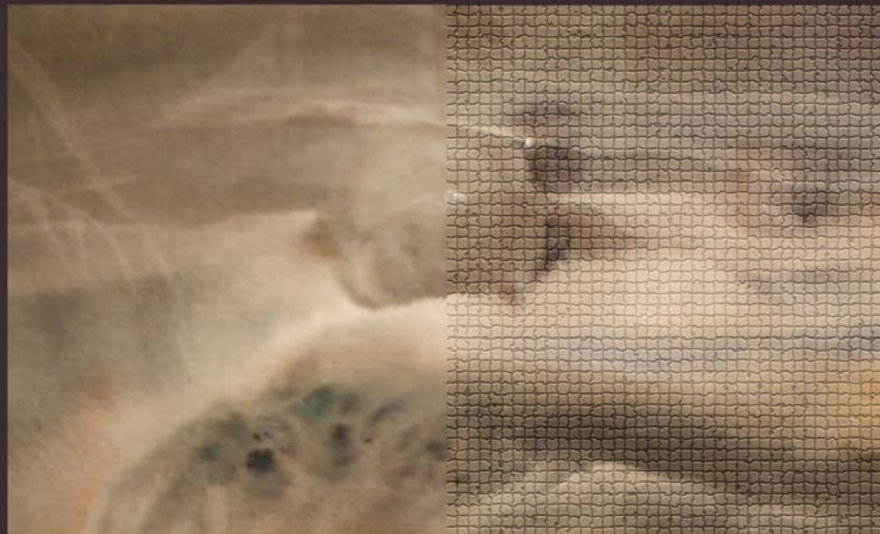
BOGDAN GROZA



INTRODUCERE ÎN INTELIGENȚĂ ARTIFICIALĂ
APLICAȚII CU STRATEGII DE CĂUTARE NEINFORMATE ŞI INFORMATE

Coperta: după lucrarea „Astrul Dimineții" de Virginia Baz-Baroiu

Cartea „Introducere în inteligenţă artificială – Aplicaţii cu strategii de căutare neinformate şi informate” scrisă de dl. dr. ing. Bogdan Groza abordează ca subiect rezolvarea problemelor prin strategii de căutare, oferind totodată o manieră de evaluare comparativă a performanţelor acestor strategii. Lucrarea are un pronunţat caracter aplicativ, materialul fiind organizat într-o structură tipică lucrărilor de laborator. Deşi lucrarea se adresează în principal stundenţilor, maniera cursivă şi concisă în care este scrisă oferă o lectură plăcută şi utilă oricărui cititor interesat de subiect. În plus, doresc să subliniez rigoarea ştiinţifică a materialului prezentat în lucrare, rezultată în mod firesc din experienţa deosebită de cercetător a autorului.

Referent ştiinţific: şef lucrări dr.ing. Dorina Petrică

INTRODUCERE ÎN INTELIGENȚĂ ARTIFICIALĂ – APLICAȚII CU STRATEGII DE CĂUTARE NEINFORMATE ŞI INFORMATE

INTRODUCERE ÎN INTELIGENȚĂ ARTIFICIALĂ – APLICAȚII CU STRATEGII DE CĂUTARE NEINFORMATE ŞI INFORMATE

Bogdan Groza[1]

[1] Universitatea Politehnica Timişoara, Facultatea de Automatică şi Calculatoare, Bd. Vasile Pârvan nr. 2, Room A304, 300223, Timişoara, România, e-mail: bogdan.groza@aut.upt.ro, web: www.aut.upt.ro/~bgroza.

CUPRINS

LUCRAREA 1 INTRODUCERE ÎN PROBLEMATICA INTELIGENŢEI ARTIFICIALE 8

1.1 AGENTUL INTELIGENT, CAPACITATEA DE A REZOLVA PROBLEME 8
1.2 FORMULAREA UNEI PROBLEME 10
1.3 ETAPELE REZOLVĂRII UNEI PROBLEME 11
1.4 EFICIENŢA REZOLVĂRII UNEI PROBLEME MĂSURI CALITATIVE ŞI CANTITATIVE ÎN EVALUAREA STRATEGIILOR DE CĂUTARE 12
1.5 IMPLEMENTAREA GENERALĂ A STRATEGIILOR INFORMATE ŞI NEINFORMATE 13
1.6 EXERCIŢII 14

LUCRAREA 2 STRATEGII NEINFORMATE DE CĂUTARE: STRATEGIILE DE CĂUTARE PE NIVEL ŞI BIDIRECŢIONALĂ 17

2.1 STRATEGII DE CĂUTARE ŞI CARACTERISTICI ALE ACESTORA 17
2.2 STRATEGIA DE CĂUTARE PE NIVEL (BREADTH-FIRST) 17
2.3 STRATEGIA DE CĂUTARE BIDIRECŢIONALĂ (BIDIRECTIONAL SEARCH) 20
2.4 EXERCIŢII 22

LUCRAREA 3 EVITAREA CICLURILOR ÎN ALGORITMII DE CĂUTARE 31

3.1 EXERCIŢII 31

LUCRAREA 4 STRATEGII NEINFORMATE DE CĂUTARE: STRATEGIILE DE CĂUTARE ÎN ADÂNCIME, ADÂNCIME LIMITATĂ ŞI ADÂNCIME ITERATIVĂ ... 34

4.1 STRATEGIA DE CĂUTAREA ÎN ADÂNCIME (DEPTH-FIRST) 34
4.2 STRATEGIA DE CĂUTARE LIMITATĂ ÎN ADÂNCIME (DEPTH-LIMITED) 35
4.3 STRATEGIA DE CĂUTARE ITERATIVĂ ÎN ADÂNCIME (ITERATIVE-DEEPENING) 36
4.4 EXERCIŢII 38

LUCRAREA 5 STRATEGII NEINFORMATE DE CĂUTARE: STRATEGIA DE CĂUTARE CU COST UNIFORM 42

5.1 STRATEGIA DE CĂUTARE CU COST UNIFORM (UNIFORM COST) 42
5.2 EXERCIŢII 44

LUCRAREA 6 STRATEGII INFORMATE DE CĂUTARE: STRATEGIILE DE CĂUTARE BEST FIRST ŞI GREEDY 54

6.1 CE ESTE O EURISTICĂ 54
6.2 STRATEGIA DE CĂUTARE BEST FIRST 55
6.3 STRATEGIA DE CĂUTARE GREEDY 55
6.4 EXERCIȚII 56

LUCRAREA 7 ALEGEREA UNEI EURISTICI 60

7.1 PROBLEME CU CONSTRÂNGERI 60
7.2 ROLUL UNEI EURISTICI 60
7.3 EXERCIȚII 61

LUCRAREA 8 STRATEGII INFORMATE DE CĂUTARE: STRATEGIA DE CĂUTARE A* 62

8.1 DEMONSTRAȚIE ASUPRA OPTIMALITĂȚII LUI A* 63
8.2 EXERCIȚII 63

LUCRAREA 9 STRATEGII DE CĂUTARE ÎN SPAȚII CU INCERTITUDINI 68

9.1 EXERCIȚII 80

LUCRAREA 10 ÎNTREBĂRI RECAPITULATIVE ŞI PROBLEME 81

Cuvânt înainte

Prezentul material este destinat pentru a fi utilizat în cadrul activității de laborator de către studenții din anul IV ai Facultății de Automatică şi Calculatoare din Universitatea "Politehnica" din Timişoara care urmează materia Bazele Inteligenței Artificiale dar poate fi desigur folosit de oricine îl consideră util. Lucrările conținute reprezintă doar o parte din lucrările de laborator efectuate în perioada 2005-2008 cu studenții anului 4 Automatică la materia Inteligență Artificială, lucrări care vor fi pe de o parte continuate cu studenții anului 4 Ingineria Sistemelor la materia Bazele Inteligenței Artificiale. Aceste lucrări reprezintă doar prima parte a laboratorului de Inteligență Artficială ținut pentru promoțiile de 5 ani, a doua parte a adresat problemele de logică (implementare în Prolog) şi rețele neuronale. Este posibil ca prezentul material să fie extins şi cu lucrările aferente acestei a doua părți în funcție de interesul acordat disciplinei de către studenți şi posibilitatea activării sale, acum materia având caracter opțional. Materialul poate fi accesat on-line la www.aut.upt.ro/~bgroza/iia.pdf.

Timişoara,
Noiembrie 2008

Bogdan Groza

Lucrarea 1 Introducere în problematica Inteligenței Artificiale

1.1 Agentul inteligent, capacitatea de a rezolva probleme

Russel şi Norvig, în binecunoscuta lor carte „Artificial Intelligence a Modern Approach”, devenită referință de nelipsit pentru orice curs în domeniu, definesc inteligența artificială ca fiind studiul agenților existenți într-un mediu care percep şi acționează.

Agentul poate fi perceput ca o entitate care percepe mediul prin intermediul senzorilor şi acționează asupra lui prin intermediul efectorilor. De asemenea un agent are o arhitectură, pe care o asociem părții fizice a acestuia, numită hardware, şi un program de funcționare, care îl numim software. Acest lucru este ilustrat în figura 1.1. De remarcat că acest concept nu adresează doar agenți de natură fizică tip robot ci chiar şi agenți de natură virtuală (sub formă de program, algoritm etc.) care au şi ei o componentă hardware pe care funcționează. Partea software, programul, este componenta care păstrează natura inteligentă.

