UNIVERSITATEA BABEȘ-BOLYAI

UBBFSEGA

Universitatea Babeș-Bolyai | Facultatea de Științe Economice și Gestiunea Afacerilor

CENTRUL DE FORMARE CONTINUĂ, ÎNVĂȚĂMÂNT LA DISTANȚĂ ȘI CU FRECVENȚĂ REDUSĂ

Programul de studii

Informatică Economică

Suport de curs

INTELIGENȚĂ ARTIFICIALĂ

Anul III Semestrul V

Cluj-Napoca, 2021

Cuprins:

Informaţii generale

Date de contact ale titularului de curs

Pop Gabriel
Universitatea Babeș-Bolyai
Facultatea de Științe Economice și Gestiunea Afacerilor
Departamentul Informatică Economică

E-mail: gabriel.pop@ubbcluj.ro

Date de identificare curs

Denumire: Inteligenta Artificiala
Cod: ELR0216

Anul: 3
Semestrul: 6
Tipul cursului: obligatoriu
Număr de credite: 4

Condiționări și cunoștințe prerechizite

Cursul nu are condiţionări prerechizite. Cunoştinţele prerechizite care pot facilita asimilarea materialului sunt legate de programare la nivel de bază şi obiectuală, reţele de calculatoare. Sugerăm ca înainte de parcurgerea materialului să se identifice următoarele cunoştinţele prerechizite:

- programare structurată; noţiuni de bază;
- sisteme de operare,

Descrierea cursului

Parcurgerea disciplinei furnizeaza posibilitatea de a rezolva probleme intr-o noua abordare: cea logica. Disciplina „Inteligenţa Artrificială" are rolul, de a familiariza studenţii cu elementele specifice acestei discipline, să ofere informaţiile şi instrumentele de aplicare ale inteligenţei artificiale în diverse domenii economice. Conţinutul cursului va traversa anumite etape importante în formarea studenţilor ca informaticieni:

- analiza diverselor definiţii cunoscute din literatură;
- asemănări şi deosebiri ale comportamentului uman de cel raţional;
- evoluţia cronologică a dezvoltării AI;
- relaţia AI cu diverse domenii ale ştiinţei dar şi cu economia şi afacerile
- familiarizarea cu definiţiile logicii şi clasificări ale acesteia;
- definirea logicii propoziţiilor;
- familiarizarea cu elemente de semantică şi în particular semantica teoriei propoziţiilor;
- analizarea principiilor logicii formale şi aplicarea acestora la logica booleană.
- Studierea sistemelor formale – în particular sintaxa logicii propoziţiilor.
- Studierea sistemului formal al algebrei Booleene, dar şi a altor sisteme formale.
- Stabilirea regulilor minferenţiale din algebra Booleană.
- Demonstrarea automată şi reguli de raţionare.

Organizarea cursului

Ordinea temelor abordate de curs, conform structurii materialului didactic ce va fi disponibil pe platforma software de învăţământ la distanţă: https://portal.portalid.ubbcluj.ro/

1.Expunerea tematicii, a obiectivelor şi cerinţelor din cadrul cursului
- -Definiţii ale inteligenţei artificiale
- -Concepte de bază din domeniul discursului
- -Relaţii cu alte discipline
- -Cronologia dezvoltării inteligenţei artificiale
- -Relaţia cu domeniile economice şi cu sistemele informatice

2.Logica simbolică
- -Semantica teoriei propoziţiilor
- -Sisteme bivalente si trivalente

3.Algebra Booleană
- -Consistenţa şi completitudinea sistemelor formale.
- -Strategii de demonstrare automată
- -Principiile raţionării directe şi inverse
- - Forme normale conjunctive si disjunctive

4.Agenţi inteligenţi şi sisteme multiagent
- 4.4.Agenţi şi medii
- 4.5.Proprietăţile mediilor in care acţionează agenţii inteligenţi
- 4.6.Structura Agenţilor Inteligenţi

5.Agenţi care rezolvă probleme
- Compararea strategiilor de cautare
- Metode de cautare neinformata (blind search)
- Metode de cautare informata

Formatul şi tipul activităţilor implicate de curs

Cursul va fi prezentat prin activităţi tutoriale periodice programate conform orarului facultăţii, afişat pe site-ul http://econ.ubbcluj.ro.
Prin adresele de e-mail oferită sau la sediul facultăţii, titularul şi tutorii cursului stau la dispoziţia studenţilor pentru consultaţii on-line sau faţă în faţă în afara activităţilor periodice preprogramate. Se încurajează studiile de caz legate de locul de muncă al acelor studenţi care sunt deja angajaţi în domeniul I.T.
Activităţile tutoriale sunt, pentru studentul la distanţă, facultative şi nu afectează nota acestuia, obţinută strict prin forma indicată: examen scris şi prezentarea unui proiect ce are ca temă rezolvarea problemelor de logică cu ajutorul agenţilor inteligenţi. Totuşi, încurajăm participarea interactivă la activităţile tutoriale în special pentru dezvoltarea incrementală a dosarului de testare care va fi notat.

Materiale bibliografice

1. S.J.Russel, P. Norvig, Artificial Intelligence, A Modern Approach, *Prentice Hall in Artificial* Intelligence, Ediţia III-a 2010.
2. Peter Flach, Simply Logical – Intelligent Reasoning by Example, 2007 by John Wiley & Sons.
3. Negnevitski M., *Artificial Intelligence*, Addison Wesley, 2002.
4. Jean H. Gallier, Logic For Computer Science Foundations of Automatic Theorem Proving Copyright 2003
5. W.Benner, R. Zarnekov, H.Witting, *Intelligent Software Agents*, Springer, 1998.
6. G. Wagner, *Foundationms of Knowledge systems*, Kluwer, 1998.

Materiale și instrumente necesare pentru curs

Calculator, materialul bibliografic, software licenţiat şi free necesar cursului;

Calendarul cursului

Sunt estimate 3 întâlniri preprogramate pe semestru, cu datele şi locaţiile afişate pe site-ul facultăţii la începutul semestrului. Premergător fiecărei întâlniri se recomandă parcurgerea materialului de faţă, pe module pentru a asigura cursivitatea discuţiilor. Conţinutul acestor întâlniri va fi, în ordine:
Prima întâlnire – discuţii pe marginea modulelor I şi II.
A doua întâlnire – discuţii pe marginea modulului III.
A treia întâlnire – discuţii pe marginea modului IV.

Politica de evaluare şi notare

Evaluare teoretică si practica – 100% din notă
Conţinut:
Test grilă cu întrebări, de dificultate şi pondere în notă echitabile (30 întrebări de 0,3 punct pe întrebare).

Elemente de deontologie academică

Tentativele de fraudare atât la examen scris cât şi în dezvoltarea proiectului practic vor fi pedepsite prin anularea examenului şi aplicarea regulamentului instituţional. Nu este admisă în timpul examenului utilizarea mijloacelor de comunicaţie.

Studenţi cu dizabilităţi

E-mail de contact pentru situaţii deosebite şi suport acordat studenţilor cu dizabilităţi:
gabriel.pop@ubbcluj.ro

Strategii de studiu recomandate

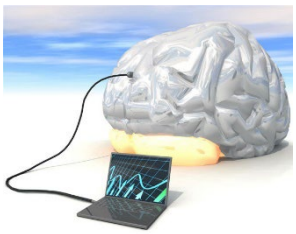
Recomandăm în ordine:

- parcurgerea materialului de faţă şi contactarea tutorilor pentru orice nelămuriri;
- parcurgerea bibliografiei obligatorii;
- cercetarea individuală pe tema cursului, folosind Internetul;
- parcurgerea documentaţiilor on-line;

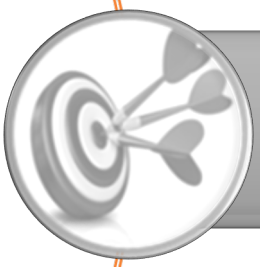
Disciplină: Inteligență Artificială

Modulul I.

Istoric



Descriere generală

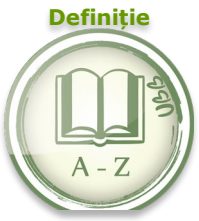


Cronologia Inteligenței Artificială

Legătura dintre Informatica Economică și Inteligența Artificială

1.1 Descriere Generală

1.1.1 Definiţii ale Inteligenţei Artificiale



(D.W.Patterson, 1990) – ramură a informaticii care se ocupă de studierea şi realizarea SC cu elemente de inteligenţă:

- învaţă concepte şi sarcini noi;
- analizează, trage concluzii şi reacţionează la mediul înconjurător;
- înţelege limbajul natural;
- efectuează activităţi care necesită elemente de inteligenţă umană.

(N. Nilson, 1998) – „comportament inteligent în artificial; comportament inteligent – percepţie, raţionament, instruire, comunicare şi acţiune într-un mediu complex”

Luger - 1992 „ramura informaticii destinată automatizării comportamentului intelligent” (Microsoft1999)

Larouse - „Ansamblu de tehnici utilizate pentru realizarea de automate apropiate de gândirea şi modul de acţiune umană”.

(Rusell&Norvig – 2003) rezumă definiţia dată în 8 cărţi – într-un tabel bidimensional:

- orizontal legat de:
  - stânga – apropierea faţă de performanţa umană;
  - dreapta – apropierea de conceptul ideal de inteligenţă – raţionalitate;
- vertical legat de:
  - sus – procesul de gândire;
  - jos – comportament;

Sistem ce raţionează uman	Sistem care raţionează
Haugeland 1985 – „efortul de a face maşinile să gândească… maşini care gândesc în sens complet ad-literam” Bellman **1978 –** „automatizarea activităţilor asociate gândirii umane, activităţi cum ar fi: luarea deciziilor, rezolvarea problemelor, învăţarea…”	**Cahrniak & McDermott** 1985–„studiul modelului mental prin utilizarea modelelor computationale” **Winson 1992** – „studiul calculului care face posibilă percepţia, raţionamentul şi acţiunea”
Sisteme care se comportă ca oamenii	**Sisteme care acţionează raţional**
Kurzweil 1992 - „arta de a crea maşini care realizează funcţii ce necesită inteligenţă analogă oamenilor” **Rich & Knight 1991 –** „studiul de a face calculatoarele să efectueze lucruri în care deocamdată oamenii sunt mai buni”	**Poole & alţii 1998** – „inteligenţa computaţională este studiul proiectării agenţilor inteligenţi” **Nilson 1998** – „comportament inteligent în artificial”

Concluzie. Domeniu recent al informaticii – dezideratul primilor constructori de calculatoare – emulează gândirea umană.

Acţionarea umană – Testul Turing (Alan Turing1950) – definiţia operaţională a inteligenţei umane – punând în scris un număr de întrebări, calculatorul răspunde ca un om. Caracteristici pentru calculatoare:

- prelucrarea limbajului natural;
- reprezentarea cunoştinţelor;
- raţionarea automată – memorează informaţii pentru a răspunde la întrebări şi trage concluzii noi;
- machine learning: extrage şi extrapolează modele;

Pentru test Turing:

- vedere artificială – să recepţioneze obiecte;
- robotică – să manipuleze obiecte;

AI nu trebuie să copieze cea umană – exemplul cu avionul, care nu copiază zborul păsărilor sau maşina de spălat, care nu copiază spălatul manual.

Gândire umană – abordarea cognitivă -> trebuie cunoscut modul de raţionare uman – 2 căi:

- introspecţie – analiza gândirii;
- experienţe psihologice;

Studiul comportamentului uman prin – analiza I/O & comportamentul în timp. Exemplu: Allen Newel & Herbert Simon 1961 – GPS „General Program Solver" – nu au urmărit rezolvarea unor categorii de probleme concrete şi nici dacă programul rezolvă corect problemele, ci trasarea raţionamentului sistemului comparativ cu cel uman. Diferenţele AI faţă de ştiinţe cognitive: ştiinţele cognitive încearcă să înţeleagă raţionamentul şi limbajul uman; AI încercă să transpună aceste raţionamente pe sisteme de calcul.

Gândire raţională – „legea gândirii" – Aristotel – logica formală– „gândirea corectă" – silogismul (modus ponens) „Socrate este un om – Orice om este muritor – Socrate este muritor".
Premize corecte -> concluzii corecte – prima abordare în logică. Notaţia matematică din secolul 19 (logica formală, matematică) -> 1965 program care poate rezolva orice problemă descriptibilă prin FOPL.

Probleme:

- formalizarea unor probleme neformale sau parţial formalizabile;
- trecerea de la „în principiu" la realizare ~ complexitatea calculului la câteva sute de fapte.

Acţionarea raţională – abordarea agentului raţional; definiţie agent – agent raţional (acţiune autonomă, percepe mediul, persistă un timp îndelungat, adaptare la schimbări, alegerea ţintei) – caută cea mai bună soluţie sau în condiţie de incertitudine – o soluţie bună.

Artificial intelligence (**AI**) is defined as intelligence exhibited by an artificial entity. Such a system is generally assumed to be a computer. Although AI has a strong science fiction connotation, it forms a vital branch of computer science, dealing with intelligent behavior, learning and adaptation in machines. Research in AI is concerned with producing machines to automate tasks requiring intelligent behavior. Examples include control, planning and scheduling, the ability to answer diagnostic and consumer questions, handwriting, speech, and facial recognition. As such, it has become a scientific discipline, focused on providing solutions to real life problems. AI systems are now in routine use in economics, medicine, engineering and the military, as well as being built into many common home computer software applications, traditional strategy games like computer chess and other video games[1].

[1] http://en.wikipedia.org/wiki/Artificial_intelligence

Ce este inteligenţa artificială? (John McCharty, 24 nov. 2004)[2]

1.1.2 Noţiuni legate de domeniul discursului

Ca în alte domenii complexe, în AI se presupun înţelese o serie de noţiuni şi categorii: inteligenţă, cunoaştere, raţionare, învăţare, gândire, etc. – legate de informatică. Înainte de o descriere riguroasă – una semantică.

Oxford Dictionary: *inteligenţa* – activitatea de a achiziţiona, înţelege şi aplica cunoştinţe, de a executa raţionamente şi judecăţi. Mai mult – inteligenţa înseamnă înglobarea unor cunoştinţe şi fapte achiziţionate experimental sau prin raţionament, conştient sau inconştient.

E. Feigebaum – defineşte *inteligenţa* ca: o percepţie superioară a imaginilor şi sunetelor, raţionament, imaginaţie, abilitatea de a citi, a scrie, a conversa, a conduce maşina, a memora, a reaminti fapte petrecute cu mult timp în urmă, a exprima stări emoţionale, etc.

Fundamentul inteligenţei sunt cunoştinţele.

1.1.3 Relaţia reală dintre AI şi cea umană – specifică şi generală

Există posibilitatea de a construi sisteme inteligente? – Da!

- sisteme care învaţă (case base reasoning, reinforcement learning, machine learning, reţele neuronale);
- sintetizează experienţe anterioare (sisteme expert);
- raţionează şi explică raţionamentele (sisteme expert, agenţi inteligenţi);
- rezolvă probleme complexe de matematică sau de joc (şah);
- planifică sarcini (sisteme suport de decizie);
- determină configuraţii optimale pentru sisteme complexe;
- elaborează strategii militare şi macroeconomice complexe;
- diagnostichează boli;
- înţeleg limbajul natural;
- recunosc imagini şi forme preluate prin camere video sau senzori (roboţi inteligenţi, robotul Honda [Wikipedia]), etc.

[2] http://www-formal.stanford.edu/jmc/whatisai/

Poate AI rezolva probleme generale de inteligenţă? – Nu – nu depăşeşte inteligenţa unui copil de 3 ani:

- recunoaşterea unui număr mare de obiecte din mediu;
- învăţarea unor sunete noi şi asocierea cu obiecte şi concepte;
- adaptarea la situaţii noi, neprevăzute.

1.1.4 Scopul şi domeniile

Nu se ocupă: sistemele de calcul convenţionale, corpul uman, limbajul uman, comportamentul uman (psihologia, filozofia, lingvistica, etc.) – relaţia dintre acestea şi AI.

Scopul – realizarea unor sisteme cu un grad înalt de inteligenţă, în anumite domenii depăşeşte inteligenţa umană.

Domeniile de bază:

- robotica inteligentă şi domeniile conexe;
- organizarea memoriei;
- reprezentarea cunoştinţelor;
- memorare, căutare şi regăsirea a informaţiilor;
- modele de învăţare;
- tehnici de interfaţă inteligentă;
- rezolvarea unor probleme logice complexe;
- recunoaşterea formelor, vedere artificială şi realitatea virtuală;
- recunoaşterea şi sinteza vocii;
- suport decizional în condiţii de incertitudine;
- limbaj natural;
- varietate de instrumente (sisteme expert, reţele neouronale, sisteme fuzzy, algoritmi genetici, etc).
- AI distribuită şi sisteme de agenţi inteligenţi, etc.
- teoria jocurilor şi planificare strategică;
- creativitate artificială;
- sisteme inteligente hibride;
- control inteligent;
- Data Mining, etc.

1.1.5 Importanţa

AI cea mai importantă dezvoltare a secolului nostru. Relaţia cu ţările dezvoltate în domeniu.

Proiectul japonez a societăţii informaţionale (1981) generaţia V-a de sisteme de calcul pe baza conlucrării universităţilor, agenţiilor guvernamentale, firme, etc. – calculatoare inteligente: traducere în limbaj natural, prelucrări de imagini şi a vocii, rafinarea şi învăţarea cunoştinţelor, luarea automată de decizii, dezvoltarea unor mecanisme de învăţare etc.

E.Feigebaum & P.Cordruc - „The Fifth Generation" (1993).

Programe lansate în acest sens:

Anglia – ALVEG, UE – Esprit; Franţa, Rusia, Canada, Austria, Italia, Singapore – proiecte proprii de dezvoltare a AI.

SUA – nu a avut un proiect guvernamental, dar agenţii şi firme private au investit enorm în domeniu.

Dezvoltarea tehnologiei VSA (MCC – Micro Computer Technology)

DARPA 3 proiecte esenţiale:
- ALVINN (Autonomous Land Vehicle) – vehicule terestre (CMU Naval Lab, 2850 mile, 98% automat)
- ASSOCIATED PILOT – SE de pilotare automată;
- STRATEGIC COMPUTING PROGRAM – proiect de supercalculator militar.

Firme cu proiecte proprii: IBM, AT&T-BELL, HP, XEROX.

Comentariile în jurul proiectului japonez.

Calculatorul NEXT (Steve Jobs): orientare în spaţiu - recunoaştere, sinteză şi reproducere a vocii umane – traducere din limbaj natural în codificat.

1.1.6 Discipline care au contribuit la dezvoltarea AI.

Domeniu distinct în jurul anilor 50 – domenii discutate atunci au devenit ramuri ale AI. Detalii relativ la domenii [Norvig2003].

Filosofia

Probleme:
- Regulile formale pot duce la concluzii corecte?
- Cum se poate reprezenta raţionamentul din creierul uman formal pe un creier fizic?

- Cum se creează cunoştinţele şi din ce provin?
- Cum guvernează cunoştinţele acţiunile?

Începuturile – Aristotel (384-322 î.e.n) silogismul : „orice om este muritor – Socrate este om -> socrate este muritor” – modus ponens, se generează concluziile mecanic. A dezvoltat primul set de reguli relativ la partea raţională a gândirii.

Rene Descartes (1596-1615) – fondatorul ideii de raţionare ca sistem fizic şi a puterii raţionamentului. Este autorul teoriei *dualismului*:

- o parte a gândirii (spiritului) uman este deasupra naturii şi a legilor fizice – gândirea umană are un caracter dual;
- gândirea animalelor nu are caracter dual – pot fi tratate ca mecanisme.

Opusul dualismului – *materialismul* – creierul uman funcţionează fizic conform structurii sale.

O altă problemă – sursa cunoştinţelor:

- *Empirism* – începe cu Francis Bacon (1561-1626): „nimic nu se înţelege dacă nu a fost în prealabil perceput”.
- Curentul inducţionist David Hume (1711-1770): regulile generale se obţin prin asocieri repetate ale elementelor lor.
- Cercul de la Viena – Bernard Rusell (1872-1970), Rudolf Carnap (1891-1970) – doctrina pozitivismului logic: cunoştinţele se pot lega de teorii logice legate de experienţe. Cartea lui Carnap *Logical Structure of the World* (1928) – teoria raţionamentului prin procese computaţionale.

Legătura dintre cunoştinţe şi acţiune este vitală pentru AI deoarece inteligenţa nu înseamnă numai raţionament ci şi acţiune. Stă la baza unor domenii de o importanţă deosebită în IE: de ex., teoria agenţilor inteligenţi sau a deciziilor bazate pe mecanismele AI.

Matematica

Probleme:

- Care sunt regulile formale care duc la concluzii corecte?
- Ce poate fi calculat?
- Cum putem raţiona cu informaţii incerte sau incomplete?

Domeniile principale ale formalizării: logica, calculul, probabilităţile, teoria mulţimilor fuzzy etc.

Legătura cu logica simbolică:

- George Bool (1815-1864) – logica propoziţiilor.
- Gottlob Frege (1848-1925) – include obiecte şi relaţii -> FOPL.

- ’20 – Whitehead, Rusell, Tarski, Chuma – bazele aplicării logicii formale (FOPL) în reprezentarea lumii reale;
- ’30- 40 – Alonso Church, Kurt Goedel, Emile Post, Alan Turing (părintele AI, 1912-1954). Maşinile Turing – (1936) – limba engleză poate fi descrisă cu un automat şi un singur procesor de poate prelucra orice informaţie simbolică sau numerică.

Legătura cu teoria algoritmilor:

- David Hilbert (1862-1943): 23 de probleme fundamentale ale matematicii: ultima este decidabilitatea algoritmilor, prin care a vrut să stabilească limitele demonstrabilităţii.
- Kurt Godel (1906-1978) – există o metodă numerică pentru orice problemă decidabilă din FOPL, dar FOPL nu acoperă principiul inducţiei complete.
- Church-Turing – există funcţii care nu pot fi reprezentate printr-un algoritm -> nu pot fi calculate; care sunt limitele calculabilităţii. De exemplu, nu se poate determina pe cale mecanică dacă un program va da un rezultat pentru un input dat sau va intra în ciclu infinit.
- Problema intractabilităţii: timpul de rezolvare a unei instanţe a problemei creşte exponenţial cu dimensiunea instanţei – problema complexităţii algoritmilor & NP-completitudinii (Cook – 1971 & Karp – 1972).

Legătura cu teoria probabilităţii:

- Gerolamo Cardano (1501-1576) – probabiltăţile legate de jocuri.
- Piere Fermat (1601-1660), Blaise Pascal (1623-1790), James Bernoulli (1654-1705), Pierre Laplace (1749-1827), teoria probabilităţilor & statistică.
- Thomas Bayes (1702-1761) determinarea probabilităţilor unor evenimente viitoare pe baza unora cunoscute şi a unor probabilităţi de trecere între evenimente -> baza raţionamentelor în condiţii de incertitudine în AI.

Teoria informaţiilor: Claude Shanon.

Mulţimile Fuzzy şi logica fuzzy a fost introdusă de Lotfy Zadeh, Universitatea Berkley în 1965, incertirudinea lingvistică –dezvoltată de şcoala japoneză, dar şi de alţii, prin includerea în aplicaţii de electronică casnică (maşini de spălat, televizoare, instalaţii de aer condiţionat etc.). Aplicaţii diverse în reţele neuronale, în reprezentarea cunoştinţelor şi sisteme decizionale.

Ştiinţe Economice

Probleme:

- Cum trebuie luate decizii pentru a obţine un profit maxim?
- Cum procedăm dacă beneficiul se va obţine după o perioadă lungă de timp?
- Cum procedăm dacă o afacere merge prost?

1776 Adam Smith – gândirea economică tratată ştiinţific – colaborarea unor agenţi individuali pentru a-şi asigura bunăstarea.

Economie, nu înseamnă numai bani – modul în care se ajunge la un anumit rezultat economic. Reprezentarea matematică a utilităţii (rezultatului preferat) Leon Warlas (1834-1910), Frank Ramsey (1931) – John von Neumann & Oskar Morgenstern în *The Theory of Games and Economic Bahaviour*, 1944.

Teoria deciziilor: combinarea teoriei utilităţii cu teoria probabilităţilor şi altele în vederea creării unui mediu decizional, nu numai economic. Se bazează pe teoria jocurilor (Neumann & Morgerstern), teoria deciziilor secvenţiale de tip Markov (formalizate de Richard Bellman – 1957)

Simularea euristică: Herbert Simon (1916-2001) laureat al premiului Nobel în economie în 1978 – înlocuirea soluţiilor optimale cu „soluţii suficient de bune".

- **Cibernetica** - Wiener, Odobeja (`40-`50) – studiul reglajului automat şi a comunicării om-maşină; combină elemente de teoria informaţiilor cu elemente de control bazate pe feed-back utilizate la sisteme de calcul.

- **Gramaticile formale** – 1900 extindere a logicii formale – gramatici comparative -> lingvistică, lingvistică computaţională.

