memoriei), au reușit să înlăture neajunsul legat de problema spațiului de memorie folosit, fără a sacrifica optimalitatea sau completitudinea. Unul dintre aceștia este algoritmul IDA*.

Iterative Deepening A* (IDA*)

- Algoritmul IDA* se referă la o căutare iterativă în adâncime de tip A* și este o extensie logică a lui Iterative Deepening Search care folosește, în plus, informația euristică.
- În cadrul acestui algoritm fiecare iterație reprezintă o căutare de tip depth-first, iar căutarea de tip depth-first este modificată astfel încât ea să folosească o limită a costului și nu o limită a adâncimii.

Faptul că în cadrul algoritmului A^* f nu descrește niciodată de-a lungul oricărui drum care pleacă din rădăcină ne permite să trasăm, din punct de vedere conceptual, *contururi* în spațiul stărilor. Astfel, în interiorul unui contur, toate nodurile au valoarea $f(n)$ mai mică sau egală cu o aceeași valoare. În cazul algoritmului IDA* fiecare iterație extinde toate nodurile din interiorul conturului determinat de costul f curent, după care se trece la conturul următor. De îndată ce căutarea în interiorul unui contur dat a fost completată, este declanșată o nouă iterație, folosind un nou cost f , corespunzător următorului contur. Figura următoare prezintă căutări iterative în interiorul câte unui contur.



Algoritmul IDA* este optim cu aceleași amendamente ca și A*. Deoarece este de tip depth-first *nu necesită decât un spațiu proporțional cu cel mai lung drum pe care îl explorează.*

Dacă δ este cel mai mic cost de operator, iar f^* este costul soluției optime, atunci, în cazul cel mai nefavorabil, IDA* va necesita spațiu pentru memorarea a $\frac{bf^*}{\delta}$ noduri, unde b este același factor de ramificare.

Complexitatea de timp a algoritmului depinde în mare măsură de numărul valorilor diferite pe care le poate lua funcția euristică.

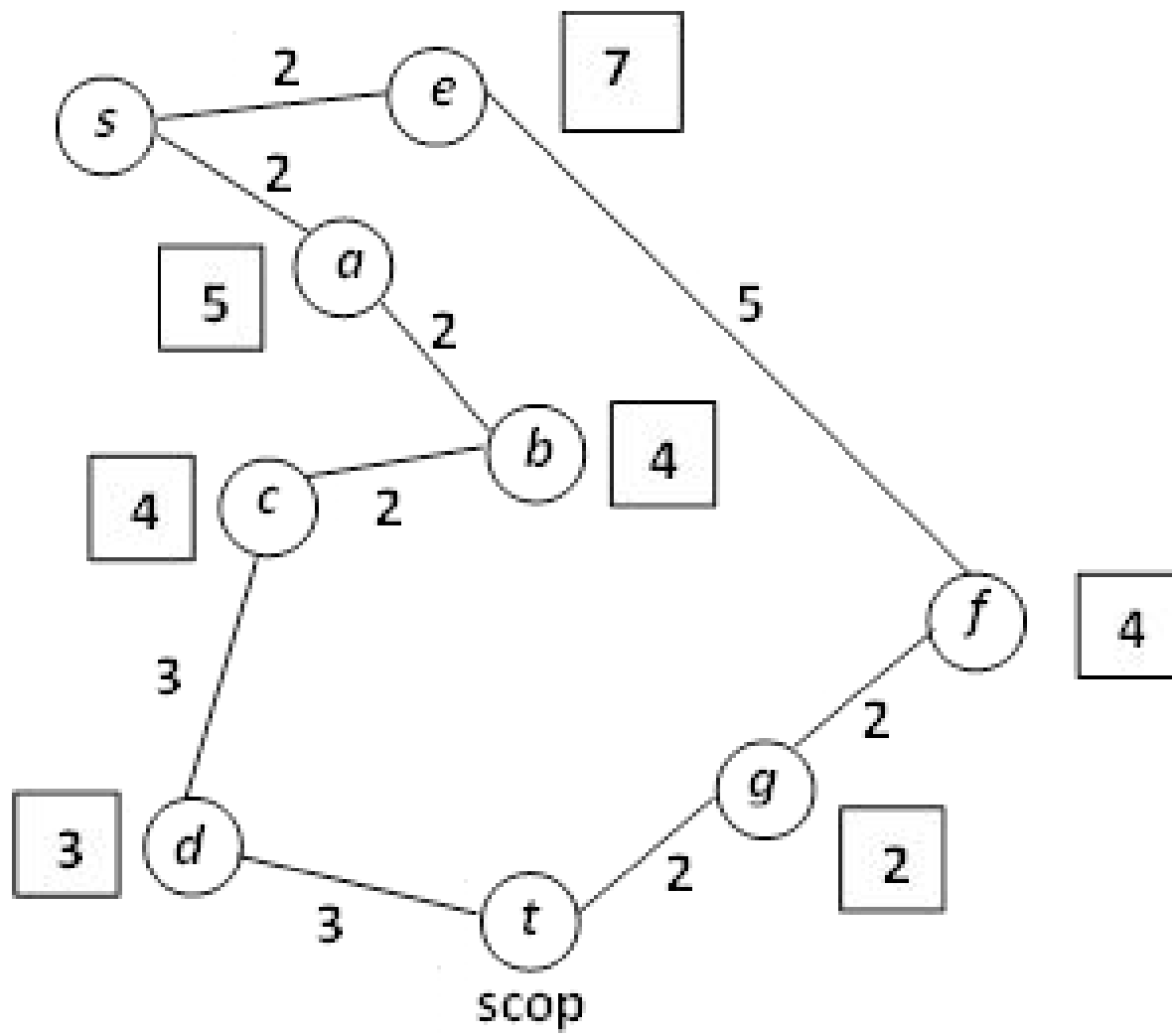
Implementarea căutării de tip best-first

- Vom imagina căutarea de tip best-first funcționând în felul următor: căutarea constă dintr-un număr de subprocese "concurente", fiecare explorând alternativa sa, adică propriul subarbore. Subarborii au subarbori, care vor fi la rândul lor explorați de subprocese ale subproceselor, ș.a.m.d..
- Dintre toate aceste subprocese doar unul este activ la un moment dat și anume cel care se ocupă de alternativa cea mai promițătoare (adică alternativa corespunzătoare celei mai mici \hat{f} - valori). Celelalte procese așteaptă până când \hat{f} - valorile se schimbă astfel încât o altă alternativă devine mai promițătoare, caz în care procesul corespunzător acesteia devine activ.

➤ Acest mecanism de activare-dezactivare poate fi privit după cum urmează: procesului corespunzător alternativei curente de prioritate maximă i se alocă un buget și, atâta vreme cât acest buget nu este epuizat, procesul este activ. Pe durata activității sale, procesul își expandează propriul subarbore, iar în cazul atingerii unei stări-scop este anunțată găsirea unei soluții. Bugetul acestei funcționări este determinat de \hat{f} -valoarea corespunzătoare celei mai apropiate alternative concurente.

Exemplu

- Considerăm orașele $s, a, b, c, d, e, f, g, t$ unite printr-o rețea de drumuri ca în figura care urmează. Aici fiecare drum direct între două orașe este etichetat cu lungimea sa; numărul din căsuța alăturată unui oraș reprezintă distanța în linie dreaptă între orașul respectiv și orașul t . Ne punem problema determinării celui mai scurt drum între orașul s și orașul t utilizând strategia best-first.
- Definim, în acest scop, funcția \hat{h} , bazându-ne pe distanța în linie dreaptă între două orașe. Astfel, pentru un oraș X , definim
$$\hat{f}(X) = \hat{g}(X) + \hat{h}(X) = \hat{g}(X) + \text{dist}(X, t)$$
unde: $\text{dist}(X, t)$ reprezintă distanța în linie dreaptă între X și t .
- În acest exemplu, căutarea de tip best-first este efectuată prin intermediul a două procese, P_1 și P_2 , ce explorează fiecare câte una din cele două căi alternative. Calea de la s la t via nodul a corespunde procesului P_1 , iar calea prin nodul e corespunde procesului P_2 .



În stadiile inițiale, procesul P_1 este mai activ, deoarece \hat{f} - valorile de-a lungul căii corespunzătoare lui sunt mai mici decât \hat{f} - valorile de-a lungul celeilalte căi. Atunci când P_1 explorează c , iar procesul P_2 este încă la e , $\hat{f}(c) = \hat{g}(c) + \hat{h}(c) = 6 + 4 = 10$, $\hat{f}(e) = \hat{g}(e) + \hat{h}(e) = 2 + 7 = 9$ și deci $\hat{f}(e) < \hat{f}(c)$. În acest moment, situația se schimbă: procesul P_2 devine activ, iar procesul P_1 intră în așteptare. În continuare, $\hat{f}(c) = 10$, $\hat{f}(f) = 11$, $\hat{f}(c) < \hat{f}(f)$ și deci P_1 devine activ și P_2 intră în așteptare. Pentru că $\hat{f}(d) = 12 > 11$, procesul P_1 va reintra în așteptare, iar procesul P_2 va rămâne activ până când se va atinge starea scop t .

Căutarea schițată mai sus pornește din nodul inițial și este continuată cu generarea unor noduri noi, conform relației de succesiune. În timpul acestui proces, este generat un arbore de căutare, a cărui rădăcină este nodul de start. Acest arbore este expandat în direcția cea mai promițătoare conform \hat{f} - valorilor, până la găsirea unei soluții.

JOCURILE CA PROBLEME DE CĂUTARE

- Jocurile reprezintă o arie de aplicație pentru algoritmii euristici.
- Jocurile de două persoane sunt în general complicate datorită existenței unui oponent ostil și imprevizibil. De aceea ele sunt interesante din punctul de vedere al dezvoltării euristicilor, dar aduc multe dificultăți în dezvoltarea și aplicarea algoritmilor de căutare.
- Oponentul introduce incertitudinea, întrucât nu se știe niciodată ce va face acesta la pasul următor.
- În esență, toate programele referitoare la jocuri trebuie să trateze așa-numita problemă de contingență. (Aici *contingență* are sensul de *întâmplare*).

- Incertitudinea care intervine în cazul jocurilor nu este de aceeași natură cu cea introdusă, de pildă, prin aruncarea unui zar sau cu cea determinată de starea vremii. Oponentul va încerca, pe cât posibil, să facă mutarea cea mai puțin benignă, în timp ce zarul sau vremea sunt presupuse a nu lua în considerație scopurile agentului.
- Complexitatea jocurilor introduce *un tip de incertitudine complet nou*. Astfel, incertitudinea se naște nu datorită faptului că există informație care lipsește, ci datorită faptului că jucătorul nu are timp să calculeze consecințele exacte ale oricărei mutări. Din acest punct de vedere, jocurile se aseamănă infinit mai mult cu lumea reală decât problemele de căutare standard.

JOCURI DE DOUA PERSOANE CU INFORMATIE COMPLETA

Ne vom referi la tehnici de joc corespunzătoare unor jocuri de două persoane cu informație completă, cum ar fi șahul.

În cazul jocurilor interesante, arborii rezultați sunt mult prea complecși pentru a se putea realiza o căutare exhaustivă, astfel încât sunt necesare abordări de o natură diferită. Una dintre metodele clasice se bazează pe „principiul minimax”, implementat în mod eficient sub forma Algoritmului Alpha-Beta (bazat pe așa-numita tehnică de alpha-beta retezare).

- Întrucât, în cadrul unui joc, există, de regulă, limite de timp, jocurile penalizează ineficiența extrem de sever. Astfel, dacă o implementare a căutării de tip A^* , care este cu 10% mai puțin eficientă, este considerată satisfăcătoare, un program pentru jocul de șah care este cu 10% mai puțin eficient în folosirea timpului disponibil va duce la pierderea partidei. Din această cauză, studiul nostru se va concentra asupra *tehnicilor de alegere a unei bune mutări atunci când timpul este limitat*.
- Tehnica de “retezare” ne va permite să ignorăm porțiuni ale arborelui de căutare care nu pot avea nici un rol în stabilirea alegerii finale, iar funcțiile de evaluare euristice ne vor permite să aproximăm utilitatea reală a unei stări fără a executa o căutare completă.

O definire formală a jocurilor

Tipul de jocuri la care ne vom referi în continuare este acela al jocurilor de două persoane cu informație perfectă sau completă. În astfel de jocuri:

- există doi jucători care efectuează mutări în mod alternativ;
- ambii jucători dispun de informația completă asupra situației curente a jocului;
- jocul se încheie atunci când este atinsă o poziție calificată ca fiind “terminală” de către regulile jocului - spre exemplu, “mat” în jocul de șah.

Un asemenea joc poate fi reprezentat printr-un arbore de joc în care nodurile corespund situațiilor (stărilor), iar arcele corespund mutărilor. Situația inițială a jocului este reprezentată de nodul rădăcină, iar frunzele arborelui corespund pozițiilor terminale.