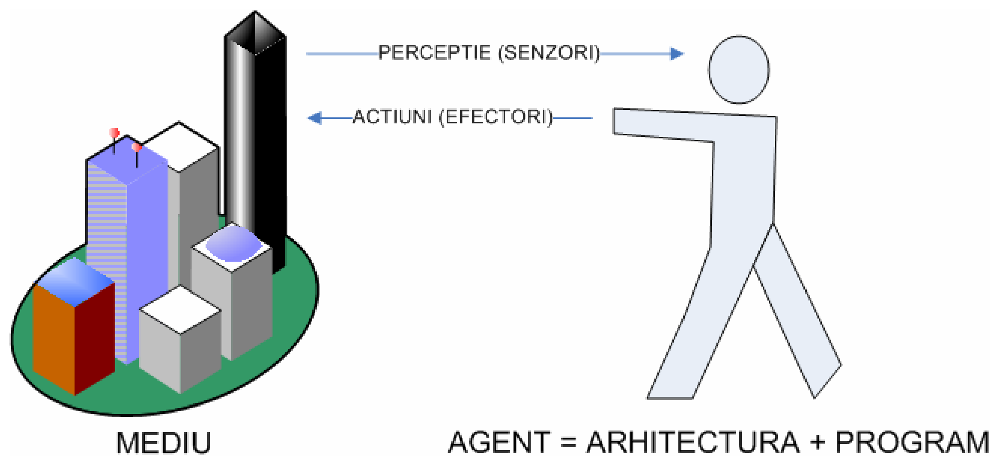


Figura 1.1 *Agentul inteligent în interacțiunea sa cu mediul*

Aşadar obiectivul Inteligenței Artificiale este construirea Agenților Inteligenți. Întrebarea este însă ce ne face să numim un agent ca fiind inteligent. Răspunsul la această întrebare nu este simplu şi de-a lungul anilor au fost dezvoltate diverse potențiale răspunsuri. Poate cel mai cunoscut criteriu pentru a demonstra că o maşină

este inteligentă este testul Turring ilustrat în figura 1.2, descris de Alan Turing în 1950 în lucrarea "Computing Machinery and Intelligence". În acest test, un participant A discută pe o linie cu o maşină de calcul B (agentul inteligent) şi pe altă linie cu un alt participant uman C. Toți participanții la comunicare sunt în incinte izolate fără contact cu mediul, iar la final participantul A trebuie să poată face distincția între B şi C, care este maşină şi care este om. Dacă nu se poate face această distincție maşina a trecut testul şi este inteligentă.

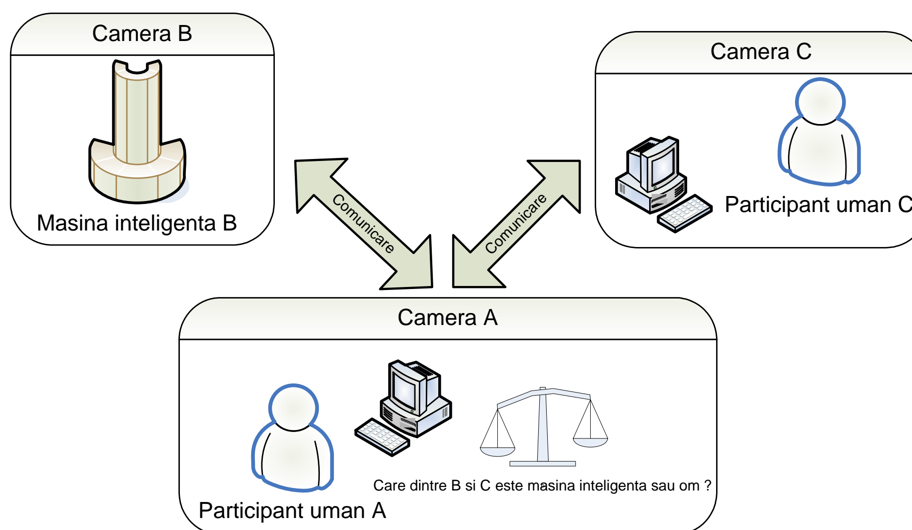


Figura 1.2 *Test Turing*

Sigur, trecerea acestui test este un deziderat greu de atins. Pentru contextul de față vom opta pentru un răspuns mai simplu, care chiar dacă mai redus ca profunzime şi complexitate, nu este lipsit de semnificație. Vom defini un agent inteligent ca fiind o entitate capabilă de a rezolva probleme. Într-adevăr, capacitatea de a rezolva probleme este o dovadă de inteligență. Mai mult, chiar şi noi, ne-am format inteligența încă din cele mai timpurii stagii, rezolvând probleme.

Dar ce înseamnă a rezolva o problemă? Indiferent de problema adresată şi de mediul de care aparține, chiar este recomandabil să gândim acest lucru nu în contextul agenților inteligenți artificiali ci al vieții noastre de zi cu zi, rezolvarea unei probleme poate fi întotdeauna văzută ca fiind echivalentă cu capacitatea de a lua nişte decizii, deci orice problemă este o problemă decizională în cele din urmă. Exemplele sunt desigur numeroase, de exemplu a juca şah înseamnă a putea lua în mod corect o decizie asupra piesei care trebuie mutată, sau a trece strada înseamnă a lua în mod corect o decizie cu privire la momentul în care trebuie să facem un anume pas etc.

Acest lucru ne duce în cele din urmă la a defini un agent inteligent ca fiind o entitate capabilă de lua decizii în scopul rezolvării unei probleme. Pentru a accentua caracterul inteligent al luării de decizii este de dorit ca entitățile inteligente să poată

evalua efectul decizilor (plan-ahead) iar această evaluare se face în baza informaților disponibile agentului, de cele mai multe ori acestea fiind însă incomplete (altfel spus, entitățile inteligente nu sunt tot timpul omnisciente).

1.2 Formularea unei probleme

Trebuie acum să lămurim ce înseamnă din punct de vedere formal rezolvarea unei probleme. Înainte de a rezolva o problemă aceasta trebuie formulată. Formularea sau descrierea unei probleme, indiferent de natura ei, se poate face în baza a trei noțiuni:

- Stare inițială (SI) – starea din care începe problema.
- Stare finală (SF) – starea în care se termină problema.
- Operatorii de generare a stărilor - acțiunile care pot fi întreprinse.

Se impune observația că starea finală poate fi specificată în mod direct, adică explicit printr-o valoare, sau indirect, printr-o funcție de test. Ca exemplu putem considera problema în care un agent trebuie să se deplaseze din punctul A în punctul B. În acest caz starea finală este explicit dată ca fiind punctul B. Pentru cel de-al doilea caz putem considera altă problemă, de exemplu jocul de şah. În acest caz starea finală nu este explicit specificată, pentru că starea finală este cea de şah mat, dar cum arată şah mat? Evident acest lucru nu poate fi ştiut la începutul jocului şi din acest motiv trebuie să existe o funcție de test cu ajutorul căreia la fiecare pas se poate testa dacă starea atinsă este şah mat. De asemenea pot fi una sau mai multe stări finale valide care rezolvă problema. În general specificarea explicită a unei stări finale şi unicitatea acesteia poate fi folosită ca avantaj în rezolvarea unei probleme deoarece se pot aplica strategii de căutare ceva mai eficiente ca timp şi spațiu precum căutarea bidirecțională aşa cum se va vedea în capitolul 2.

Spațiul stărilor este totalitatea de stări în care se poate ajunge, aplicând succesiv operatori pe stările nou rezultate din starea curentă. Putem de asemenea defini spațiul stărilor ca fiind ansamblul de stări şi tranziții posibile între acestea, acest lucru creând o imagine de tip graf asupra problemei (o astfel de imagine este în general un bun punct de plecare în rezolvarea problemelor).

Prin drum sau cale în spațiul stărilor înțelegem orice secvență de stări legate prin operatori în spațiul starilor. Fiecare drum în spațiul stărilor are un cost, care este evaluat de o funcție care o vom nota cu g şi care asociază unui drum un cost. Costul drumului prin spațiul stărilor de la starea inițială la starea curentă se mai numeşte şi costul stării sau al nodului, în momentul în care discutăm de arbori de căutare.

În acest context soluția unei probleme este drumul prin spațiul stărilor de la starea inițială la starea finală. Trebuie evitată confuzia că starea finală este soluția problemei, soluția problemei este drumul până la starea finală şi nu starea finală în sine. Acum putem defini foarte simplu şi ce presupune rezolvarea unei probleme: rezolvarea unei probleme este căutarea unei căi de la starea inițială la starea finală a

problemei şi deci echivalează cu găsirea unei căi în spațiu stărilor de la starea inițială la starea finală.

Poate fi util să ne oprim asupra unui exemplu ilustrativ. Una dintre problemele care vor fi studiate în acest material, datorită simplității dar şi a puterii de exemplificare este puzzleul *3x3*, ilustrat în figura 1.3, ce poate fi generalizat sub forma *nxn*. Pentru acesta putem extrage următoarele caracteristici:

- Stare inițială: orice aranjare a pieselor din careu care este punctul de plecare al problemei.
- Stare finală: orice aranjare prestabilită a pieselor.
- Operatori: toate mutările posibile ale spațiului liber: stânga, dreapta, sus, jos (desigur, e vorba de mutarea fizică a pieselor din jurul acestuia)
- Rezolvare: găsirea unui drum prin spațiul stărilor de la starea inițială către starea finală prin aplicarea operatorilor (deci mutările care trebuie făcute pentru a ajunge de la configurația care se dă la configurația care se cere)
- Spațiul stărilor: numărul de variante posibile pentru a aranja piesele pe tablă, anume permutări de 9 care înseamnă 9!= 362880 stări

Se impune însă o observație esențială, şi anume, că de fapt sunt mult mai puține stări decât 9! pentru că există şi poziții în care nu se poate ajunge, de exemplu orice poziție care s-ar obține prin interschimbarea a două piese alăturate. Din acest motiv dimensiunea spațiului stărilor o vom trata sub indicatorul de complexitate *O* astfel putem spune în mod riguros că dimensiunea spațiului stărilor pentru puzzleul de *n* piese este *O(n!)* prin aceasta indicând în fapt doar o limită superioară asupra numărului de stări ale problemei.