Ingineria calculatoarelor

- Leonardo de Vinci (1452-1519) – proiecte, dar nu a realizat calculatorul mecanic.
- Wilhem Ashickard (1592-1635) în 1623 primul calculator mecanic, continuate cu Blaise Pascal (1623-1662) în 1642, Wilhelm Leibnitz (1646-1716).
- Holleritz (1912);
- Charles Babbage (1842);
- 1944 Mark I (Harvard 1944);
- ENIAC (University of Pensylvania 1947) şi UNIVAC (Sperry-Land).

1.2 Cronologia AI

1.2.1 Faza pre-incipientă (1943-1955)

Primele cercetări AI de Waren McCulloch & Walter Pitts (1943) cele 3 surse:

- cunoştinţe despre funcţionarea neuronilor şi a creierului;
- analiza formală a logicii propoziţiilor al lui Rusell & Whitehead;

- teoria computaţională a lui Turig.

Teoria reţelelor neuronale care pot fi instruite.

Donald Hebb (1949) – regulile de modificare a conexiunilor între neuroni -> regula de învăţare a lui Hebb utilizată şi azi.

Marvin Minsky & Dean Edmonds, absolvenţi de Princeton – prima reţea neuronală (SNARC) în 1951 cu 40 de noduri. Teza a fost primită cu reţinere dar von Neumann a spus „chiar dacă nu funcţionează va fi operaţional". Minsky a elaborat ulterior o teoremă care a stopat pentru mult timp cercetările în reţele neuronale.

Alan Turing – 1950, „Computing Machinery and Intelligence" – a introdus o serie de concepte moderne: testul Turing, machine learning, algoritmi genetici şi reinforcement learning.

Jocuri – caracter euristic cu strategii de rezolvare a problemelor.

- 1950 – jocurile de şah, go etc.
- 1952, 1955 Programele Claude Shanon la MIT de şah.
- Allen Nevel – jocuri complexe (GO) – RAND Corporation (logica binară de ordinul I - Nevel & Simon - 1972).
- 1955 – Warr – traducere automată cu dicţionare mari – abordare simplistă.

1.2.2 Perioada iniţială (1956)

John McCarty – Princeton după doctorat trece la Darmounth College şi convinge pe Shanon, Minsky şi Rochester să organizeze în iunie 1956 – ***IBM*** (***Dartmounth College*** – Workshop de 2 luni cu 10 participanţi). Problemele discutate cum ar fi: reţele neuronale, reprezentarea cunoştinţelor, demonstrarea teoremelor, etc. au devenit domenii în AI.-> s-a introdus numele propus de McCarthy **„Artificial Intelligence"** – mai corect „raţionalitate computaţională" -> explozia AI, dominată peste 20 de ani de MIT, CMU, Stanford şi IBM.

1956-1957 - programul Logic Theorist (demonstarea automată a teoremelor) Nevel, Shaw, Simon (Nevel & Simon - 1972) de la Carnegie Mellon University, în limbajul IPL (Information Processing Language) predecesorul LISP, ALGOL. Simon: „am construit un program care poate rezolva probleme nenumerice". Ei au dezvoltat limbajul IPL şi au compilat manual programele. IPL a influenţat apariţia limbajului Fortran (1954) şi teoria gramaticilor generative a lui Naom Chomsky (1955-1957) ->

a influenţat domenii ale AI ca de ex., lingvistica computaţională şi tratarea limbajului natural.

1.2.3 Epoca entuziastă (1956-1969)

1958 – Rosenbloth – teoria percepţiei – perceptroni, recunoaşterea, învăţarea liniară.

1958 – John McCarthy – LISP limbaj în care s-au dezvoltat multe aplicaţii ale AI.

1958 MIT – primul laborator specializat sub conducerea lui McCharty mutat de la Dorthmounth.

1958 GPS (General Problem Solver dezvoltat de Newel, Shaw & Simon) – simulează modul de gândire umană – printre probleme rezolvate a fost: misionar-canibal => ipoteza sistemelor fizice simbolice (Nevel & Simon, 1976) „condiţia necesară şi suficientă pentru a rezolva o problemă care necesită inteligenţă este să se dispună de un sistem fizic uman sau maşină, care prelucrează simboluri".

Echipa lui Nathaniel Rochester IBM – mai multe programe de AI.

- ⇨ Geometry Theorem Power - Herbert Galetner (1959) probleme de geometrie sintetică de nivel mediu din liceu -> puzle & jocuri, roboţi, integrarea simbolică etc. 60-70.
- ⇨ Arthur Samuel (MIT) 1956-1965 - joc de şah care învaţă şi ajunge la nivel de maestru, demonstrat la televizor iniţial în februarie 1956.

1958 EPAM (Elementary Perceiver and Monitorizer, Feigebaum & Simon) – în IPL primul program care învaţă -> Machine Learning.

1958 McCrathy – Advice Taker, descrie un program care poate rezolva orice problemă care poate fi descrisă prin FOPL, de exemplu, organizarea primirii automate a avionalor pe un aeroport, reorientarea automată a unui curs, etc.

1963 McCharty - trece la Stanford şi dezvoltă pe baza Advice Taker, teoria roboţilor inteligenţi.

Minsky dezvoltă la MIT programe cu studenţi, formând domeniul microworlds, care rezolvă problemele de colegiu de anul 1, de teste IQ etc. (1961-1976), programe de tratare a limbajului natural etc.

1965 Robinson – program care rezolvă orice Problemă a Logicii inferenţiale – reprezentabilă actual în Prolog.

1.2.4 Epoca de maturitate

Probleme de traducere automată – finanţată de National Research Council, din rusă în americană: Proiect Sputnik (1957).

Rezolvarea problemelor din microworld nu se putea aplica la probleme de dimensiuni mari -> algoritmii genetici (Ffriedberg 1958, 1959) – ideea de bază: aplicând mici mutaţii asupra unui program se poate obţine un program suficient de bun pentru anumite sarcini simple -> sute de ore de rulare fără succes.

A doua problemă: raportul Lighthill, 1973, critică cercetarea AI şi Guvernul Britanic suspendă subvenţiile.

A treia problemă: apar limitările sistemelor inteligente, de exemplu Minky & Papret în cartea *Perceptrons* (1969) demonstrează că sistemele au o putere de reprezentare foarte limitată -> abandonarea reţelelor neuronale până în `80 multilayer.

S-au dezvoltat sistemele bazate pe cunoaştere:

- ⇨ 1965-1971- DEDRAL (Feigebaum – elev al lui Simon, Buchanan – filosof devenit informatician, Lederberger, laureat al premiului Nobel în genetică - Stanford) – primul SE, structura moleculară (masa şi compoziţia moleculară);
- ⇨ Feigebaum – HPP (Heuristic Programming Project) extinderea SE la alte activităţi umane;
- ⇨ Feigebaum, Buchanan, Shortliffe -> MYCIN - 450 de reguli pentru infecţia sangvină; actualmente peste 2000 de reguli; multe alte sisteme de diagnoză [P.Korduk 1979].

1968 – MACSYMA (MIT, Carl Engleman, William Martin, Sell Moses) – orice problemă de logică formală bazată pe axiome.

Explozia de după 1965.

Roberts 1963 – analiza vizuală a mediului – vedere artificială 1993 Nawal.

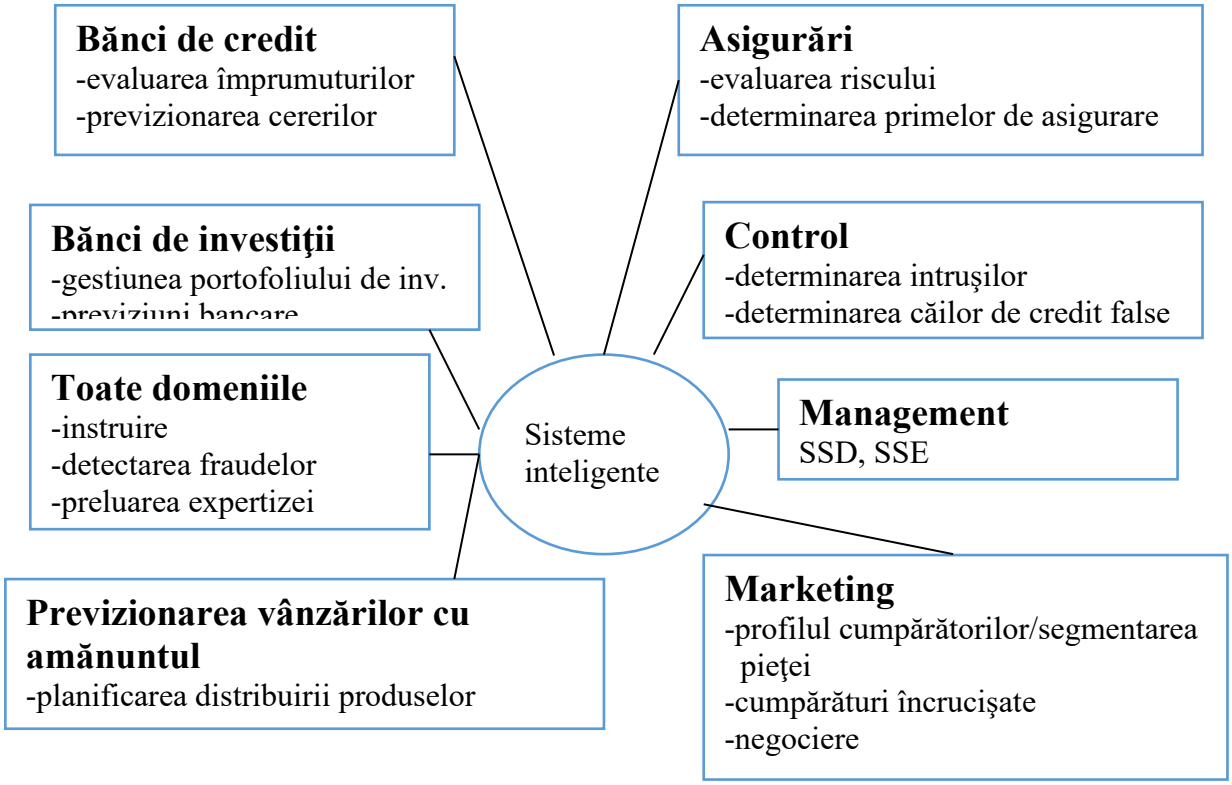
Limbaj natural Winograd 1972 (SHRLDU) – recunoaşterea limbajului natural, referenţierea pronumelor, blocuri de cuvinte. Elevul său de la MIT, Eugen Charniak – pentru tratarea limbajului natural trebuie cunoştinţe generale despre lume.

Yale – Roger Schank dezvoltă cu studenţii o serie de programe de înţelegere a limbajului natural.

1973 LUNAR System (Woods) – NASA.

11 mai 1997, DEEP BLUE, IBM, bate pe Garry Kasparov 3,5 la 2,5 în 6 partide: algoritm performant, calculator de viteză mare, hard specializat.

1.3.Legătura cu IE



- utilizarea SI de principalele bănci şi instituţii financiare;
- Country Wide Founding – SE adaptive;
- Fuji Bank – sisteme fuzzy – creşterea serviciilor, reducerea costurilor;
- American Express de la 15$/tranzacţie la 1,44;
- Visa Internaţional – 6 luni 40.000.000 USD – reţea neuronală;

Avantaje:

- lucrează non-stop;
- lucrează repede;
- nu sunt influenţabile ca şi experţii umani;
- lucrează în orice condiţii.

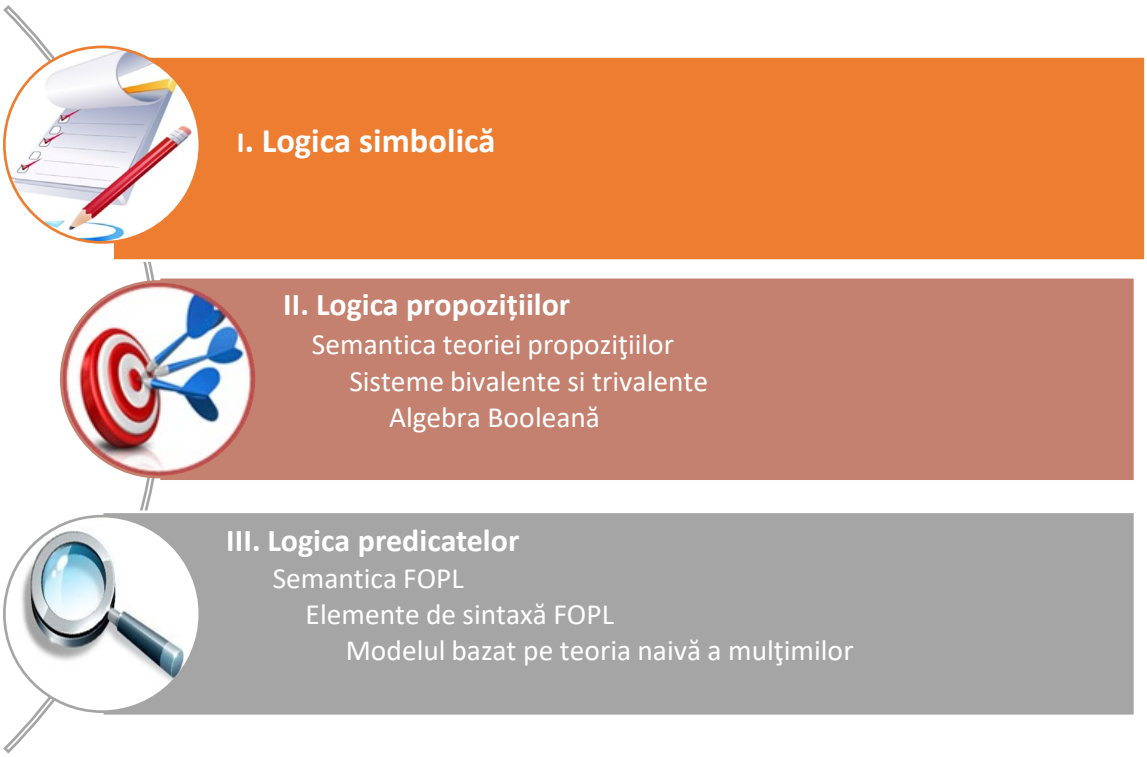
Caracteristici care le recomandă în afaceri:

- Capacitate de învăţare – burse, bănci, asigurări – reţele neuronale, algoritmi genetici;
- Flexibilitate, adaptabilitate – date incomplete şi incorecte – reţele neuronale; imprecizie lingvistică – sistemele fuzzy;
- Explicaţie – domeniul bancar – SE;
- Descoperire de modele, Data Mining – reţele neuronale.
- Extragerea de informaţii distribuite – agenţi inteligenţi, mocbili, sisteme colaborative, multi-expert etc.

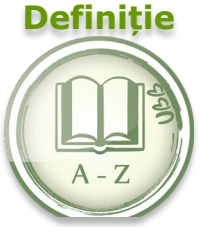
Disciplină: Inteligență Artificială

Modulul II.

Bazele logice ale Inteligenței Artificiale



2.1. Logica simbolică



[DEX03] *"**LOGICA** este ştiinţa demonstraţiei al cărui obiect este stabilirea condiţiilor corectitudinii gândirii, a formelor şi legilor generale ale raţionării corecte, conforme prin ordinea ideilor cu organizarea legică a realităţii obiective".*

[Larouse1998] *„Teorie ştiinţifică a raţionamentului ce exclude procesele fiziologice".*

http://en.wikipedia.org/wiki/Logic

Logic, from Classical Greek λόγος (logos), originally meaning *the word*, or *what is spoken*, (but coming to mean *thought* or *reason*) is most often said to be the study of criteria for the evaluation of arguments, although the exact definition of logic is a matter of controversy among philosophers. However the subject is grounded, the task of the logician is the same: to advance an account of valid and fallacious inference to allow one to distinguish good from bad arguments.

Traditionally, logic is studied as a branch of philosophy. Since the mid-1800s logic has been commonly studied in mathematics, and, even more recently, in computer science. As

a formal science, logic investigates and classifies the structure of statements and arguments, both through the study of formal systems of inference and through the study of arguments in natural language. The scope of logic can therefore be very large, ranging from core topics such as the study of fallacies and paradoxes, to specialist analyses of reasoning such as probably correct reasoning and arguments involving causality.

Una dintre cele mai vechi ştiinţe – grecii antici pentru a obţine avantaj verbal asupra oponenţilor în retorică (Aristotel 384-322 î.e.n) [http://www.neurocomputing.org/Logic_History/body_logic_history.html] - The Rationale for Analog Truth Value Operations in the History of Logic by David D. Olmsted (2000)

Dintre principalele aplicaţii ale logicii din domeniul IE:

- baze de date şi de cunoştinţe, a căror interogare se bazează pe logică.
- SE, SSD, SSE, dar şi deciziile de zi cu zi ale managerului sau omului de afaceri se bazează pe raţionamente logice.
- Practic nu există domeniu de afaceri în care logica să nu joace un rol major.

Logica are două ramuri esenţiale:

- **logica clasică**, sau **logica aristoteliană**[3] **-** categoriile logice fundamentale: noţiunea, judecata sau raţionamentul;
- **logica matematică, logica formală sau simbolică**[4]**.**

Logica simbolică fundamentul raţionamentului uman[5] - avantaje:

- asigură expresivitatea şi rigurozitatea în reprezentarea cunoştinţelor;
- asigură deducerea unor cunoştinţe noi pe baze altora deja existente.

Utilizează simboluri - reprezentarea obiectelor şi a operaţiilor executate asupra simbolurilor.

[3] Aristotel (384-322î.e.n) a fost cel care a definit pentru prima dată precis o serie de reguli care guvernează partea raţională a gândirii. El a dezvoltat un sistem informal al silogismelor proprii raţionamentelor, în baza cărora pornind de la premise se pot genera mecanic concluzii [Rusell&Norwig03]

[4]Exprimarea ideilor logicii formale aşa cum a fost ea definită de grecii antici sub o formă matematică a început cu lucrările lui George Bool (1815-1864), care a dezvoltat în 1847 logica propoziţiilor. Gottlab Forge (1848-1925) a dezvoltat teoria lui Bool incluzând obiecte şi relaţii creând astfel logica predicatelor de ordinul I utilizată astăzi în reprezentarea cunoştinţelor. Alfred Tarski (1902-1983) a introdus o teorie referenţială prin care leagă obiectele logicii de cele din lumea reală [Rusell&Norvig03, Luger02].

[5] [Maliţa&Maliţa87]

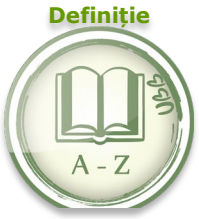
Pentru informatică şi pentru IE în special, logica formală prezintă un interes aparte. Dicţionarele de informatică se mărginesc în general numai la acestea:

- [Colin90]: "**logica** = substantiv, ştiinţa care se ocupă cu gândirea şi raţionamentele; **logica formală** = tratarea formei şi structurii, ignorând conţinutul";
- [Oxford91] "Logica este un formalism de reprezentare a cunoştinţelor şi a raţionamentelor, dezvoltat iniţial de către matematicieni pentru a formaliza raţionamentele matematice. In logica matematică, investigaţia cuprinde metode matematice împrumutate din algebră şi teoria algoritmilor. Sistemele cele mai uzuale sunt calcul propoziţiilor şi cel al predicatelor."

[Boden87] Calculatorul şi deci, informatica prelucrează simboluri. Din această cauză logica formală, în tratatele de informatică [Patterson90], apare în general sub denumirea de ***logică simbolică***.

***Logica simbolică de ordinul I*, *logica propoziţiilor* + *logica predicatelor de ordinul I*[6].**

2.2 Logica propoziţiilor (PL)



Conform [DEX03], *noţiunea de* ***propoziţie*** *are atât semnificaţia de CEA MAI MICĂ UNITATE SEMANTICĂ care exprimă o idee, o judecată etc., utilizată în gramatică, cât şi cea de enunţ a cărui valoare de adevăr este întemeiată pe bază de reguli explicit exprimate, utilizată în logica simbolică.*

Exemplu: simbolul p - ataşat propoziţiei "Grivei latră" ia valoarea **adevărat** (True - T) dacă într-adevăr latră, respectiv **fals** (False-F) dacă nu latră.

Diferenţa dintre gramatică şi logica simbolică. În PL, valoarea de adevăr a propoziţiei este calitatea acesteia de a fi adevărată sau falsă în întregul ei şi nu interesează obiectele constitutive ale sale. Exemplu: propoziţia „Ionescu este managerul societăţii comerciale" sau „Ionescu manager".

Teoria logică este în esenţă un limbaj de reprezentare a cunoştinţelor. Ca orice limbaj, are două aspecte esenţiale:

- aspectul *semantic* sau *abordarea semantică*;
- aspectul *sintactic* sau *abordarea sintactică*.

[6] Bibliografie foarte vastă în domeniu chiar şi în limba română [Florea&Boangiu94, Maliţa&Maliţa87, Mihăiescu, 66] sau [D.vanDalen84, Gray85, Graham88, Nilson98, Turner84]

Alţi autori [Nilson98] consideră logica sub 3 aspecte:

- *limbajul* (cu sintaxă care specifică expresiile corecte în acest limbaj);
- *regulile inferenţiale*, prin care se manevrează propoziţiile limbajului;
- *semantica* pentru asocierea elementelor limbajului de semnificaţia lor.

Semantica - aspectele intime (interne) ale universului problemei - obiectivele:

- notarea propoziţiilor ataşate universului problemei cu ajutorul unor simboluri şi fixarea valorii de adevăr a acestor simboluri;
- stabilirea simbolurilor care joacă rolul de conectori, adică leagă simbolurile ataşate propoziţiilor;
- stabilirea valorii de adevăr a noilor propoziţii astfel obţinute (compuse).

Conceptul central în abordarea semantică este cel de „*valoare de adevăr*”: „Este o formulă, o tautologie, adică este ea adevărată indiferent de faptul că are părţile componente adevărate sau false?”

Within the study of logic, a tautology is a statement that is true by its own definition. [http://en.wikipedia.org/wiki/Tautology]

Spre deosebire de abordarea semantică, cea *sintactică* are ca şi concept central “*demonstraţia logică*” şi anume, trebuie să răspundă la întrebarea: “Este o formulă demonstrabilă în cadrul unui sistem logic, sau nu?”.

Din acest motiv, de regulă, semantica este asemănată cu studiul expresiilor din algebră, unde se demonstrează corectitudinea formulelor, în timp ce sintaxa se aseamănă cu rezolvarea sistemelor de ecuaţii prin metoda substituţiei.

2.2.1 Abordarea semantică

În cadrul abordării semantice trebuie fixate 5 elemente de bază [Patterson90]:

- limbajul de descriere a formulelor logice (alfabetul limbajului);
- valoarea de adevăr a simbolurilor ataşate propoziţiilor (respectiv predicatelor);
- funcţiile de evaluare;
- mecanismul de raţionament reprezentat de consecinţele logice;
- principiile teoriei logice.

PL- propoziţii simple - tratate atomic, ca un tot unitar - se vor nota cu câte un simbol - **litere mari sau mici de la mijlocul alfabetului P,Q,R,…,** respectiv **p,q,r,...** *Propoziţiile simple - nici o parte a lor nu este o propoziţie.*

PL clasic - orice propoziţie poate fi T sau F, dar nu amândouă deodată **(legea terţului exclus) - logică bivalentă.**

Teoremele în logică, sisteme în care se pleacă cu valoarea de adevăr a unor propoziţii, numite ***ipoteze (premise)***, şi aplicând o serie de reguli de raţionare (reguli inferenţiale), operaţii şi funcţii logice, se ajunge la alte propoziţii, numite ***concluzii***.

[Wikipedia]

A **theorem** is a proposition that has been or is to be proved on the basis of explicit assumptions. Proving theorems is a central activity of mathematicians.

A theorem has two parts, stated in a formal language – a set of assumptions, and a conclusion that can be derived from the given assumptions according to the inference rules of the formal system comprising the formal language. The proof, though necessary to the statement's classification as a *theorem*, is not considered part of the theorem.

In general, a statement must not have a trivially simple derivation to be called a theorem. Less important statements are called:

- ***lemma***: a statement that forms part of the proof of a larger theorem. The distinction between theorems and lemmas is rather arbitrary, since one mathematician's major result is another's minor claim. Gauss' lemma and Zorn's lemma, for example, are interesting enough *per se* that some authors present the nominal lemma without going on to use it in the proof of any theorem.
- ***corollary***: a proposition that follows with little or no proof from one already proven. A proposition *B* is a corollary of a proposition or theorem *A* if *B* can be deduced quickly and easily from *A*.
- ***proposition***: a result not associated with any particular theorem.
- ***claim***: a very easily proven, but necessary or interesting result which may be part of the proof of another statement. Despite the name, claims are proven.
- ***remark***: similar to claim. Usually presented without proof, which is assumed to be obvious.

A mathematical statement which is believed to be true but has not been proven is known as a conjecture. Gödel's incompleteness theorem establishes very general conditions under which a formal system will contain a true statement for which there exists no derivation within the system.

As noted above, a theorem must exist in the context of some formal system. This will consist of a basic set of axioms (see *axiomatic system*), as well as a process of inference, which allows one to derive new theorems from axioms and other theorems that have been derived earlier. In mathematical logic, any provable statement is called a theorem.