Un joc poate fi definit, în mod formal, ca fiind *un anumit tip de problemă de căutare având următoarele componente:*

- *starea inițială*, care include poziția de pe tabla de joc și o indicație referitoare la cine face prima mutare;
- *o mulțime de operatori*, care definesc mișcările permise (“legale”) unui jucător;
- *un test terminal*, care determină momentul în care jocul ia sfârșit;
- *o funcție de utilitate* (numită și *funcție de plată*), care acordă o valoare numerică rezultatului unui joc; în cazul jocului de șah, spre exemplu, rezultatul poate fi *câștig*, *pierdere* sau *remiză*, situații care pot fi reprezentate prin valorile 1, -1 sau 0.

- Vom lua în considerație cazul general al unui joc cu doi jucători, pe care îi vom numi MAX și respectiv MIN.
- MAX va face prima mutare, după care jucătorii vor efectua mutări pe rând, până când jocul ia sfârșit.
- La finalul jocului vor fi acordate puncte jucătorului câștigător (sau vor fi acordate anumite penalizări celui care a pierdut).

- Dacă un joc ar reprezenta o problemă standard de căutare, atunci acțiunea jucătorului MAX ar consta din căutarea unei secvențe de mutări care conduc la o stare terminală reprezentând o stare câștigătoare (conform funcției de utilitate) și din efectuarea primei mutări aparținând acestei secvențe.
- Acțiunea lui MAX interacționează însă cu cea a jucătorului MIN. Prin urmare, MAX trebuie să găsească o strategie care va conduce la o stare terminală câștigătoare, indiferent de acțiunea lui MIN. Această strategie include mutarea corectă a lui MAX corespunzătoare fiecărei mutări posibile a lui MIN.
- În cele ce urmează, vom arăta cum poate fi găsită strategia optimă (sau rațională), deși în realitate nu vom dispune de timpul necesar pentru a o calcula.

Algoritmul Minimax

➤ JUCATORI: MIN și MAX

- MAX reprezintă jucătorul care încearcă să câștige sau să își maximizeze avantajul avut.
- MIN este oponentul care încearcă să minimizeze scorul lui MAX.
- Se presupune că MIN folosește aceeași informație și încearcă întotdeauna să se mute la acea stare care este cea mai nefavorabilă lui MAX.

➤ Algoritmul Minimax este conceput pentru a determina strategia optimă corespunzătoare lui MAX și, în acest fel, pentru a decide care este cea mai bună primă mutare.

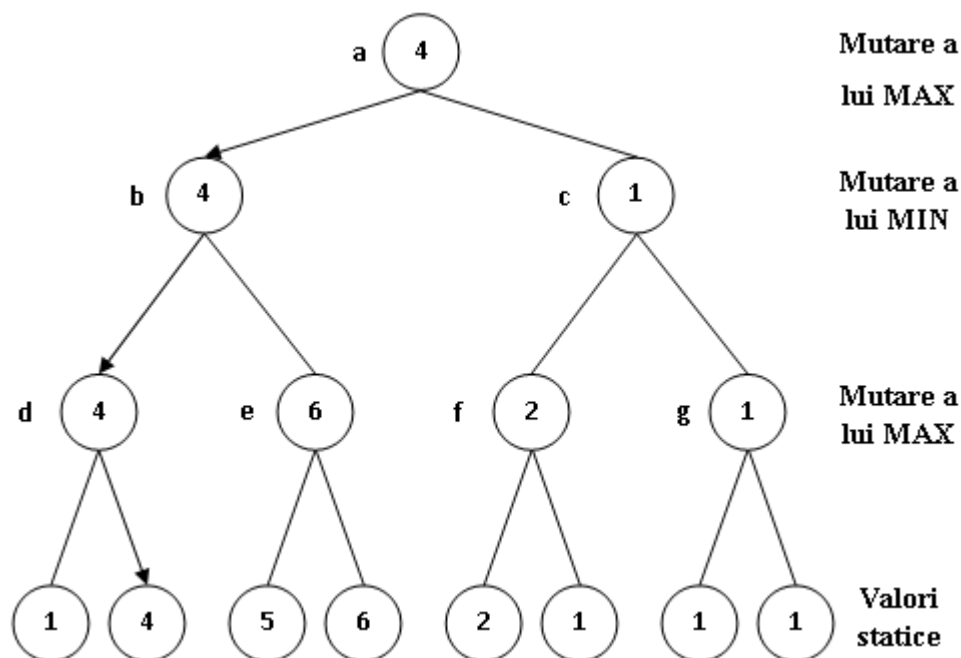
Algoritmul Minimax

1. Generează întregul arbore de joc, până la stările terminale.
2. Aplică funcția de utilitate fiecărei stări terminale pentru a obține valoarea corespunzătoare stării.
3. Deplasează-te înapoi în arbore, de la nodurile-frunze spre nodul-rădăcină, determinând, corespunzător fiecărui nivel al arborelui, valorile care reprezintă utilitatea nodurilor aflate la acel nivel. Propagarea acestor valori la niveluri anterioare se face prin intermediul nodurilor- părinte succesive, conform următoarei reguli:
 - dacă starea-părinte este un nod de tip MAX, atribuie-i maximul dintre valorile avute de fiii săi;
 - dacă starea-părinte este un nod de tip MIN, atribuie-i minimul dintre valorile avute de fiii săi.
4. Ajuns în nodul-rădăcină, alege pentru MAX acea mutare care conduce la valoarea maximă.

- **Observație:**

Decizia luată la pasul 4 al algoritmului se numește *decizia minimax*, întrucât ea maximizează utilitatea, în ipoteza că oponentul joacă perfect cu scopul de a o minimiza.

Un arbore de căutare cu valori minimax determinate conform Algoritmului Minimax este cel din figura următoare:



Valorile pozițiilor de la ultimul nivel sunt determinate de către funcția de utilitate și se numesc valori statice. Valorile minimax ale nodurilor interne sunt calculate în mod dinamic, în manieră bottom-up, nivel cu nivel, până când este atins nodul-rădăcină. Valoarea rezultată, corespunzătoare acestuia, este 4 și, prin urmare, cea mai bună mutare a lui MAX din poziția *a* este *a-b*. Cel mai bun răspuns al lui MIN este *b-d*. Această secvență a jocului poartă denumirea de variație principală. Ea definește jocul optim de tip minimax pentru ambele părți.

- Se observă că valoarea pozițiilor de-a lungul variației principale nu variază. Prin urmare, mutările corecte sunt cele care *conservă valoarea jocului*.

➤ Complexitate:

Dacă adâncimea maximă a arborelui este m și dacă există b “mutări legale” la fiecare punct, atunci complexitatea de timp a Algoritmului Minimax este $O(b^m)$. Algoritmul reprezintă o căutare de tip depth-first (deși aici este sugerată o implementare bazată pe recursivitate și nu una care folosește o coadă de noduri), astfel încât cerințele sale de spațiu sunt numai liniare în m și b .

Observație:

În cazul jocurilor reale, cerințele de timp ale algoritmului sunt total nepractice, dar acest algoritm stă la baza atât a unor metode mai realiste, cât și a analizei matematice a jocurilor.

Observație:

Întrucât, pentru majoritatea jocurilor interesante, arborele de joc nu poate fi alcătuit în mod exhaustiv, au fost concepute diverse metode care se bazează pe căutarea efectuată numai într-o anumită porțiune a arborelui de joc. Printre acestea se numără și tehnica Minimax, care, în majoritatea cazurilor, va căuta în arborele de joc numai până la o anumită adâncime, de obicei constând în numai câteva mutări.

- ✓ Algoritmul general Minimax a fost amendat în două moduri: funcția de utilitate a fost înlocuită cu o funcție de evaluare, iar testul terminal a fost înlocuit de către un așa-numit test de tăiere.

➤ Cea mai directă abordare a problemei deținerii controlului asupra cantității de căutare care se efectuează este aceea de a fixa o limită a adâncimii, astfel încât testul de tăiere să aibă succes pentru toate nodurile aflate la sau sub adâncimea d . Limita de adâncime va fi aleasă astfel încât cantitatea de timp folosită să nu depășească ceea ce permit regulile jocului. O abordare mai robustă a acestei probleme este aceea care aplică „iterative deepening”. În acest caz, atunci când timpul expiră, programul întoarce mutarea selectată de către cea mai adâncă căutare completă.

Funcții de evaluare

O funcție de evaluare întoarce o estimație, realizată dintr-o poziție dată, a utilității așteptate a jocului. Ea are la bază evaluarea șanselor de câștigare a jocului de către fiecare dintre părți, pe baza calculării caracteristicilor unei poziții. Performanța unui program referitor la jocuri este extrem de dependentă de calitatea funcției de evaluare utilizate.

- **Funcția de evaluare trebuie să îndeplinească anumite condiții evidente:**
 - ✓ trebuie să concorde cu funcția de utilitate în ceea ce privește stările terminale;
 - ✓ calculele efectuate nu trebuie să dureze prea mult;
 - ✓ trebuie să reflecte în mod corect șansele efective de câștig.

- **O valoare a funcției de evaluare acoperă mai multe poziții diferite, grupate laolaltă într-o categorie de poziții etichetată cu o anumită valoare.** (Spre exemplu, în jocul de șah, fiecare *pion* poate avea valoarea 1, un *nebun* poate avea valoarea 3 șamd.. În poziția de deschidere evaluarea este 0 și toate pozițiile până la prima captură vor avea aceeași evaluare. Dacă MAX reușește să captureze un nebun fără a pierde o piesă, atunci poziția rezultată va fi evaluată la valoarea 3. Toate pozițiile de acest fel ale lui MAX vor fi grupate într-o *categorie* etichetată cu “3”).

- **Funcția de evaluare trebuie să reflecte șansa ca o poziție aleasă la întâmplare dintr-o asemenea categorie să conducă la câștig (sau la pierdere sau la remiză) pe baza experienței anterioare (VEZI tehnici de învățare).**

- Funcția de evaluare cel mai frecvent utilizată presupune că valoarea unei piese poate fi stabilită independent de celelalte piese existente pe tablă. Un asemenea tip de funcție de evaluare se numește funcție liniară ponderată, întrucât are o expresie de forma

$$w_1 f_1 + w_2 f_2 + \dots + w_n f_n,$$

unde valorile w_i , $i = \overline{1, n}$ reprezintă ponderile, iar f_i , $i = \overline{1, n}$ sunt caracteristicile unei anumite poziții. În cazul jocului de șah, spre exemplu w_i , $i = \overline{1, n}$ ar putea fi valorile pieselor (1 pentru pion, 3 pentru nebun etc.), iar f_i , $i = \overline{1, n}$ ar reprezenta numărul pieselor de un anumit tip aflate pe tabla de șah.

În construirea formulei liniare trebuie mai întâi alese caracteristicile, operație urmată de ajustarea ponderilor până în momentul în care programul joacă suficient de bine. Această a doua operație poate fi automatizată punând programul să joace multe partide cu el însuși, dar alegerea unor caracteristici adecvate nu a fost încă realizată în mod automat.

(caracteristica \equiv feature)

O implementare eficientă a principiului Minimax: Algoritmul Alpha-Beta

Tehnica pe care o vom examina, în cele ce urmează, este numită în literatura de specialitate alpha-beta pruning (“alpha-beta retezare”). Atunci când este aplicată unui arbore de tip minimax standard, ea va întoarce aceeași mutare pe care ar furniza-o și Algoritmul Minimax, dar într-un timp mai scurt, întrucât realizează o retezare a unor ramuri ale arborelui care nu pot influența decizia finală.

Principiul general al acestei tehnici constă în a considera un nod oarecare n al arborelui, astfel încât jucătorul poate alege să facă o mutare la acel nod. Dacă același jucător dispune de o alegere mai avantajoasă, m , fie la nivelul nodului părinte al lui n , fie în orice punct de decizie aflat mai sus în arbore, atunci n nu va fi niciodată atins în timpul jocului. Prin urmare, de îndată ce, în urma examinării unora dintre descendenții nodului n , ajungem să deținem suficientă informație relativ la acesta, îl putem înlătura.

- **Ideea tehnicii de alpha-beta retezare: a găsi o mutare “suficient de bună”, nu neapărat cea mai bună, dar suficient de bună pentru a se lua decizia corectă. Această idee poate fi formalizată prin introducerea a *două limite, alpha și beta*, reprezentând limitări ale valorii de tip minimax corespunzătoare unui nod intern.**
- **Semnificația acestor limite este următoarea:**
 - *alpha este valoarea minimă* pe care este deja garantat că o va obține MAX;
 - *beta este valoarea maximă* pe care MAX poate spera să o atingă.
- Din punctul de vedere al jucătorului MIN, beta este valoarea cea mai nefavorabilă pentru MIN pe care acesta o va atinge.
- **Valoarea efectivă** care va fi găsită se află între *alpha și beta*.

Valoarea alpha, asociată nodurilor de tip MAX, nu poate niciodată să descrească, iar valoarea beta, asociată nodurilor de tip MIN, nu poate niciodată să crească.