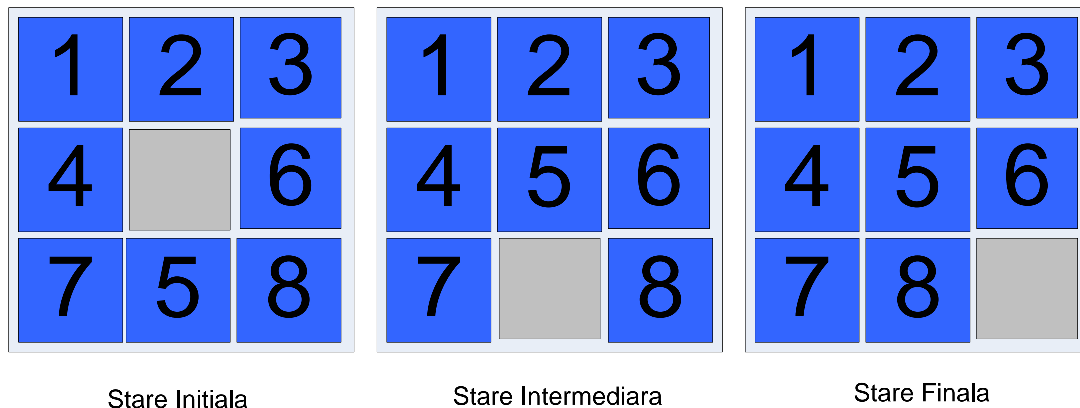


Figura 1.3 *Exemplu de puzzle 3x3*

1.3 Etapele rezolvării unei probleme

În baza preambulului făcut, putem enumera următorii paşi care trebuie să îi urmăm în rezolvarea unei probleme:

1) Abstractizarea - înseamnă eliminarea detaliilor inutile şi păstrarea elementelor esențiale rezolvării problemei
2) Determinarea stării inițiale
3) Determinare stării finale
4) Extragerea operatorilor (acțiunilor posibile)
5) Pornirea unei strategii de căutare în spațiul stărilor

Este de remarcat că paşii 1,2,3 şi 4 țin de formularea problemei în timp ce pasul 6 reprezintă aplicarea strategiei de căutare şi rezolvarea efectivă a problemei. Generarea unor stări noi se face prin aplicarea operatorilor asupra stării curente în timp ce alegerea efectivă a stării care urmează să fie explorată (expandată) este determinată de strategia de căutare folosită.

Odată cu pornirea unei strategii de căutare se ajunge implicit la generarea spațiului stărilor şi mai exact se va ajunge la o structură ce poate fi asociată unui Arbore de Căutare. Este de remarcat că arborele de căutare nu trebuie să apară explicit ca structură de date, ci acesta poate fi simulat de către algoritmul de căutare. Aşadar structura de date asociată în mod concret sau virtual unei strategii de căutare este arborele de căutare. În mod uzual asociem următoarele informații unui nod al arborelui de căutare:

- Starea – starea căreia nodul îi este asociat
- Părintele nodului – nodul care a generat nodul curent
- Operator – operatorul prin care a fost generat nodul
- Adâncime – numărul de nivele până la acest nod
- Cost – costul drumului de la nodul rădăcină la nodul curent
- Euristica – valoare estimativă a distanței până la starea finală

Este important de observat că arborele de căutare nu este acelaşi lucru cu spațiul stărilor, de exemplu Arborele de Căutare poate avea un număr infinit de stări chiar dacă Spațiul Starilor este finit în cazul în care strategia de căutare are cicluri (aspect discutat în capitolul 3). De asemenea un nod dintr-un Arbore de Cautare corespunde unei stări dar nu este echivalent cu aceasta deoarece de exemplu un nod are un singur nod părinte în timp ce într-o stare se poate ajunge din mai multe stări şi în acest caz o stare poate avea mai multe stări părinte.

1.4 Eficiența rezolvării unei probleme măsuri calitative şi cantitative în evaluarea strategiilor de căutare

La alegerea unei strategii de căutare pentru rezolvarea unei probleme, evaluarea eficienței rezolvării problemei poate fi pusă din diverse puncte de vedere. De exemplu iată câteva întrebări orientative: Căutarea ajunge la o soluție? Soluția găsită a fost cea mai bună? Soluția a fost furnizată în timp util? Care au fost costurile găsirii

acestei soluții? De cele mai multe ori, strategia de căutare aleasă poate fi un compromis între calitatea soluției (costul căii Stare inițială – Stare finală) şi costul căutarii (timp, memorie, costuri de implementare etc.).

Pe lângă astfel de întrebări orientative, există măsuri cantitative şi calitative exacte pentru a estima eficiența unei strategii de căutare. Acestea sunt următoarele:

- Măsuri cantitative:
  - Complexitatea de timp a strategiei T - în cât timp (măsurat ca număr de paşi ai algoritmului) este găsită soluția.
  - Complexitatea de spațiu a strategiei S – de cât spațiu (este vorba de memorie) este nevoie pentru găsirea soluției.
- Măsuri calitative:
  - Completitudine – o strategie se numeşte completă dacă garantează găsirea unei soluții.
  - Optimalitate – o strategie se numeşte optimală dacă garantează găsirea celei mai bune soluții din punctul de vedere al costului căii între starea inițială şi starea finală.

Trebuie precizat că în timp ce complexitățile de timp şi spațiu sunt aprecieri pur cantitative, măsurate cu ajutorul indicatorului de complexitate *O*, completitudinea este întotdeuna o proprietate binară: o strategie este sau nu este completă. În ceea ce priveşte optimalitatea, aceasta este în general tot o proprietate binară: o strategie este sau nu este optimală, dar, există şi strategii, numite sub-optimale, care nu găsesc cea mai bună soluție dar găsesc o soluție foarte apropiată ca şi cost de cea mai bună soluție (de exemplu A* atunci când nu folosim euristici admisibile, un astfel de caz va fi discutat în secțiunea 8).

1.5 Implementarea generală a strategiilor informate şi neinformate

Toate strategiile de căutare neinformate şi informate discutate în acest material urmează aceeaşi paradigmă generală în implementare. Pe lângă paşii anterior amintiți de abstractizare, extragere a stărilor inițială şi finală, respectiv a operatorilor, trebuie să adăugăm şi pasul în care se realizează căutarea efectivă pe care îl vom denumi funcție de căutare (funcție search). Acest lucru este ilustrat în figura 1.4.

În ceea ce priveşte funcția search aceasta are ca şi componentă de bază o buclă în care se realizează succesiv: extragerea noului nod successor, testarea dacă acest nod corespunde stării finale, adăugarea succesorilor nodului curent în lista utilizată pentru memorarea stărilor. Lista de noduri este cea care face diferența între diversele strategii de căutare, aceasta fiind stivă, coadă, listă ordonată după euristică etc. Pseudocodul funcției search este următorul:

```
funcție search {
nod_curent.stare = stare_inițială
adaugă nod_curent în lista_noduri
        repetă la infinit {
         dacă lista_noduri este vidă atunci return(failure)
```

nod_curent = extrage_nod(listă_noduri)
dacă nod_curent.stare=stare_finală atunci soluție()
adaugă în lista_noduri succesori(nod_curent)
}
} sfârşit funcție

1.6 Exerciții

1. Se consideră un puzzle 3x3 (asemenea celui din figura 2) şi un tablou cu leduri de dimensiune 3x3 la care prin apăsarea unui led acesta îşi schimbă starea proprie din stins în aprins (şi invers) precum şi a ledurilor aflate în stânga, dreapta, sus, jos față de acesta. În starea inițială toate ledurile sunt stinse şi se doreşte o stare finală cu toate ledurile aprinse. Se cere:

a) Care este dimensiunea spațiului starilor pentru cele 2 probleme?
b) Care sunt operatorii asociați?
c) Desenați arborele de căutare.

2. Una dintre cele mai vechi probleme abordate de strategiile de căutare este problema canibalilor şi misionarilor. Trei canibali şi trei misionari se află pe marginea unui râu, ştiind că barca de care dispun nu poate lua decât 2 persoane şi pe nici unul dintre țărmuri nu pot fi mai mulți canbali decât misionari să se găsească o soluție pentru ca aceştia să traverseze râul (în figura 1.5 este sintetizat arborele de căutare pentru această problemă).