1. Limbajul de descriere a formulelor (sistemul notaţional)

În PL - alfabetul este format din simboluri propoziţionale definite astfel:
- litere mici, p,q,r,…sau mari P,Q,R,... ataşate propoziţiilor;
- conectori logici:

~ sau ¬ - negaţia
^ - conjuncţia
V – disjuncţia
→ - implicaţia
↔ - echivalenţa

- alte simboluri, cum ar fi de exemplu: (,).

PL - *formulele corecte, corect formulate* sau *bine formulate*, notate în literatură cu *wff – well formatted formulas* - definesc recursiv:
(i) un atom (simbol ataşat unei propoziţii simple) este un *wff;* un *wff* în paranteză;
(ii) dacă P este un *wff* , negatul său ~P este un *wff;*
(iii) dacă P şi Q sunt *wff,* atunci P^Q, PVQ, P→Q şi P↔Q au aceeaşi proprietate;
(iv) mulţimea *wff*-urilor este generată de regulile (i)-(iii).

(P→ (Q^~R)); (P→).

Propoziţiile compuse sunt wff-uri care se realizează pe baza unor conectori sau operatori logici. Principalii operatori logici, aşa după cum s-a prezentat mai sus, sunt: ***negaţia, conjuncţia şi disjuncţia***.

Negaţia – unei propoziţii, **"non P"** - [French91]: "not.P", ¬P, $\overline{P}$, sau ~P. Operaţiile logice - cu ajutorul *tablelor de a*devăr (corespunzătoare tablelor operaţiilor aritmetice elementare). Aceste table indică valoarea de adevăr a rezultatului în funcţie de valorile de adevăr ale componentelor.

P	**~P**
F	T
T	F

Exemplu: P = "Bugetul trebuie aprobat anual", care are valoarea T, negaţia ~P, "Bugetul nu trebuie aprobat anual", are valoarea F.

Conjuncţia – propoziţiilor P şi Q - "P şi Q" - se notează cu P ∧ Q. Notaţii [French91]: P.Q, P .and. Q sau P&Q. Exemplu: P = "Pământul este rotund" şi Q= "impozitul este o datorie faţă de stat". P&Q va fi "Pământul este rotund" şi „impozitul este o datorie faţă de stat".

Disjuncţia – "P sau Q" şi se notează cu "P ∨ Q". Notaţii şi cu: P+Q, P.or.Q sau P ∪ Q. P ∨ Q = T dacă P=T, sau Q=T, sau ambele propoziţii sunt T. Sau exclusiv "sau Cezar sau nimic". "sau" logic trebuie interpretat în sensul limbajului curent astfel: dacă P este propoziţia "Firma X practică comerţ en-gros" şi Q este propoziţia "Firma X practică comerţ en-detail", propoziţia "P sau Q" - **"sau exclusiv"** şi se notează cu **xor**.

2. Valoarea de adevăr

O teorie logică - cel puţin două valori de adevăr: true (T) şi false (F) - exact două valori, avem de a face cu *logica bivalentă*. Aceasta are diferite forme, cum ar fi: logica propoziţiilor, cea a predicatelor, modală sau temporală[7].

Klee, Bochvar şi Lukasiewicz - *logica trivalentă* "adevărat" şi "fals", poate exista şi o a treia valoare, "nu ştiu" sau "necunoscut" sau „nedeterminat în momentul de faţă dar urmează să fie determinat în viitor", „imposibil", „absurd". etc . Ulterior - alte logici trivalente sau cu 4 valori de adevăr, logici denumite *neclasice*, cum ar fi: logicile lui Lucasiewicz, Post, Klee sau Bochvar. -> [Turner85], -> logicile fuzzy în care pot exista o infinitate de valori de adevăr, logici bazate pe teoria introdusă de Lotfi Zadeh [Zadeh65].

3. Funcţia de evaluare

Funcţie de evaluare - valorizare - de adevăr a unui atom, o funcţie care ataşează atomului respectiv o valoare T sau F în logica bivalentă. -> funcţia de evaluare a unei propoziţii ataşează o valoare de adevăr propoziţiei în funcţie de valorile de adevăr ale componentelor, pe baza regulilor de evaluare a conectorilor.

Regulile de evaluare a conectorilor sunt date prin tablele de adevăr. În general, valoarea de adevăr a unei propoziţii p se notează cu p^V.

Interpretare a unei formule - atribuirea unei valori de adevăr fiecărei componente a formulei respective. Exemplu, pVq - 4 interpretări posibile, care formează domeniul de interpretare:

$$I_1=\{T,T\}, \quad I_2=\{T,F\}, \quad I_3=\{F,T\}, \quad I_4=\{F,F\}$$

Considerând, de exemplu, interpretarea I_2, valorile componentelor sunt:

$$p^V = T \text{ şi } q^V = F$$

iar valoarea obţinută prin funcţia de evaluare este:

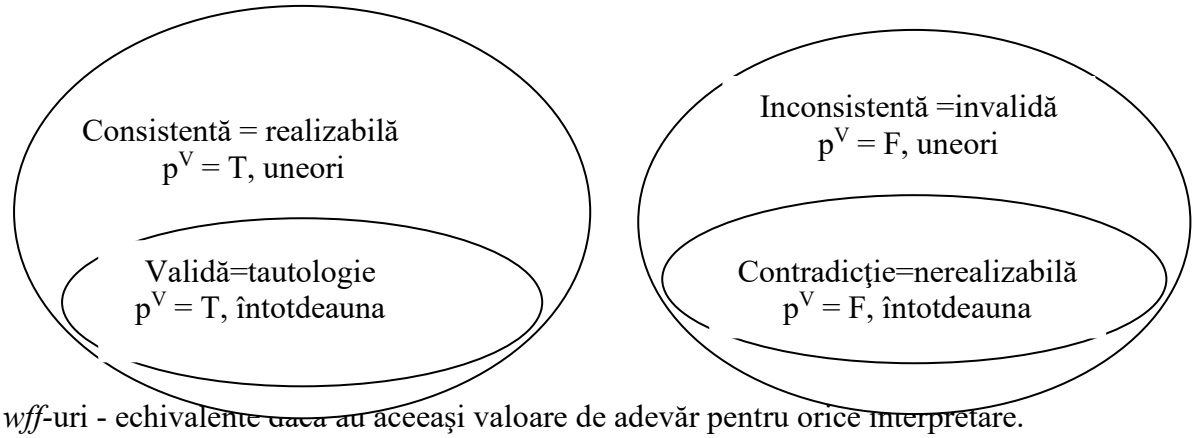
[7] [Turner85, http://plato.stanford.edu/enties]

$$(p \vee q)^V = T$$

Pe baza domeniilor de interpretare şi a funcţiilor de evaluare, *wff* – urile se clasifică în:

- *tautologii* sau *formule valide* – T indiferent de interpretare;
- *consistente* – care iau valoarea T pentru unele interpretări;
- *inconsistente (invalide)* – F pentru unele interpretări;
- *contradicţii* - F pentru orice interpretare.

Relaţia dintre aceste tipuri de *wff* - schemele Wyne:



wff-uri - echivalente ~~dacă au~~ aceeaşi valoare de adevăr pentru orice ~~interpretare~~.

modelul unui wff - o interpretare pentru care formula ia valoarea T (în exemplul nostru I_2). Analog, se poate defini *modelul unui ansamblu de formule* ca fiind o interpretare pentru care toate formulele sunt adevărate.

4. Principiile teoriei logice

Orice teorie logică trebuie să respecte anumite principii fundamentale. Respectarea sau dimpotrivă, eliminarea unor restricţii, stabileşte tipul logicii.

a. Principiul fixării numărului de valori logice

Teoriile logice - un număr fixat, n de valori logice distincte cu $2 \leq n \leq \infty$. |V| cardinalul mulţimii V - de cele mai multe ori acest cardinal are o valoare finită. Există însă şi logici infinite.

Logică este cea bivalentă, în care

$$V = \{T,F\} \text{ şi deci } |V|=2$$

În general se consideră 3 conectori de bază şi anume: ~, V, ^ cu tablele de adevăr:

P	~P
T	F
F	T

P^Q	T	F
T	T	F
F	F	F

PVQ	T	F
T	T	T
F	T	F

Funcţie logică - oricărei combinaţii de valori logice să-i corespundă o valoare logică. Funcţiile logice binare, adică acelea care au 2 argumente. Aceste funcţii fac ca la combinaţiile FF, FT ,TF şi TT să le corespundă valorile F sau T. Cum în domeniul de definiţie există patru combinaţii posibile, iar în cel al valorilor, două -> [Gray85] există numai 2^4=16 funcţii logice binare distincte. Se poate demonstra, de asemenea, că orice funcţie binară se poate reprezenta cu ajutorul celor 3 operatori definiţi anterior.

Dintre funcţiile logice, cele mai importante sunt considerate:

- **Implicaţia - inferenţa logică -** operaţia principală în domeniul bazelor de cunoştinţe; notată cu → şi are semnificaţia **"dacă P, atunci Q".** Propoziţia P→Q ia valoarea fals numai dacă din P „adevărat" rezultă Q „fals". Avem, deci, următoarele situaţii posibile:
  - dacă P este adevărată, atunci şi Q trebuie să fie adevărată;
  - dacă P este falsă, Q poate fi adevărată sau falsă;

- **Echivalenţa logică – echivalenţa** - notată cu ↔ - reprezentată de dubla implicaţie şi deci, generează o **propoziţie compusă care este adevărată dacă cele două propoziţii P şi Q sunt concomitent adevărate sau false**.

Pentru a reprezenta cele două funcţii binare, vom recurge la tabla de adevăr:

P	Q	P→Q	Q→P	~P	~P∨Q	P↔Q	P→Q∧Q→P
T	T	T	T	F	T	T	T
T	F	F	T	F	F	F	F
F	T	T	F	T	T	F	F
F	F	T	T	T	T	T	T

Din tabel se pot deduce o serie de relaţii utile, dintre care amintim:

$$P \to Q = \sim P \vee Q$$

$$P \to Q = \sim P \vee Q = \sim(\sim Q) \vee \sim P = (\sim Q) \to (\sim P) \qquad (1)$$

$$P \leftrightarrow Q = P \to Q \wedge Q \to P$$

Observaţie. În logica formală relaţiile de mai sus sunt deosebit de importante. (1) joacă un rol deosebit în raţionamentele matematice. De exemplu, o funcţie f este injectivă dacă:

pentru orice $x \neq y$ are loc $f(x) \neq f(y)$

P → Q

Verificarea injectivităţii se realizează cu implicaţia echivalentă:

$\sim Q \to \sim P$, adică $f(x) = f(y) \to x = y$.

Observaţie. Formula (1) trebuie tratată cu grijă - (1) nu asigură egalitatea $P \to Q = Q \to P$. Pentru ilustrare să considerăm următorul exemplu:

„avem o creştere a veniturilor „→ „avem o creştere a vânzărilor"

aceasta este echivalentă cu „nu avem o creştere a vânzărilor” → ”nu avem o creştere a veniturilor” dar aceasta nu este echivalentă cu „creşterea vânzărilor” → „creşterea veniturilor”, deoarece vânzările pot creşte şi din alte motive, cum ar fi, solduri, perioadă de sărbători, acţiuni promoţionale, etc.

Algebra booleană nu este singurul sistem logic bivalent - sistemul logic al lui Hilbert-Ackermann are la bază 3 conectori ~, V şi →.

Observaţie. Pornind de la faptul că au loc relaţiile:

$$P \wedge Q = \sim (P \rightarrow \sim Q)$$
$$P \vee Q = (\sim P) \rightarrow Q$$

Lukasiewicz a definit un sistem logic în care operaţiile de bază sunt negaţia şi implicaţia – sistemul logic Lukasiewicz.

În cazul logicii trivalente, deci în cazul |V|=3, avem mai multe sisteme.

Astfel, avem sistemul lui Klee cu V={T,F,U}, unde U este „necunoscut”, având 2 conectori de bază, ~ şi ^, definiţi astfel:

p	~p
T	F
F	T
U	U

p^q	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

De unde se poate deduce:

pVq	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

p→q	T	F	U
T	T	F	U
F	T	T	T
U	T	U	U

p↔q	T	F	U
T	T	F	U
F	F	T	U
U	U	U	U

Sistemul Lukasiewicz, are V= {T,F,I}, unde I indică „un eveniment din viitor nerealizabil în prezent, dar nu un necunoscut”. În acest caz conectorii sunt ~ şi →, cu tablele de adevăr:

p	~p
T	F
F	T
I	I

p→q	T	F	I
T	T	F	I
F	T	T	T
I	T	I	T

de unde se deduce:

pVq	T	F	I
T	T	T	T
F	T	F	I
I	T	I	I

p^q	T	F	I
T	T	F	I
F	F	F	I
I	I	I	I

p↔q	T	F	I
T	T	F	I
F	F	T	I
I	I	I	T

Sistemul Bochvar, creat pentru explicarea unor paradoxuri semantice, are V={T,F,M}, cu M indicând „absurd” şi cu conectorii ~ şi →, având tablele de adevăr:

p	~p
T	F
F	T
M	M

p→q	T	F	M
T	T	F	M
F	T	T	M
M	M	M	T

de unde se deduce:

pVq	T	F	M
T	T	T	M
F	T	F	M
M	M	M	M

p^q	T	F	M
T	T	F	M
F	F	F	M
M	M	M	M

p↔q	T	F	M
T	T	F	M
F	F	T	M
M	M	M	M

Logicile infinite, cele mai cunoscute, - logicile fuzzy, introduse de Lotfy Zadeh, pentru a reprezenta cunoştinţele descrise cu ajutorul incertitudinii lingvistice. V este o submulţime mărginită a lui R_+. Această mulţime - normalizată luând în locul lui V, mulţimea V/|V|, înlocuind fiecare element v din V cu v/|V|. -> V=[0,1] în toate cazurile. De exemplu, considerând o firmă şi notând cu x beneficiul obţinut de firmă, putem defini măsurile fuzzy de exemplu astfel:

- dacă beneficiul este de x=1.000.000, Beneficiu (x)=0
- dacă beneficiul este de x=50.000.000, Beneficiu (x)=0.2
- dacă beneficiul este de x=1.000.000.000, Beneficiu (x)=0.6 …

b. Principiul consistenţei şi non-contradicţiei

Pentru un domeniu de interpretare dat, o propoziţie - o valoare unică din V → principiul valorizări *non-contradictorii* şi *consistente*. Cu alte cuvinte, fie D mulţimea propoziţiilor, considerând V={T,F} şi

$$D_T=\{p\in D \mid p^V=T\}, \text{ respectiv } D_F=\{p\in D \mid p^V=F\}; \text{ se obţine } D_T \cap D_F=\Phi$$

Logicile polivalente, cu $V=\{v_1, v_2, \ldots, v_n\}$, $D_i=\{p\in D \mid p^V=v_i\}$,condiţia de consistenţă se poate scrie:

$$D_i \cap D_j = \Phi, \text{ pentru } i\neq j.$$

Logicile care nu sunt consistente se numesc ***paraconsistente*** sau ***logici suprasaturate***. În aceste logici propoziţiile pot primi deodată două sau mai multe valori de adevăr, deoarece în general

$$D_i \cap D_j \neq \Phi, \text{ pentru } i\neq j.$$

c. Principiul excluderii celei de a n+1-a valori

Unui atom i se poate asocia o valoare din mulţimea V, pe domeniul de interpretare → funcţia de evaluare este total definită pe domeniul de interpretare.

Principiul excluderii celei de a n+1-a valori - în logica bivalentă, legea terţului exclus:

$$D_T \cup D_F = D$$

→ orice propoziţie este ori adevărată, ori falsă, nu poate lua o a treia valoare.

În general, legea excluderii celei de a (n+1)-a valori logice are forma:

$$\bigcup_{n} D_i = D, \text{ pentru } V = \{v_1, v_2, \ldots, v_n\}.$$

i=1

Pornind de la două valori logice, admiterea sau respingerea principiilor b. sau c generează diferite logici. Astfel,

- admiterea ambelor, duce la ***logica cla***sică;
- admiterea consistenţei b şi respingerea c duce la ***logici lacunare*** de tipul logicilor trivalente a lui Klee, Lukasiewicz, Bochvar sau la ***logici intuitiviste***, unde se admite şi a 3-a valoare;
- respingerea lui b şi admiterea lui c duce la ***logici paraconsistente***;
- respingerea ambelor principii, duce la ***logici cu semantici aproximative***, numite şi ***logici relevante***, unde o propoziţie poate admite mai multe valori sau rămâne neevaluată.

d. Principiul constanţei de valorizare a unei propoziţii elementare

Unei valori de adevăr unei propoziţii elementare date trebuie să rămână aceeaşi la toate apariţiile acesteia atâta timp cât se consideră o anumită interpretare. Renunţarea la acest principiu duce la ***logici paraconsistente*** sau ***relevante***.

5. Mecanismul raţionamentului

Realizarea raţionamentului într-un sistem logic → obţinerea unor formule noi pe baza celor existente → o extindere consistentă a cunoştinţelor despre universul problemei.

O formulă Q - consecinţă logică a unei formule P, dacă Q ia valoarea T pentru toate interpretările pentru care P ia valoarea T. În general, Q - consecinţă logică a unei mulţimi de formule $P_1, P_2, \ldots, P_n$, dacă Q primeşte valoarea T pentru orice interpretare pentru care $P_1, P_2, \ldots, P_n$ iau valoarea T. Consecinţa logică se notează în general cu simbolul implicaţiei „→”.

(P1). Q este consecinţa logică a lui $P_1, P_2, \ldots, P_n$ este echivalent cu faptul că
$P_1 \wedge P_2 \wedge \ldots \wedge P_n \rightarrow Q$ este validă.

Pe baza proprietăţilor implicaţiei, propoziţia de mai sus mai poate fi enunţată şi astfel*:*
(P2). Q este consecinţa logică a lui $P_1, P_2, \ldots, P_n$ este echivalent cu faptul că

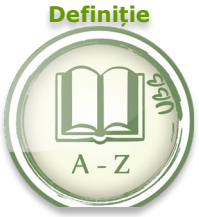
$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \vee \sim Q$$

este inconsistentă.

→ (P1), (P2) **-** constituie baza raţionamentului logic şi a demonstrării corectitudinii, deoarece reduce problema consecinţelor logice la cea a demonstrării validităţii sau inconsistenţei unor formule.

2.2.2. Abordarea sintactică a teoriei logice

2.2.2.1. Sistemul formal - concept de bază a abordării sintactice a teoriei limbajelor.



Sistemul formal este un instrument de analiză, prelucrare şi generare a simbolurilor de bază (semnelor limbajului) şi care duce la structuri de simboluri, respectiv şiruri de simboluri (cuvinte, propoziţii, fraze) acceptate în limbaj.

Din cele de mai sus rezultă că un sistem formal se defineşte ca un cvadruplu:

$$S = \{ A, F, \Upsilon, \Re \}$$

unde:

A - alfabetul sistemului (mulţimea simbolurilor de bază)

F - mulţimea formulelor corecte (wff-urilor); $F \subseteq A^*$, adică F este o submulţime a şirurilor de caractere sau a şirurilor de simboluri din alfabetul A;

Υ este mulţimea axiomelor, adică $\Upsilon \subseteq A^*$; despre Υ se poate demonstra că este decidabilă, adică pentru orice element x al lui A^* se poate demonstra că face parte sau nu din mulţimea Υ a axiomelor;

$\Re$ - mulţimea regulilor de deducţie sau inferenţiale. O regulă de deducţie este o relaţie de aritate n+1 în mulţimea wff-urilo, - orice $R \in \Re$ este o submulţime a produsului cartezian de ordinul n+1 a lui F, adică $R \subseteq F^n x F$, astfel ca, pentru orice $Y=<y_1, y_2, \ldots, y_n>$ corespunde prin R un $x \in F$, cu y_i, i=1,2,...,n. wff-urile y_i, i=1,2,...,n se numesc antecedente, iar x se numeşte consecinţa lui R.

$\Gamma=\{y_1, y_2, \ldots, y_n\}$, *mulţimea premizelor*. Se notează cu $E_0 = \Gamma \cup \Upsilon$ reuniunea dintre mulţimea premizelor şi cea a axiomelor. Se consideră:

$$E_1 = E_0 \cup \{x \mid \exists\, Y \in E_0, \text{astfel încât } R: Y \rightarrow x, \text{ cu } R \in \Re\}$$

adică E_1 este format din E_0 şi imaginile elementelor din E_0 prin reguli de deducţie din $\Re$. Având construit E_1, putem construi - E_2, E_3, …

Dacă în mulţimea E_0, $\Gamma=\Phi$, adică E_0 este format numai din axiome, atunci elementele lui E_i, i=1,2,..., - teoreme ale sistemului formal → un rezultat cunoscut din matematică şi anume, *teoremele unui sistem formal* sunt acele formule care pot fi demonstrate utilizând numai axiome.

Fie $x \in E_i$ o teoremă - printr-o secvenţă de deducţii $D=\{ E_0, E_1, \ldots, E_{i-1}\}$ - secvenţă de deducţii D formează o *demonstraţie a teoremei*.

Un sistem S de formule este decidibil, dacă există o procedură efectivă prin care se poate decide dacă o formulă din S este sau nu o teoremă.

Decidabilitatea - esenţială în teoria sistemelor formale - baza teoriei demonstrabilităţii teoremelor, deci a extinderii cunoştinţelor în cadrul sistemelor logice. Se poate demonstra:

- (PL) este decidabilă
- (FOPL) nu este.

2.2.2.2. Algebra Booleană

Algebra booleană - cel mai important sistem logic bivalent George Boole (1815-1864).

Semantica -Alfabetul şi cuvintele (wff)

1. **A** - simbolurile propoziţionale notate cu p,q,r,… sau P,Q,R,…, constante - A,B,... precum şi pe cei trei conectori: ~, ^ şi ∨.
2. **F** – wff
3. **ϒ - Axiomele de bază ale algebrei booleene sunt:**

Υ_1 – comutativitatea:

$$P \wedge Q = Q \wedge P, \text{ respectiv } P \vee Q = Q \vee P$$

Υ_2 – asociativitatea:

$$P \wedge (Q \wedge R) = (P \wedge Q) \wedge R, \text{ respectiv } P \vee (Q \vee R) = (P \vee Q) \vee R$$

Υ_3 – proprietatea lui ^ şi ∨, adică:

$$P \wedge T = P \text{ şi } P \wedge F = F, \text{ respectiv } P \vee T = T \text{ şi } P \vee F = P$$

Υ_4 – proprietatea negaţiei, adică: $P \wedge \sim P = F$ respectiv $P \vee \sim P = T$

Υ_5 – distributivitatea lui ^ faţă de ∨ şi a lui ∨ faţă de ^, adică:

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R),$$

respectiv $P \vee (Q \wedge R) =(P \vee Q) \wedge (P \vee R)$.

Axiomele - 5 legi de bază ale algebrei booleene.

Din cele de mai sus se pot deduce o serie de alte legi. Dintre legile cele mai importante deductibile amintim:

- legea dublei negaţii sau a complementului $\sim\sim P=P$;
- idempotenţa: $P \wedge P=P$, respectiv $P \vee P=P$;
- legea absorbţiei, $P \wedge (P \vee Q)=P$.

Pentru ilustrarea celor de mai sus vom demonstra legea absorbţiei [Gray85];

$P \wedge (P \vee Q)$ $= (P \vee F) \wedge (P \vee Q)$ {proprietatea operatorului ∨}

$= P \vee (F \wedge Q)$ {distributivitatea lui ∨ faţă de ∧}

$= P \vee (Q \wedge F)$ {comutativitatea}

$= P \vee F$ {proprietatea operatorului ∧}

$= P$ {proprietatea operatorului ∨}

Celelalte legi se demonstrează analog.

Legile booleene - caracter dual: *în orice teoremă din algebra booleană dacă se înlocuieşte ∨ cu ^ şi invers, teorema rămâne adevărată.*

Legile deductibile: ***legile lui DeMorgan:***

$$\sim(P \wedge Q) = \sim P \vee \sim Q, \text{ respective dualul, } \sim(P \vee Q) = \sim P \wedge \sim Q$$

Prin inducţie completă -> generalizarea legilor lui DeMorgan la n propoziţii:

$$\sim (P_1 \wedge P_2 \wedge \dots \wedge P_n) = \sim P_1 \vee \sim P_2 \vee \dots \vee \sim P_n, \text{ respectiv}$$

$$\sim (P_1 \vee P_2 \vee \dots \vee P_n) = \sim P_1 \wedge \sim P_2 \wedge \dots \wedge \sim P_n$$

-> ”***Pentru a nega o formulă, schimbă semnul şi operatorul cu complementarul***”. De exemplu:

$$\sim(P \vee \sim Q)=\sim P \wedge \sim(\sim Q)= \sim P \wedge Q.$$

Observaţie. Proprietăţile lui ∨ se aseamănă cu +, iar ale lui ^ cu ale operatorului de înmulţire *******. Proprietatea care le diferenţiază este însă distributivitatea; distributivităţii lui ^ faţă de ∨ îi

corespunde distributivitatea înmulţirii faţă de adunare A * (B + C) = A * B + A * C, invers nu este adevărat; adunarea nu este distributivă faţă de înmulţire, adică

$$(A * B) + C \neq (A + C) * (B + C)$$

în timp ce distributivitatea disjuncţiei $\vee$ faţă de conjuncţie $\wedge$ are loc.