- **Alpha este scorul cel mai prost pe care îl poate obține MAX, presupunând că MIN joacă perfect.**

Dacă, spre exemplu, valoarea alpha a unui nod intern de tip MAX este 6, atunci MAX nu mai trebuie să ia în considerație nici o valoare internă mai mică sau egală cu 6 care este asociată oricărui nod de tip MIN situat sub el. În mod similar, dacă MIN are valoarea beta 6, el nu mai trebuie să ia în considerație nici un nod de tip MAX situat sub el care are valoarea 6 sau o valoare mai mare decât acest număr.

➤ Cele două reguli pentru încheierea căutării, bazată pe valori alpha și beta, sunt:

1. Căutarea poate fi oprită dedesubtul oricărui nod de tip MIN care are o valoare beta mai mică sau egală cu valoarea alpha a oricărui dintre strămoșii săi de tip MAX.
2. Căutarea poate fi oprită dedesubtul oricărui nod de tip MAX care are o valoare alpha mai mare sau egală cu valoarea beta a oricărui dintre strămoșii săi de tip MIN.

Dacă, referitor la o poziție, se arată că valoarea corespunzătoare ei se află în afara intervalului alpha-beta, atunci această informație este suficientă pentru a ști că poziția respectivă nu se află de-a lungul variației principale, chiar dacă nu este cunoscută valoarea *exactă* corespunzătoare ei.

- **Cunoașterea valorii exacte a unei poziții este necesară numai atunci când această valoare se află între alpha și beta.**

Din punct de vedere formal, putem defini o valoare de tip minimax a unui nod intern, P , $V(P, \alpha, \beta)$, ca fiind “*suficient de bună*” dacă satisface următoarele cerințe:

$$V(P, \alpha, \beta) < \alpha, \text{ dacă } V(P) < \alpha \quad (1)$$

$$V(P, \alpha, \beta) = V(P), \text{ dacă } \alpha \leq V(P) \leq \beta$$

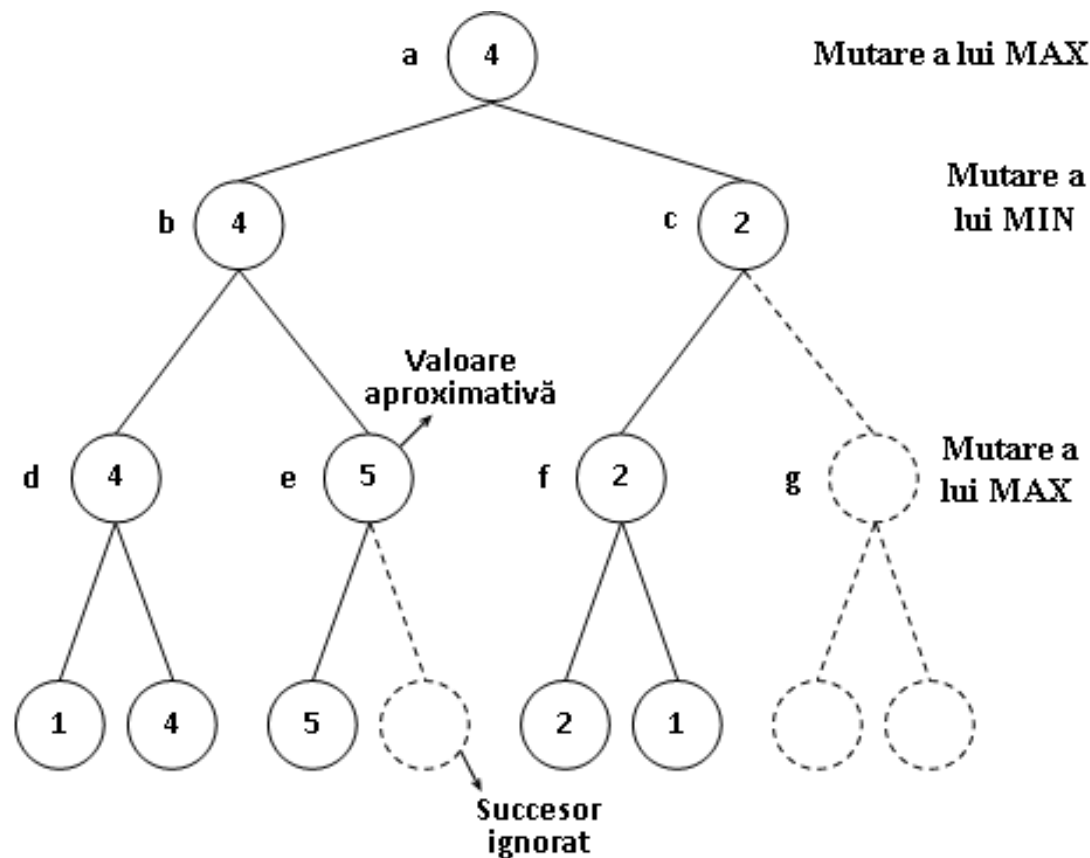
$$V(P, \alpha, \beta) > \beta, \text{ dacă } V(P) > \beta,$$

unde prin $V(P)$ am notat valoarea de tip minimax corespunzătoare unui nod intern.

Valoarea exactă a unui nod-rădăcină P poate fi întotdeauna calculată prin setarea limitelor după cum urmează:

$$V(P, -\infty, +\infty) = V(P).$$

Figura următoare ilustrează acțiunea Algoritmului Alpha-Beta în cazul arborelui anterior. Așa cum se vede în figură, unele dintre valorile de tip minimax ale nodurilor interne sunt aproximative. Totuși, aceste aproximări sunt suficiente pentru a se determina în mod exact valoarea rădăcinii. Se observă că Algoritmul Alpha-Beta reduce complexitatea căutării de la 8 evaluări statice la numai 5 evaluări de acest tip.



Căutarea de tip alpha-beta retează nodurile figurate în mod discontinuu. Ca rezultat, câteva dintre valorile intermediare nu sunt exacte (nodurile *c*, *e*), dar aproximările făcute sunt suficiente pentru a determina atât valoarea corespunzătoare rădăcinii, cât și variația principală, în mod exact.

Corespunzător arborelui din figură procesul de căutare decurge după cum urmează:

- 1. Începe din poziția a .**
- 2. Mutare la b .**
- 3. Mutare la d .**
- 4. Alege valoarea maximă a succesorilor lui d , ceea ce conduce la $V(d) = 4$.**
- 5. Întoarce-te în nodul b și execută o mutare de aici la e .**
- 6. Ia în considerație primul succesor al lui e a cărui valoare este 5. În acest moment, MAX, a cărui mutare urmează, are garantată, aflându-se în poziția e , cel puțin valoarea 5, indiferent care ar fi celelalte alternative plecând din e . Această informație este suficientă pentru ca MIN să realizeze că, la nodul b , alternativa e este inferioară alternativei d . Această concluzie poate fi trasă fără a cunoaște valoarea *exactă* a lui e . Pe această bază, cel de-al doilea succesor al lui e poate fi neglijat, iar nodului e i se poate atribui valoarea *aproximativă* 5.**

Un alt exemplu

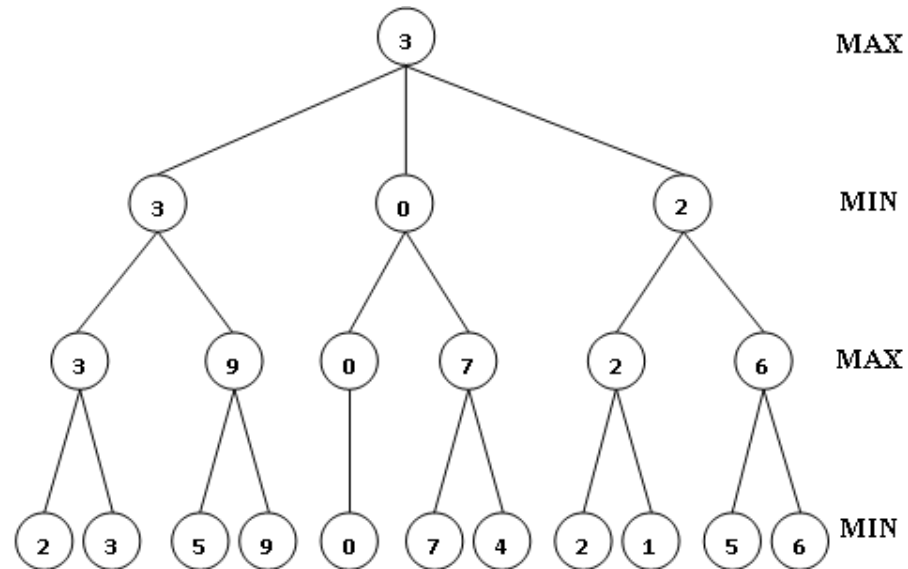


Fig. 1 (Alg. Minimax)

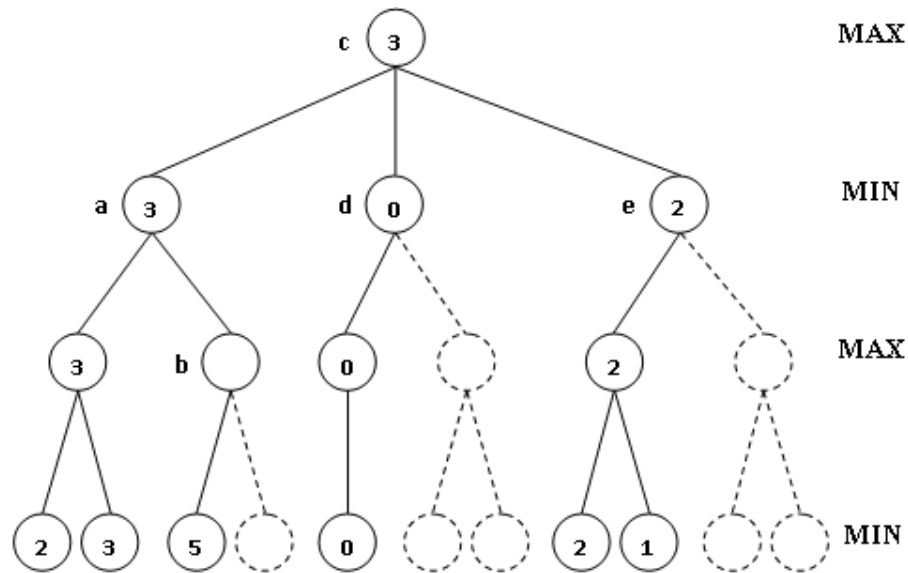


Fig. 2 (Alg. Alpha-Beta)

În Fig. 2: *a* are $\beta = 3$ (valoarea lui *a* nu va depăși 3);
b este β - retezat, deoarece $5 > 3$;
c are $\alpha = 3$ (valoarea lui *c* nu va fi mai mică decât 3);
d este α – retezat, deoarece $0 < 3$;
e este α – retezat, deoarece $2 < 3$;
c este 3.

Considerații privitoare la eficiență

Eficiența Algoritmului Alpha-Beta depinde de *ordinea în care sunt examinați succesorii*. Este preferabil să fie examinați mai întâi succesorii despre care se crede că ar putea fi cei mai buni. În mod evident, acest lucru nu poate fi realizat în întregime. Dacă el ar fi posibil, funcția care ordonează succesorii ar putea fi utilizată pentru a se juca un joc perfect. În ipoteza în care această ordonare ar putea fi realizată, s-a arătat că Algoritmul Alpha-Beta nu trebuie să examineze, pentru a alege cea mai bună mutare, decât $O(b^{d/2})$ noduri, în loc de $O(b^d)$, ca în cazul Algoritmului Minimax. Aceasta arată că factorul de ramificare efectiv este \sqrt{b} în loc de b – în cazul jocului de șah 6, în loc de 35. Cu alte cuvinte, Algoritmul Alpha-Beta poate “privi înainte” la o adâncime dublă față de Algoritmul Minimax, pentru a găsi același cost.

În cazul unei ordonări neprevăzute a stărilor în spațiul de căutare, Algoritmul Alpha-Beta poate dubla adâncimea spațiului de căutare (Nilsson 1980). Dacă există o anumită ordonare nefavorabilă a nodurilor, acest algoritm nu va căuta mai mult decât Algoritmul Minimax. Prin urmare, în cazul cel mai nefavorabil, Algoritmul Alpha-Beta nu va oferi nici un avantaj în comparație cu căutarea exhaustivă de tip minimax. În cazul unei ordonări favorabile însă, dacă notăm prin N numărul pozițiilor de căutare terminale evaluate în mod static de către Algoritmul Minimax, s-a arătat că, în cazul cel mai bun, adică atunci când mutarea cea mai puternică este prima luată în considerație, Algoritmul Alpha-Beta nu va evalua în mod static decât \sqrt{N} poziții. În practică, o funcție de ordonare relativ simplă (cum ar fi încercarea mai întâi a capturilor, apoi a amenințărilor, apoi a mutărilor înainte, apoi a celor înapoi) ne poate apropia suficient de mult de rezultatul obținut în cazul cel mai favorabil.