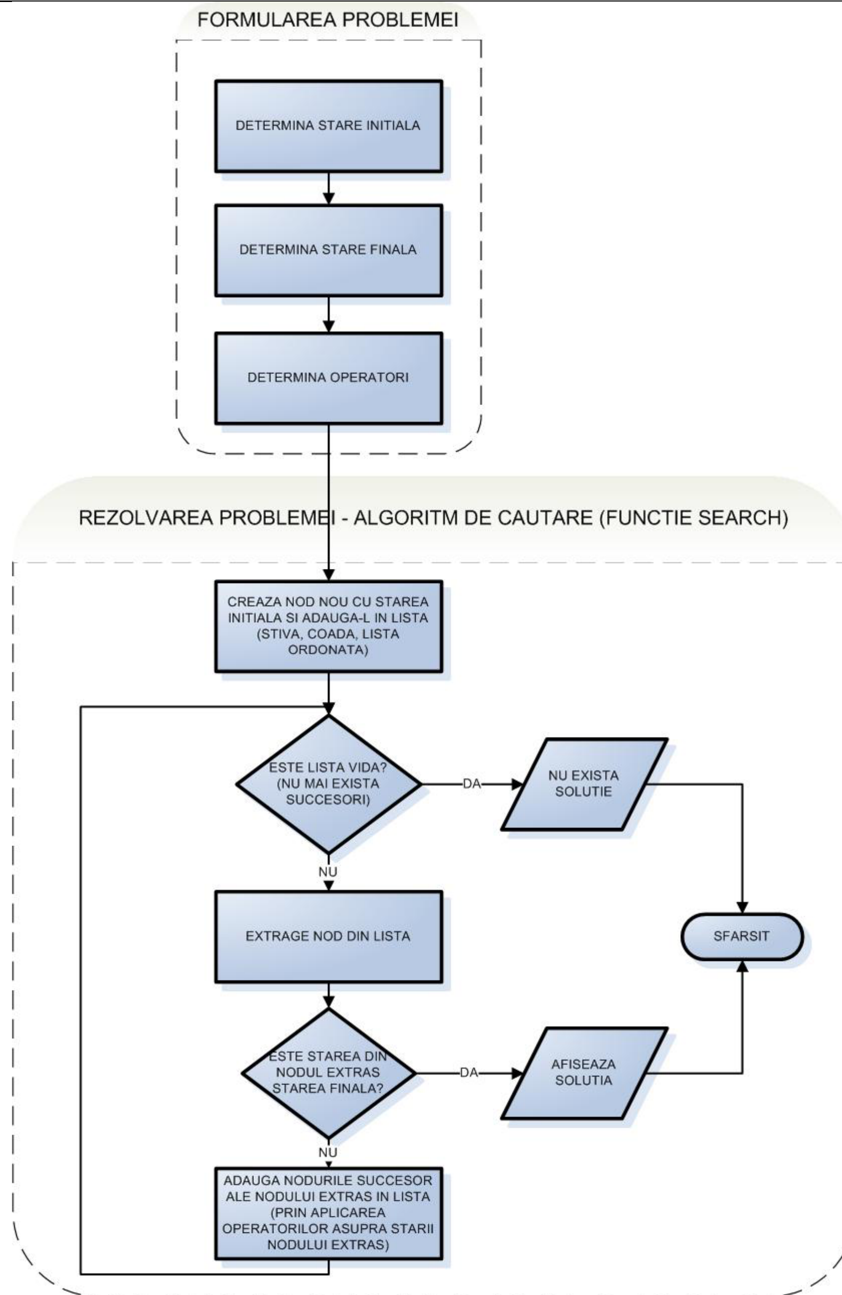


Figura 1.4. *Abordarea generală a unei probleme folosind o strategie de căutare*

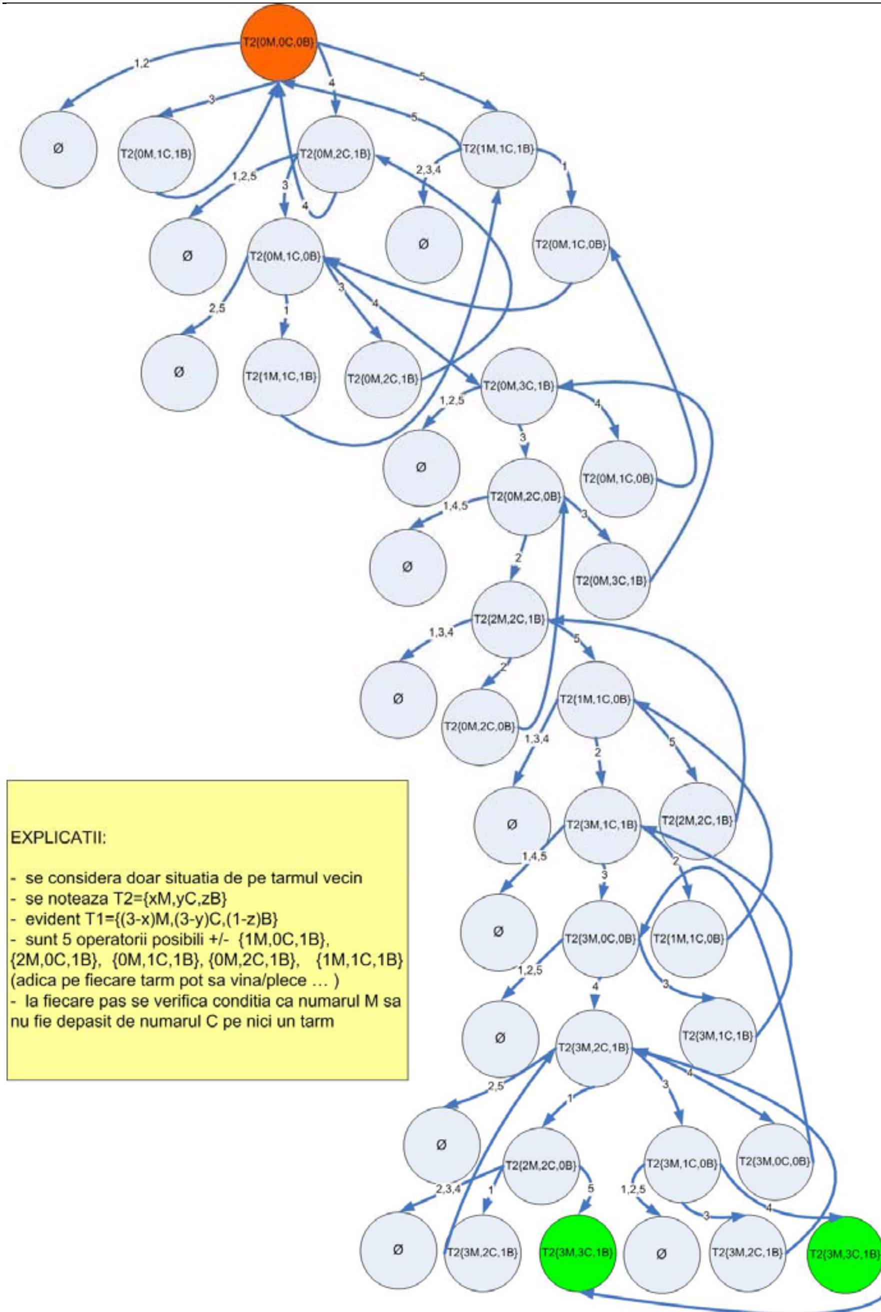


Figura 1.5. *Arborele de căutare pentru problema canibalilor şi misionarilor*

Lucrarea 2 Strategii neinformate de căutare: strategiile de căutare pe nivel şi bidirecțională

2.1 Strategii de căutare şi caracteristici ale acestora

Strategiile de căutare care nu folosesc euristici pentru a alege nodul successor se mai numesc şi strategii de căutare neinformate sau oarbe (blind-search). Şapte strategii de căutare neinformate sunt abordate în acest material:

- Strategia de căutare pe nivel (Breadth-First)
- Strategia de căutare în adâncime (Depth-First)
- Strategia de căutare cu cost uniform (Uniform Cost)
- Strategia de căutare limitată în adâncime (Depth-Limited)
- Strategia de căutare iterativă în adâncime (Iterative-Deepening)
- Strategia de căutare bidirecțională (Bidirectional search)

Cu privire la toate acestea, inclusiv strategiile informate ce vor urma, interesează aflarea răspunsului la următoarele întrebări:

a) Care este complexitatea de timp a strategiei?
b) Care este complexitatea de spațiu a strategiei?
c) Este strategia completă?
d) Este strategia optimală?
e) Cum se implementează?
f) Ce avantaje şi dezavataje are?

Răspunsul la întrebările de la a) la f) va fi dat în cele ce urmează în cazul particular al fiecărei strategii în parte. În ceea ce priveşte întrebarea e) aceasta se referă la tipul structurii de date în care se rețin nodurile arborelui de căutare. Aşa cum se va observa este vorba de structuri de tip coadă, stivă sau liste sortate. În ceea ce priveşte întrebarea de la f) aceasta presupune o analiză în baza caracteristicilor proprii de la a) la e) dar şi o analiză comparativă cu celelalte strategii.

2.2 Strategia de căutare pe nivel (Breadth-First)

Cea mai simplă strategie de căutare este căutarea pe nivel, numită şi breadth-first. Această strategie explorează nodurile în ordinea nivelelor, altfel spus nodurile de pe nivelul d sunt explorate înaintea nodurilor de pe nivelul $d+1$. Acest aspect este ilustrat în figura 2.1, doar pentru simplitatea figurii s-a optat pentru un arbore binar, deci există doar doi operatori, altfel numărul de operatori poate fi variabil.

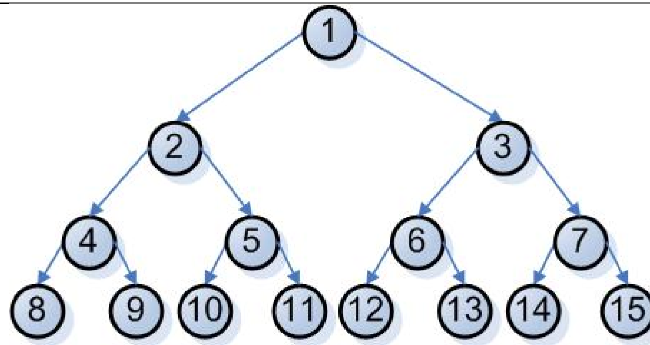


Figura 2.1 *Arborele de căutare în cazul strategiei breadth-first*

- ***Complexitate de timp şi spațiu***

Complexitatea de timp a strategiei poate fi simplu calculată ca $T = 1 + b + b^2 + ... + b^d = O(b^d)$, unde d este adâncimea primei soluții a problemei (depth), iar b este factorul de ramificație al problemei (branching factor).