→ 5 legi + legile lui DeMorgan, permit aducerea oricărei expresii din algebra booleană la una din următoarele forme:

- **forma normală conjunctivă** - forma $(C_1 \wedge C_2 \wedge C_3 \wedge \ldots)$ unde C_i se numeşte clauză; fiecare clauză - din propoziţii simple sau disjuncţii de propoziţii şi eventual negaţii ale acestora. Forma normală conjunctivă - teoria demonstraţiei, deoarece propoziţia scrisă sub formă clauzală este adevărată dacă şi numai dacă toate propoziţiile componente sunt adevărate şi invers.
- **forma normală disjunctivă** – expresia sub forma unor disjuncţii de expresii, expresiile din propoziţii simple sau conjuncţii de propoziţii şi eventual negaţii ale acestora. Această formă este utilă în teoria circuitelor.

Observaţii.

(i) Orice propoziţie logică poate fi adusă la una dintre formele normale, în următorii paşi:

1. se reduc funcţiile logice din expresie la cei trei conectori de bază, la implicaţii şi echivalenţe;
2. se înlocuiesc echivalenţele cu implicaţii duble;
3. se înlocuiesc implicaţiile cu forma lor disjunctivă;
4. se aplică legile lui De Morgan şi cele cinci legi fundamentale ale algebrei booleene.

(ii) Orice formulă se poate demonstra fie cu ajutorul tablei de adevăr, fie utilizând cele cinci legi fundamentale ale algebrei booleene. Utilizarea tablei de adevăr este mai sigură dar mai lungă, în timp ce utilizarea celor cinci legi are un caracter euristic mai pronunţat.

2.2.2.3. Regulile inferenţiale în logica propoziţiilor

Implicaţia - în ***deducţiile logice*** - din propoziţii adevărate → propoziţii adevărate. Acest mod de utilizare a mecanismului logicii diferă de cel al algebrei booleene, deoarece aceasta, ca orice algebră, permite determinarea valorii de adevăr a unor formule pe baza valorii de adevăr a componentelor, dar nu şi manipularea acestora indiferent de valoarea de adevăr a componentelor.

În deducţia logică - două ***reguli inferenţiale*** clasice, cunoscute de mult timp.

1. „***modus-ponens***” sau „***mod-pons***”, care se enunţă astfel:
   Fiind dat P adevărat P

din P→Q . notat şi cu P→Q
rezultă Q adevărat Q

2. ***regula de înlănţuire a inferenţelor*** sau ***închiderea tranzitivă a inferenţelor***. Ea permite ca pe baza a două implicaţii să se deducă o a treia. Astfel, această regulă se poate scrie:
dacă P→Q şi Q→R atunci P→R

Exemplu. Considerăm următoarele formule:
(1) (P→Q) →((P∨Q) →(R∨Q))
(2) (R∨Q) → (R∨S)
(3) P→Q
modus ponens la (3) şi (1), rezultă
(4) (P∨Q) →(R∨Q)
înlănţuirea inferenţelor (4) şi (2) rezultă
(5) (P∨Q) →(R∨S).

2.2.2.4. Strategii de demonstrare automată

Constituie o ramură importantă a inteligenţei artificiale. In demonstraţie, în general, trebuie arătat că o formulă B este „T" folosind pentru aceasta mecanismul modus-ponens, adică presupunând că A est "T" şi are loc implicaţia A→B, adică

$$\left(\frac{A, A-B}{B}\right).$$

Această regulă este dificil de aplicat direct şi din această cauză se utilizează diferite strategii de demonstraţie pe care se bazează demonstrarea automată.

A. Strategia demonstrării prin adoptarea unei premise/ipoteze/aserţiuni auxiliare sau suplimentare. Această strategie se bazează pe cele 2 teoreme ale lui Stoll (1961) :

1. Demonstrarea concluziei B din premisa A este echivalentă cu demonstrarea lui B fără a presupune premise speciale asupra lui A. Adică
$A \vdash B \leftrightarrow A \rightarrow B$ (dacă A atunci B ↔ A→B)
2. Dacă propoziţia B depinde individual de $A_1, A_2 \ldots A_n$ ea depinde şi de conjuncţia acestor premise şi invers.
$A_1, A2 \ldots An \vdash B \leftrightarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow B$

La cele două teoreme ale lui Stoll se mai pot adăuga o serie de tautologii auxiliare, mai importante:

$$(X \rightarrow (Y \rightarrow Z)) \leftrightarrow ((X \wedge Y) \rightarrow Z)$$

Demonstraţie. Prin transcrierea implicaţiei şi aplicarea asociativităţii şi a legii lui deMorgan

$$\sim X \vee (\sim Y \vee Z) \leftrightarrow (\sim X \vee \sim Y) \vee Z \leftrightarrow \sim (X \wedge Y) \vee Z$$

Demonstrarea prin adăugarea unei premise suplimentare constă în următoarele: *dacă prin adăugarea la ipotezele $A_1 \ldots A_n$ a unei premise auxiliare P, are loc concluzia Q, adică $(A_1 \ldots . A_n, P \rightarrow Q)$ atunci $A_1, A_2 \ldots A_n \rightarrow (P \rightarrow Q)$*

Demonstraţie : aplicăm T_2 a lui Stoll $\rightarrow A_1 \wedge A_2 \wedge \ldots . \wedge A_n \wedge P \rightarrow Q$

Aplicăm T_1 a lui Stoll $\rightarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow (P \rightarrow Q)$

Aplicăm T_2 a lui Stoll $\rightarrow \vdash A_1, \ldots A_n \rightarrow (P \rightarrow Q)$

Această metodă este foarte lungă şi este dificilă de aplicat → se pot folosi rar în cadrul demonstraţiei (exemplu : laturile egale ale triunghiului isoscel) – importanţa istorică.

B. Strategia reducerii la absurd – se bazează pe faptul că se adoptă ipotezele B şi ~C şi trebuie să ajungem la ~(B→C)

$$\sim(B \rightarrow C) = \sim(C \vee \sim B) = B \wedge \sim C$$

3 metode de bază în strategia reducerii la absurd :

i) se porneşte de la B şi se ajunge la C
ii) se porneşte de la ~C şi se demonstrează că are loc ~B ; B- premisă → contradicţie ;
iii) se porneşte de la B^ ~C şi se aplică regulile şi axiomele SF până se ajunge la o propoziţie p ^ ~p (se contrazic)

Observaţie. Această metodă se poate utiliza doar în cazurile în care implicaţia B→C este 'T'. Dacă B→C este 'F', metoda reducerii la absurd poate să ducă la raţionamente infinite. Exemplul tipic de astfel de raţionament - a 5-a axiomă a lui Euclid : printr-un punct exterior unei drepte se poate duce o singură paralelă la acea dreaptă. Această afirmaţie este adevărată doar în cadrul unui plan, într-un spaţiu multidimensional nu este adevărat şi deci demonstrarea pe baza axiomelor lui Euclid a dus la un raţionament infinit ce a generat geometriile neeuclidiene de tip Lobacevski-Bolyai sau Hilbert.

C. Strategia bazată pe rezoluţie/rezolvare – regula de rezoluţie – fundamentală în demonstrarea automată, regula de bază alături de modus-ponens şi înlănţuirea inferenţelor (le include pe ambele)

Legea rezoluţiei	**Înlănţuirea inferenţelor**	**Modus-ponens**
din $X \vee A$ şi $Y \vee \sim A$	din $\sim X \rightarrow A$ şi $A \rightarrow Y$	din A şi $A \rightarrow Y$
rezultă $X \vee Y$	rezultă $\sim X \rightarrow Y$	Rezultă Y

Regula rezolvării permite să combinăm 2 formule disjuncte în care apare atomul A şi negatul lui, eliminând atomul respectiv.

Modus-ponens – caz particular al regulii rezoluţiei pentru că de considerăm $X \in \Phi$ sau X = propoziţie falsă în ipoteza că are loc premiza A=.T. rezultă Y.

Demonstraţia legii rezoluţiei:

$$(X \vee A), (Y \vee \sim A) \vdash (X \vee Y)$$

T_2 Stoll & definiţia implicaţiei → $\sim((X \vee A) \wedge (Y \vee \sim A)) \vee (X \vee Y)$

Din regula lui DeMorgan şi dubla negaţie $(\sim X \wedge \sim A) \vee (\sim Y \wedge A)) \vee (X \vee Y)$ din asociativitatea lui $\vee$: $(X \vee (\sim X \wedge \sim A)) \vee (Y \vee (\sim Y \wedge A))$
din distributiv lui $\wedge$:

$$((X \vee \sim X) \wedge (X \vee A)) \vee (Y \vee \sim Y) \wedge (Y \vee A) =$$
$$(X \vee \sim A) \vee (Y \vee A) = (X \vee Y) \vee (A \vee \sim A) = T$$

Argumente pentru generalizarea demonstraţiei pe baza regulii rezoluţiei – automatismul şi simplitatea.
Paşi utilizaţi în demonstrarea prin metoda rezoluţiei:

P_1 : se presupune prin absurd că avem concluzia falsă ; se transformă echivalenţele în implicaţii duble şi se transcriu implicaţiile prin disjuncţii;
P_2 : pentru negarea din faţa parantezelor se aplică regula lui De Morgan;
P_3 : se aplică distributivitatea $\vee$ faţă de $\wedge$ şi invers.
P_4 : fiecare premisă se aduce la forma clauzală, adică la atomi, negaţi de atomi şi de clauze, adică atomi negaţi de atomi, legaţi prin disjuncţie.
P5: Se aplică principiul rezoluţiei până se ajunge la o propoziţie şi negatul acesteia, adică la o contradicţie.

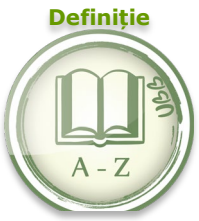
Exemplu. Să se demonstreze $[(P \vee Q) \wedge (P \rightarrow R) \wedge (Q \rightarrow S) \vdash R \vee S]$
Aceasta este echivalentă cu: $P \vee Q, P \rightarrow R, Q \rightarrow S \vdash R \vee S$ – foloseşte metode bazate pe rezoluţie pentru demonstraţie:
P1. Se consideră ipotezele, negatul concluziei şi definiţia implicaţiei se obţine:

(1) $P \vee Q$
(2) $\sim P \vee R$
(3) $\sim Q \vee S$
din $\sim(R \vee S) = \sim R \wedge \sim S \rightarrow$ (4)$\sim R$, şi (5)$\sim S$ – sunt .T.

din (4), (2) & perinc. rezoluţiei → (6) $(\sim P \vee R)$, $(\sim R) \rightarrow \sim P$
din (6), (1) → (7) $(P \vee Q)$, $(\sim P) =$ ..... → Q
(3),(5) → (8) $(\sim Q \vee S)$, $(\sim S) =$ ..... → ~Q
} Contradiţie

→ cond.de la care am plecat e falsă, rezultă formula iniţială este „T”.

FORME NORMALE ŞI CONSECINŢE LOGICE



Logica matematică = logica simbolică în care afirmaţiile (pe care le vom numi propoziţii) sunt notate cu literele alfabetului.

Logica propoziţiilor = un limbaj formal care conţine un alfabet, reguli de sintaxă, axiome şi o regulă de deducţie. (studiază legăturile dintre propoziţii. **(!"Această propoziţie este falsă"?)**).

Propoziţii = *atom* = afirmaţii cărora li se ataşează o valoare de adevăr.

Logica simbolică = utilizează simboluri pentru reprezentarea elementelor universului problemei şi a operaţiilor asupra acestora.

Echivalenţa		
p	q	p↔q
1	1	1
1	0	0
0	1	0
0	0	1

Operaţii logice:

Negaţia	
p	~p
1	0
0	1

Disjuncţia		
p	q	pvq
1	1	1
1	0	1
0	1	1
0	0	0

Disjuncţia exclusivă		
p	q	pvq
1	1	0
1	0	1
0	1	1
0	0	0

Conjuncţia		
p	q	P^q
1	1	1
1	0	0
0	1	0
0	0	0

Implicaţia		
p	q	p→q
1	1	1
1	0	0
0	1	1
0	0	1

ordinea efectuării operaţiilor logice: **~, ^, v, \/ , ->, <->**

Calcul propoziţional:

atom = propoziţie cu valoare de adevăr
formulă = propoziţie compusă

Formulă bine formată (sau **bine formulată - wff**): se construieşte după următoarele reguli:

R1. Un atom este o formulă.

R2. Dacă F este o formulă, atunci ~F este formulă bine formată.

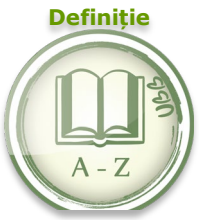
R3. Dacă F şi G sunt formule, atunci : FvG, F^G, F->G, F<->G sunt bine formate.

R4. Orice f.b.f. se obţine prin aplicarea regulilor R1,R2,R3.

O formulă bine formată poate fi:

<table>
<tr><td>VALIDĂ</td><td colspan="2">INVALIDĂ</td></tr>
<tr><td rowspan="2">Totdeauna adevărată.</td><td>Nu totdeauna adevărată.</td><td rowspan="2">Totdeauna falsă.</td></tr>
<tr><td>Nu totdeauna falsă.</td></tr>
<tr><td colspan="2">CONSISTENTĂ</td><td>INCONSISTENTĂ</td></tr>
</table>

FORME NORMALE



Echivalenţa (se notează F=G): F şi G sunt echivalente atunci când F şi G au aceeaşi valoare pentru orice interpretare I.
Reguli de echivalenţă în utilizarea lui .T. şi .F.:

P v .F. = P P v .T. = .T.
P ^ .T. = P P ^ .F. = .F.

Forma normală conjunctivă (FNC): Dacă $F_1,F_2,..,F_n$ sunt formule bine formate care conţin doar literale (atomi sau negaţie de atomi) şi sunt de forma $L_1vL_2v...vL_m$ (disjuncţie de literale), atunci $F_1^F_2^...^F_n$ este o **fnc**.

Forma normală disjunctivă (FND): Dacă $F_1,..,F_n$ sunt formule bine formate care conţin doar literale (atomi sau negaţii de atomi) şi sunt de forma $L_1^L_2^..^L_m$ (conjuncţie de literale) atunci $F_1vF_2v...vF_n$ este o **fnd**.

Exemple: **- fnc: (PvQv~R) ^ (Pv~QvR) ^ (~PvQvR)**

- fnd: (P ^ ~Q) v (~P ^ ~Q) v (P ^ Q).

Algoritm de transformare a unei formule bine formate într-o *formă normală*:

Pas 1: Se elimină echivalenţa şi implicaţia:

F<->G = (F->G) ^ (G->F)
F ->G = ~F v G

Pas 2: Se elimină dubla negaţie şi se distribuie negaţia (Legile lui de Morgan):

~(~F)=F
~(FvG)=~F ^ ~G
~(F^G)=~F v ~G

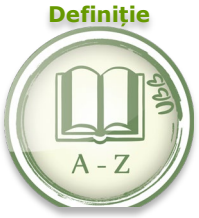
Pas 3: Se aplică distributivitatea pentru separarea operaţiilor ^,v:

Fv(G^H)=(FvG)^(FvH)
F^(GvH)=(F^G)v(F^H)

Exemple:

a) **fnd**: ((P->Q)->(R->S))^(Q->~(P^R))...
...(P^~Q)v(P^~Q^~R)v(Q^~P)v(Q^~P^~R)v(~R^~P)v(~R^~P)v~Rv(S^~Q)v
v(S^~P)v(S^~R)
b) **fnc**: P^(Q->R)->S ... (Sv~PvQ)^(Sv~Pv~R)

Consecinţe logice:



Dacă $F_1,..,F_n,G$ sunt f.b.f., şi *dacă* pentru orice interpretare I pentru care $F_1^F_2^...^F_n$ este adevărată, atunci G este adevărată atunci se spune că G este o **consecinţă logică** a lui $F_1,...,F_n$, iar $F_1,...,F_n$ sunt **axiome** pentru G.

Teorema1: G este o consecinţă logică a lui $F_1,..,F_n$ dacă şi *numai dacă* formula $F_1^...^F_n$->G este validă.

Teorema2: G este o consecinţă logică a lui $F_1,..,F_n$ dacă şi *numai dacă* formula $F_1^...^F_n$ ^ ~G este inconsistentă.

Problema1:
Fie F_1: P->Q
F_2: ~Q
G: ~P

Să se demonstreze că: G este o consecinţă logică a lui F_1 şi F_2.
Indicaţie: Din T1 => $F_1^F_2$->G - validă sau din T2 => $F_1^F_2^$~G - inconsistentă.

Problema2:
Se dau:
P = Parlamentul refuză să acţioneze.
Q = Greva s-a terminat.
R = Directorul firmei demisionează.
S = Greva continuă de mai mult de un an.

F_1 : Dacă parlamentul refuză să acţioneze atunci directorul firmai demisionează şi greva continuă mai mult de un an.
F_2 : Parlamentul refuză să acţioneze.
F_3 : Greva nu s-a terminat.

Se cere:

i) să se transpună în formule F_1, F_2 şi F_3;
ii) să se demonstreze că F_3 este o consecinţă logică a lui F_1 ^ F_2 ^ F_3.

Tema 1

Să se construiască o problemă în stilul problemei 2, să se transpună apoi în formule şi să se rezolve.

Ex de rezv cu tabele. Tema voastra este sa demonstrati cu ajutorul teoremelor.

Se cunosc:

N = se formează numărul;
T = telefonul sună;
L = linie ocupată;
A = abonatul răspunde;
M = linie ocupată pe timpul convorbirii.

F_1 : Se formează numărul, telefonul sună sau linie ocupată.
F_2 : Telefonul sună, abonatul răspunde, linia este ocupată pe timpul convorbirii.
F_3 : Dacă linia nu e ocupată telefonul sună.

				F_1		F_2		F_3		
T	**L**	**A**	**M**	**N**	**N→T**	**N→T∨L**	**T→A**	**T→A→M**	**~L**	**~L→T**
1	0	1	1	0	1	1	1	1	1	1
1	1	0	0	1	1	1	0	1	0	1
0	0	1	1	1	0	0	1	1	1	0
0	1	0	0	0	1	1	1	0	0	1

2.3. Logica predicatelor de ordinul I (FOPL) și aplicațiile ei în Informatica Economică și de Afaceri

Introducere

Logica Propoziţiilor (PL) - săracă să reprezinte lumea înconjurătoare - clase similare, neputând aplica propoziţiile - reduce modelarea lumii reale.

Exemplu, propoziţia:

Ion este student la IE

Adăugăm: Orice student la IE → Student învaţă C#
nu putem pe baza PL să tragem concluzia „Ion învaţă C#”, deoarece nu sunt satisfăcute condiţiile din modus-ponens.

Pe de altă parte:
Ion este student la IE

poate fi transcris sub forma:

Student_informatică_economică (ion),

„ion” este o constantă. Tot aşa de bine putem scrie şi

Student_informatică_economică (vasile).

Rezultă deci că putem scrie generic:

Student_informatică_economică (x)

unde x = ion sau x = vasile etc. → x mulţimea oamenilor şi predicatul ia valoarea T sau F, după cum persoana în cauză este sau nu student la IE.

→ propoziţie se înlocuieşte cu cea de predicat.

Semnificaţii[8]:

- matematică, predicatul = relaţie.
- gramatică - o parte a propoziţiei.
- programare, - un operator sau o funcţie, care returnează o valoare booleană adică T sau F.
- teoria lui Bernard Rusell a tipurilor, este un act de stabilirea unui tip.

A treia definiţie este cea mai adecvată – în accepţiunea noastră.

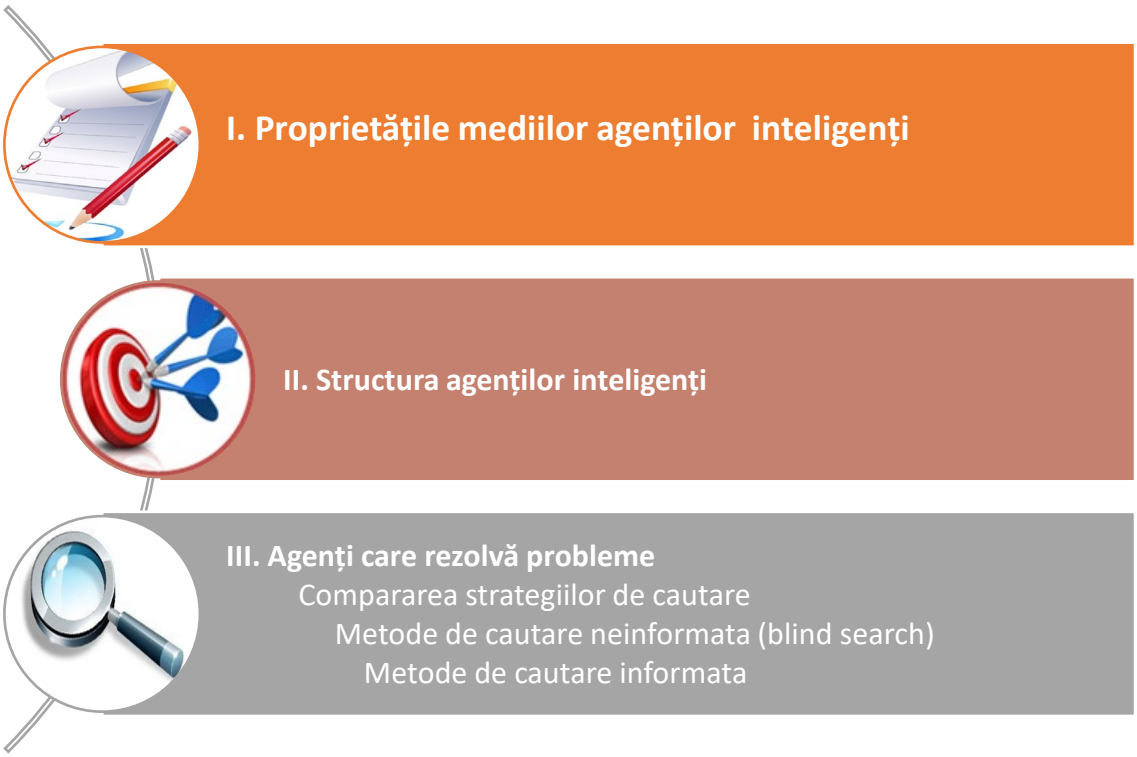
Studiul predicatelor se utilizează logica predicatelor, în particular Logica Predicatelor de Ordinul I -i First Order Predicat Logic (FOPL) - limbaj al teoriei logicii unde, spre deosebire de PL, pot apare sintaxa şi semantica.

[8] http://en.wikipedia.org/wiki/Predicate - From Wikipedia - free encyclopedia

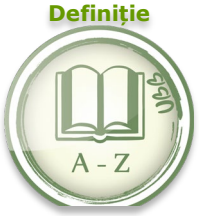
Disciplină: Inteligență Artificială

Modulul III.

Agenți Inteligenţi



Un **agent** este o entitate care percepe mediul înconjurător prin senzori și acționează asupra lui prin efecotori. Un agent uman are ochi, urechi și alte organe ca senzori și mâini, picioare etc. ca și efectori. Un agent robotic poate avea camere sau dispozitive cu infraroșu pe post de senzori și diferite motoare pe post de efectori.



Folosim termenul **"percepție"** pentru a ne referi la inputul perceptual pe care îl are agentul la un moment dat. O secvență de percepții a unui agent reprezintă o istorie completă a tot ce a perceput agentul. Dacă specificăm acțiunile pe care poate să le facă agentul pentru fiecare secvență de percepții posibilă, precizăm, într-o mare măsură, tot ce este de zis despre agent. Din punct de vedere matematic, spunem că, comportamentul agentului este descris de către funcția agent care mapează fiecare secvență de percepții pe o acțiune. Putem să ne imaginăm funcția agent ca și un tabel. Tabelul este o caracterizare externă a agentului. Din punct de vedere intern, funcția agent pentru un agent artificial va fi implementată de un program agent.

Un **agent rațional** este acela care face lucrul potrivit -adică fiecare intrare în tabelul cu funcțiile agentului este completată în mod corect. Ce înseamnă să faci lucrul bun/corect? Considerăm consecințele comportamentului agentului. Când un agent este într-un mediu, generează o secvență de acțiuni în funcție de perceptorii pe care îi primește. Secvența aceasta determină mediul să treacă printr-o secvență de stări. Dacă secvența este cea dorită, atunci agentul a efectuat în mod corect ce avea de făcut. Noțiunea de "dorită" este dată de o masură a performanței care evaluează fiecare secvență dată de stări ale mediului.

Nu există o masură a performanței fixă pentru toți agenții și activitățile. În mod normal, un designer va decide o masura potrivită în funcție de circumstanțe. Când vine vorba de un agent rațional, ceea ce ceri este ceea ce primesti. Este mai bine să decizi o masură de performanță în funcție de ceea ce vrea cineva în mediu, decât în funcție de cum crede cineva că trebuie să se comporte agentul.