Tipuri de cunoștințe

Cunoștințe relaționale simple

- Cea mai simplă modalitate de reprezentare a faptelor declarative constă în folosirea unei mulțimi de relații de același tip cu cele utilizate în sistemele de baze de date.
- Cunoștințele relaționale din acest tabel corespund unei mulțimi de atribute și de valori asociate, care împreună descriu obiectele bazei de cunoștințe.

| Student | Vârstă | An de studiu | Note la informatică |
|----------------|--------|--------------|---------------------|
| Popescu Andrei | 18 | I | 8-9 |
| Ionescu Maria | 18 | I | 9-10 |
| Hristea Oana | 20 | I | 7-8 |
| Pârvu Ana | 19 | II | 8-9 |
| Savu Andrei | 19 | II | 7-8 |
| Popescu Ana | 20 | III | 9-10 |

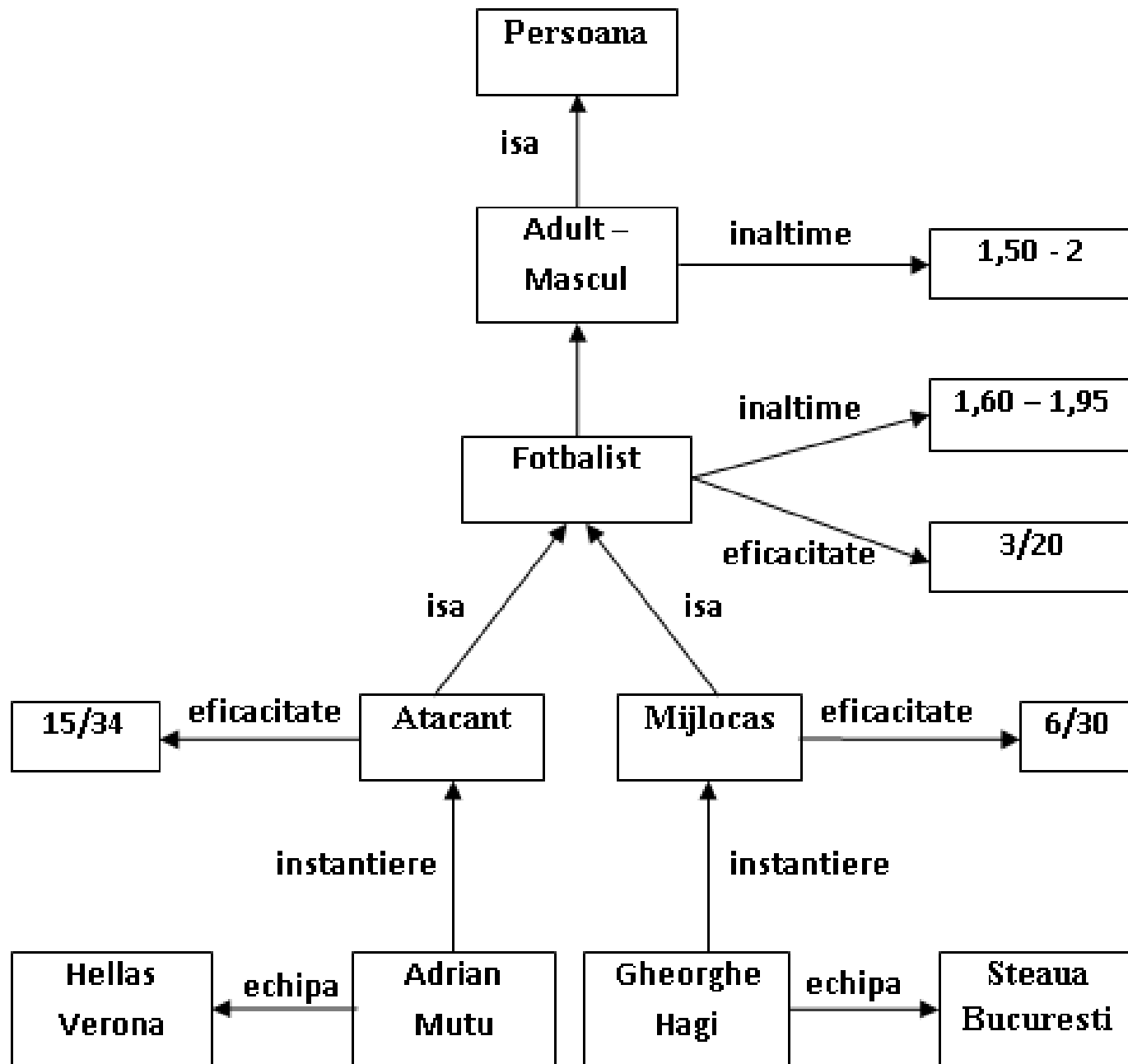
Cunoștințe care se moștenesc

- Este posibil ca reprezentarea de bază să fie îmbogățită cu mecanisme de inferență care operează asupra structurii reprezentării.
- Pentru ca această modalitate de reprezentare să fie eficientă, structura trebuie proiectată în așa fel încât ea să corespundă mecanismelor de inferență dorite.
- Una dintre cele mai utilizate forme de inferență este moștenirea proprietăților, prin care elemente aparținând anumitor clase moștenesc atribute și valori provenite de la clase mai generale, în care sunt incluse.
- Pentru a admite moștenirea proprietăților, obiectele trebuie să fie organizate în clase, iar clasele trebuie să fie aranjate în cadrul unei ierarhii.

➤ În fig. care urmează, sunt reprezentate cunoștințe legate de jocul de fotbal, cunoștințe organizate într-o structură de acest tip. În această reprezentare, liniile desemnează atribute. Nodurile figurate prin dreptunghiuri reprezintă obiecte și valori ale atributelor obiectelor.

- ✓ Aceste valori pot fi, la rândul lor, privite ca obiecte având atribute și valori ș.a.m.d..
- ✓ Săgețile corespunzătoare liniilor sunt orientate de la un obiect la valoarea lui (de-a lungul liniei desemnând atributul corespunzător).

- În exemplul din figura, toate obiectele și majoritatea atributelor care intervin corespund domeniului sportiv al jocului de fotbal și nu au o semnificație generală. Singurele două excepții sunt atributul isa, utilizat pentru a desemna *incluziunea între clase* și atributul instanțiere, folosit pentru a arăta *apartenența la o clasă*. Aceste două atribute, extrem de folosite, se află la baza *moștenirii proprietăților* ca tehnică de inferență.



Utilizând această tehnică de inferență, baza de cunoștințe poate asigura atât regăsirea faptelor care au fost memorate în mod explicit, precum și a faptelor care derivă din cele memorate în mod explicit, ca în următorul exemplu:

$$\text{eficacitate(Adrian_Mutu)} = 15/34$$

Este una dintre cele mai folosite tehnici de inferență!

- În acest exemplu, structura corespunzătoare reprezintă o *structură de tip “slot-and-filler”*. Ea mai poate fi privită și ca o rețea semantică sau ca o colecție de cadre.
- În cel din urmă caz (*colecție de cadre*), fiecare cadru individual reprezintă colecția atributelor și a valorilor asociate cu un anumit nod.
- O diferențiere exactă a acestor tipuri de reprezentări este greu de făcut. În general, termenul de sistem de cadre implică existența unei mai mari structurări a atributelor și a mecanismelor de inferență care le pot fi aplicate decât în cazul rețelelor semantice.

Cunoștințe inferențiale

- Puterea logicii tradiționale este adesea utilă pentru a se descrie toate inferențele necesare.
- Astfel de cunoștințe nu sunt utile decât în prezența unei proceduri de inferență care să le poată exploata.
- Există multe asemenea proceduri, dintre care unele fac raționamente de tipul “înainte”, de la fapte date către concluzii, iar altele raționează “înapoi”, de la concluziile dorite la faptele date. Una dintre procedurile cele mai folosite de acest tip este rezoluția, care folosește strategia contradicției.
- În general, logica furnizează o structură puternică în cadrul căreia sunt descrise legăturile dintre valori. Ea se combină adesea cu un alt limbaj puternic de descriere, cum ar fi o ierarhie de tip isa.

Cunoștințe procedurale

- Reprezentarea cunoștințelor descrisă până în prezent s-a concentrat asupra faptelor statice, declarative.
- Un alt tip de cunoștințe extrem de utile sunt *cunoștințele procedurale sau operaționale*, care specifică ce anume trebuie făcut și când.
- Cea mai simplă modalitate de reprezentare a cunoștințelor procedurale este cea sub formă de cod, într-un anumit limbaj de programare.
- În acest caz, mașina folosește cunoștințele atunci când execută codul pentru a efectua o anumită sarcină.
- Acest mod de reprezentare a cunoștințelor procedurale nu este însă cel mai fericit din punctul de vedere al adecvării inferențiale, precum și al eficienței în achiziție.

- Cea mai folosită tehnică de reprezentare a cunoștințelor procedurale în programele de inteligență artificială este aceea a utilizării regulilor de producție.
- Atunci când sunt îmbogățite cu informații asupra felului în care trebuie să fie folosite, regulile de producție sunt mai procedurale decât alte metode existente de reprezentare a cunoștințelor.
- Regulile de producție, numite și reguli de tip if-then, sunt instrucțiuni condiționale, care pot avea diverse interpretări, cum ar fi:
 - if condiție P then concluzie C
 - if situație S then acțiune A
 - if condițiile C1 și C2 sunt verificate then condiția C nu este verificată
- Regulile de producție sunt foarte utilizate în proiectarea sistemelor expert.

- Regulile de tip if-then adesea definesc relații logice între conceptele aparținând domeniului problemei. Relațiile pur logice pot fi caracterizate ca aparținând așa-numitelor cunoștințe categorice, adică acelor cunoștințe care vor fi întotdeauna adevărate.
- În unele domenii, cum ar fi diagnosticarea în medicină, predomină cunoștințele “moi” sau probabiliste. În cazul acestui tip de cunoștințe, regularitățile empirice sunt valide numai până la un anumit punct (adesea, dar nu întotdeauna). În astfel de cazuri, regulile de producție sunt modificate prin adăugarea la interpretarea lor logică a unei calificări de verosimilitate, obținându-se reguli de forma:

if conditie A then concluzie B cu certitudinea F

unde:

$F =$ factor de certitudine, măsură a încrederii sau certitudine subiectivă

Pentru calculul lui F: statistica Bayesiană

Reprezentarea cunoștințelor în sistemele expert

- Un sistem expert este un program care se comportă ca un expert într-un domeniu relativ restrans.
- Caracteristica majoră a sistemelor expert, numite și sisteme bazate pe cunoștințe, este aceea că ele se bazează pe cunoștințele unui expert uman în domeniul care este studiat.
- La baza sistemelor expert se află utilizarea în rezolvarea problemelor a unor mari cantități de cunoștințe specifice domeniului.

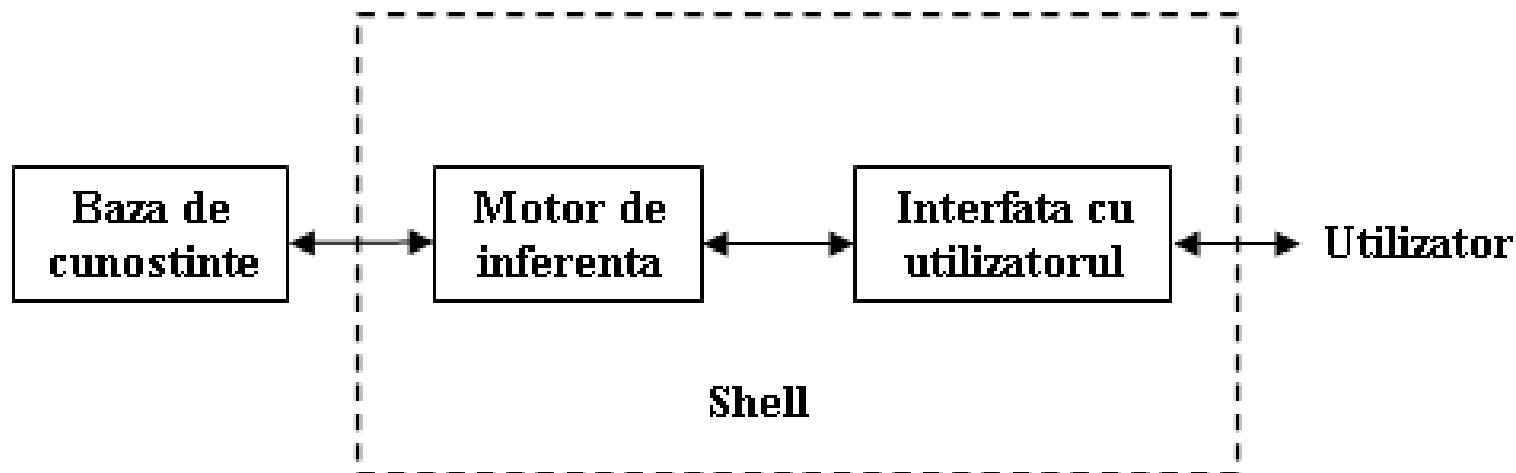
Alte caracteristici ale sistemului expert:

- să fie capabil să explice comportamentul său și deciziile luate - la fel cum o fac experții umani - prin generarea răspunsului pentru două tipuri de întrebări ale utilizatorului:
 - ✓ întrebare de tipul “*cum*”: *Cum* ai ajuns la această concluzie?
 - ✓ întrebare de tipul “*de ce*”: *De ce* te interesează această informație?
- să lucreze cu incertitudinea sau starea de a fi incomplet - informații incerte, incomplete sau care lipsesc; relații aproximative în domeniul problemei (de ex. efectul unui medicament asupra stării pacientului).