Factorul de ramificație maxim este egal cu numărul de operatori. Trebuie însă observat că nu toți operatorii pot fi aplicați pe orice stare şi din acest motiv factorul de ramificație nu este tot timpul egal cu numărul de operatori. Astfel putem avea un factor de ramificație minim, maxim şi mediu. Dacă luăm ca exemplu un puzzle 3x3 aşa cum se poate observa şi în figura 2.2 factorul de ramificație maxim este 4 şi minim 2 dar în medie va fi 3. Se poate defini deasemenea un factor de ramificație efectiv. Acesta se calculează pe cale experimentală şi este rezultatul ecuației $n = 1 + b_e + b_e^2 + ... + b_e^d \Rightarrow b_e \approx \sqrt[d]{n}$ unde n este numărul de noduri explorate obținut pe cale experimentală.

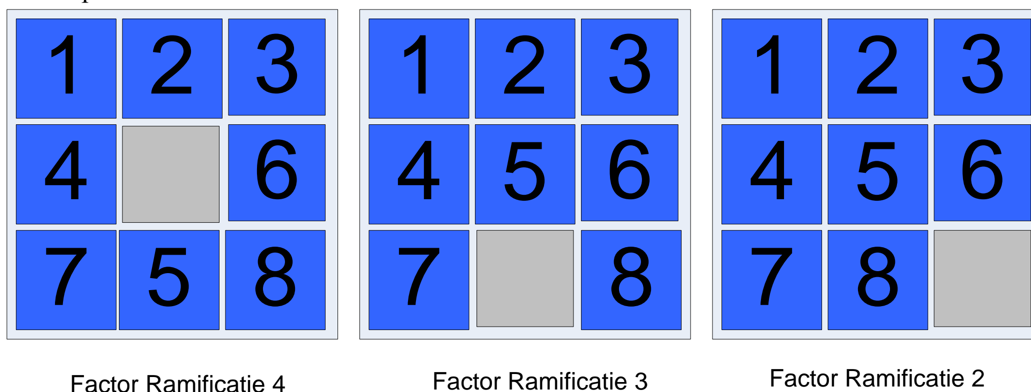


Figura 2.2 *Factori de ramificație variabili pe problema puzzleului 3x3*

Complexitatea de spațiu este $S = b^d = O(b^d)$ deoarece strategia reține tot timpul doar nodurile de pe ultimul nivel al arborelui de căutare.

➢ *Completitudine şi optimalitate*

Aşa cum se poate observa strategia de căutare pe nivel este completă atunci când numărul de operatori este finit deoarece parcurge în mod exhaustiv toate nodurile arborelui de căutare.

Strategia este însă optimală doar dacă costul creşte proporțional cu adâncimea, deci dacă orice nod este mai scump decât orice alt nod aflat pe un nivel deasupra lui. Altfel spus, dacă pentru oricare două noduri: $\forall x, y, D(x) < D(y) \Rightarrow g(x) < g(y)$ unde *D* este funcția care returnează adâncimea nodului şi *g* reprezintă costul. Deci nici un nod de pe nivelul *d+1* nu poate fi mai ieftin decat un nod de pe nivelul *d* , această condiție este în particular satisfacută când costul operatorilor este egal, lucru valabil pentru multe probleme întâlnite în practică, inclusiv problema puzzleului 3x3 anterior aminitită.

➢ *Implementare*

Strategia breadth-first poate fi uşor implementată folosind o structură de tip coadă (queue) în care la fiecare pas se extrage din coadă nodul curent şi se adaugă la sfârşit nodurile rezultate din explorarea nodului curent.

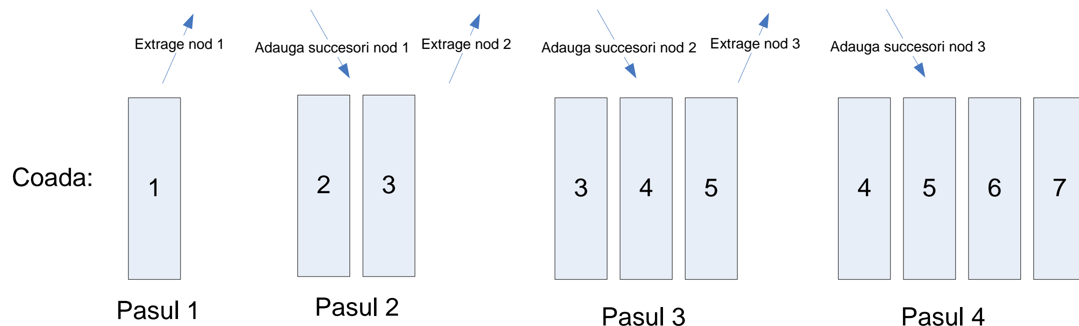


Figura 2.3. *Evoluția cozii la strategia de căutare Breadth First*

➢ *Avantaje şi dezavantaje*

Avantajul major este faptul că strategia este completă, mai mult este chiar şi optimală în condițiile anterior menționate.

Dezavantajul major este faptul că are necesități de memorie mult prea mari, creşterea spațiului fiind tot exponențială. În general spațiul este o problemă mult mai mare decât 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 nicioadată un rezultat. Tabelul 2.1 ilustrează acest lucru pe o creştere exponențială de timp şi memorie.

Adâncime	Timp	Memorie
16	0.06 secunde	64 KB
32	1.19 ore	4 GB
40	12 zile	1 TB
56	2284 ani	64×10^{6} TB

Tabel 2.1. *Creşterea necesităților de timp şi spațiu pentru un algoritm de complexitate timp şi spațiu* $O(2^n)$ *pentru cazul în care dimensiunea unei stări este 1 octet iar durata unui pas al algoritmului* 10^{-6} *secunde.*

2.3 Strategia de căutare bidirecțională (Bidirectional search)

O potențială îmbunătățire la adresa strategiei de căutare pe nivel poate fi adusă prin pornirea a două căutări simultane: una de la starea inițială către starea finală şi una în sens invers. Acest lucru duce în mod evident la înjumătățirea resurselor de timp şi memorie necesare având singura cerință suplimentară de a păstra o listă a frontierelor pentru cele două căutări în care să se verifice periodic dacă nu există noduri care coincid, figura 2.4 ilustrează acest lucru. Această strategie poartă numele de strategia de căutare bidirecțională şi menționăm încă de acum că pentru economie de memorie pe una dintre direcții poate fi aplicată şi altă strategie precum strategia de căutare în adâncime ce va fi discutată în secțiunea următoare.

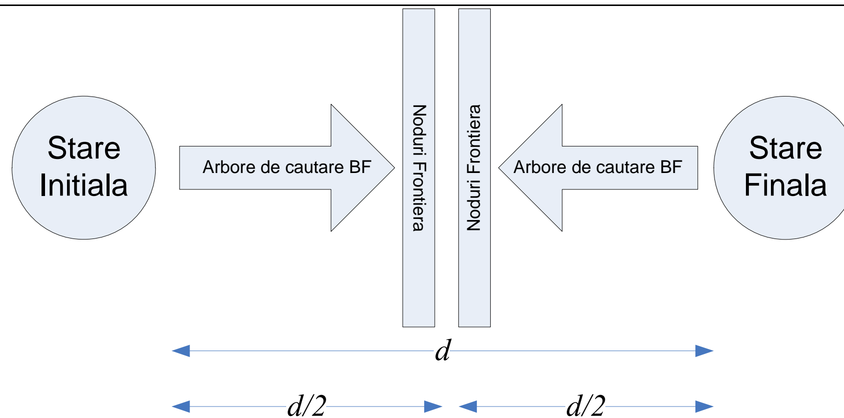


Figura 2.4. *Strategia de căutare bidirecțională*

➢ ***Complexitate de timp şi spațiu***

Deoarece adâncimea arborelui de căutare se înjumătățeşte, complexitățile de timp şi spațiu vor deveni $T = 2 \cdot \left(1 + b + b^2 + ... + b^{d/2}\right) = O\left(b^{d/2}\right)$ respectiv $S = 2 \cdot b^{d/2} = O\left(b^{d/2}\right)$. Acest lucru este sugerat şi în figura 2.4. La complexitatea de timp trebuie adăugat şi timpul de calcul necesar pentru verificarea coliziunilor între cele două frontiere. Deoarece acest lucru implică o căutare binară, sub indicatorul O complexitatea nu se schimbă, mai exact însă timpul de calcul va fi $T = 1 + b + b^2 ... + b^d + 1 + b\log_2 b + b^2 \log_2 b^2 + ... + b^{d/2} \log_2 b^{d/2} = O\left(b^{d/2}\right)$ în cazul în care presupunem că la fiecare nod nou explorat se verifică dacă acesta nu cumva există în lista de frontiere a căutării pornite din cealaltă parte.

➢ ***Completitudine şi optimalitate***

Căutarea bidirecțională este completă şi optimală după cum sunt cele două strategii care le implică. Astfel pentru cazul în care se foloseşte breadth-first, este completă tot timpul (atunci când factorul de ramificație este finit) şi este optimală pe toate cazurile unde este şi breadth-first optimală.

➢ ***Implementare***

Se implementează cele două strategii de căutare, astfel se porneşte cu breadth-first simultan de la starea inițială şi de la starea finală. Pe lângă aceasta se păstrează şi două liste ale frontierelor între care se verifică periodic dacă nu există coliziuni. Se

poate alege breadth-first dintr-o parte şi o căutare diferită din cealaltă parte, atenție însă la riscurile ca nodurile din cele două frontiere să nu coincidă.