Ceea ce este rațional la un moment dat, depinde de **4 lucruri**:

- Măsura de performanță care definește gradul de succes
- Secvența de percepții
- Cunoștințele pe care le-a dobândit agentul despre mediu până în momentul de față
- Acțiunile pe care agentul le poate face

Definiția unui agent rațional :

Pentru fiecare secvență de percepții posibilă, un agent rațional ar trebui să selecteze o acțiune care se așteaptă să maximizeze măsura sa de performanță, având în vedere informațiile pe care i le dă secvența de percepții și orice alte cunoștințe pe care le are.

Omniscientă vs raționalitate

Un agent atotștiutor cunoaște foarte bine ceea ce ar trebui sa facă și acționează asemenea. Dar omniscienta este imposibilă în realitate. Spre exemplu, vreau să traversez strada, mă asigur, mă uit în stânga și în dreapta și traversez. În timp ce traversez, sunt lovit de către un meteorit. Pot spune că am acționat irațional când am traversat strada ?

Raționalitatea nu este același lucru cu perfecțiunea. Raționalitatea maximizează performanța care se așteaptă, în timp ce perfecțiunea maximizează performanța actuala.

Definiția rațonalității nu implică omniscienta pentru că o decizie rațională depinde doar de secvența de perceptii de până în prezent. Spre exemplu, dacă un agent nu se uită în ambele părți când vrea să traverseze, atunci secvența de percepții nu o să-i spună că este un camion care se apropie cu viteză mare, pentru că nu are de unde să știe. Din punct de vedere rațional este bine să treci strada în cazul de față ? Nu . În primul rând, nu este rațional pentru că eu știu din secvența de percepții că riscul de accident în cazul în care treci strada fără să te asiguri este mare. În al doilea rand, un agent rațional ar trebui să aleagă acțiunea de a se uita în stânga și în dreapta înainte să facă pasul pe stradă, pentru că asigurarea ajută maximizarea performanței așteptate. A face acțiuni pentru a modifica percepții viitoare este o parte importantă a raționalității și se numește adunarea de informații. O altă metodă de a aduna informații este explorarea.

Raționalitatea cere nu numai adunarea de informații, ci și învățarea pe cât de mult din ceea ce percepe agentul. Configurarea inițială a agentului ar putea reflecta niște cunoștințe anterioare despre eveniment, dar , pe parcurs ce agentul capătă experiență, asta s-ar putea modifica. Sunt putine cazuri în care evenimentul este cunoscut în totalitate dinainte.

În măsura în care un agent se bazează pe cunoștința anterioară a designerului său decât pe propriile percepții, putem spune că agentul duce lipsă de autonomie. Un agent rațional ar trebui să fie autonom, ar trebui să învețe cum să compenseze cunoștințele incorecte sau incomplete de până atunci. Spre exemplu, un aspirator care invață cum să anticipeze unde și când va apărea mai multă mizerie, se va

descurca mai bine decât unul care nu va face asta . Bineînțeles, ar trebui ca desinger-ul să îi asigure agentului niște cunoștințe inițiale precum și capacitatea de a învăța. După o experiență suficientă în mediul său înconjurător, comportamentul unui agent rațional poate deveni independent de cunoștințele anterioare. Așadar, dacă îi este oferită oportunitatea de a învăța, un singur agent poate avea succes într-o varietate de medii.

Dupa ce am înțeles ce înseamnă raționalitatea, putem să ne gandim să construim un agent rational. Primul aspect care trebuie luat în considerare este mediul, care imi definește, în esență, problemele pentru care agenții rationali sunt soluțiile.

Cand am vorbit despre aspirator, a trebuit să specificăm măsura de performantă, mediul, senzorii și efectorii săi . Toate aceste lucruri adunate poartă denumirea de mediu de activitate sau descriere PEAS. Prima dată când concepem un agent, trebuie să specificăm cât mai concret mediul de activitate.

Să luăm exemplul unui șofer de taxi automat. Trebuie specifcat că acest caz este într-un fel peste capabilitățile tehnologiilor existente. Prima dată, vom preciza care sunt măsurile de peformanță pe care dorim să le aibă șoferul nostru automat: să ne ducă la destinația corectă , să minimizeze consumul de combustibil, timpul călătoriei și costul, încălcarea legilor, să maximizeze siguranța pasagerilor și confortul, profitul. Următorul pas este reprezentat de mediu: drumuri, trafic, pietoni, clienți, mașini, animale etc. Vremea poate face parte, de asemenea, din mediu.

Efectorii unui șofer automat coincid în mare cu cei ai unui șofer uman. Controlul asupra accelerației, ambreiajului, franie. Pe lângă acestea, mai avem nevoie și de un ecran prin intremediul căruia să comunice cu pasagerii sau alte vehicule.

Senzorii vor include una sau mai multe camere video cu ajutorul cărora să poată vedea drumul. Mai pot avea nevoie de senzori de sunet sau infrarosu ca să detecteze distanța față de alte mașini. De asemenea, ca să evite amenzile de viteză, ar trebui să aibă un vitezometru și accelerometru. Evident, un motor, combustibil. Totodata, un GPS ca să nu se piardă și o tastatură sau un microfon ca pasagerii să introducă o destinație.

3.1 Proprietăți:

În ceea ce privește intelegența artificială , putem vorbi despre un număr mare de medii care pot să apară. Pentru acestea se poate determina un număr relativ minim de dimensiuni după care se poate face o categorizare a acestora.

Aceste dimensiuni determina designul potrivit și aplicabilitatea fiecărei grupe de tehnici pentru implementare.

Exemplu:

Tip	Perfomanță	Mediu	Actori	Senzori
Sistem de diagnosticare medical	Pacienți sănătoși, Costuri reduse	Pacient, spital, staff	Intrebări , teste, diagnostice	Tastatură, răspunsurile pacienților

3.1.1 Complet observabil vs parțial observabil

Un mediu este complet observabil dacă senzorii agentului detecteaza toate aspectele mediului care sunt relvante și determină alegerea acțiunii. Practic senzorii au acces la întregul univers în fiecare moment de timp. Alegerea acțiunii este un aspect care este influențat de relevanța acesteia și de asemenea depinde de performanță. Un astfel de mediu este necesar pentru ca agenții nu trebuie să mențină starea internă pentru a ține pasul cu lumea.

Un **mediu este considerat parțial observabil** datorită inacurateții senzoriilor sau deoarece lipsesc părți din datele senzoriilor. De asemenea mai apare situația in care lipsesc senzorii și atunci mediul este considerat **neobservabil**

Exemple:

- Aspiratorul cu locație nu poate sa prezică dacă exista mizerie și în alte locuri.
- Șoferul de taxi automat nu poate să determine ce gândesc ceilalți șoferi.

3.1.2 Deterministic vs stochastic

Un mediu este considerat **deterministic** atunci când următoarea stare a mediului este determininată de starea curentă și de acțiunile selectate de agenți.Daca un mediu este complet observabil și deterministic rezultatul este inexitența incertitudinii.

Stochastic – este un termen folosit în inteligența artificiala atunci cand vorbim despre programe care rezolva problemele cu probabilitați .

Un mediu **stochastic** poate sa fie un mediu parțial observabil. De exemplu sofer de taxi automat care nu poate sa prezică exact cum o să fie traficul pe parcursul cursei. Pentru situațiile complexe este convenabil să fie considerate stochastice.

De asemenea mai apar încă doua notații:

-**incert** : mediu parțial observabil sau nedeterministic

-**nondeterministic** –acțiunile sunt caracterizate dupa rezultatele lor posibile, dar nicio probabilitate nu au atașată

3.1.3 Episodic vs secvențial

În ceea ce priveste un **mediu episodic**, întreaga experiență a agentului este împărțită in episoade atomice. În fiecare episod agentul primeste o percepție (input) și realizează o singură acțiune . În acest caz este crucial ca următorul episod sa nu depindă de cel anterior. Aceste medii sunt mai preferabile deoarece agenții nu trebuie să se gândeasca inainte la ce urmează.

În cadrul unui **mediu secvențial** , decizia curentă afecteaza toate deciziile viitoare. Se merge pe ideea " acțiunile pe termen scurt pot avea consecințe pe termen lung". Atât in ceea ce priveste exemplu cu soferul de taxi automat cât și un simplu joc de sah , putem vorbi despre medii secventiale, deoarece orice decizie curenta va afecta deciiziile viitoare.

3.1.4 Static vs dinamic

Ne putem referii la un mediu ca fiind **dinamic** dacă mediul se poate schimba în timp ce agentul hotăraste ce actiune va face in continuare, iar in caz contrat mediul este **static.** Acesta din urmă este mai preferabil deoarece agenții nu trebuie să tot fie atenti la mediu sau să își facă griji în privinta timpului.

Un mediu dinamic intreabă constant agentii ce vor face in continuare, dacă acestia nu actioneaza se presupune ca nu vor face nimic.Aici apare notiunea de mediu semidinamic atunci cand mediul nu se schimba însă performanta agentului se schimbă.

Exemple:

- Sofer de taxi- mediu dinamic
- Șah (cu ceas) –mediu semidinamic
- Rebus – mediu static

3.1.5 Discret vs continuu

Aceasta distincție se aplică stării mediilor, în felul in care timpul este administrat și pe baza la imputurile si actiunile agentiilor. Dacă timpul, numărul de percepții sau acțiuni diferite sunt finite atunci mediul este **discret**. Spre exemplu , șahul are un număr finit de stări diferite si un set discret de perceptii si actiuni.

Sper deosebire de jocul de șah, în cazul șoferul de taxi automat putem vorbi despre o stare **continuă** la care se adaugă și o problemă constantă cu timpul.

3.1.6 Cunoscut vs necunoscut

Această proprietate se referă la starea de cunoștiințe ale agentiilor despre legile fizici ale mediilor:

- **Mediu cunoscut** – rezultatele pe toate acțiunile sunt date
- **Mediu necunocut**- agentii trebuie să învețe cum funcționează pentru a lua decizii bune.

Trebuie scoasă în evidentă diferența dintre un mediu cunoscut- necusoscut și un mediu complet-partial observabil.

- Un mediu cunoscut poate fi parțial observabil
  - Solitaire- cunoscutul joc de carți pentru ca știm regulile, însă nu putem stii ce carte o sa vină in continuare.
- Un mediu necunoscut poate fi complet observabil
  - Joc video- putem vedea întreaga suprafață de joc, însă nu știm ce funcționalitati au toate butoanele.

3.1.7 Cu un singur agent vs multiagent

Fiecare mediu poate să aibă mai unul sau mai mulți agenti. De exemplu un rebus sau o integramă reprezinta un mediu cu un singur agent , pe când un joc șah impune din strat 2 agenți.

Aici apare noțiunea de entitate, care poate fi vazută ca și un agent. Însă problema se pune în felul urmator : cum poate să fie văzută o entitate ca agent?Dacă avem doi agenti A și B , A poate sa vadă agentul B fie ca agent fie ca un obiect care acționează dupa regulile fizicii.

Cheia constă in felul in care comportamentul lui B este cel mai bine descris ca maximizare a masuratorii performanței lui A. De exemplu, în cadrul jocul de șah dacă B își maximizeaza performantele are ca și consecință o minimizare a performanței lui A.

În cadrul mediilor multiagent apar o nouă clasificare:

- Medii competitive – jocul de șah
  - Putem vorbi aici de *Comportament random*- pentru evitarea predictibilitatii in mediile competive
- Medii cooperative – evitarea unui accident in trafic , creste performanta tuturor soferilor.

Exercițiu :

Medii	Observabil	Deterministic	Episodic	Static	Discret
Șah cu ceas	Da	Da	Nu	Semi	Da
Șah fară ceas	Da	Da	Nu	Da	Da
Poker	Nu	Nu	Nu	Da	Da
Solitaire	Nu	Da	Nu	Da	Da

3.2 Structura agenților

Sarcina AI este de a construi programe agent care implementează funcţia agentului, practic toate percepţiile sunt mapate în acţiuni. Programele funcţionează pe anumite dispozitive cu senzori și efectori fizici. Aceasta reprezintă arhitectura.
Toate programele agent au acelaşi schelet: acceptă percepţiile curente de la senzori ca şi input şi returnează o acţiune. Trebuie menţionat faptul că există o diferenţă între programale agent şi funcţiile agent: programele agent iau ca şi input percepţia curentă, în timp ce funcţiile agent iau ca şi input întreaga istorie de percepţii. Dacă acţiunea agentului depinde de o secvenţă de percepţii atunci acesta trebuie să îşi amintească întreaga secvenţă de percepţii.

În continuare sunt prezentate 4 tipuri de bază de programe agent :

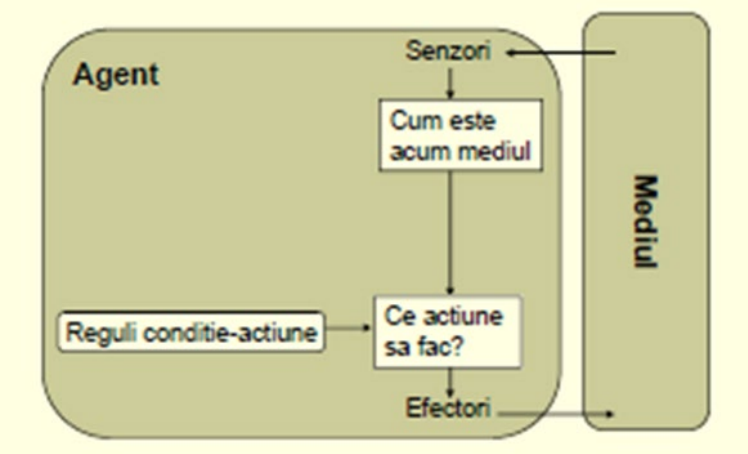
3.2.1 Agentul reflex simplu

Aceşti agenţi selectează acţiunea pe care o returnează, bazându-se doar pe percepţia curentă, ignorând istoria de percepţii.
În ceea ce priveşte exemplul cu aspiratorul este vorba despre un agent reflex simplu, deoarece decizia se bazează doar pe locaţia curentă şi pe faptul dacă aceasta conţine sau nu mizerie.

Un alt exemplu relevant are în vedere un taxi automat. Dacă maşina din faţă frânează , şi lumina de la frână se aprinde, agentul observă si iniţiază frânarea. Practic, se realizează o procesare pe baza inputului vizual pentru a stabili condiţia pe baza căreia se stabileşte condiţia : ”maşina din faţă frânează”. Declanșatoarele stabilesc o conexiune la nivelul agentului program pentru acţiunea de

frânare. Astfel regula de acţiune pe baza condiţiei este: Dacă maşina din faţă frânează atunci iniţializez frânarea.

Agentul percepe starea curentă a mediului prin intermediul senzorilor, iar pe baza unei reguli a cărei condiţie se potriveşte cu starea curentă, selectează acţiunea pe care o realizează prin intermediul efectorilor. Sunt simpli şi cu inteligenţă limitată.

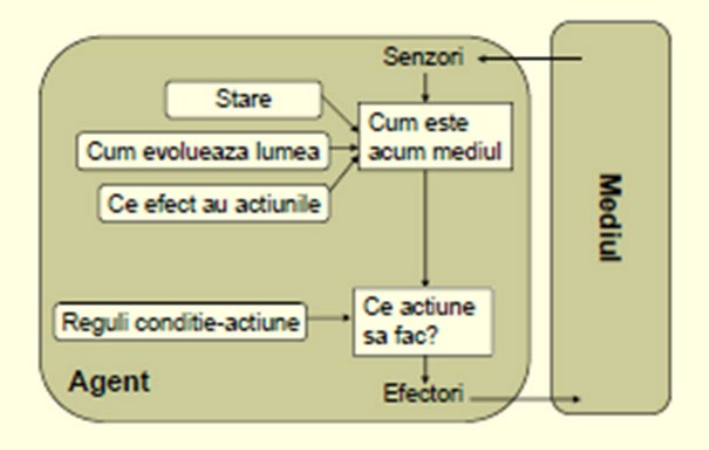


3.2.2 Agent reflex cu stare internă

Cel mai eficient mod al unui agent de a rezolva problemele referitoare la observabilitatea parţială este de a păstra urma părţii mediului pe care nu o vede. Astfel agentul menţine o stare internă care depinde de istoria de percepţii, şi care reflectă unele aspecte neobservate ale stării curente. Referitor la exemplu anterior,cu taxi-ul automat, în cazul în care se doreşte schimbarea benzii , trebuie păstrată urma celorlalte maşini, în cazul în care nu are o vedere de ansamblu asupra celorlalte maşini.

Actualizarea stării interne se realizează ţinând cont atât de modul în care mediul evoluează independent de agent, de exemplu dacă o maşină în depăşire e mai aproape decât a fost cu un moment în urmă , cât şi de informaţia privind modul în care acţiunea agentului afectează mediul: de exemplu ,o maşină conduce 5 km spre nord, ea va fi cu 5 km mai în nord decât a fost în urma cu câteva minute.

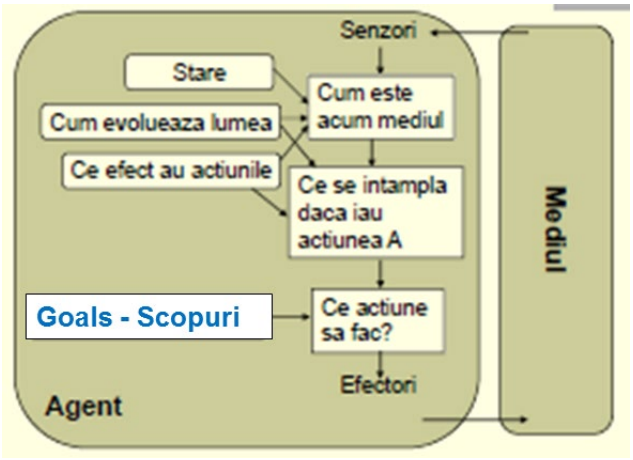
În cazul de faţă , se observă modul în care percepţia curentă este combinată cu starea internă veche pentru a genera o descriere actualizată a stării curente, bazându-se pe modul în care evoluează mediul. Cu toate acestea sunt foarte rare situaţiile în care agentul determină starea curentă a unui mediu parţial observabil, cu exactitate . De exemplu : în cazul taxi-ului, dacă are în faţă un tir, şi acesta opreşte , nu ştie motivul pentru care opreşte. Ştie doar că opreşte şi trebuie să reacţioneze, trebuie să ia o decizie.



3.2.3Agent cu scop exact

În anumite situaţii, nu este suficient să fie cunoscută doar starea curentă a mediului pentru a decide ce trebuie făcut. De exemplu, în cazul unei intersecţii, taxi-ul poate să meargă la stânga, la dreapta sau în faţă. Decizia corectă depinde de destinaţia taxi-ului. Astfel pe lângă descrierea stării curente, agentul, are nevoie de informaţie în ceea ce priveşte scopul care să descrie situaţia care este dorită să se întâmple: destinaţia pasagerului.

Este păstrată atât urma stării mediului, cât şi a unui set de scopuri, pe care încearcă să le atingă. Când o acţiune este aleasă, decizia este luată astfel încât să conducă în cele din urmă la realizarea scopurilor. Deşi pare mai puţin eficientă, furnizează mai multă flexibilitate, deoarece cunoaşterea pe baza căreia se ia o decizie este reprezentată cu exactitate şi poate fii modificată. De exemplu, dacă începe să plouă, agentul îşi actualizează cunoaşterea referitor la cât de eficient vor funcţiona frânele. Astfel toate comportamentele relevante se adaptează noilor condiţii. În cazul unui agent reflex, trebuie rescrise regulile condiţiilor de acţiune.



3.2.4 Agent bazat pe funcţionalitate

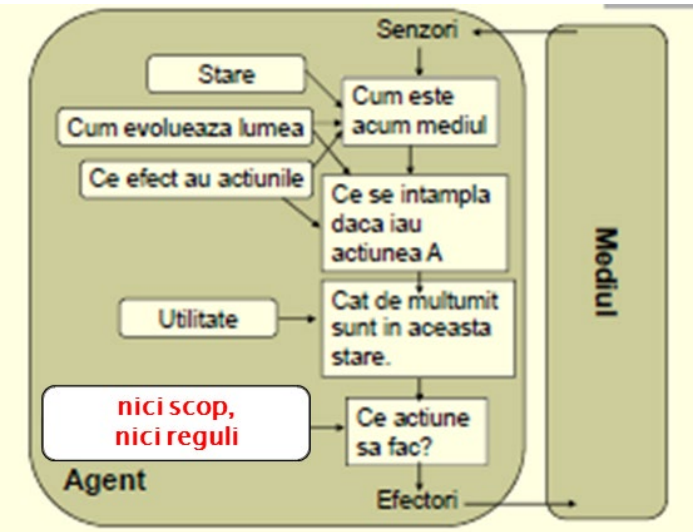
În cele mai multe medii , doar scopurile nu sunt suficiente pentru a genera comportamente de înaltă calitate. De exemplu: mai multe secvenţe de acţiuni vor duce taxi-ul la destinaţie (scopul este atins) , dar nu toate sunt la fel de rapide, sigure, mai rentabile, mai ieftine decât celelalte).

Scopul are rolul doar de a face distincţia dintre o stare fericită şi o stare nefericită.

O măsură de măsurare a performanţei mai generală trebuie să permită să stabilim exact cât de fericit va face agentul, în urma acţiunilor alese .

Un agent bazat de funcţionalitate, foloseşte un model al mediului împreună cu o funcţie a utilităţii care îi masoară preferinţa în ceea ce priveşte stările mediului. Ulterior este aleasă acţiunea care conduce la cea mai bună utilitate asteptată, unde utilitatea aşteptată este calculată.
Funcţia de utilitate a unui agent este o internalizare a măsurii de performanţă ; dacă funcţia de utilitate internă este în concordanţă cu măsura de performanţă externă , atunci agentul care alege să acţioneze astfel încât să îşi maximizeze utilitatea va fi raţional conform măsurii de performanţă externă. Asemenea agenţilor cu un scop exact , există avantaje în ceea ce priveşte flexibilitatea şi învăţarea.

Exista situaţii în care scopurile sunt inadecvate , dar un agent bazat pe funcţionalitate poate să ia decizii raţionale: scopurile aflate în conflict (viteza si siguranţa) , sau în cazul în care există mai multe scopuri cerute , dar niciunul nu poate fi atins cu certitudine.



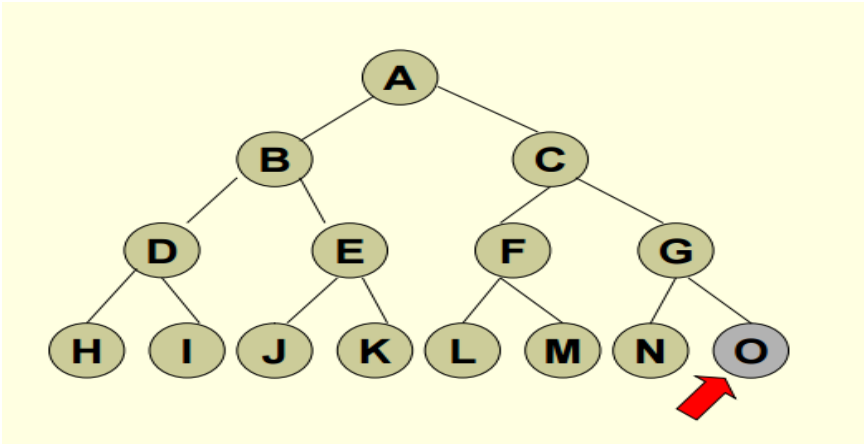
3. Agenți care rezolvă probleme

3.1 Metode de căutare neinformată

3.1.1 Căutarea în lățime

Cea mai simplă strategie de căutare este căutarea in lăţime, numită şi breadthfirst search. Această strategie explorează toate nodurile în ordinea nivelelor, altfel spus nodurile de pe nivelul d, sunt explorate înaintea nodurilor de pe nivelul d+1 de aceea sunt considerate toate drumurile de lungime 1, apoi toate de lungime 2 etc..

Graful din figură pleacă de la nodul start A. Parcurgerea in laţime exploreaza in mod sistematic toate arcele din graful de fată ca să descopere fiecare nod care poate fi identificat pornind de la nodul de start. Prima dată se calculează distanţa de la nodul A la fiecare nod din graf care este conectat la A. Deci algoritmul explorează o frontieră dintre nodurile descoperite si cele nedescoperite in laţime , adică descoperă toate nodurile de la distanţa d, apoi pe cele de la distanţa d+1 etc.