Structura de bază a unui sistem expert

Un sistem expert conține trei module principale, și anume:

- o bază de cunoștințe;
- un motor de inferență;
- o interfață cu utilizatorul.



Concluzii

- Regulile if-then formează lanțuri de forma
informatie input $\rightarrow \dots \rightarrow$ informatie dedusa
- Informația de tip input mai poartă denumirea de date sau manifestări.
- Informația dedusă constituie ipotezele care trebuie demonstrate sau cauzele manifestărilor sau diagnostice sau explicații.
- Atât înlănțuirea înainte, cât și cea înapoi (ca metode de inferență) presupun căutare, dar direcția de căutare este diferită pentru fiecare în parte.
- Înlănțuirea înapoi execută o căutare de la scopuri înspre date, din care cauză se spune despre ea că este orientată către scop.
- Înlănțuirea înainte caută pornind de la date înspre scopuri, fiind orientată către date.

Clase de metode pentru reprezentarea cunoștințelor

Principalele tipuri de reprezentări ale cunoștințelor sunt reprezentările *bazate pe logică* și cele de tip “*slot-filler*” (“deschizătură-umplutură”).

Reprezentările bazate pe logică aparțin unor două mari categorii, în funcție de instrumentele folosite în reprezentare, și anume:

- Logica - mecanismul principal îl constituie inferența logică.
- Regulile (folosite, de pildă, în sistemele expert) - principalele mecanisme sunt “înlănțuirea înainte” și “înlănțuirea înapoi”. O regulă este similară unei implicații logice, dar nu are o valoare proprie (regulile sunt aplicate, ele nu au una dintre valorile „true” sau „false”).

Reprezentările de tip slot-filler folosesc două categorii diferite de structuri:

- **Rețele semantice și grafuri conceptuale - o reprezentare distribuită (concepte legate între ele prin diverse relații). Principalul mecanism folosit este *căutarea*.**
- **Cadre și scripturi - o reprezentare structurată (grupuri de concepte și relații); sunt foarte utile în reprezentarea tipicității. Principalul mecanism folosit este împerecherea (potrivirea) sabloanelor (tiparelor) – „pattern matching”.**

REȚELE BAYESIENE

- Ideea: În lumea reală majoritatea evenimentelor sunt independente de celelalte, astfel încât interacțiunile dintre ele nu trebuie luate în considerație. În schimb, se poate folosi o reprezentare locală având rolul de a descrie grupuri de evenimente care interacționează (dependenta locala intre variabile).
- Relațiile de independență condiționată dintre variabile pot reduce substanțial numărul probabilităților condiționate care trebuie specificate.
- Structura de date folosită cel mai frecvent pentru a reprezenta dependența dintre variabile și pentru a da o specificație concisă a repartiției de probabilitate multidimensionale este rețeaua Bayesiană.

- Retelele Bayesiene sunt un tip de model grafic probabilist care utilizeaza inferenta Bayesiană pentru calculul probabilitatilor.
- Retelele Bayesiene isi propun sa modeleze dependenta conditionata si, prin urmare, cauzalitatea, prin reprezentarea dependentei conditionate sub forma muchiilor dintr-un graf directionat. Prin aceste relatii, se poate efectua inferenta asupra variabilelor aleatoare din graf.
- Reteaua Bayesiană constituie un graf directionat aciclic in care fiecare muchie corespunde unei dependente conditionate si fiecare nod corespunde unei variabile aleatoare unice.
- Pentru retelele Bayesiene s-au creat algoritmi performanti care efectueaza
 - ✓ Inferenta
 - sau
 - ✓ Invatarein retea Bayesiană.
- Vom studia un algoritm de inferenta in retele Bayesiene.

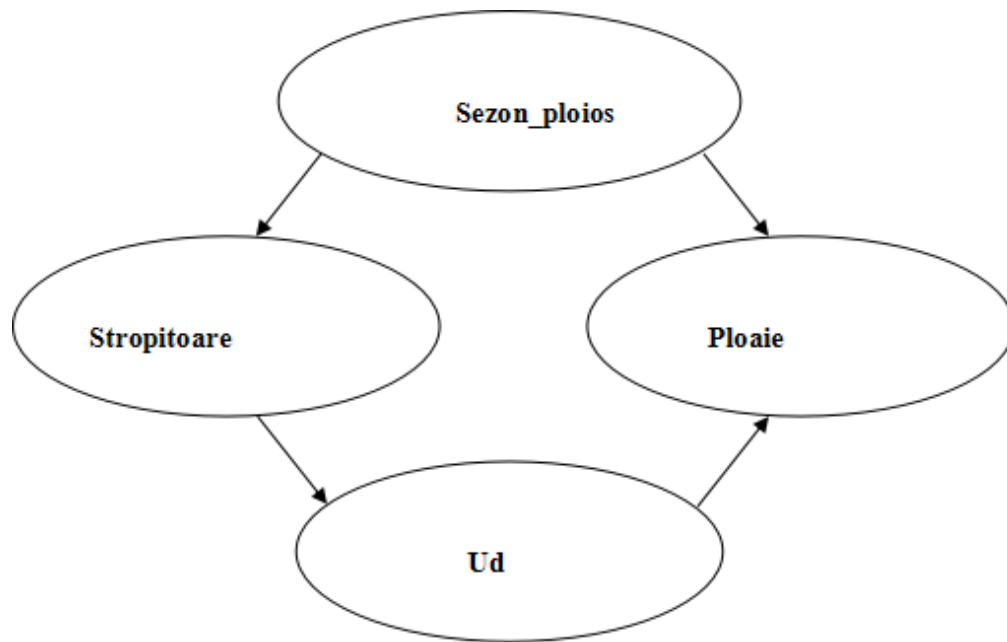
Definiția 1

O rețea Bayesiană este un graf cu următoarele proprietăți:

- (1) Graful este direcționat și aciclic.**
- (2) O mulțime de variabile aleatoare constituie nodurile rețelei.**
- (3) O mulțime de arce direcționate conectează perechi de noduri. În mod intuitiv, semnificația unui arc direcționat de la nodul X la nodul Y este aceea că X are o influență directă asupra lui Y .**
- (4) Fiecărui nod îi corespunde un tabel de probabilități condiționate care cuantifică efectele pe care părinții le au asupra nodului respectiv. (Părinții unui nod sunt toate acele noduri din care pleacă arce direcționate înspre acesta).**

- Un expert în domeniu decide relațiile directe de dependență condiționată care sunt valabile în domeniu.
- După stabilirea topologiei rețelei Bayesiene, nu mai este necesară decât specificarea probabilităților condiționate ale acelor noduri care participă în dependențe directe. Acestea vor fi folosite în calculul oricăror alte probabilități.
- Concret: se construiește un graf direcționat aciclic care reprezintă relațiile de cauzalitate dintre variabile. Variabilele dintr-un astfel de graf pot fi propoziționale (caz în care pot lua valorile TRUE sau FALSE) sau pot fi variabile care primesc valori de un alt tip (spre exemplu o temperatură a corpului sau măsurători făcute de către un dispozitiv de diagnosticare).

- Pentru a putea folosi graful cauzalității (rețea) ca bază a unui raționament de tip probabilist sunt necesare însă mai multe informații.
- În particular, este necesar să cunoaștem, pentru o valoare a unui nod părinte, ce dovezi sunt furnizate referitor la valorile pe care le poate lua nodul fiu.



Probabilitățile condiționate corespunzătoare pot fi date sub forma unui tabel de tipul:

| Atribut | Probabilitate |
|--|---------------|
| $p(\text{Ud} \text{Stropitoare}, \text{Ploaie})$ | 0.95 |
| $p(\text{Ud} \text{Stropitoare}, \neg \text{Ploaie})$ | 0.9 |
| $p(\text{Ud} \neg \text{Stropitoare}, \text{Ploaie})$ | 0.8 |
| $p(\text{Ud} \neg \text{Stropitoare}, \neg \text{Ploaie})$ | 0.1 |
| | |
| $p(\text{Stropitoare} \text{Sezon ploios})$ | 0.0 |
| $p(\text{Stropitoare} \neg \text{Sezon ploios})$ | 1.0 |
| | |
| $p(\text{Ploaie} \text{Sezon ploios})$ | 0.9 |
| $p(\text{Ploaie} \neg \text{Sezon ploios})$ | 0.1 |
| | |
| $p(\text{Sezon ploios})$ | 0.5 |

Exemple:

- **probabilitatea a priori de a avea un sezon ploios este 0.5**
- **dacă suntem în timpul unui sezon ploios, probabilitatea de a avea ploaie într-o noapte dată este 0.9**

- Pentru a putea folosi o asemenea reprezentare în rezolvarea problemelor este necesar un mecanism de calcul al influenței oricărui nod arbitrar asupra oricărui alt nod.
- Pentru a obține acest mecanism de calcul este necesar ca:
 - graful inițial să fie convertit la un graf nedirecționat, în care arcele să poată fi folosite pentru a se transmite probabilități în oricare dintre direcții, în funcție de locul din care provin dovezile;
 - să existe un mecanism de folosire a grafului care să garanteze transmiterea corectă a probabilităților (exemplu: iarba udă să nu constituie o dovadă a ploii, care să fie considerată apoi o dovadă a existenței ierbii ude).

- Există trei mari clase de algoritmi care efectuează aceste calcule.
- Ideea care stă la baza tuturor acestor metode este aceea că nodurile au domenii limitate de influență.
- Metoda cea mai folosită este probabil cea a transmiterii mesajelor [Pearl, 1988], abordare care se bazează pe observația că, pentru a calcula probabilitatea unui nod A condiționat de ceea ce se știe despre celelalte noduri din rețea, este necesară cunoașterea a trei tipuri de informații:
 - suportul total care sosește în A de la nodurile sale părinte (reprezentând cauzele sale);
 - suportul total care sosește în A de la fiii acestuia (reprezentând simptomele sale);
 - intrarea în matricea fixată de probabilități condiționate care face legătura dintre nodul A și cauzele sale.

Relații de independență condiționată în rețele Bayesiene

- O rețea Bayesiană exprimă independența condiționată a unui nod și a predecesorilor săi, fiind dați părinții acestuia și folosește această independență pentru a proiecta o metodă de construcție a rețelelor.
- Pentru a stabili algoritmi de inferență trebuie însă să știm dacă se verifică independențe condiționate mai generale, adică:
 - ✓ Fiind dată o rețea, dorim să putem ”citi” dacă o mulțime de noduri X este independentă de o altă mulțime Y fiind dată o mulțime de ”noduri dovezi” E .
- Metoda de stabilire a acestui fapt se bazează pe noțiunea de separare dependentă de direcție sau d -separare.

Definiția 2

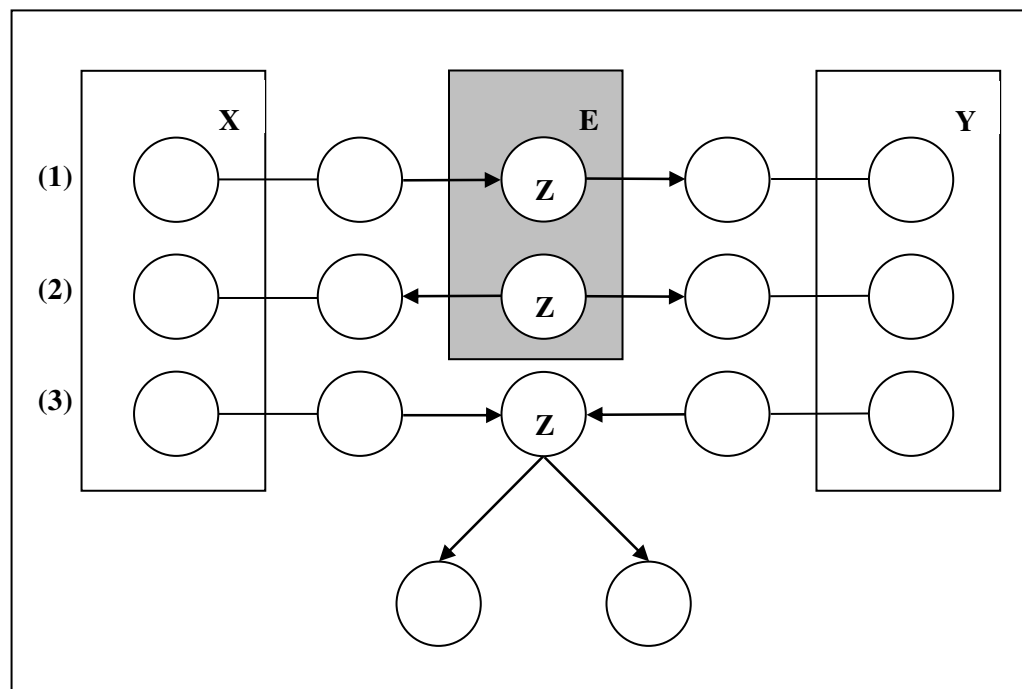
O mulțime de noduri E ***d-separă*** două mulțimi de noduri X și Y dacă orice drum nedirecționat de la un nod din X la un nod din Y este ***blocat*** condiționat de E .