- ***Avantaje şi dezavantaje***

Avantajul major este că reduce semnificativ costurile căutării Breath First, dar mai mult faptul că este mult mai rapidă decât orice altă strategie neinformată. Ca dezavantaj rămâne consumul de memorie care este tot exponențial. 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.

2.4 Exerciții

1. Doi frați au un vas de 8 litri cu "apă" şi doresc sa îl împartă în mod egal. Ei mai dispun de două vase de 3 respectiv 5 litri goale. Se cere:

a) abstractizați starea inițiala şi finală

SI={R_1=0,R_2=0,R_3=8} SF={R_1=0,R_2=4,R_3=4}

b) extrageți operatorii

1. $R_3 \rightarrow R_2$
2. $R_3 \rightarrow R_1$
3. $R_2 \rightarrow R_1$
4. $R_2 \rightarrow R_3$
5. $R_1 \rightarrow R_2$
6. $R_1 \rightarrow R_3$

$\rightarrow$ semnifică toarnă conținutul lui R_i în R_j până când R_i este gol sau R_j este la capacitate maximă

c) Propuneți o strategie de căutare optimală şi completă şi precizați complexitatea acesteia la o adâncime *d=8*

Breadth-first: $S=T=O(b^d)=6^8$

d) Căutați soluția folosind strategia de la c) (se va evita explorarea stărilor care deja au mai fost întâlnite)

	1	2	3	4	5	6

{0,0,8}	{0,5,3}	{3,0,5}	Ø	Ø	Ø	Ø
{0,5,3}	Ø	{3,5,0}	{3,2,3}	{0,0,8}[r]	Ø	Ø
{3,0,5}	{3,5,0}[r]	Ø	Ø	Ø	{0,3,5}	{0,0,8}[r]
{3,5,0}	Ø	Ø	Ø	{3,0,5}[r]	Ø	{0,5,3}[r]
{3,2,3}	{3,5,0}[r]	Ø	Ø	{3,0,5}[r]	{0,5,3}[r]	{0,2,6}
{0,3,5}	{0,5,3}[r]	{3,3,2}	{3,0,5}[r]	{0,0,8}[r]	Ø	Ø
{0,2,6}	{0,5,3}[r]	{3,2,3}[r]	{2,0,6}	{0,0,8}[r]	Ø	Ø
{3,3,2}	{3,5,0}[r]	Ø	Ø	{3,0,5}[r]	{1,5,2}	{0,3,5}[r]
{2,0,6}	{2,5,1}	{3,0,5}[r]	Ø	Ø	{0,2,6}[r]	{0,0,8}[r]
{1,5,2}	Ø	{3,5,0}[r]	{3,3,2}[r]	{1,0,7}	Ø	{0,5,3}[r]
{2,5,1}	Ø	{3,5,0}[r]	{3,4,1}	{2,0,6}[r]	Ø	{0,5,3}[r]
{1,0,7}	{1,5,2}[r]	{3,0,5}[r]	Ø	Ø	{0,1,7}	{0,0,8}[r]
{3,4,1}	{3,5,0}[r]	Ø	Ø	{3,0,5}[r]	{2,5,1}[r]	{0,4,4}
{0,1,7}	{0,5,3}[r]	{3,1,4}	{1,0,7}[r]	{0,0,8}[r]	Ø	Ø
{0,4,4}						

[r] – înseamnă stare repetată şi nu se va dezvolta în continuare
Ø – înseamnă că operatorul nu poate fi aplicat

e) Determinați câte noduri au fost explorate, ce adâncime are soluția şi care este factorul efectiv de ramificație

- adâncimea este 8 şi au fost explorate doar 14 noduri la un factor efectiv de ramificație b≈1,39

2. Să se scrie programul C# care rezolvă problema unui puzzle 3x3 folosind strategia breadth-first urmărind interfața din figura 2.5 de mai jos. Se cere: a) utilizarea unor casete text TextBox pentru introducerea stării inițiale şi finale b) pentru consum minim de memorie se impune codificarea fiecărei stări a puzzle-ului ca întreg pe 32 de biți c) utilizarea unei cozi Queue pentru pentru implementarea strategiei e) afişarea pe ecran a soluției într-un obiect ListBox f) afişarea pe ecran a cantității de memorie utilizate, a numărului de stări explorate şi a adâncimii soluției.

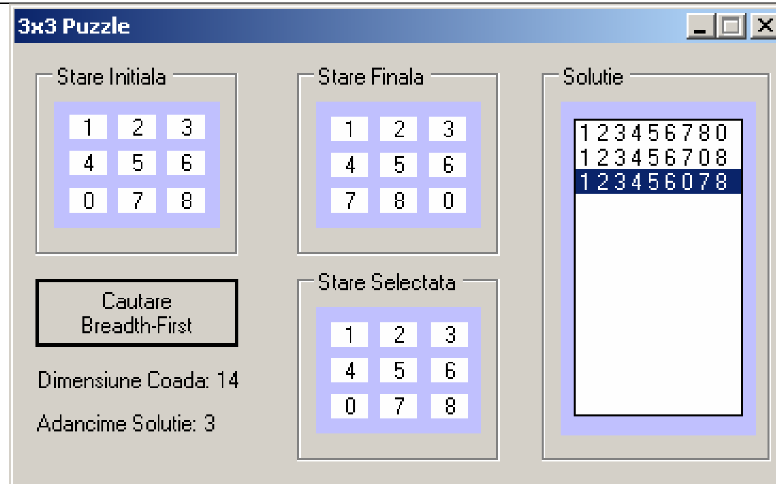


Figura 2.5. *Exemplu de interfață pentru un puzzle 3x3*

Indicații pentru rezolvare:

- Definim o clasă pentru reprezentarea nodurilor din arborele de căutare. Clasa va conține un constructor care primeşte nodul părinte, starea care corespunde nodului şi adâncimea acestuia. Toate aceste valori sunt private şi pot fi returnate cu metode Get.

```
    class SearchTreeNode
    {
        //nodul parinte
        private SearchTreeNode parrentNode = null;
        //starea asociata nodului curent
        private int state;
        //adancimea nodului
        private int depth;

        public SearchTreeNode(SearchTreeNode pn, int s, 
int d)
        {
            parrentNode = pn;
            state = s;
            d = depth;
        }

        public SearchTreeNode GetParrentNode()
```

```
        {
            return parrentNode;
        }

        public int GetState()
        {
            return state;
        }

        public int GetDepth()
        {
            return depth;
        }
    }
```

- Coada se declară ca obiect de tip Queue. Vom mai defini ca valori globale starea inițială respectiv starea finală.

```
private Queue nodesToExplore = new Queue();
// coada care retine nodurile ce vor fi explorate
private int initialState; // starea initiala
private int finalState; // starea finala
```

- Codificarea şi decodificarea stărilor din şir în întreg şi invers, utilă pentru consum scăzut de memorie şi pentru compararea simplă a stărilor, este realizată de următoarele metode. Vom memora puzzleul ca şir şi nu ca matrice (deci ca matrice liniarizată).

```
// functile DecodeState si EncodeState se utilizeaza
pentru a decodifica starea din intreg in
// vector (pentru a putea face deplsarea spatiului sus,
jos, stanga, dreapta) respectiv codificarea
// din vector in intreg pentru a face memorarea starii in
nodul arborelui de cautare
 private byte[] DecodeState(int state)
        {
            byte[] dstate = new byte[9];
            int i = 0;
            for (i = 0; i<9; i++)
            {
                dstate[8 - i] = (byte) (state % 10);
                state = state / 10;
            }
            return dstate;
```

```
        }

        private int EncodeState(byte[] state)
        {
            int estate = 0, i = 0;
            for (i = 0; i < 9; i++)
            {
                estate = estate * 10 + state[i];
            }
            return estate;
        }
```

➢ Succesorii se generează în baza metodelor de mai jos

```
//generare stare succesor in functie de operatorul aplicat
up, down, left, right (daca e 0 nu se aplica, daca e 1 se
aplica)
private byte[] GenerateState(byte[] state, int pos, int
up, int down, int left, int right)
        {
            int i = 0;
            byte[] newstate =new byte[9];
            for (i = 0; i < 9; i++)
            {
                newstate[i] = state[i];
            }
            newstate[(pos / 3)*3 + pos % 3] =
newstate[((pos / 3) + up * (-1) + down * 1) * 3 + (pos %
3) + left * (-1) + right * 1];
            newstate[((pos / 3) + up *(-1) + down * 1) * 3
+ (pos % 3) + left * (-1) + right * 1] = 0;
            return newstate;
        }

//adaugare succesori ai starii curente in coada (maxim 4
succesori)
private void AddSuccessors(SearchTreeNode currentNode){
            SearchTreeNode left, right, up, down;
            int i = 0;
            while ((i < 9) &&
(DecodeState(currentNode.GetState())[i] != 0)) i++;
//determina pozitia spatiului liber, se observa ca i div 3
e linia pe care se afla spatiul liber si i mod 3 e coloana
            if ( (i / 3) != 0)
```

```
//nu este pe prim linie deci se poate muta sus
            {
                up = new SearchTreeNode( currentNode,
EncodeState(GenerateState(DecodeState(currentNode.GetState
()), i, 1, 0, 0, 0)), currentNode.GetDepth() + 1);
                nodesToExplore.Enqueue(up);
            }
            if ((i/3) != 2)
//nu este pe linia de jos deci se poate muta jos
            {
                down = new SearchTreeNode( currentNode,
EncodeState(GenerateState(DecodeState(currentNode.GetState
()), i, 0, 1, 0, 0)), currentNode.GetDepth() + 1);
                nodesToExplore.Enqueue(down);
            }
            if ((i%3) != 0)
//nu este pe coloana din stanga deci se poate muta stanga
            {
                left = new SearchTreeNode(currentNode,
EncodeState(GenerateState(DecodeState(currentNode.GetState
()), i, 0, 0, 1, 0)), currentNode.GetDepth() + 1);
                nodesToExplore.Enqueue(left);
            }
            if ((i%3) != 2)
//nu este pe coloana din dreapta deci se poate muta
dreapta
            {
                right = new SearchTreeNode(currentNode,
EncodeState(GenerateState(DecodeState(currentNode.GetState
()), i, 0, 0, 0, 1)), currentNode.GetDepth() + 1);
                nodesToExplore.Enqueue(right);
            }
        }
```