PSEUDOCOD:

Avem o funcţie de căutare în lăţime care primește parametrul problema, întoarce o soluţie sau un eșec. Mai întâi se generează o frontieră care primește toate nodurile nedescoperite și o listă care este o coadă FIFO și un set care o să fie gol la început cu lista nodurilor descoperite.
Cât timp nu este o soluţie se verifică dacă lista nodurilor nedescoperite e goală si dacă e goală întoarce un eșec că nu are cum să fie goală la început. Pe urmă îmi testează fiecare nod, dacă e ok și poate fi descoperit atunci nodul e descoperit îl întoarce și altfel se adaugă în lista de noduri descoperite.

```
functia cautare_latime(problema) intoarce solutie sau esec
noduri = genereaza_lista(genereaza_nod(stare_initiala[problema]))

Cat timp solutie negasita si noduri ≠ ∅ executa
Daca noduri = vida atunci
                                    intoarce esec
nod = scoate_din_fata(noduri)
Daca testare_tinta[problema] se aplica la stare(nod) atunci
intoarce nod
Altfel
  noduri = adauga(noduri, expandare(nod, adauga_la_sfarsit))
  Sfarsit cat timp
```

CRITERII:

Algoritmul de căutare în lățime trebuie să satisfacă mai multe condiții:

- **Completitudine**(Garantează strategia găsirea unei soluții atunci când există una?)- strategia de căutare în lăţime este completă atunci când numărul de operatori este finit deoarece parcurge pe rând toate nodurile arborelui de căutare, altfel spus dacă soluţia există căutarea în lătime o găseşte.
- **Optimalitate**(Găsește strategia soluția cea mai bună calitativ când există mai multe soluții diferite?) doar dacă costul crește proporțional cu adâncimea. Deci fiecare operațiune trebuie să aibă același cost.

La un arbore fiecare nivel are b succesori, nodul rădacina genereaza b noduri la primul nivel iar fiecare nod generează alte b noduri, in total avem b l la a2 noduri pe al doilea nivel, astfel b la a 3a pe al treilea nivel s.a.m.d , deci **complexitatea în timp,** un alt criteriu(Cat dureaza gasirea unei solutii?) va fi simplu calculată după formula $1 + b + b^2 + b^3 + \ldots + b^d$, unde d este adâncimea primei soluţii a problemei(depth) iar b este factorul de ramificaţie a problemei , numărul de noduri în care se expandează fiecare nod(branching factor) .

Exemplu: Dacă b=10 se procesează 1000 de noduri pe secundă şi fiecare nod necesită 100 byte de memorie.

- **Complexitatea de spaţiu**(De câtă memorie este nevoie pentru a face căutarea?) este $O(b^d)$ deoarece algoritmul reţine tot timpul doar nodurile de pe ultimul nivel al arborelui de căutare.

Adancime	Noduri	Timp	Memorie
0	1	1 milisecunde	100 bytes
2	111	0.1 secunde	11 kilobytes
4	11 111	11 secunde	1 megabyte
6	10^6	18 minute	111 megabytes
8	10^8	31 ore	11 gigabytes
10	10^{10}	128 zile	1 terabyte
12	10^{12}	35 ani	111 terabytes
14	10^{14}	3 500 ani	11 111 terabytes

În tabel sunt prezentate mai multe valori a soluţiei adâncimii, timpul și memoria necesară pentru căutarea în lăţime. De exemplu, dacă o anumită problemă ar avea o soluţie cu adancimea de 14 se va necesita in jur de 3500 de ani pentru ca aceasta cautare sa ofere raspunsul.

Se poate deduce astfel dezavantajul major al căutării in lăţime si anume faptul ca are necesităţi de memorie mult prea mari, creşterea spaţiului fiind tot exponentială. In general spaţiul este o problemă mult mai mare decat timpul, astfel, dacă pentru un algoritm care solicită mult timp de calcul putem în cele din urmă aştepta până când furnizează rezultatul, în cazul în care memoria nu este suficientă, algoritmul evident nu poate rula şi nu va furniza niciodată un rezultat.

3.1.2 Căutarea cu cost uniform

Algoritmul de căutare cu cost uniform rezolvă optimalitatea pentru cazul în care operatorii nu au costuri egale, deci costul unor noduri de pe un nivel mai mare (aflat mai jos în arbore) nu este neapărat mai mare decât costul unor noduri de pe un nivel mai mic.
Strategia mai este cunoscută sub numele de Dijkstra's Source Shortest Path sau simplu algoritmul lui Dijkstra.

Algoritmul funcționează similar cu strategia de căutare in lățime, doar că de această dată nodurile nu sunt explorate în ordinea nivelelor, ci în ordinea costurilor, explorându-se întotdeauna nodul cu costul cel mai mic. În cazul în care costul operatorilor este egal, strategia de căutare cu cost uniform se comportă identic cu strategia de căutare pe nivel.

PSEUDOCOD:

Acesta este similar cu cel al căutării în lățime, însă ultima linie este schimbată, fiindcă în lista de noduri nedescoperite se adaugă întotdeauna nodul cu costul cel mai mic.

```
functia cautare_cost_uniform(problema) intoarce solutie sau esec
noduri = genereaza_lista(genereaza_nod(stare_initiala[problema]))
Cat timp solutie negasita si noduri ≠ ∅ executa
Daca noduri = vida atunci
                                intoarce esec
nod = scoate_din_fata(noduri)
Daca testare_tinta[problema] se aplica la stare(nod) atunci
        intoarce nod
Altfel
  noduri = adauga(noduri, expandare(nod, ordoneaza_dupa_cost))
  Sfarsit cat timp
```

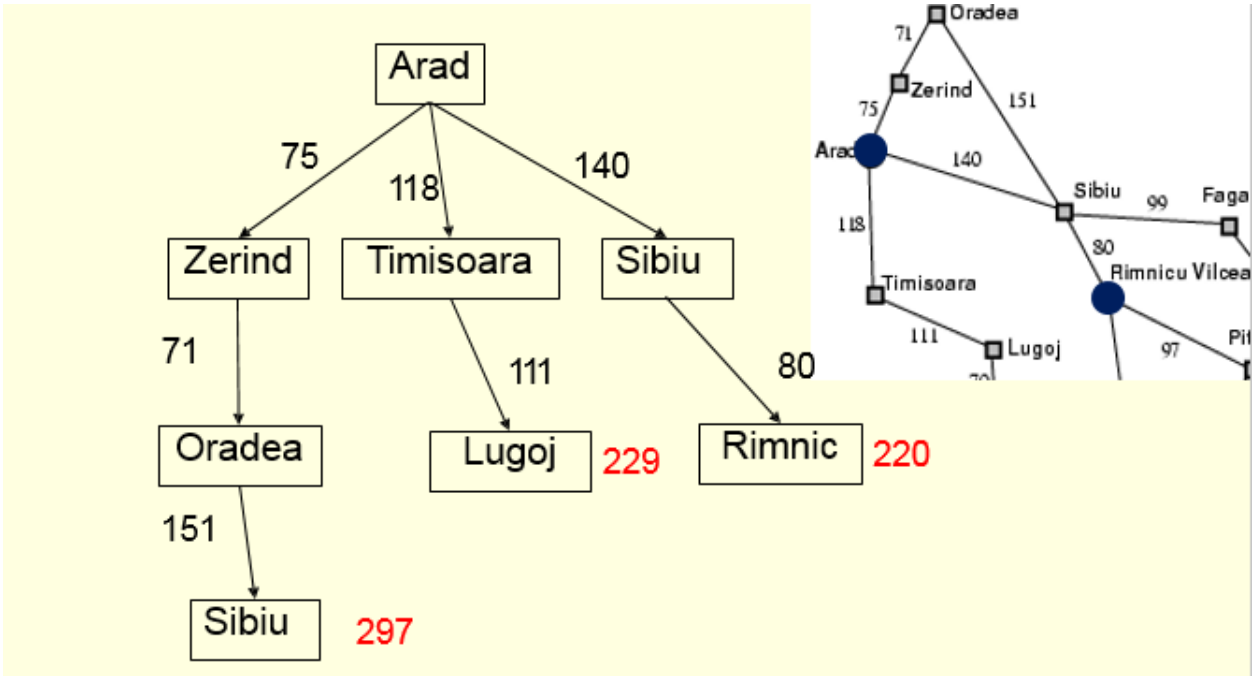
Se permit noduri duplicate, insa si ele sunt ordonate in functie de cost.

Pe lângă ordonarea nodurilor in funcție de cost, mai există două diferențe majore față de parcurgerea în lățime.

Prima este faptul că nodul care urmează sa fie extins este selectat și nu este primul nod generat. Motivul este că primul nod generat poate fi pe o cale mai puțin optimă, adică nu este cel mai mic cost. A doua diferență este reprezentată de faptul căci chiar dacă s-a găsit o soluție, căutarea continua, pentru a se verifica daca nu se găsește o soluție mai bună.

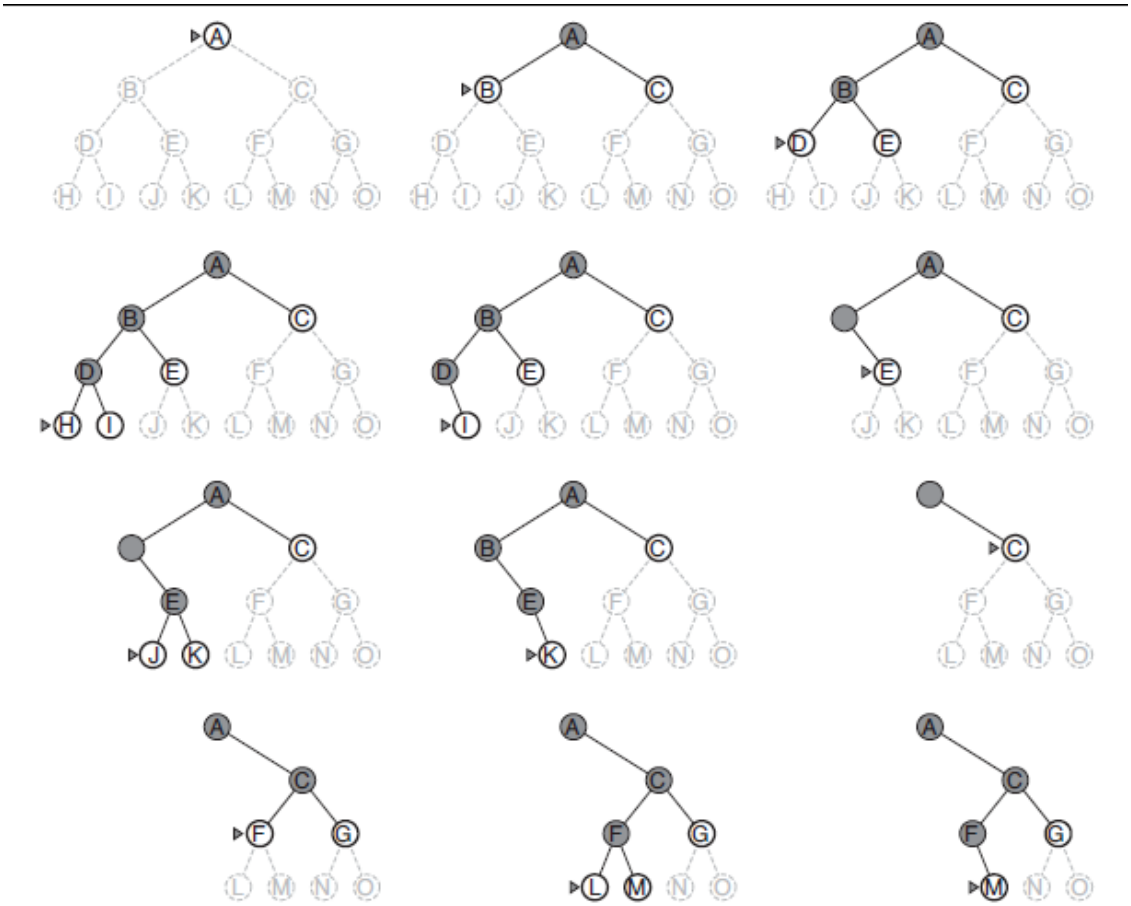
Exemplificare:

Ambele modificări sunt ilustrate în problema de a ajunge de la Arad la Rîmnicu Vâlcea. Nodurile succesoare sunt Zerind cu costul 75, Sibiu cu costul 140 si Timișoara cu costul 118. Ele sunt așezate în ordine crescătoare a distanței față de nodul rădăcina. Următorul nod extins este nodul cu costul cel mai mic, adică Zerind adăugând nodul Oradea cu costul 71, având un total de 146. Următorul nod cu costul cel mai mic este Timișoara și se adaugă nodul Lugoj cu costul de 111, având un total de 229 și apoi se extinde nodul Sibiu, adăugându-se nodul Râmnic cu costul 80, rezultând un total de 220. Astfel am ajuns la destinație, nodul ținta fiind generat. Însă căutarea nu a luat sfârșit, fiindcă se dorește găsirea celei mai bune căi și se verifică dacă celelalte sunt mai bune decât varianta deja găsită. Astfel, la nodul Timișoara, se adaugă nodul Lugoj cu costul 111, având un total de 229. A depășit deja valoarea 220 și este eliminat. Mergem la ultima ramură și adăugam la nodul Zerind nodul Oradea, cu costul 71, acumulând un cost de 151 și apoi nodul Sibiu. S-a depășit deja valoarea 220 și se elimină și această variantă. Astfel soluția poate fi returnată.



3.1.3 Căutarea în adâncime

Algoritmul de căutare în adâncime ajută la parcurgerea arborilor binari. Această parcurgere a arborelui începe de la nodul rădăcină apoi se merge pe un drum pana drum ajunge la cel mai adânc nivel al arborelui, in momentul când se ajunge la nodurile frunza, căutarea se întoarce și se parcurg nodurile de la nivele mai puțin adânci.



PSEUDOCOD:

Avem o funcţie de căutare în adâncime care primește parametrul problema, întoarce o soluţie sau un eșec. Mai întâi se generează o frontieră care primește toate nodurile nedescoperite și o listă care este o coadă LIFO. O astfel de coadă conține cel mai recent nod generat care este ales pentru a fi descoperit. Acest nod este cel mai adânc nod nedescoperit, fiind mai adânc ca părintele său- care la rândul său a fost cel mai adânc nod nedescoperit când a fost selectat.

```
functia cautare_adancime(problema) intoarce solutie sau esec
noduri = genereaza_lista(genereaza_nod(stare_initiala[problema]))
Cat timp solutie negasita si noduri ≠ ∅ executa
Daca noduri = vida atunci
                                    intoarce esec
nod = scoate_din_fata(noduri)
Daca testare_tinta[problema] se aplica la stare(nod) atunci
intoarce nod
Altfel
  noduri = adauga(noduri, expandare(nod, adauga_la_inceput))
  Sfarsit cat timp
```

Analiza complexității:

b – factor de ramificare (nr de noduri fii ale unui nod)
d - lungimea (adâncimea) la care se găsește o soluție
m – adâncimea maxima din arbore

- Complexitate temporală: b^m
- Complexitate spațială : b*m
- Completitudine : Nu ->algoritmul nu se termină pentru drumurile infinite (neexistând suficientă memorie pentru reținerea nodurilor deja vizitate)
- Optimalitate : Nu -> căutarea în adâncime poate găsi un drum soluție mai lung decât drumul optim

Avantaje:

- Nu necesită multă memorie (fată de căutarea în lățime)– stochează un singur drum de la rădăcină la o frunză împreună cu nodurile neexpandate.

Dezavantaje:

- Se poate bloca pe anumite drumuri greșite (nenorocoase) fără a putea reveni
    - Ciclu infinit
    - Găsirea unei soluții mai “lungi” decât soluția optimă

Acest algoritm nu este recomandat să fie folosit la arbori care au o adâncime foarte mare.

3.1.4 Căutarea limitată în adâncime

Algoritmul de căutarea limitată în adâncime reprezintă o îmbunătățire a căutării în adâncime, astfel acesta poate fi folosit și atunci când dorim să parcurgem arbori cu o adâncime foarte mare. Acest algoritm impune o margine superioară pentru lungimea unui drum, el se poate utiliza atunci când cunoaștem la ce adâncime maximă trebuie găsită soluția.

PSEUDOCOD:

Algoritmul de căutare limitată în adâncime, reține pentru fiecare nod și adâncimea la care se află, începând cu rădăcina care este la 0, iar la while se adaugă si condiția să nu se depășească limita prestabilită.

```
functia cautare_adancime_limitata(problema) intoarce solutie sau esec
noduri = genereaza_lista(genereaza_nod(stare_initiala[problema]))
Cat timp solutie negasita si noduri ≠ ∅ executa
Daca noduri = vida atunci
                                    intoarce esec
nod = scoate_din_fata(noduri)
Daca testare_tinta[problema] se aplica la stare(nod) atunci
intoarce nod
Altfel
  noduri = adauga(noduri, expandare(nod, adauga_la_inceput_daca_
  limita_permite))
  Sfarsit cat timp
```

Analiza complexității:

l - limita adâncime
b – factor de ramificare (nr de noduri fii ale unui nod)
d - lungimea (adâncimea) la care se găsește o soluție

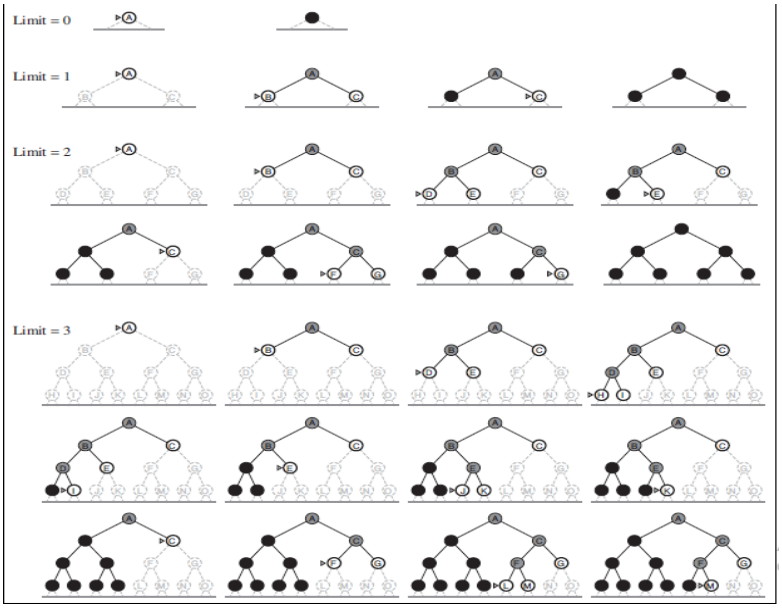
- Complexitate temporală: b^l
- Complexitate spațială: b*l
- Completitudine -> Da, doar daca $l > d$ sau $l = d$
- Optimalitate -> Nu, căutarea în adâncime poate găsi un drum soluție mai lung decât drumul optim

Dezavantaje:

- Este greu sa alegem o limită, daca l < d, nu se va găsi soluția căutată.

3.1.5 Căutarea în adâncime iterativă

Algoritmul de căutarea în adâncime iterativă soluționează problema stabilirii limitei optime, acesta parcurgând iterativ fiecare limită până la găsirea nodului căutat.



PSEUDOCOD:

Algoritmul de căutare în adâncime iterativă, aplică în mod repetat căutarea limitată în adâncime cu limite în creștere. Încetează atunci când se găsește soluția sau dacă căutarea limitată în adâncime returnează eșec, ceea ce înseamnă că nu există nicio soluție.

```
functia cautare_adancime_iterativa(problema) intoarce solutie
Pentru adancime = 0 pana la ∞ executa
Daca cautare_adancime_limitata(problema, adancime)
gaseste solutia atunci
intoarce solutia
Sfarsit daca
Sfarsit pentru
```

Analiza complexității:

b – factor de ramificare (nr de noduri fii ale unui nod)
d - lungimea (adâncimea) la care se găsește o soluție

- Complexitate temporală: b^d
- Complexitate spațială: b*d
- Completitudine -> Da
- Optimalitate -> Da

Avantaje:

- Necesită memorie liniară
- Asigură atingerea nodului țintă urmând un drum de lungime minimă
- Mai rapidă decât căutarea în lățime și adâncime

Dezavantaje:

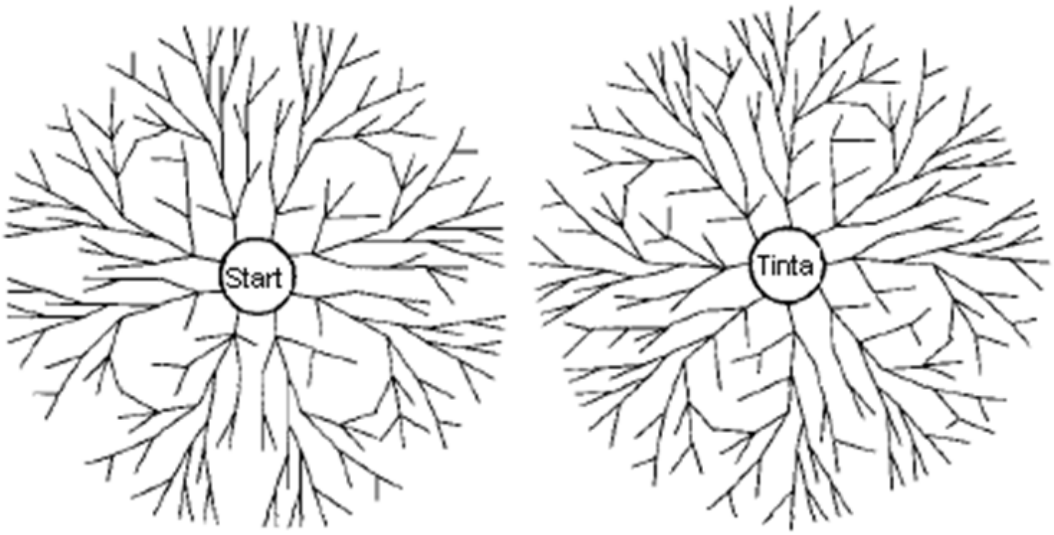
- Unele stări sunt expandate de mai multe ori (cot mare)

Acest tip de căutare este preferat când spațiul de căutare este foarte mare și nu se cunoaște adâncimea soluției.

3.1.6 Căutarea bidirecţională

Ideea din spatele căutarii bidirecţionale este de a rula simultan 2 căutari – una pornind de la starea iniţială către starea finală(de la rădăcina spre frunze), iar cealalta în sens invers(de la frunze spre rădăcina), care se opresc atunci când ajung la un nod comun. Acest lucru duce în mod evident la înjumătăţirea resurselor de timp şi memorie necesare. Pentru economia de memorie pe una dintre direcţii poate fi aplicată oricare dintre strategiile de cautare anterioare.
Deoarece adâncimea arborelui de căutare se înjumătăţeşte, complexităţile de timp şi spaţiu vor deveni $T = O(b^{d/2})$, respectiv S = $O(b^{d/2})$.

B – numărul de noduri în care se expandează/ extinde fiecare nod(adică câţi copii are)
D – adâncimea la care se găseşte soluţia



Implementare:

Necesită stabilirea succesorilor, respectiv a predecesorilor unui nod şi stabilirea locului de intâlnire. (adica nodul curent este 2 şi predecesorul este 1). Se implementează cele două strategii de căutare, astfel se porneşte cu căutarea în lăţime simultan de la starea iniţială şi de la starea finală. Se poate alege căutarea în lăţime dintr-o parte şi o căutare diferită din cealaltă parte, atenţie însă la riscurile ca nodurile din cele două extreme să nu coincidă. Se foloseşte algoritmul în funcţie de strategia de căutare folosită.

Avantaje şi dezavantaje:

Avantajul major este că reduce semnificativ costurile căutării in latime, dar mai mult faptul că este mult mai rapidă decât orice altă strategie neinformată.
Ca dezavantaj rămâne consumul de memorie care este tot exponenţial. Alt dezavantaj este faptul că nu poate fi implementată dacă nu este cunoscută starea finală sau dacă operatorii de generare a stărilor nu sunt inversabili sau sunt greu de calculat predecesorii.Exemple de aplicaţii: cel mai scurt drum.Căutările sunt complete dacă numărul de noduri este finit.

Căutarea bidirecţională este completă şi optimală după cum sunt cele două strategii care le implică. Astfel pentru cazul în care se foloseşte cautarea in latime, este completă tot timpul (atunci când factorul de ramificaţie este finit) şi este optimală pe toate cazurile unde este şi cautarea in latime optimală.

Stările repetate, stări în care agentul deja a mai fost conduc la creşterea inutilă a necesităţilor de timp şi spaţiu precum şi la arbori de căutare de dimensiune infinită. Mai mult, arborii infiniţi fac în unele cazuri ca strategia de căutare să nu mai găsească niciodată soluţia problemei, intrându-se într-o buclă infinită.

Pentru evitarea acestui neajuns există câteva restricţii care pot fi aplicate:

- Pentru un nod, să nu existe posibilitatea de a se intoarce in nodul părinte;

- prin extindere să nu apară noduri care au fost găsite la noduri predecesor;
- Să nu se genereze o stare care a mai fost intâlnită anterior.