Definiția 3

Fiind dată o mulțime de noduri E , spunem că un drum este blocat condiționat de E dacă există un nod Z aparținând drumului, pentru care una dintre următoarele trei condiții se verifică:

- (1) Z aparține lui E și Z are o săgeată a drumului intrând în E și o săgeată a drumului ieșind din E .
- (2) Z aparține lui E și Z are ambele săgeți ale drumului ieșind din E .
- (3) Nici Z și nici vreunul dintre descendenții săi nu aparțin lui E , iar ambele săgeți ale drumului țintesc înspre Z .

Figura următoare ilustrează cele trei cazuri posibile:



- S-a demonstrat că (teorema fundamentală a rețelelor Bayesiene, demonstrată de Verma și Pearl):
 - ✓ *dacă orice drum nedirecționat de la un nod din X la un nod din Y este d-separat de E , atunci X și Y sunt independente condiționat de E .*
- Procesul construirii și folosirii rețelelor Bayesiene nu utilizează d-separarea.
- Noțiunea de d-separare este fundamentală în construcția algoritmilor de inferență.

Inferența în rețele Bayesiene

- Sarcina principală a oricărui sistem probabilist de inferență este aceea de a calcula probabilități a posteriori de tipul
$$P(\text{Interogare}|\text{Dovezi})$$
corespunzător unei mulțimi de variabile de interogare condiționat de valori exacte ale unor variabile dovezi.
 - ✓ Exemplu: În exemplul considerat, *Ud* este o variabilă de interogare, iar *Stropitoare* și *Sezon_ploios* ar putea fi variabile dovezi.
- Rețelele Bayesiene sunt suficient de flexibile pentru ca orice nod să poată servi fie ca o variabilă de interogare, fie ca o variabilă dovadă.
- Un agent primește valori ale variabilelor dovezi de la senzorii săi (sau în urma altor raționamente) și întreabă despre posibilele valori ale altor variabile astfel încât să poată decide ce acțiune trebuie întreprinsă.

- **Sarcina cea mai frecventă și mai utilă a rețelelor Bayesiene:**
 - ✓ **determinarea probabilităților condiționate a posteriori ale variabilelor de interogare.**

Literatura de specialitate discută patru tipuri distincte de inferență, care poate fi realizată de rețelele Bayesiene:

- inferență de tip diagnostic (de la efecte la cauze);
- inferență cauzală (de la cauze la efecte);
- inferență intercauzală (între cauze ale unui efect comun);
- inferențe mixte (reprezentând combinații a două sau mai multe dintre inferențele anterioare).

Exemplu: Setând efectul *Stropitoare* la valoarea "adevărat" și cauza *Sezon_ploios* la valoarea "fals", ne propunem să calculăm

$$P(U_d | \text{Stropitoare}, \neg \text{Sezon_ploios}).$$

Aceasta este o inferență mixtă, care reprezintă o utilizare simultană a inferenței de tip diagnostic și a celei cauzale.

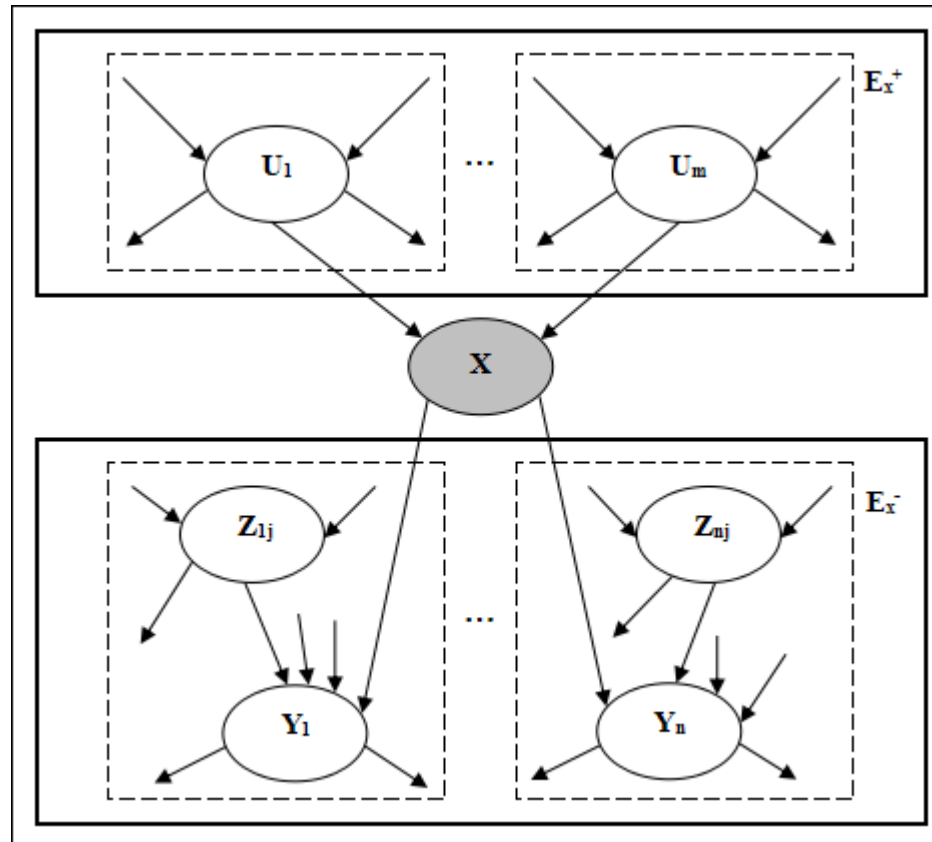
Un algoritm pentru răspunsul la interogări
**(algoritm de calcul al probabilităților condiționate a
posteriori ale variabilelor de interogare)**

- Algoritmul va fi de tip **înlănțuire înapoi** prin faptul că pleacă de la variabila de interogare și urmează drumurile de la acel nod până la nodurile dovezi.
- Algoritmul se referă numai la **rețele unic conectate**, numite și **poli-arbori**.
(In poli-arbori există cel mult un drum nedirecționat între oricare două noduri ale rețelei).
- Algoritmii pentru rețele generale vor folosi algoritmii referitori la poli-arbori ca principală subrutină.

- Figura care urmează prezintă o rețea generică unic conectată.
- În această rețea nodul X are părinții $U = U_1 \dots U_m$ și fiii $Y = Y_1 \dots Y_n$. Corespunzător fiecărui fiu și fiecărui părinte a fost desenat un dreptunghi care include toți descendenții nodului și toți strămoșii lui (cu excepția lui X).
- Proprietatea de unică conectare înseamnă că toate dreptunghiurile sunt disjuncte și că nu există legături care să le conecteze între ele.
- Se presupune că X este variabila de interogare și că există o mulțime E de variabile dovezi.
- Se urmărește calcularea probabilității condiționate

$$P(X | E)$$
- Dacă însuși X este o variabilă dovadă din E , atunci calcularea lui $P(X | E)$ este banală. Presupunem că X nu aparține lui E .

- Rețeaua din figură este **partiționată în conformitate cu părinții și cu fiii variabilei de interogare X** .
- Pentru a concepe un **algoritm**, va fi util să ne putem referi la diferite **porțiuni ale dovezilor**:
 - E^+_X reprezintă **suportul cauzal** pentru X - variabilele dovezi aflate “deasupra” lui X , care sunt conectate la X prin intermediul părinților săi.
 - E^-_X reprezintă **suportul probatoriu** pentru X - variabilele dovezi aflate “dedesubtul” lui X și care sunt conectate la X prin intermediul fiilor săi.



Fie rețeaua generică unic conectată din figura anterioară, partiționată în conformitate cu părinții și cu fiii variabilei de interogare X . Nodul X are părinții $U = U_1 \dots U_m$ și fiii $Y = Y_1 \dots Y_n$. Presupunem ca există o mulțime E de variabile dovezi, $E = \{E^-_X, E^+_X\} = \{U_1, \dots, U_m, Y_1, \dots, Y_n\}$, mulțime de variabile aleatoare discrete, unde:

- E^+_X reprezintă suportul cauzal pentru X - variabilele dovezi aflate "deasupra" lui X , care sunt conectate la X prin intermediul părinților săi.**
- E^-_X reprezintă suportul probatoriu pentru X - variabilele dovezi aflate "dedesubtul" lui X și care sunt conectate la X prin intermediul fiilor săi.**

Se urmărește calcularea probabilității condiționate $P(X | E)$.

Strategia generală pentru calculul lui $P(X | E)$ este următoarea:

- **Exprimă $P(X | E)$ în termenii contribuțiilor lui E^+_X și E^-_X .**
- **Calculează contribuția mulțimii E^+_X calculând efectul ei asupra părinților lui X și apoi transmițând acest efect lui X .**
- **Calculează contribuția mulțimii E^-_X calculând efectul ei asupra fiilor lui X și apoi transmițând acest efect lui X .**

Exprimarea lui $P(X | E)$ în termenii contribuțiilor lui E^+_X și E^-_X este următoarea:

$$P(X | E) = \alpha P(X | E^+_X) P(E^-_X | X) \quad (1)$$

În urma efectuării calculelor, se obține contribuția mulțimii E^+_X ca fiind dată de

$$P(X | E^+_X) = \sum_{(u_1, \dots, u_m)} P(X | \{U_1 = u_1, \dots, U_m = u_m\}) \prod_{i=1}^m P(\{U_i = u_i\} | E_{U_i \setminus X}) \quad (2)$$

și, introducând (2) în (1), avem:

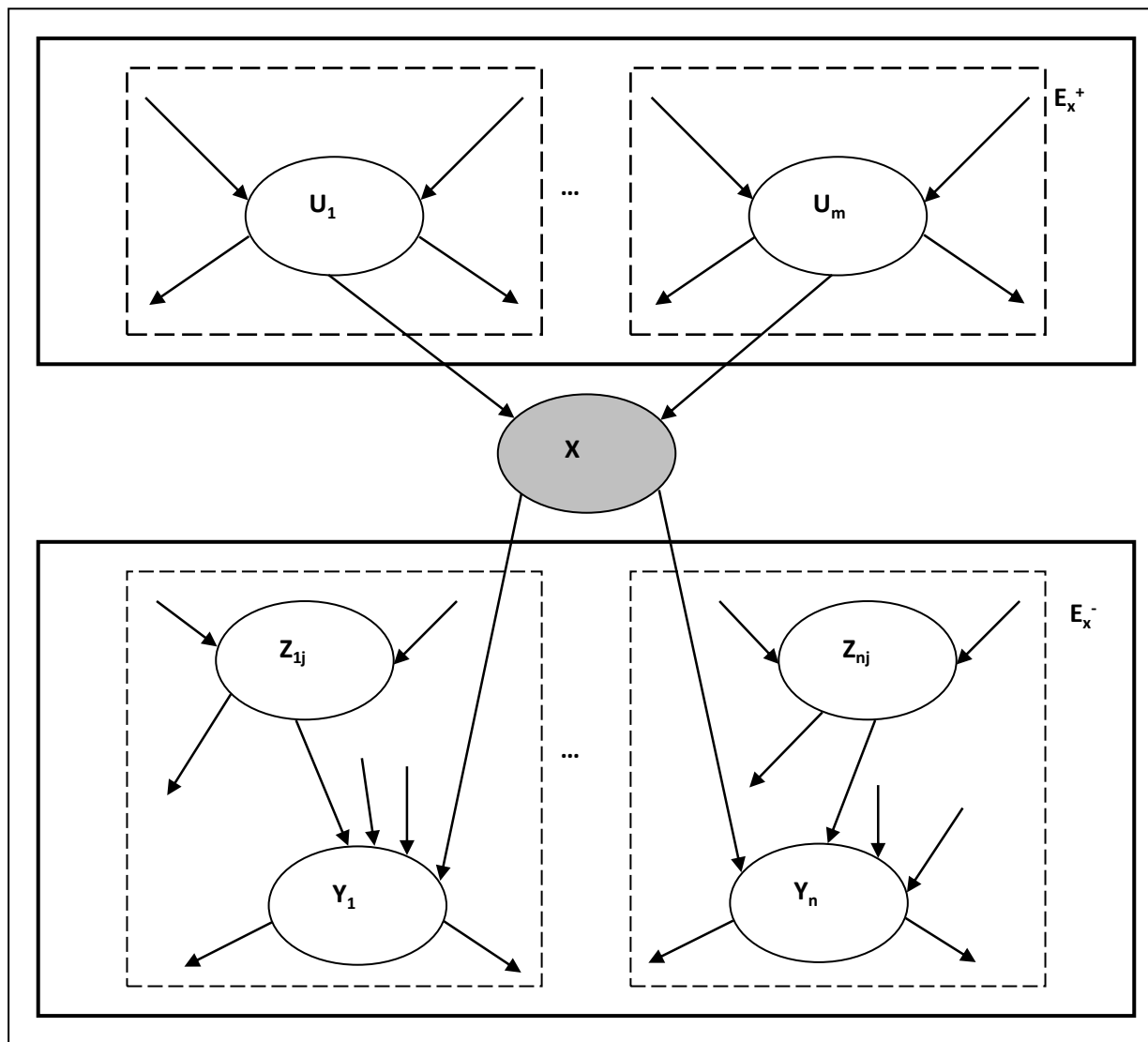
$$P(X | E) = \alpha P(E^-_X | X) \sum_{\mathbf{u}} P(X | \mathbf{u}) \prod_{i=1}^m P(\{U_i = u_i\} | E_{U_i \setminus X})$$