- Funcția search care implementează algoritmul de căutare este următoarea

```
//functie search (algoritm de cautare)
        private void Search()
        {
            SearchTreeNode currentNode = new
SearchTreeNode(null, initialState, 0);
            nodesToExplore.Enqueue(currentNode);
            do
            {
```

```
                    if ( nodesToExplore.Count == 0 )
//verifica daca mai sunt stari de explorat
                    {

System.Windows.Forms.MessageBox.Show("Nu exista solutie");
                        break;
                    }
                    currentNode = (SearchTreeNode)
nodesToExplore.Dequeue();
                    if ( currentNode.GetState() == finalState)
//verifica daca s-a ajuns la solutie
                    {

System.Windows.Forms.MessageBox.Show("S-a gasit solutia");
                        ShowSolution(currentNode);
                        break;
                   }
                   AddSuccessors(currentNode);
// adauga succesorii starii curente
               }while(true);
          }
```

- Soluția este afişată prin parcurgerea arborelui de căutare de la nodul cu starea finală până la rădăcină care conține starea inițială

```
//afiseaza solutia in ListBox
private void ShowSolution(SearchTreeNode fs)
{
SearchTreeNode currentnode = fs;
byte[] dstate;
while (currentnode != null)
            {
dstate = DecodeState(currentnode.GetState());
listaSolutie.Items.Add(dstate[0].ToString() + " " +
dstate[1].ToString() + " " + dstate[2].ToString() + " " +
dstate[3].ToString() + " " + dstate[4].ToString() + " " +
dstate[5].ToString() + " " + dstate[6].ToString() + " " +
dstate[7].ToString() + " " + dstate[8].ToString());
currentnode = currentnode.GetParrentNode();
            }
listaSolutie.SelectedIndex = listaSolutie.Items.Count - 1;
labelAdancime.Text = "Adancime Solutie: " +
listaSolutie.Items.Count.ToString();
```

```
labelDim.Text  = "Dimensiune Coada: " +
nodesToExplore.Count.ToString();
}
```

➢ Alte câteva funcţii legate de interfaţă sunt următoarele

```
//pentru evenimentul Click pe buton
private void startButton_Click(object sender, EventArgs e)
        {
            nodesToExplore.Clear();
            listaSolutie.Items.Clear();
            InitStates();
            Search();
        }

//initializeaza starea initiala si starea finala din
casetele text
private void InitStates()
{
byte[] iState = { Convert.ToByte(SI0.Text),
Convert.ToByte(SI1.Text), Convert.ToByte(SI2.Text),
Convert.ToByte(SI3.Text), Convert.ToByte(SI4.Text),
Convert.ToByte(SI5.Text), Convert.ToByte(SI6.Text),
Convert.ToByte(SI7.Text), Convert.ToByte(SI8.Text) };
byte[] gState = { Convert.ToByte(SF0.Text),
Convert.ToByte(SF1.Text), Convert.ToByte(SF2.Text),
Convert.ToByte(SF3.Text), Convert.ToByte(SF4.Text),
Convert.ToByte(SF5.Text), Convert.ToByte(SF6.Text),
Convert.ToByte(SF7.Text), Convert.ToByte(SF8.Text) };
initialState = EncodeState(iState);
finalState = EncodeState(gState);
}

//afiseaza starea selectata din ListBox in casetele text
private void listaSolutie_SelectedIndexChanged(object
sender, EventArgs e)
{
char[] state =
listaSolutie.Items[listaSolutie.SelectedIndex].ToString().
ToCharArray();
            SS0.Text = state[0].ToString();
            SS1.Text = state[2].ToString();
            SS2.Text = state[4].ToString();
            SS3.Text = state[6].ToString();
```

```
                SS4.Text = state[8].ToString();
                SS5.Text = state[10].ToString();
                SS6.Text = state[12].ToString();
                SS7.Text = state[14].ToString();
                SS8.Text = state[16].ToString();
}
```

3. Să se rezolve în C# problema anterioară folosind căutarea bidirecțională.

4. Să se scrie programul C# care rezolvă problema tabloului cu leduri. Se consideră un tablou cu leduri de dimensiune 3x3 la care prin apasarea unui led acesta îşi schimbă starea proprie din stins în aprins (şi invers) precum şi a ledurilor aflate în stânga, dreapta, sus, jos față de acesta. În starea inițială toate ledurile sunt stinse iar tabloul trebuie adus în starea finală în care toate ledurile vor fi aprinse. Se va folosi o interfață apropiată de cea de la problema 2.

Lucrarea 3 Evitarea ciclurilor în algoritmii de căutare

Stările repetate, stări în care agentul deja a mai fost şi care conduc la adăugarea în listele de succesori a unor noduri ce conțin stări deja explorate, conduc la creşterea inutilă a necesităților de timp şi spațiu precum şi la arbori de căutare de dimensiune infinită. De exemplu, datorită stărilor repetate, strategia breadth-first anterior implementată pentru puzzleul 3x3 nu găseşte în timp util soluția chiar la adâncimi relativ mici. Mai mult, arborii infiniți fac în unele cazuri ca strategia de căutare să nu mai găsească niciodată soluția problemei, intrându-se într-o buclă infinită. Chiar dacă strategiile orientate pe căutarea pe nivel evită buclele infinite, şi sunt astfel complete indiferent de situație, acelaşi lucru nu se întâmplă în cazul căutărilor în adâncime ce vor fi discutate în capitolul următor şi care în cazul intrării într-un ciclu nu vor mai furniza nicicând o soluție.

Pentru evitarea acestui neajuns există câteva restricții care pot fi aplicate. În ordinea crescătoare a eficienței dar şi a dificultății implementare şi a resurselor solicitate acestea sunt:

- dintr-un nod fiu nu se acceptă succesori cu aceeaşi stare cu a nodului părinte,
- dintr-un nod nu se acceptă succesori cu aceeaşi stare cu a unui nod care i-a fost strămoş,
- dintr-un nod nu se acceptă succesori cu stări care au mai fost generate vreodată.

Desigur, implementarea ultimei metode necesită stocarea tuturor stărilor deja explorate, pentru aceasta, complexitatea de spațiu este cea care induce problemele cele mai mari, altfel complexitatea de timp este logaritmică datorită posibilității de a plasa stările parcurse în liste ordonate şi a efectua căutări binare.

3.1 Exerciții

1. Pentru implemetarea strategiei breadth-first pentru puzzleul 3x3 din capitolul anterior aplicați restricțiile pentru evitarea stărilor repetate şi extrageți caracterisiticile de performanță pentru fiecare dintre restricții. Se va folosi o interfață apropiată de cea din figura 3.1, care afişează în plus față de problema anterioară şi numărul de stări explorate.

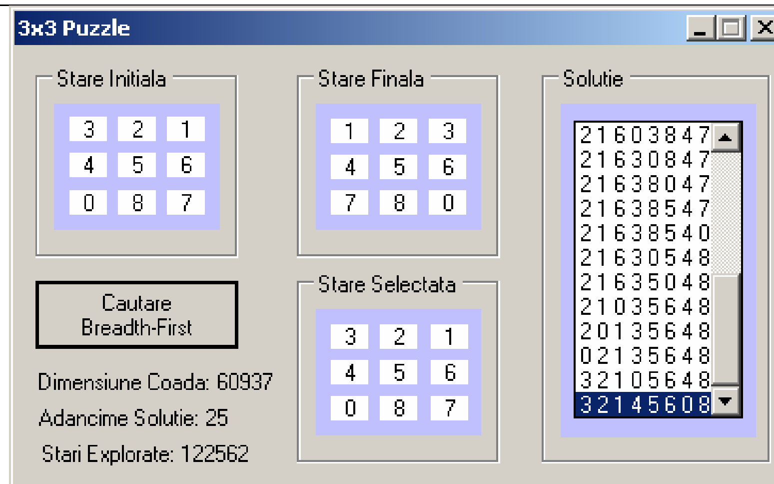


Figura 3.1. *Exemplu de interfață*

Indicații pentru rezolvare:

- Se foloseşte o tabelă HashTable pentru memorarea stărilor care sunt deja explorate

```
private Hashtable exploredStates = new Hashtable(); //
tabela care retine starile ce au fost deja explorate
```

- În funcția search, înainte de a explora o stare (a îi adăuga succesorii) se verifică în tabela HashTable dacă stărea nu a fost deja explorată

```
private void Search()
        {
            SearchTreeNode currentNode = new
SearchTreeNode(null, initialState, 0);
            nodesToExplore.Enqueue(currentNode);
            do
            {
                if ( nodesToExplore.Count == 0 )
//verifica daca mai sunt stari de explorat
                {

System.Windows.Forms.MessageBox.Show("Nu exista solutie");
                    break;
```

```
                }
                currentNode = (SearchTreeNode)
nodesToExplore.Dequeue();
                if ( currentNode.GetState() == finalState)
//verifica daca s-a ajuns la solutie
                {

System.Windows.Forms.MessageBox.Show("S-a gasit solutia");
                    ShowSolution(currentNode);
                    break;
                }
                if
(!exploredStates.ContainsKey(currentNode.GetState()))
//verifica daca starea curenta a mai fost explorata
                {
                    AddSuccessors(currentNode); // adauga
succesorii starii curente

exploredStates.Add(currentNode.GetState(), currentNode);
//adauga starea curenta intre starile explorate
                }
            }while(true);
        }
```

- La repornirea căutării se goleşte lista de noduri explorate şi a celor care urmează a fi explorate

```
private void startButton_Click(object sender, EventArgs e)
        {
            nodesToExplore.Clear();
            listaSolutie.Items.Clear();
            exploredStates.Clear();
            InitStates();
            Search();
        }
```

3. Să se rezolve în C# problema anterioară cu restricții pentru stări repetate şi folosind căutarea bidirecțională.

4. Să se scrie programul C# care rezolvă problema tabloului cu leduri folosind restricțiile pentru evitarea stărilor repetate. Se va folosi o interfață apropiată de cea de la problema 1.