3.1.7 Agenți care rezolvă probleme

Formularea problemelor

Un agent american venit în România cu scopul de a vizita țara, aflându-se momentan la Arad. În ziua următoare el trebuie să ajungă la București deoarece el are un bilet de avion nerambursabil pentru ziua respectivă. În cazul acesta **obiectivul** agentului este să ajungă la București acesta reprezentând primul lucru care trebuie stabilit.
O schemă simplă pentru un agent constă în: formularea problemei, căutarea și execuția.
Formularea problemei este un proces de a decide ce acțiuni și stări trebuie luate în considerare pentru a atinge scopul final.
Agentul trebuie să decidă traseul de la Arad la București, el având posibilitatea de-a alege între cele trei ieșiri din Arad: Sibiu, Timișoara sau Zerind. Din cauza că agentul nu are cunoștințe adiționale despre România, el nu poate decide calea optimă petru atingea scopului. Stabilirea traseului este o acțiune aleasă în mod aleator. Dacă agentul are o hartă, el poate examina diferite secvențe de acțiuni posibile care duc la starea finală, apoi alege cea mai bună secvență de acțiuni.
Căutarea este procesul de a examina secvențe de acțiuni pentru alegerea celei mai bune dintre ele. Căutarea poate fii realizată prin intermediul unui algoritm de căutare, care are ca intrare o problemă și întoarce o soluție sub forma unei secvențe de acțiuni. După găsirea soluției se trece la faza de execuție.

```
functia agent_simplu_rezolvitor_de_probleme(p) intoarce actiune
Persista la fiecare reapelare
- s – o secventa de actiuni initial vida
- stare – descriere a starii curente in care se afla lumea
- g – scop, initial nul
- problema - formulare

stare = actualizeaza_stare(stare, p)
daca s este vida atunci
g = formulare_scop(stare)
problema = formulare_problema(stare, g)
s = cautare(problema)
actiune = recomandare(s, stare)
s = rest(s, stare)
intoarce actiune
```

Există patru tipuri de probleme:
- probleme cu o singură stare,
- probleme cu mai multe stări,
- probleme contigente,

- probleme explorative.

În cazul problemei cu o singură stare, mediul este determinist, complet observabil, agentul știe exact în ce stare se va găsi, el știe exact ce efecte au acțiunile sale, iar soluția este o secvență. La problema cu mai multe stări mediul este neobservabil, agentul știe exact ce efecte au acțiunile sale, dar nu știe în ce stare se găsește. Agentul trebuie să raționeze în raport cu mulțimea de stări la care se ajunge. În cazul prblemelor contigente, mediul este unul nedeterminist, parțial observabil, iar perceptorii oferă agentului informații noi despre starea curentă. La problemele explorative, spațiul stărilor este unul necunoscut, de aceea agentul trebuie să experimenteze, să descopere gradual care sunt efectele acțiunuilor lui și ce tipuri de stări există.
În tipologia problemelor explorative putem încadra agentul american aflat în Arad, deoarece el nu are o hartă și nu știe nici alte cunoștințe despre România.
O problemă este definită prin patru puncte: starea inițială, acțiuni sau funcția succesor, testarea țintei problemei, funcția de cost al drumului. Starea inițială exprimă starea în care se află agentul, de exemplu Arad. Acțiunile sau funcția succesor, notată cu S(x), exprimă o stare x, unde S(x) întoarce mulțimile de stări în care se poate ajunge din x printr-o singură acțiune, de exemplu S(Arad) = {Zerind, Sibiu, Timișoara}. Testarea țintei problemei verifică dacă starea curentă a atins ținta problemei, în exemplul nostru se verifică dacă x = (București, sah_mat(x)). Funcția de cost al drumului calculează un cost *g* pentru drumul curent (suma distanțelor, numărul acțiunilor executate,etc.) . În final soluția este definite, o secvență de acțiuni care merg de la o stare inițială la o stare țintă.

Probleme din viaţa reală:

1. Algoritmi de găsire de rute:
   - Rutarea în reţele de calculatoare
   - Sisteme de planificare pentru transportul aerian

2. Problema comis-voiajorului

   - Să se găsească cel mai scurt tur astfel încât să se viziteze fiecare oraş exact o dată plecând şi terminând din/în acelaşi oraş.
   - Spre deosebire de problemele cu găsire de rute, aici trebuie reţinute oraşele vizitate.

3. Problema misionarilor şi canibalilor

   Trei misionari şi trei canibali se află de o parte a râului. Ei au o barcă ce poate duce cel mult doi oameni. Găsiţi o posibilitate să traverseze toţi de cealaltă parte a râului cu condiţia să nu existe mai mulţi canibali decât misionari într-o parte.

- Stări: secvenţe ordonate de 3 numere reprezentând numărul de misionari, canibali şi bărci de partea în care se aflau iniţial (3, 3, 1).
- Acţiuni: mutarea unui misionar sau canibal sau 2 canibali, 2 misionari sau un misionar şi un canibal de pe o parte pe alta.
- Testarea ţintei: starea (0, 0, 0).
- Costul drumului: numărul de traversări

4. Problema celor 8 dame

- Stări: orice aranjament de 0 până la 8 dame care nu se atacă.
- Acţiuni: adaugă o damă la orice pătrat. (pot avea loc diferite acţiuni: adaugă o damă pe coloana cea mai din stânga a.î. să nu fie atacată de alta damă.)
- Testarea ţintei: 8 dame care nu se atacă pe tablă.
- Costul drumului: 0.
- 64^8 posibilităţi.

3.2 Metode de căutare informată

3.2.1 Best First Search

- Strategia de căutare întâi cel mai bun este o instanță a căutării arbore (tree search)
- Tree search este un algoritm
- În acest algoritm, un nod este selectat utilizându-se o funcție de evaluare pentru fiecare nod
- Cu ajutorul funcției de evaluare se iau decizii bazate pe evaluare

Exemplu:
-> funcția de evaluare interpretează un cost estimat, în consecință nodul cu cea mai scăzută evaluare va fi expandat primul

- Implementarea căutării best first graph este identică cu strategia de căutare a costului uniform
- Deosebirea constă în utilizarea lui f în loc de g pentru a ordona prioritățile
- Alegerea lui f determină ce strategie de căutare vom folosi
- Algoritmii bazați pe best first search includ ca parte din *f* o funcție euristică notată cu *h (funcție folosită în estimarea costurilor)*

Există două abordări:
1. Căutarea Greedy
2. Căutarea A*

3.2.1.1 Căutarea Greedy

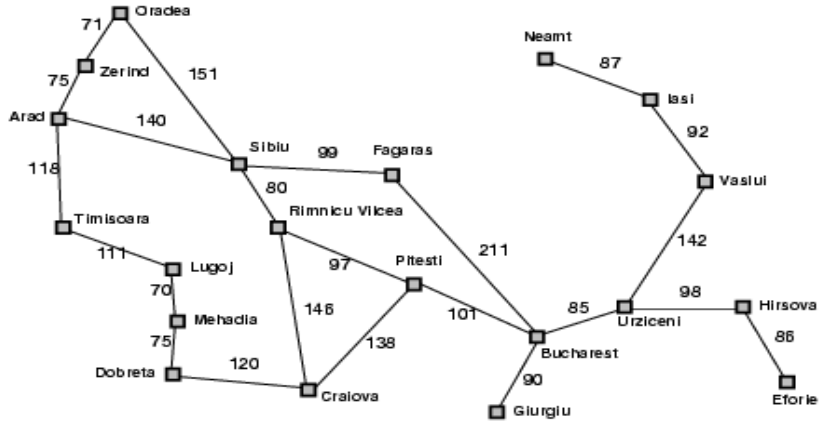
- Încearcă să expandeze nodul cel mai apropiat de țintă pentru a ajunge cât mai rapid la o soluție
- Evaluează nodurile bazându-se doar pe funcția euristică

$f(n) = h(n)$

- Se păstrează ideea de a minimiza costul de ajungere la nodul țintă,chiar dacă la cele mai multe dintre probleme costul nu poate fi determinat cu exactitate, ci doar estimat
- Aici intervine funcția euristică amintită si notată cu *h*
- Funcția euristică h(n) = costul estimat pentru cea mai ieftină cale de a ajunge dintr-o stare la nodul n până la starea țintă
- Dacă n = nodul țintă, atunci h(n) = 0
- Căutarea Greedy are la bază utilizarea funcției euristice

Demonstrarea căutarii Greedy

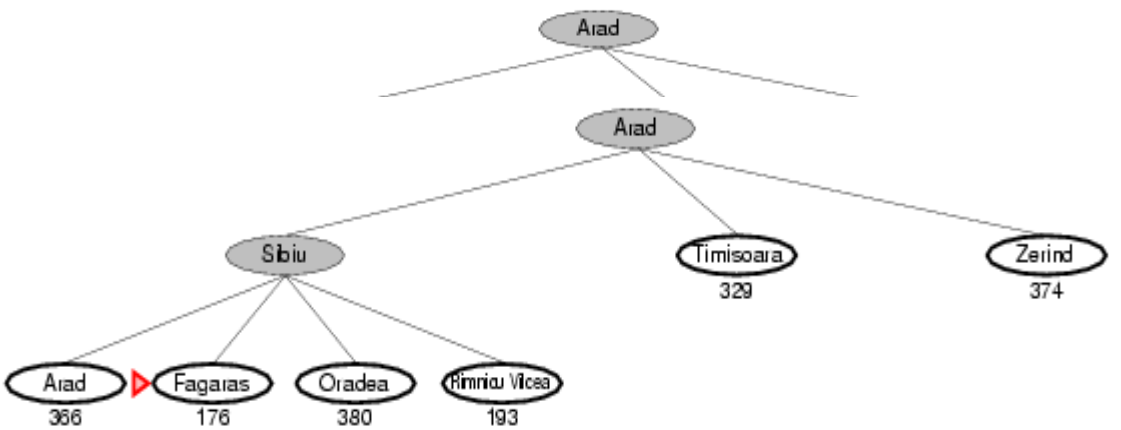
Se cunoaște distanța de la un oraș n până la București
Orașului n i se mai adaugă alte orașe intermediare până se ajunge la București

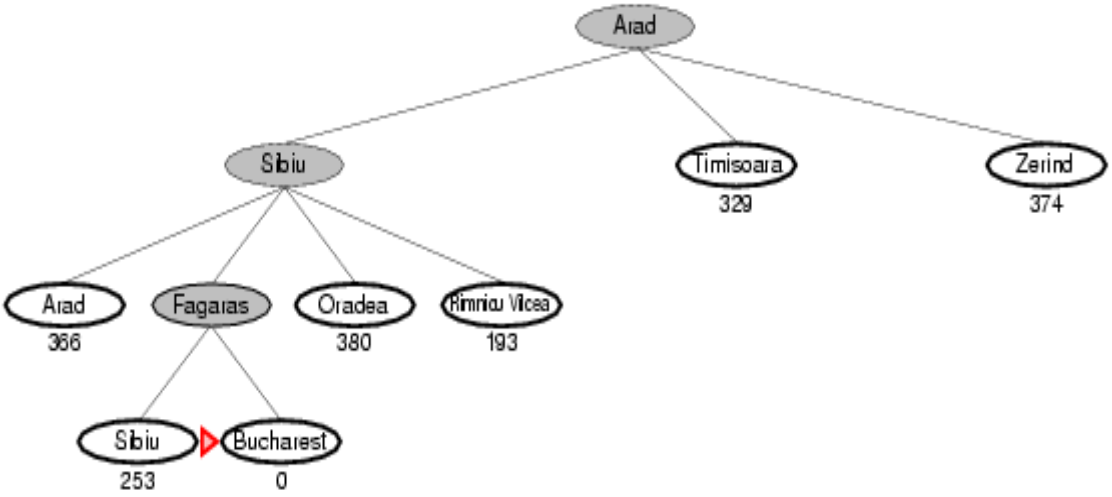


Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Obiectiv: Găsirea celui mai rapid drum Arad-București

- Deoarece dorim să fim cât mai eficienți ne vom folosi de o metodă care utilizează linia dreapta (straight line distance heuristic)
- Primul nod pe care îl vom expanda din Arad va fi Sibiu; de ce? Pentru că Sibiu este mai aproape de București decât Zerind și Timișoara
- Următorul nod care va fi expandat este Făgăraș deoarece ne duce exact la București, care este și ținta noastră.





Căutarea Greedy

- Totuși soluția găsită de noi nu este cea optimală
- Această problemă ne arată de ce algoritmul se numește "greedy" (lacom); la fiecare execuție încearcă să se apropie cât mai mult de soluție indiferent că parcursul nu este unul optim
- Se urmărește rezultatul și nu parcursul
- Dacă se alegea calea via Sibiu -> Făgăraș -> București = mai puțin cu 32 km

Căutarea Greedy are și dezavantaje:

- Este incompletă chiar și într-un spațiu finit (asemănătoare căutarii în adâncime)
- Ca și în cazul căutării în adâncime, se merge pe o singură cale și dacă nu se găsește nodul țintă se intoarce pe o altă cale
- Să considerăm că vrem să ajungem de la Iaşi la Făgăraș

  -> funcția euristică sugerează că Neamț ar trebui expandat prima dată deoarece e mai aproape de Făgăraș, dar drumul nu duce nicăieri
  -> soluția ar fi să mergem către Vaslui -> Urziceni -> București -> Făgăraș
  -> algoritmul nu va găsi această soluție niciodată deoarece alegând Neamț se ignoră Iași, care este mai aproape de Făgăraș decât Vaslui, ceea ce duce la declanșarea unui loop infinit

- Totuși, cu o funcție euristică potrivită, complexitatea poate fi redusă substanțial
- Cantitatea care trebuie redusă depinde de fiecare problemă în particular, dar și de calitatea funcției euristice folosite
- Complexitate temporală și spațială $O(b^m)$, unde:

  b este numărul de noduri în care se expandează fiecare nod
  m este adâncimea maximă a spațiului de căutare

3.2.1.2 Căutarea A*

- Cea mai cunoscută formă a algoritmului best first se numeşte căutarea A*
- Evaluează nodurile combinând *g(n)* = costul pentru a ajunge la nod și *h(n)* = costul de a ajunge de la nod la țintă

$$f(n) = g(n) + h(n).$$

- g(n) -> costul de la calea unde se află nodul de început până la nodul n
- h(n) -> costul estimat de la cea mai ieftină cale de a ajunge de la n la țintă
- Atunci avem f(n) = costul estimat al celei mai ieftine soluții prin n
- Dacă încercăm să căutăm cea mai ieftină soluție trebuie să încercăm să folosim nodul cu cea mai mică valoare a g(n) + h(n)
- Această soluție ne arată că h(n) îndeplinește anumite condiții
- Căutarea A* este optimală și completă dacă h nu supraestimează costul de ajungere la soluție
- Este identică cu căutarea costului uniform
- Diferența este că: A* folosește g + h în loc de g

Condiții pentru optimalitate, admisibilitate și consistență

- Pentru optimalitate avem nevoie ca h(n) să fie o funcție euristică admisibilă
- O funcție euristică admisibilă este o funcție care nu supraestimează costul pentru a atinge ținta
- g(n) -> costul de a-l atinge pe n în calea curentă
- f(n) = g(n) + h(n) consecința imediată a faptului că f(n) nu supraestimează costul soluției pentru calea curentă prin n
- O funcție euristică admisibilă este optimistă deoarece ele tind spre un cost mai mic decât este de fapt
- Un exemplu care demonstrează acest fapt este hLSD pe care am utilizat-o pentru a ajunge la București în căutarea Greedy

Optimalitatea lui A*

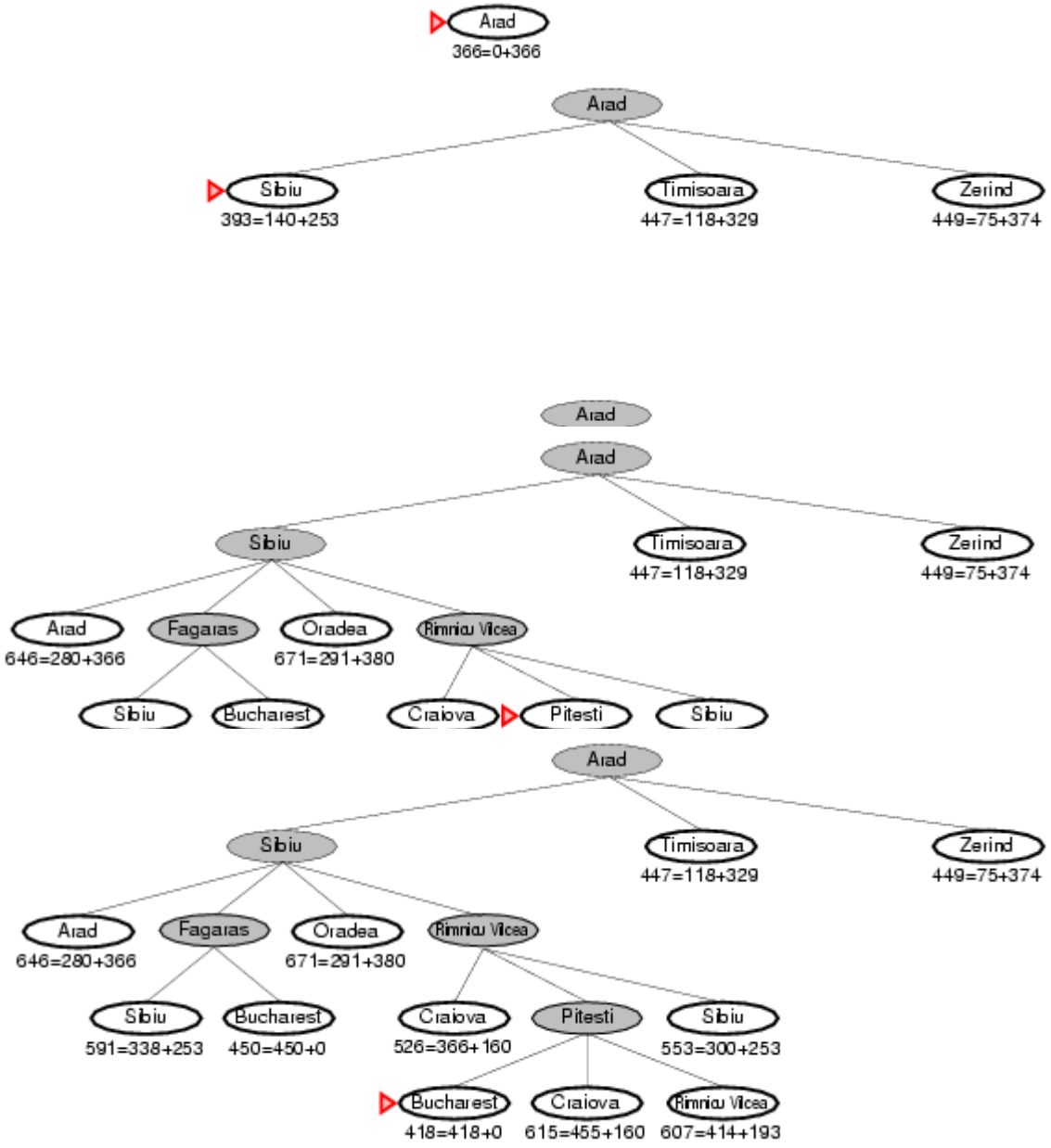
- Să ne reamintim; A* are următoarele proprietăți:

1. Căutarea arbore în versiunea A* este optimală dacă h(n) este admisibilă

2. Căutarea graf în versiunea A* este optimală daca h(n) este consistentă

- Dacă h(n) = consistentă atunci valorile lui f(n) = nu sunt în scădere
- n' este succesorul lui n => g(n') = g(n) + c(n,a,n') => f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') >= g(n) + h(n) = f(n)

Următorul pas este să demonstrăm că oricând A* selectează un nod pentru expandare, calea optimă spre acel nod a fost găsită. Unde nu se aplică acest lucru va trebui să existe un alt n' pe calea optimă de la nodul de start.



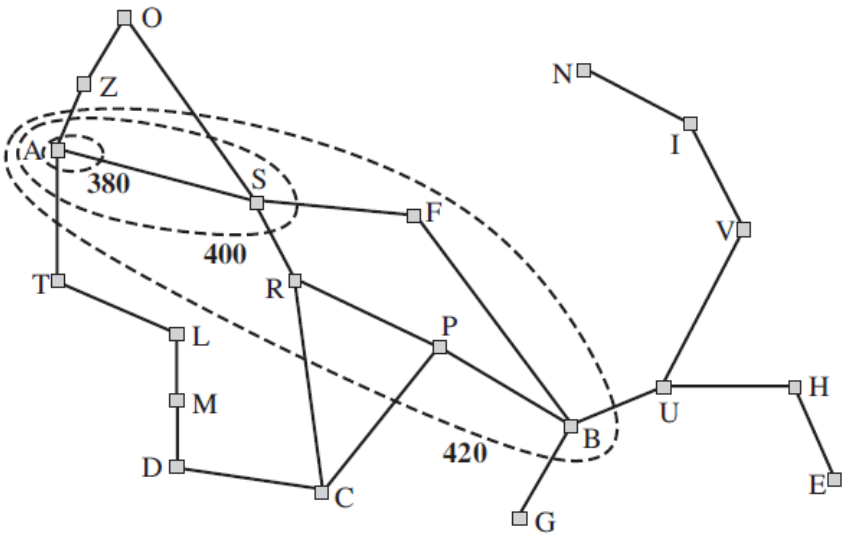


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

Din cele 2 constatări reiese că **nodurile expandate de A* folosindu-ne de metoda GRAPH SEARCH nu sunt în scădere dată de f(n).** Primul nod tintă selectat pentru expandare trebuie să fie o soluție optimală deoarece f este costul pentru nodurile țintă (care au h=0) și mai târziu toate nodurile țintă vor fi cel puțin la fel de scumpe.
Chiar dacă costurile lui f nu sunt în scădere în nici o cale înseamnă și că putem desena contururi în spațiul de stări la fel ca și contururile din harta topografică. (Figura 3.25 ne arată un exemplu)
În interiorul etichetei cu numărul 400, toate nodurile au f(n) mai puțin sau egal cu 400, şi așa mai departe.

Apoi, deoarece A* expandează frontiera nodului cu cel mai scăzut cost f, putem vedea că o căutare A* apare din nodul de start adăugând noduri în sensuri concentrice ale costului f de creștere.
Dacă avem costuri de căutare uniforme (căutarea A* folosind h(n) = 0), sensurile vor fi circulare în jurul stării de început. Cu funcții euristice mai potrivite, sensurile se vor întinde în jurul stării țintă și vor deveni mai atente la calea optimală. Daca C* este costul soluției căii optimale putem spune următoarele:

- A* expandează nodurile f(n) < C*
- A* ar putea expanda unele noduri chiar pe conturul țintă (unde f(n) = C*) înainte să se selecteze un nod țintă.

Caracterul complet cere existența unui număr finit de noduri cu cost mai mic sau egal cu C*, o condiție care este adevărată dacă toţi paşii pe care îi parcurge costul sunt finiți şi b este finit.

Dacă A* nu expandează nici un nod cu f(n) > C* de exemplu, Timișoara nu este expandată chiar dacă este un fiu al rădăcinii. Spunem că sub-ramificația Timișoara este tăiată deoarece hSLD este admisibilă; algoritmul poate ignora această sub-ramificație și optimalitatea tot va fi garantată.
Conceptul tăierii (eliminarea posibilităților care pot fi luate în considerare fără a fi examinate) este important pentru numeroase ramuri ale inteligenței artificiale.

O ultimă observație pe care o facem este că printre atâția algoritmi optimali de acest tip – algoritmi care extind calea de căutare de la rădăcină și folosesc aceeași informație euristică – A* este optimal eficientă pentru orice consistenţă euristică dată. Nici un alt algoritm optimal nu garantează expandarea nodurilor care sunt mai puține decât A* (cu excepția posibilității tie-breaking printre nodurile f(n)=C*). Acest lucru se întâmplă la orice algoritm care nu expandează toate nodurile f(n)<C*; poate întâmpina riscul de a rata soluția optimală.

Căutarea A* este completă, optimală și optimal eficientă printre algoritmii de același tip. Din păcate, nu înseamnă că A* este răspunsul pentru toate căutarile de care avem nevoie. Secretul este că, pentru majoritatea problemelor, numărul de stări din căutarea conturul țintă este exponențial în lungimea soluției. Pentru probleme cu costuri constante, creșterea ca o funcție pentru o soluție optimală, adâncime (depth) d este analizată în termeni de eroare absolută și eroare relativă a euristicului.

Eroarea absolută este definită ca $\Delta = h^* - h$, unde h^* este costul actual de a ajunge de la rădăcina țintă, iar eroarea relativă este definită ca $\varepsilon = (h^*-h)/h^*$.

Complexitatea lui A* de multe ori face ca găsirea unei solutii optimale să nu fie practică.
Se pot utiliza variante ale A* care găsesc rapid soluții sub-optime sau se pot întocmi funcții euristice care sunt mai potrivite dar nu și neapărat admisibile. În orice caz, utilizarea unei funcții euristice potrivite furnizează un mod de a economisi enorm în comparație cu folosirea unei căutări neinformate.

Timpul de calcul nu este principalul dezavantaj al căutării A*. Deoarece generează noduri în memorie (la fel ca și algoritmul GRAPH-SEARCH), de obicei A* rămâne fără spațiu deoarece rămâne fără timp. Din acest motiv, A* nu este practică pentru probleme la scară largă. Există totuși algoritmi care pot acoperi problema spațiu-timp fără a sacrifica optimalitatea sau complexitatea, la un cost mic de execuție.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

Figure 3.26 The algorithm for recursive best-first search.