Contribuția mulțimii E^-_X se obține ca fiind:

$$P(E^-_X|X) = \beta \prod_i \sum_{y_i} P(E^-_{Y_i}|y_i) \sum_{z_i} P(y_i|X, z_i) \prod_j P(z_{ij}|E_{Z_{ij}\setminus Y_i})$$

Menționăm că:

- **Notația $E_{U_i\setminus X}$ este folosită pentru a se face referire la toate dovezile conectate cu nodul U_i , mai puțin cele prin drumul de la X ;**
- **U este vectorul părinților $U_1 \dots U_m$ și u reprezintă o atribuire de valori pentru aceștia;**
- **Y este vectorul fiilor Y_1, \dots, Y_n și fie $y = (y_1, \dots, y_n)$ o realizare a acestuia;**
- **Z_i sunt părinții lui Y_i , alții decât X și fie z_i o atribuire de valori ale acestor părinți (vezi figura următoare).**



Folosind aceste notații, scriem, în pseudocod, algoritmul pentru răspunsul la interogări care calculează probabilitatea condiționată a posteriori a variabilei de interogare X , adică $P(X \mid E)$.

Algoritmul

function INTEROGARE-REȚEA (X) **return**

o distribuție de probabilitate a valorilor lui X

input: X , o variabilă aleatoare

SUPPORT-EXCEPT (X , nul)

function SUPPORT-EXCEPT (X, V) **return** $P(X|E_{X \setminus V})$

if DOVEZI ? (X) **then return** distribuția observată pentru X

else

calculează $P(E_{X \setminus V}^- | X) = \text{DOVEZI-EXCEPT}(X, V)$

$U \leftarrow \text{PĂRINȚI}[X]$

if U este vid

then return $\alpha P(E_{X \setminus V}^- | X) P(X)$

else

pentru fiecare U_i **in** U **do**

calculează și memorează $P(U_i | E_{U_i \setminus X}) =$
 $= \text{SUPPORT-EXCEPT}(U_i, X)$

return $\alpha P(E_{X \setminus V}^- | X) \sum_{\mathbf{u}} P(X | \mathbf{u}) \prod_i P(U_i | E_{U_i \setminus X})$

function DOVEZI-EXCEPT (X, V) **return** $P(E_{X \setminus V}^- | X)$

$Y \leftarrow \text{FII}[X] - V$

if Y este vid

then return o repartiție uniformă

else

pentru fiecare Y_i **in** Y **do**

calculează $P(E_{Y_i}^- | y_i) = \text{DOVEZI-EXCEPT}(Y_i, \text{nul})$

$Z_i \leftarrow \text{PĂRINȚI}[Y_i] - X$

pentru fiecare Z_{ij} **in** Z_i

calculează $P(Z_{ij} | E_{Z_{ij} \setminus Y_i}) =$
 $= \text{SUPPORT-EXCEPT}(Z_{ij}, Y_i)$

return $\beta \prod_i \sum_{y_i} P(E_{Y_i}^- | y_i) \sum_{\mathbf{z}_i} P(y_i | X, \mathbf{z}_i) \prod_j P(z_{ij} | E_{Z_{ij} \setminus Y_i})$

➤ **Problema:**

Fiind dat sirul de intrare

In aceasta familie masa se ia la ore fixe.

sa se conceapa un soft care sa inteleaga sensul sirului de intrare.

➤ **Indicatie:**

Pentru a se intelege sensul sirului de intrare, softul trebuie, mai intai, sa determine cu ce sens este folosit aici cuvantul polisemantic *masa*. (Trebuie sa se determine faptul ca *masa* este folosit cu sensul de mancare si nu cele de mobila, multime – masa de oameni, masa din fizica etc.)

➤ **Terminologie:**

Masa este un cuvant polisemantic (cu mai multe sensuri), deci ambiguu. Precizarea sensului cu care *masa* este folosit in sirul de intrare dat reprezinta operatia de dezambiguizare a sensului.

➤ **WSD \equiv Word Sense Disambiguation (dezambiguizare automata a sensului cuvintelor polisemantice)**

O retea semantica pentru reprezentarea si procesarea limbajului natural

WordNet (WN)

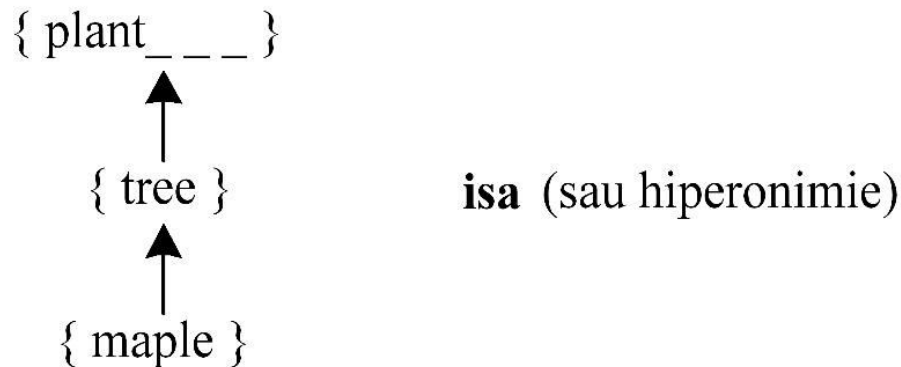
- WordNet (WN) este o baza de date lexicala a limbii engleze, conceputa si creata de Profesorul George Miller, la Universitatea Princeton, in anii '90. Ea este permanent actualizata de catre un colectiv de cercetatori.
- WN cuprinde toate “cuvintele cu continut” ale limbii engleze, adica toate substantivele, adjectivele, verbele si adverbele. Substantivele si verbele sunt organizate in ierarhii, iar adjectivele si adverbele in clustere.
- “Piatra de temelie” in WN o constituie *synset*-ul, sau multimea de sinonime.
- Aici cuvinte sinonime inseamna cuvinte care se refera la un acelasi concept. (De pilda, cuvintele *scaun* si *masa* sunt considerate aici sinonime, intrucat ambele se refera la conceptul de “mobila”.)

- Fiecare *synset* din WN se refera la un anumit *concept*.
- Toate sinonimele care apar intr-un synset au aceeasi parte de vorbire (substantiv, adjectiv, verb sau adverb).
- Un cuvânt polisemantic (cu mai multe sensuri) se refera la un alt concept prin fiecare dintre sensurile lui. Prin urmare, un cuvânt polisemantic va interveni in mai multe synset-uri WN. El apartine cate unui synset prin fiecare sens al lui. Spre exemplu, cuvântul polisemantic *masa* poate apartine fiecaruia dintre synset-urile care se refera la conceptele de *mancare*, *mobila*, *multime*, *masa din fizica* etc.
- Un synset WN contine toate sinonimele care lexicalizeaza conceptul la care synset-ul se refera si care au aceeasi parte de vorbire (substantiv, verb etc.), precum si un string numit *glosa*, care seamana cu o definitie clasica a sensului (cum sunt cele din dictionar). Glosa poate contine si un exemplu de utilizare a sinonimelor din synset.

- Synset-urile WN sunt legate între ele prin relatii semantice. Aceste relatii leaga între ele conceptele la care se refera synset-urile si care sunt aceleasi in toate limbile (conceptele sunt independente de limba).

Synset-uri de substantive

Synset-urile substantivale sunt legate între ele prin relatii semantice cum ar fi hiperonimia si inversa acestei relatii, hiponimia, care reprezinta relatia ISA din inteligenta artificiala in domeniul procesarii limbajului natural. Pe baza acestei relatii este construita ierarhia substantivala din WN:



Hiperonimul este conceptul parinte, iar hiponimul este conceptul fiu. Mecanismul de inferenta este mostenirea proprietatilor. (Hiponimele mostenesc toate proprietatile hiperonimului).

Synset-uri substantivale – continuare:

- alta relatie semantica importanta in cazul substantivelor este meronimia sau relatia *parte-din* (“the *part-of* relation”). Inversa acestei relatii este holonimia.

Exemplu: substantivele *clanta* si *usa*.

(Clanta este parte din usa. Substantivul *usa* este holonim al substantivului *clanta*. Iar *clanta* este meronim al lui *usa*).

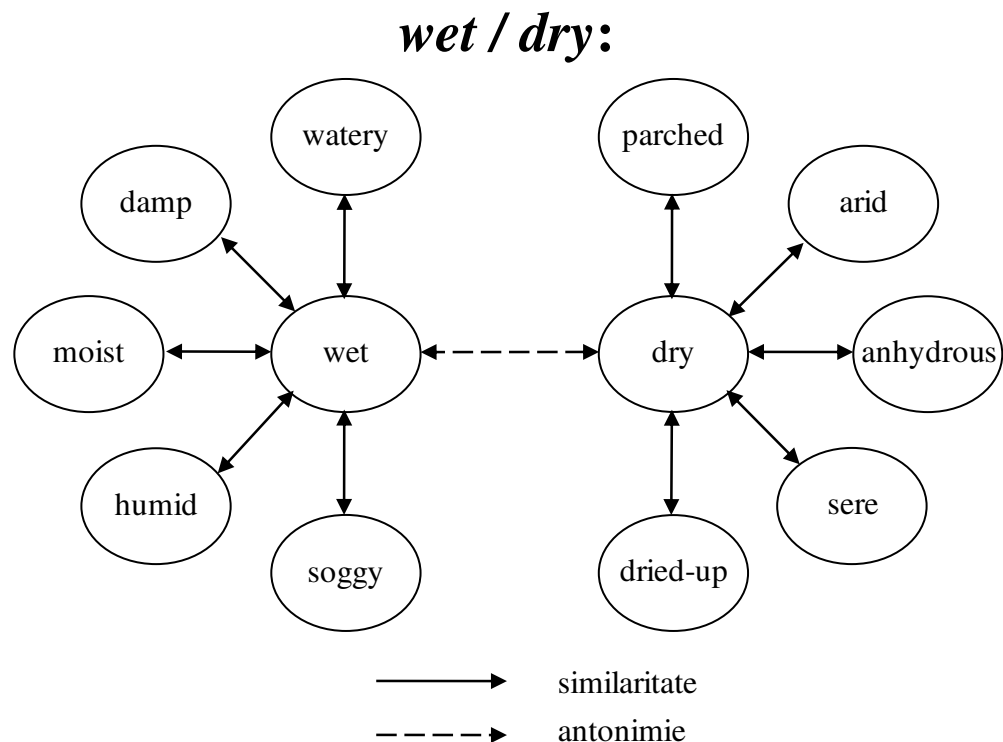
WordNet

- Structura de retea semantica a WN este data de relatiile semantice care leaga synset-urile intre ele.
- Prin intermediul relatiei de hiperonimie conceptele mostenesc toate proprietatile conceptelor parinte. Mostenirea proprietatilor transforma WN intr-o baza de cunostinte.
- Concluzie: WN este o baza de date lexicala a limbii engleze, o retea semantica si o baza de cunostinte. Faptul ca WN reprezinta o baza de cunostinte o face extrem de utila in diverse aplicatii din IA.

SYNSET-uri de adjective in WN

Relatii semantice de o alta natura leaga intre ele synset-urile de adjective din WN. Principala relatie semantica este considerata a fi antonimia.

Prin intermediul relatiei de antonimie synset-urile de adjective sunt organizate in cluster-e. Mai jos este dat ca exemplu cluster-ul format in jurul antonimelor



Synset-uri de adjective

Relatia de similaritate exista in WN numai intre synset-uri de adjective. Ea aduce, in plus, antonime indirecte. (In exemplul dat, *wet* si *dry* sunt antonime directe, iar *wet* si *arid* sunt antonime indirecte. Adjectivul *moist* nu are un antonim direct, dar un antonim indirect al lui poate fi gasit urmand drumul *moist* → *wet* → *dry*).

S-a dovedit ca relatia de antonimie furnizeaza informatie importanta procesului de dezambiguizare automata a sensului.

Alte relatii semantice importante in cazul adjectivelor sunt: *also-see*, *pertaining-to*, *attribute*.

SYNSET-URI DE VERBE SI ADVERBE

- **Synset-urile de verbe** sunt organizate in ierarhii, ca si cele de substantive, prin intermediul relatiei entailment (to entail = a atrage dupa sine).
- Alta relatie semantica importanta pentru verbe este relatia cauzala.
- **Synset-urile de adverbe** sunt organizate in cluster-e, ca si cele de adjective.

Similaritate si inrudire (intre concepte si cuvintele care le lexicalizeaza)

- Inrudirea semantica este un concept mai general decat similaritatea semantica. Similaritatea este un caz particular de inrudire.
Exemplu: *Masina* si *anvelopa* nu sunt similare, dar sunt inrudite (anvelopa este o “parte din” masina). *Doctorii* si *spitalele* nu sunt concepte similare, dar sunt inrudite.
- In literatura exista diverse masuri pentru calcularea similaritatii si/sau inrudirii. Vom vedea o asemenea masura in algoritmul care urmeaza.
- S-a demonstrat ca, pentru a realiza dezambiguizarea automata a sensului cuvintelor, conceptul de inrudire este mai util decat cel de similaritate.

- **Studiem, in continuare, un algoritm de dezambiguizare a sensului cuvintelor bazat pe conceptul de inrudire si care foloseste cunostinte furnizate de reteaua semantica WordNet.**
- **Problema:**
Fiind dat sirul de intrare
In aceasta familie masa se ia la ore fixe.
sa se conceapa un soft care sa inteleaga sensul sirului de intrare.
- **Indicatie:**
Pentru a se intelege sensul sirului de intrare, softul trebuie, mai intai, sa determine cu ce sens este folosit aici cuvantul polisemantic *masa*. (Trebuie sa se determine faptul ca *masa* este folosit cu sensul de mancare si nu cele de mobila, multime – masa de oameni, masa din fizica etc.) Algoritmul care urmeaza dezambiguizeaza cuvantul ambiguu *masa*.
Cuvantul de dezambiguizat \equiv cuvânt tinta

Algoritmul Lesk extins

- Banerjee si Pedersen (2003) prezinta o noua masura a inrudirii semantice intre concepte, care se bazeaza pe numarul de cuvinte pe care le au in comun definitiile lor (suprapuneri de glose – “gloss overlaps”).
- Aceasta masura primeste ca input doua concepte (reprezentate de doua synset-uri WN) si ofera ca output o valoare numerica, care cuantifica gradul lor de inrudire semantica. Aceasta valoare numerica este apoi folosita pentru a realiza dezambiguizarea sensului.
- Masura de inrudire si algoritmul de dezambiguizare corespunzator reprezinta o varianta a Algoritmului lui Lesk clasic (care se bazeaza pe suprapuneri de glose). Aceasta varianta extinde glosele conceptelor luate in considerare.
- Extinderea gloselor se face astfel incat sa fie luate in considerare si glosele altor concepte, inrudite cu cele date prin intermediul relatiilor semantice furnizate de WN.

Algoritmul lui Lesk

Suprapunerile de glose (definitii) au fost introduse de Lesk (1986) cu scopul de a servi în dezambiguizarea sensului.

Algoritmul lui Lesk atribuie un sens unui cuvânt tinta (de dezambiguizat), într-un context dat, prin compararea gloselor diverselor sensuri ale cuvântului cu cele ale celorlalte cuvinte din context. Sensul cuvântului tinta a cărui glosa are cele mai multe cuvinte în comun cu glosele cuvintelor învecinate este cel atribuit cuvântului tinta.

Exemplu: considerăm următoarele glose ale cuvintelor *car* și *tire*

car: *four wheel motor vehicle usually propelled by an internal combustion engine*

tire: *hoop that covers a wheel, usually made of rubber and filled with compressed air*

✓ glosele acestor concepte au în comun cuvântul wheel

Limitare: Algoritmul Lesk original ia în considerare numai suprapunerile dintre glosele cuvântului tinta și cele ale cuvintelor din jurul lui, în contextul dat. Definițiile de dicționar sunt însă scurte și nu furnizează un vocabular suficient de mare (vor fi puține cuvinte comune).

- Masura suprapunerii extinse de glose introdusa de Banerjee si Pedersen (2003) extinde glosele conceptelor luate in considerare prin includerea gloselor conceptelor inrudite cu acestea. Conceptele inrudite sunt alese prin intermediul relatiilor semantice din WN.

Masura suprapunerii extinse de glose

- **Atunci cand masuram inrudirea dintre doua synset-uri de input, cautam suprapuneri nu numai intre glosele lor, dar si intre glosele synset-urilor hiperonime, hiponime, meronime, holonime etc. ale lor.**
- **Alte relatii WN de luat in considerare sunt attribute, similar-to, also-see, antonymy etc.**
- **Observatie: NU toate aceste relatii au aceeasi importanta in procesul de dezambiguizare. Alegerea relatiilor depinde de partea de vorbire a cuvântului tinta (substantiv, adjectiv, verb, adverb), dar si de tipul de aplicatie pentru care se studiaza inrudirea.**
- **Vom aplica aici aceasta masura a inrudirii in procesul de dezambiguizare.**

Exemplu de utilizare a relatiilor semantice din WN:

In cazul substantivelor testele empirice au demonstrat ca este util in dezambiguizare sa folosim glosele hiponimelor si meronimelor synset-urilor de input. Observati ca se recomanda folosirea hiponimelor (conceptul fiu) si nu a hiperonimelor (conceptul parinte) sau a ambelor. Cu alte cuvinte, s-a demonstrat ca hiponimia furnizeaza mai multa informatie decat hiperonimia, desi ambele formeaza relatia ISA.

Mecanism de acordare a scorurilor

- Algoritmul Lesk inițial compară glosele unei perechi de concepte și calculează un scor prin numărarea cuvintelor pe care acestea le au în comun. Acest mecanism de acordare a scorurilor nu diferențiază între suprapuneri de cuvinte unice și suprapuneri de grupuri de cuvinte, ci tratează fiecare glosă ca pe un „sac de cuvinte”. Spre exemplu, acorda scorul 3 conceptelor *drawing paper* și *decal*, care au următoarele glose:

drawing paper: paper that is specially prepared for use in drafting

decal: the art of transferring designs from specially prepared paper to a wood or glass or metal surface.

Aici există 3 suprapuneri, cuvântul *paper* și grupul de cuvinte *specially prepared*.

- Banerjee și Pedersen (2003) atribuie unei suprapuneri de n cuvinte scorul n^2 (pentru că o suprapunere de n cuvinte consecutive este mult mai rară decât suprapunerile de cuvinte unice). Pentru perechea anterioară de glose se atribuie scorul 5 (în loc de 3).

Algoritm de acordare a scorurilor

1. Fiind date două șiruri, se detectează cea mai lungă suprapunere dintre acestea. Dacă două sau mai multe astfel de suprapuneri au aceeași lungime maximă, atunci este raportată acea suprapunere care intervine prima în primul șir care se compară. Cea mai lungă suprapunere detectată este înlăturată, iar în locul ei se plasează un marcaj în fiecare dintre cele două șiruri constituind input-ul. Cele două șiruri astfel obținute sunt apoi din nou verificate pentru suprapuneri, iar acest proces continuă până când nu mai există suprapuneri între ele.
2. Se atribuie scoruri tuturor suprapunerilor găsite (dimensiunile suprapunerilor detectate sunt ridicate la pătrat) și aceste scoruri sunt adunate pentru a se determina *scorul perechii de glose date*.

Calcularea scorului de înrudire dintre doua synset-uri A si B

1. Definim mulțimea RELS ca pe o mulțime nevidă de relații constând din una sau mai multe dintre relațiile puse la dispoziție de WordNet:

$$\text{RELS} \subset \{r \mid r \text{ este o relație definită în WordNet}\}.$$

Presupunem că funcția fiecărei relații r ($r \in \text{RELS}$) este cea dată de numele relației, care acceptă un synset de input și întoarce glosa synset-ului (sau synset-urilor) înrudite cu cel de input prin relația desemnată. Spre exemplu, dacă $r \in \text{RELS}$ reprezintă relația de hiperonimie, atunci $r(A)$ întoarce glosa unui synset hiperonim al lui A . r mai poate reprezenta „relația glosă”, caz în care $r(A)$ întoarce glosa synset-ului A .

OBS.: Dacă mai multe synset-uri sunt înrudite cu cel de input prin intermediul aceleiași relații, glosele acestora vor fi concatenate și returnate ca un unic șir de caractere. Această concatenare este efectuată deoarece nu se dorește să se facă o diferențiere între synset-uri care sunt înrudite cu cel de input printr-o aceeași relație, de interes fiind doar glosele acestora. Atunci când niciun synset nu este înrudit cu cel de input prin relația dată, va fi returnat șirul vid.

Definim o nouă mulțime nevidă, de perechi de relații, selectate din mulțimea RELS. Singura constrângere în formarea acestor perechi de relații va fi aceea că, dacă este aleasă perechea (r_1, r_2) , atunci va trebui aleasă și perechea (r_2, r_1) , astfel încât să se asigure reflexivitatea măsurii de înrudire ($r_1, r_2 \in \text{RELS}$). Cu alte cuvinte, trebuie să se verifice $\text{înrudire}(A, B) = \text{înrudire}(B, A)$. Vom numi această nouă mulțime RELPAIRS. Ea se definește după cum urmează:

2. $\text{RELPAIRS} = \{(R_1, R_2) \mid R_1, R_2 \in \text{RELS}; \text{dacă } (R_1, R_2) \in \text{RELPAIRS}, \text{atunci } (R_2, R_1) \in \text{RELPAIRS}\}$.

3. Presupunem că $\text{scor}()$ este o funcție care acceptă ca input două glose, găsește grupurile de cuvinte care se suprapun ale acestora și întoarce un scor calculat conform Algoritmului de acordare a scorurilor descris anterior.

➤ Folosind toate aceste elemente, calculăm scorul de înrudire dintre synset-urile A și B după cum urmează:

$$\text{înrudire}(A, B) = \sum_{\forall (R_1, R_2) \in \text{RELPAIRS}} \text{scor}(R_1(A), R_2(B)).$$

Această măsură de înrudire se bazează pe mulțimea tuturor perechilor posibile de relații furnizate de rețeaua semantică WordNet. În practică, vor fi alese acele relații care s-au dovedit a fi cele mai utile corespunzător fiecărei părți de vorbire.

Exemplu:

Vom presupune că mulțimea de relații aleasă este $RELS = \{gloss, hype, hypo\}$ (unde *hype* și *hypo* reprezintă relațiile de hiperonimie și, respectiv, hiponimie). În continuare, vom presupune că mulțimea perechilor de relații $RELPAIRS$ este dată de $RELPAIRS = \{(gloss, gloss), (hype, hype), (hypo, hypo), (hype, gloss), (gloss, hype)\}$. Atunci, înrudirea dintre synset-urile A și B poate fi calculată după cum urmează:

$$\text{înrudire}(A, B) = \text{scor}(gloss(A), gloss(B)) + \text{scor}(hype(A), hype(B)) + \text{scor}(hypo(A), hypo(B)) + \text{scor}(hype(A), gloss(B)) + \text{scor}(gloss(A), hype(B)).$$

Se observă că alegerea acestor perechi de relații asigură egalitatea $\text{înrudire}(A, B) = \text{înrudire}(B, A)$.

Aplicatie in dezambiguizarea sensului cuvintelor

Vom selecta o fereastră de context în jurul cuvântului țintă (de dezambiguizat) și vom presupune că această fereastră de context constă din $2n + 1$ cuvinte notate w_i , $-n \leq i \leq +n$, unde cuvântul țintă este w_0 . Pentru fiecare cuvânt cu conținut din fereastra de context este identificată o mulțime de sensuri candidate. Fie $|w_i|$ numărul de sensuri candidate ale cuvântului w_i . Aceste sensuri vor fi notate prin $s_{i,j}$, $1 \leq j \leq |w_i|$. Fiecărui sens k posibil al cuvântului țintă îi vom atribui un scor notat $SenseScore_k$ și calculat prin adunarea scorurilor de înrudire obținute la compararea sensului analizat al cuvântului țintă cu fiecare sens al fiecărui cuvânt cu conținut care nu este cuvânt țintă din fereastra de context. Scorul sensului $s_{0,k}$ este calculat după cum urmează:

$$SenseScore_k = \sum_{i=-n}^n \sum_{j=1}^{|w_i|} \text{înrudire}(s_{0,k}, s_{i,j}), \quad i \neq 0$$

Acel sens care are scorul cel mai ridicat este considerat a fi cel mai adecvat pentru cuvântul țintă. Dacă mai multe sensuri au scor egal, atunci, prin convenție, se alege acel sens care este considerat cel mai important (uzual) conform WordNet. O combinație candidată de glose care nu prezintă nicio suprapunere va primi scorul zero.

Complexitate

Dacă, în medie, există α sensuri pentru fiecare cuvânt, iar fereastra de context conține N cuvinte, vom avea $\alpha^2 \times (N - 1)$ perechi de mulțimi de synset-uri care trebuie comparate, ceea ce reprezintă o creștere liniară în raport cu N .