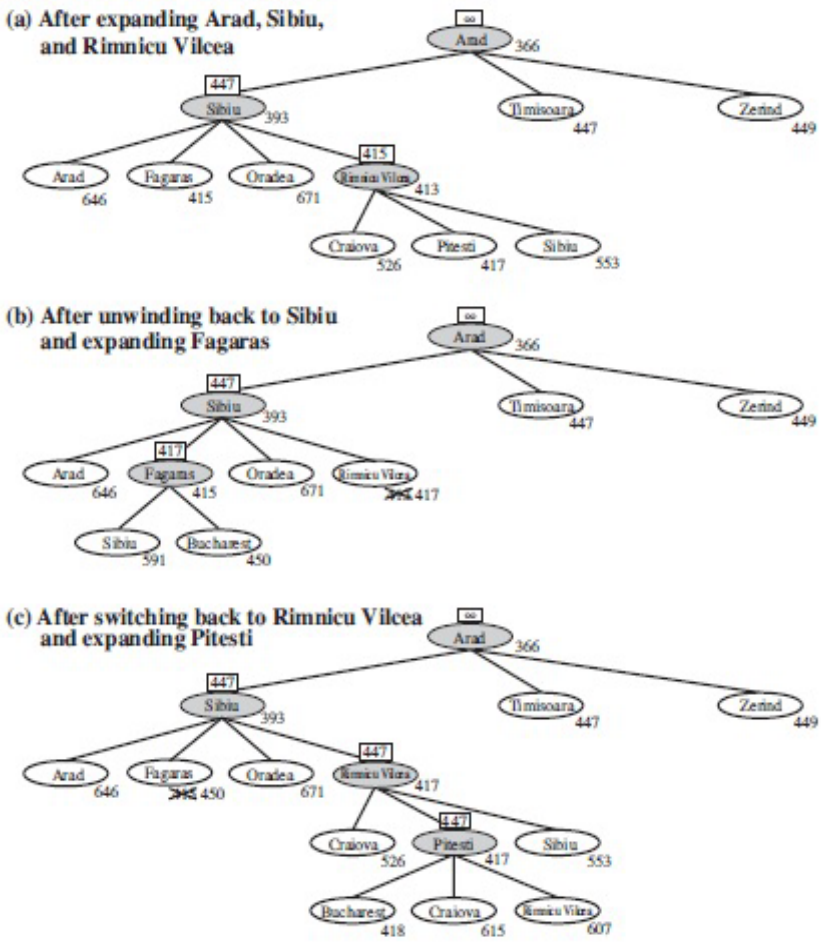
3.2.2 Căutare euristică cu memorie delimitată

Algortimul IDA* - **I**terative **D**eepening **A***

- A rezultat din adaptarea ideii de adâncire iterativă la contextul de căutare euristică
- Diferenţa principală faţă de adâncirea iterativă standard constă în contorizarea folosită, care este o funcţie de tip *f-cost* (g+h), în detrimentul celei în adâncime
- La fiecare iterare, valoarea contorizării este dată de funcţia cu cel mai mic cost al oricărui nod care depăşeşte valoarea contorizată din iterarea precedentă
- Este practic pentru multe probleme care au un cost unitar pentru fiecare pas
- Evită supra-solicitarea asociată menţinerii unei cozi de noduri sortate
- Are aceleaşi dezavantaje ca şi versiunea iterativă a căutarii cu cost uniform

Algoritmul RBFS – **R**ecursive **B**est-**F**irst **S**earch

- Este un algoritm recursiv care încearcă să imite funcţionalitatea căutarii best-first standard, folosind spaţiu liniar
- Are o structură similară cu cea a unei căutari depth-first recursivă



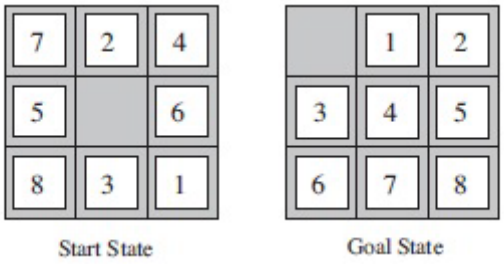
- Foloseşte variabila *f_limit* pentru a monitoriza valoarea celei mai bune căi alternative de la oricare strămoş al nodului curent
- Când un nod curent depăşeşte valoarea limită, procesul recursiv caută o cale alternativă şi valoarea fiecărui nod este înlocuită cu o valoare de back-up, care este cea mai bună valoare a fiilor nodului respectiv
- Este un algoritm puţin mai eficient decât IDA*
- Este considerat un algoritm optim atâta timp cât funcţia euristică $h(n)$ este admisibilă

Algoritmul MA* (Memory-Bounded A*) şi SMA* (Simplified MA*)

- Procedează la fel ca şi algoritmul *A**, expandând cel mai bun nod până când memoria este plină, moment în care se renunţă la cel mai slab nod (cel cu valoarea *f* cea mai mare) şi apoi valoarea nodului uitat se salvează în nodul părinte
- Algoritmul este complet dacă există o soluţie accesibilă – dacă valoarea lui *d* (adâncimea nodului ţintă) este mai mică decât mărimea memoriei
- Este optim dacă orice soluţie optimă este accesibilă – altfel, algoritmul returnează cea mai bună soluţie accesibilă
- În practică, algoritmul reprezintă o variantă robustă de găsire a soluţiilor optime

3.2.2.1 Funcţii euristice

8-Puzzle este una dintre primele probleme de căutare euristică. Rezolvarea acesteia constă în permutarea pieselor pe orizontală sau verticală până când se ajunge la starea finală (goal state).



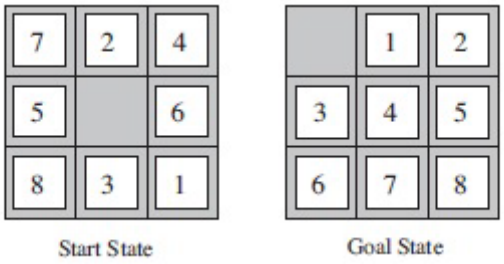
Pentru o instanţă a puzzle-ului generată aleatoriu, costul mediu al unei soluţii de rezolvare este de aproximativ 22 de paşi. Factorul de expandare este de aproximativ 3.

- Când spaţiul gol este în mijloc sunt posibile 4 mutări
- Când spaţiul gol este într-un colţ sunt posibile 2 mutări
- Când spaţiul gol este de-alungul unei muchii sunt posibile 3 mutări

De aici rezultă că o căutare în adâncime până la nivelul 22 va parcurge aproximativ 3.1 x 10^{10} stări. Pentru un sistem mai lărgit (de exemplu – 15-Puzzle), numărul corespunzător de stări ar fi de aproximativ 10^{13}.

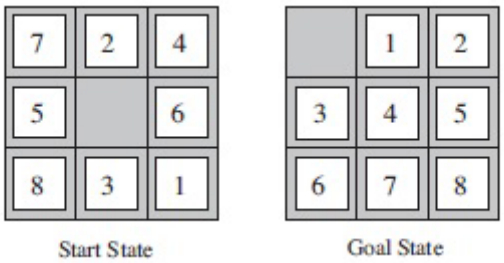
Astfel, se pune problema găsirii unei funcţii euristice bune. Dacă vrem să găsim cea mai scurtă soluţie folosind algoritmul *A**, avem nevoie de o funcţie euristică ce nu supra-estimează niciodată numărul de paşi până la starea ţintă. Două dintre cele mai folosite funcţii euristice în acest sens sunt:

- ***h1*** = numărul de piese poziţionate greşit



  - în figura de mai sus, toate cele 8 piese sunt poziţionate greşit, astfel că starea de start va avea *h1* = 8
  - *h1* este o funcţie euristică admisibilă pentru că este clar că orice piesă care nu e la locul ei trebuie mutată cel puţin o dată

- ***h2*** = suma distanţelor pieselor faţă de poziţia ţintă
  - reprezintă suma mutărilor necesare pentru ca piesele să ajungă de la poziţia curentă până la poziţia ţintă
  - piesele se pot muta doar pe orizontală sau pe verticală, pe diagonală nu
  - mai poartă denumirea de *distanţă city block* sau *distanţă Manhattan*



  - pentru figura de mai sus, *distanţa Manhattan* este:

$h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$

3.2.2.2 Efectul preciziei euristice asupra performanţei

Factorul de ramificare eficientă b^* - reprezintă un mod de a caracteriza calitatea unei funcţii euristice

- Dacă numărul total de noduri generate de către algoritmul A^* pentru o anumită problemă este N şi adâncimea soluţiei este d, atunci b^* este factorul de ramificare pe care un arbore uniform de adâncime d ar trebui să-l aibă pentru a conţine $N+1$ noduri

  $N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$
- O funcţie euristică bine concepută va avea o valoare a lui b^* cât mai aproape de 1

Pentru a testa funcţiile euristice *h1* şi *h2* menţionate mai sus, s-au generat aleator 1200 de probleme având adâncimea soluţiilor (d) între 2 şi 24, pentru a fi rezolvate cu ajutorul căutarii iterative în adăncime (IDS) şi al algoritmului A^* folosind *h1* şi *h2*. Tabelul următor prezintă numărul mediu de noduri generate de fiecare strategie şi factorul de ramificare eficientă:

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

- Rezultatele sugerează că *h2* este mai bun decât *h1* şi mult mai bun decât căutarea iterativă în adâncime (IDF)
- Se poate spune că, în principiu, *h2* este întotdeauna mai bun decât *h1*
- Din definiţia celor două funcţii euristice se poate observa că, pentru orice nod n, $h2(n) \geq h1(n)$ => spunem că ***h2* domină *h1*** => *h2* este mai eficient decât *h1*
- Cu alte cuvinte, folosirea algoritmului A^* împreună cu *h1* va expanda întotdeauna mai multe noduri decât folosirea lui A^* cu *h2*
- În general, este mai indicat a se folosi funcţii eurisitice cu valori mai mari, cu condiţia ca acestea să fie consistente şi timpul de calcul aferent lor să nu fie prea mare

3.2.2.3 Generarea funcţiilor euristice admisibile prin relaxarea problemelor

O problemă cu mai puţine restricţii asupra acţiunilor posibile se numeşte o **problemă relaxată**. Astfel, dacă regulile sistemului 8-Puzzle s-ar modifica încât să permită unei piese să fie mutată oriunde (nu doar în spaţiul gol adiacent acesteia), atunci *h1* ar oferi soluţia cea mai scurtă. În mod similar, dacă o piesă ar putea fi mutată cu o poziţie în orice direcţie (chiar şi peste o poziţie deja ocupată), atunci *h2* ar oferi cea mai scurtă soluţie.

Graful spaţiului de stări pentru problema relaxată este un supergraf al spaţiului iniţial de stări (înlăturarea restricţiilor determină adăugarea de muchii în graf). Astfel, orice soluţie optimă pentru problema originală este în acelaşi timp o soluţie şi pentru problema relaxată. Dar problema relaxată poate avea soluţii mai bune dacă muchiile adăugate la graf oferă scurtături.
În concluzie, costul unei soluţii optime pentru o problemă relaxată reprezintă o funcţie euristică admisibilă pentru problema originală.

Dacă o problemă este descrisă folosind un limbaj formal, atunci se pot construi probleme relaxate în mod automat. Pentru exemplul de mai sus cu sistemul 8-Puzzle, acţiunile posibile ar putea fi descrise în felul următor:

O piesă se poate muta din locul A în locul B dacă
A este adiacent cu B pe orizontală sau pe verticală şi B este un spaţiu gol

De aici se pot genera 3 probleme relaxate prin înlăturarea uneia sau a ambelor condiţii:

(a) O piesă se poate muta din locul A în locul B dacă A este adiacent cu B
(b) O piesă se poate muta din locul A în locul B dacă B este un loc gol
(c) O piesă se poate muta din locul A în locul B

Astfel, din funcţia euristică descrisă la punctul (a) rezultă *h2* (distanţa Manhattan), iar din funcţia euristică de la punctul (c) rezultă *h1*.

Absolver – este un program care generează funcţii euristice în mod automat din formularea problemelor, folosind metoda problemei relaxate

- A reuşit să genereze o funcţie euristică pentru sistemul 8-Puzzle care este mai bună decât orice altă funcţie euristică existentă
- A găsit prima funcţie euristică folositoare pentru celebra problemă a cubului Rubik

O problemă care se pune în momentul generării mai multor funcţii euristice este că nu se poate determina întotdeauna cu exactitate care dintre acestea este singura cea mai bună. Dacă pentru o problemă avem la dispoziţie o colecţie de funcţii euristice admisibile *h1*, … , *hm* şi nici una din ele nu le domină pe celelalte, cum ştim pe care din ele să o alegem?
Astfel se ajunge la definirea unei funcţii euristice compozite *h* care foloseşte funcţia euristică cea mai potrivită pentru nodul curent:

$$h(n) = \max\{h1(n), \dots, hm(n)\}$$

Functia euristică *h* este admisibilă pentru că şi funcţiile euristice care o compun sunt admisibile. Mai mult, *h* este consistentă şi domină funcţiile euristice din care este compusă.

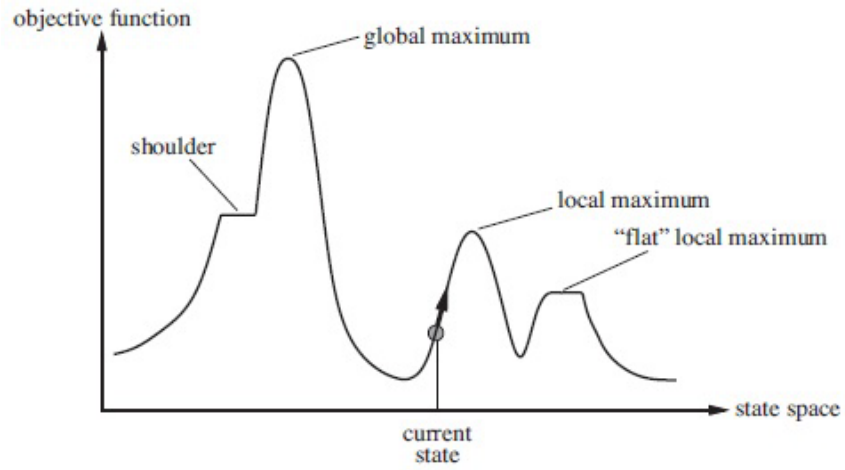
3.2.3 Algoritmi de căutare locală

În cazul multor probleme, calea către ţintă nu este relevantă. De exemplu, în cazul problemei cu 8 dame pe o tablă de şah, ceea ce contează este aşezarea finală a damelor pe tablă (astfel încât să nu se atace între ele), nu ordinea în care acestea se adaugă pe tablă.
Pentru astfel de probleme contează doar spaţiul soluţiilor, nu şi costul drumului pentru găsirea acestora. În rezolvarea acestui tip de probleme se folosesc **algoritmii de căutare locală**.
Aceştia operează folosind un singur nod curent şi caută doar în vecinătatea acestuia. Căile urmate pentru căutarea soluţiilor nu sunt reţinute.

Avantaje:

- Folosesc memorie foarte puţină (de obicei o cantitate constantă)
- Pot găsi soluţii rezonabile în spaţii de stări foarte mari (chiar infinite), unde algoritmii sistematici nu se potrivesc
- Pot fi folosiţi pentru rezolvarea unor probleme de optimizare, a căror scop este găsirea celei mai bune stări în concordanţă cu o funcţie obiectivă (sau cu o funcţie euristică de cost)

Pentru a înţelege cum funcţionează căutarea locală, avem nevoie de figura următoare, care este o reprezentare grafică a spaţiului de stări posibile pentru o problemă:



- Spaţiul stărilor are o *localizare* (axa orizontală - determinată de starea soluţiei) şi o *înălţime* (axa verticală - determinată de valoarea pe care o ia funcţia euristică de cost sau funcţia obiectivă)
- Dacă înălţimea corespunde costului -> scopul este de a se găsi cea mai joasă valoare (un minim global)
- Dacă înălţimea corespunde funcţiei obiective -> scopul este de a se găsi cea mai înaltă culme (un maxim global)

Algoritmul de căutare locală poate fi:
- Complet -> va găsi întotdeauna o stare ţintă, dacă ea există
- Optim -> va găsi întotdeauna un minim sau un maxim global

Căutarea Hill-Climbing

- Este cel mai fundamental algoritm de căutare locală
- Este o buclă care se mişcă în mod continuu în direcţia valorilor în creştere
- La fiecare pas, nodul curent este înlocuit de vecinul său cel mai bun (vecinul care are cea mai mare valoare sau cel mai mic cost euristic estimat)
- Se termină când se găseşte un vârf (culme) al cărui vecini nu au o valoare mai mare
- Nu caută mai departe de vecinii apropiaţi de starea curentă
- Dacă există o mulţime de succesori ai nodului curent care au toţi cea mai bună valoare, se va alege unul dintre ei în mod aleator

- Se aseamană cu cazul din viaţa reală în care se încearcă găsirea vârfului unui munte, străbătându-l printr-o ceaţă groasă şi suferind de amnezie
- Se mai numeşte *căutare locală lacomă* (greedy local search) pentru că se agaţă de un nod vecin mai bun fără a se gândi unde va merge mai departe în continuare

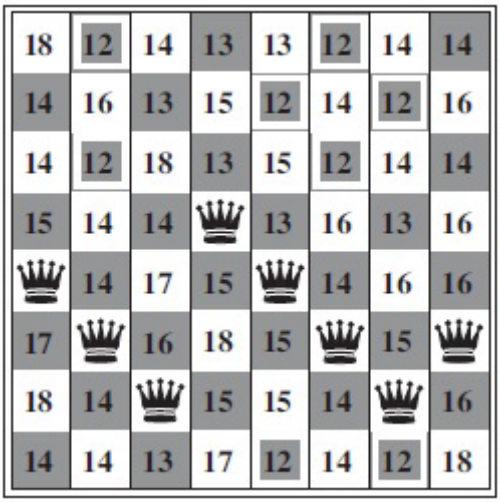
Funcţia de căutare este prezentată în figura următoare:

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

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

Pentru a ilustra algoritmul de căutare Hill-Climbing vom folosi problema celor 8 dame de pe o tablă de şah. Acestea trebuie aşezate pe tablă în aşa fel încât să nu se atace una pe alta.

- Algoritmii de căutare locală folosesc de obicei o formulare a stării complete, unde fiecare stare are 8 dame pe tablă, câte una pe fiecare coloană

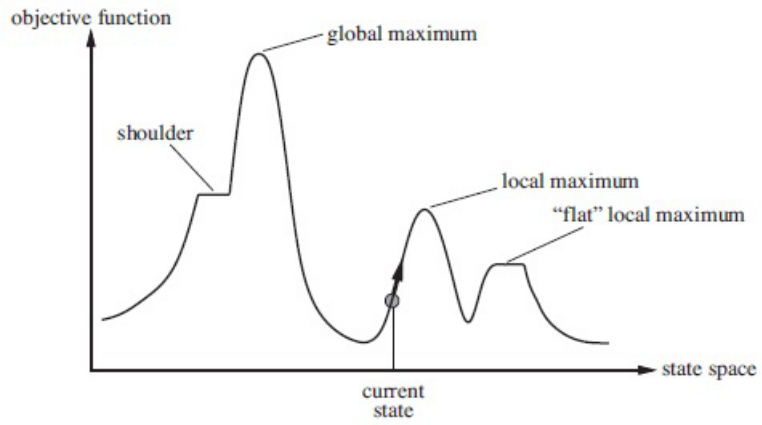


- Funcţia euristică de cost *h* reprezintă numărul de perechi de dame care se atacă între ele -> pentru figura de mai sus, *h* = 17
- Minimul global al acestei funcţii *h* este 0 -> când nici o damă nu atacă o altă damă
- Succesorii unei stări sunt orice stări posibile generate de mutarea unei dame într-o altă căsuţă de pe aceeaşi coloană -> fiecare stare are 8 x 7 = 56 de succesori
- Valorile numerice din căsuţe reprezintă valoarea functiei *h* pentru fiecare succesor al unei dame (se observă că cel mai bun succesor are *h* = 12)

Dezavantajele căutarii Hill-Climbing:

- Maxima locală – este un vârf mai înalt decât fiecare din stările sale vecine dar mai mic decât maximul global din spaţiul stărilor. Când se atinge maxima locală, algoritmul se blochează pentru că nu mai are unde să meargă (nu poate coborî dealul)

- Platouri – sunt zone plane din spaţiul de stări pentru care funcţia de evaluare are valori constante. În aceste cazuri, căutarea se va face la întâmplare



La un moment dat, algoritmul va ajunge într-un punct în care nu va mai putea face nici un progres. Pornind de la o stare generată aleator pentru aşezarea a 8 dame pe o tablă de şah, căutarea Hill-Climbing cu cea mai abruptă ascensiune se va bloca în 86% din cazuri, rezolvând doar 14% din instanţele problemelor.
Pe de altă parte însă, algoritmul funcţionează destul de repede, având nevoie, în medie, de doar 4 paşi (pentru cazurile în care nu se blochează şi ajunge la final) sau 3 paşi (pentru situatiile în care se blochează) -> ceea ce nu e deloc rău, ţinând cont că spaţiul stărilor posibile este de $8^8 \sim 17$ milioane de stări.

Hill-Climbing cu restart aleator

- Se ghidează după zicala "Dacă nu reuşeşti din prima, mai încearcă"
- Face o serie de căutari Hill-Climbing din diferite stări iniţiale generate aleator până când găseşte o stare ţintă
- Dacă fiecare căutare Hill-Climbing are o probabilitate de succes p, atunci numărul de restarturi necesar este dat de raportul $1/p$
    - Pentru problema celor 8 dame descrisă mai devreme, când nu se permit mutări în laterale (deci mutările se pot face doar pe coloană), p are o valoare aproximativă de 0.14, deci este nevoie de 7 iteraţii (mutări) pentru a găsi starea ţintă (6 mutări defectuoase şi 1 mutare reuşită)
    - Numărul necesar de paşi pentru a ajunge la starea ţintă (damele să nu se atace între ele) este dat de costul unei mutări reuşite, la care se adaugă produsul dintre costul mutărilor defectuoase şi raportul $(1 - p) / p$ => în jur de 22 de paşi
    - Putem spune că pentru problema celor 8 dame, acest algoritm de căutare este unul foarte eficient
- Succesul algoritmului depinde foarte mult de forma spaţiului stărilor (dacă există doar câteva maxime locale sau platouri, algoritmul va gasi o soluţie foarte rapid)

Simulated annealing (normalizare simulată)

Un algoritm Hill-Climbing care nu face niciodată mutări de tip downhill spre stări vecine cu valori mai joase (sau cu cost mai mare) decât starea curentă este incomplet pentru că se blochează la un maxim local.
Ca şi contrast, o parcurgere în mod aleator (schimbarea stării curente cu cea a unui succesor ales aleator dintr-un set de succesori) este completă, dar ineficientă.
O soluţie rezonabilă ar fi combinarea acestor doi algoritmi în aşa fel încât să se ofere atât eficienţă, cât şi deplinătate. **Simulated annealing** este un astfel de algoritm.
Pentru a explica acest algoritm trebuie să trecem de la Hill-Climbing la o înclinaţie descrescătoare (gradient descent) şi să ne imaginăm că trebuie să aşezăm o minge de ping-pong în cea mai adâncă fisură dintr-o suprafaţă cu denivelări:

- Dacă doar lăsăm mingea să se rostogolească, aceasta va ajunge la un minim local
- Dacă scuturăm suprafaţa în discuţie, putem determina mingea să părăsească minimul local găsit mai devreme
- Scopul este de a scutura suprafaţa destul de tare încât să determinăm mingea să părăsească minimul local, dar nu foarte tare încât să "plece" şi din minimul global
- Soluţia propusă de algoritmul *simulated annealing* este de a începe cu o scuturare intensă a suprafeţei şi apoi reducerea treptată a intensităţii scuturării

Funcţia asociată acestui algoritm este prezentată în figura următoare:

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

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

- Bucla interioară a algoritmului descris mai sus este destul de similară cu cea a algoritmului Hill-Climbing
- În loc să se aleagă cea mai bună acţiune, se alege o mutare (mişcare) în mod aleator
- Dacă mutarea îmbunătăţeşte situaţia, atunci aceasta va fi întotdeauna acceptată -> altfel, algoritmul acceptă mutarea cu o oarecare probabilitate mai mică decât 1
- Probabilitatea descreşte exponenţial în funcţie de cât de "rea" este mutarea -> coeficientul ΔE măsoară variaţia cu care s-a înrăutăţit evaluarea
- Probabilitatea este influenţată şi de parametrul T
    - T poate fi asociat cu mişcarea de scuturare imaginată mai devreme (cazul mingii de ping-pong)
    - T determină probabilitatea de a alege o acţiune mai rea
    - Cu cât T este mai mare, cu atât acţiunile mai rele au şanse mai mari să fie alese
    - Daca T scade suficient de încet, algoritmul va găsi un optim global cu o probabilitate apropiată de 1.