Lucrarea 4 Strategii neinformate de căutare: strategiile de căutare în adâncime, adâncime limitată şi adâncime iterativă

4.1 Strategia de căutarea în adâncime (Depth-First)

Strategia de căutarea în adâncime, aşa cum se va vedea în cele ce urmează, este alternativa pentru a reduce consumul ridicat de memorie de la căutarea pe nivel. În acest caz nodurile sunt explorate în ordinea adâncimii şi se revine la nivelul superior doar când căutarea ajunge într-un punct mort (dead-end). Acest lucru este sugerat în figura 4.1 pe un arbore de căutare de adâncime 4 şi factor de ramificație 2.

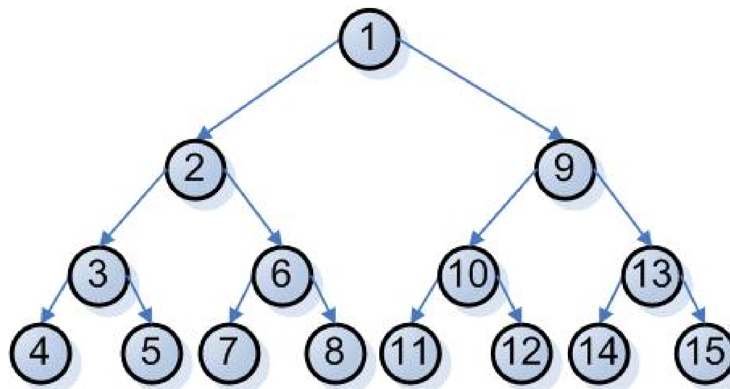


Figura 4.1. *Arborele de căutare în cazul strategiei depth-first*

- ***Complexitate de timp şi spațiu***

Este uşor de observat că în cazul cel mai dezavantajos din punct de vedere computațional complexitatea de timp este tot $T = O(b^d)$. În ceea ce priveşte spațiul însă, din cauză că explorarea se face întotdeauna în adâncime acesta ajunge la $S = O(b \cdot d)$ ceea ce înseamnă complexitate liniară şi este desigur un avantaj major comparativ cu complexitatea exponențială de la căutarea pe nivel.

- ***Completitudine şi optimalitate***

Căutarea în adâncime nu este completă în arbori infiniți, adică atunci când apar cicluri. Altfel, dacă se foloseşte o constrângere pentru evitarea stărilor repetate strategia este completă. Deoarece riscă să găsească soluția aflată la adâncimea cea mai mare, şi nu ține cont nici de costuri, strategia nu este optimală.

➤ ***Implementare***

Căutarea în adâncime se implementează uşor folosind o stivă în care sunt adăugate, evident la început, nodurile care rezultă din explorarea nodului curent şi tot aşa. Acest lucru este ilustrat în figura 4.2.

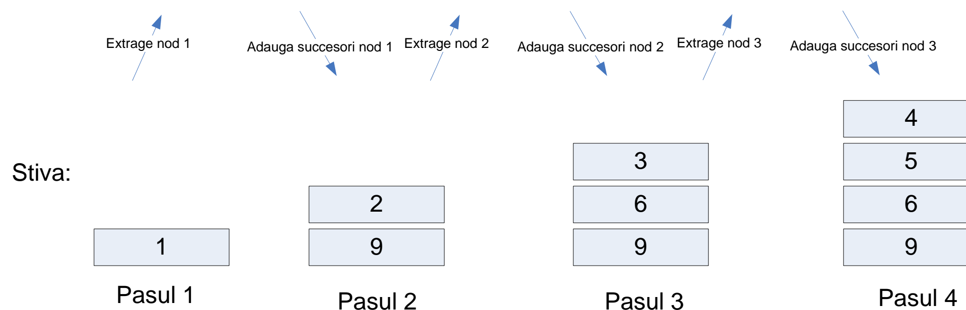


Figura 4.2. *Evoluția stivei la căutarea depth-first*

➤ ***Avantaje şi dezavantaje***

Avantajul major al strategiei de căutare în adâncime este faptul că necesitățile de memorie sunt minimale, complexitatea de spațiu fiind liniară. Dezavantajul major este că strategia nu este nici optimală şi nici completă.

4.2 Strategia de căutare limitată în adâncime (Depth-Limited)

Principalul dezavantaj al strategiei de căutare în adâncime este faptul că nu este completă. Din fericire această deficiență poate fi înlăturată parțial prin introducerea unei limite de adâncime l. Astfel, vor fi explorate doar nodurile pâna la o limită l şi dacă adâncimea soluției este mai mică decât l atunci ea va fi găsită. Utilizarea strategiei de căutare în adâncime este recomandată atunci când se cunoaşte adâncimea unei soluții. De altfel, algoritmul numit BackTracking aşa cum fost prezentat în cadrul unor materii este de fapt o căutare limitată în adancime şi motivul pentru care a funcționat era faptul că se cunoştea limita de adâncime iar atunci când nu se cunoştea adâncimea soluției BackTracking nu se mai putea aplica.

- ***Complexitate de timp şi spațiu***

Complexitatea de timp rămâne cea de la căutarea în adâncime şi anume $T = O(b^d)$, la fel şi cea de spațiu $S = O(bd)$. Deoarece explorarea se face efectiv până la limita de adâncime *l*, este mai corect să spunem că $T = O(b^l)$ şi $S = O(bl)$.

- ***Completitudine şi optimalitate***

Strategia de căutare limitată în adâncime este completă doar dacă adâncimea unei soluții este mai mică decât *l*, i.e. $d<l$. Rămâne necesară şi remarca de la breadth first, anume că arborele trebuie să fie finit, ceea ce presupune şi un număr de operatori finit.

Strategia nu este optimală deoarece este predispusă la a găsi întâi soluția cea mai adâncă, mai mult nu ține cont nici de potențiale costuri ale operatorilor. Totuşi este interesant de observat că există un caz în care strategia este optimală: atunci când toate soluțiile problemei se află pe limita de adâncime. Acesta este cazul multor probleme rezolvate cu backtracking în liceu sau primul an de facultate: aranjarea a *n* regine pe tabla de şah, umplerea unei table de şah de dimensiune *n*x*n* prin săritura calului etc.

- ***Implementare***

Strategia se implementează identic cu strategia de căutare în adâncime, doar că în acest caz se impune şi o limită de adâncime peste care funcția search nu mai desface succesori.

- ***Avantaje şi dezavantaje***

Principalul avantaj este acela că are necesități de memorie scăzute, la fel ca la căutarea în adâncime, în timp ce rămâne completă atunci când $d<l$. Dezavantajul este faptul că nu este optimală.

4.3 Strategia de căutare iterativă în adâncime (Iterative-Deepening)

Ar fi desigur de dorit să avem o strategie de căutare care este completă şi optimală precum căutarea pe nivel, dar în acelaşi timp are necesitățile de memorie scăzute de la căutarea în adâncime. Din fericire acest lucru este posibil folosind căutarea limitată în adâncime cu limite iterate (0,1,2,3,…). Această strategie îmbină cele mai bune caracteristici de la căutarea în adâncime şi căutarea pe nivel apelând

succesiv căutarea limitată în adâncime cu limite iterate. Astfel, unele noduri vor fi explorate de mai multe ori aşa cum se poate observa şi în figura 4.3. În figură, pentru a nu încărca desenul s-a omis faptul că primul nod este explorate de 4 ori, pentru cei 4 arbori de căutare generați la adâncime 1, 2, 3 respectiv 4.

- ***Complexitate de timp şi spațiu***

Complexitate de timp rămâne $T = O(b^d)$ dar deoarece unele noduri sunt explorate de mai multe ori, la fiecare iterație în adâncime, complexitatea este de fapt $T = (d+1) + d \cdot b + (d-1) \cdot b^2 + ... + b^d = O(b^d)$. Din această cauză, chiar dacă din punct de vedere asimptotic, al indicatorului *O*, strategia are aceeaşi complexitate de timp cu celelalte strategii, în realitate timpul de calcul este mai mare şi trebuie să ne aşteptăm ca strategia de căutare iterativă în adâncime să dea un răspuns într-un timp ceva mai lung decât celelalte.

Complexitatea de spațiu rămâne cea de la strategia de căutare în adâncime $S = O(b \cdot d)$.

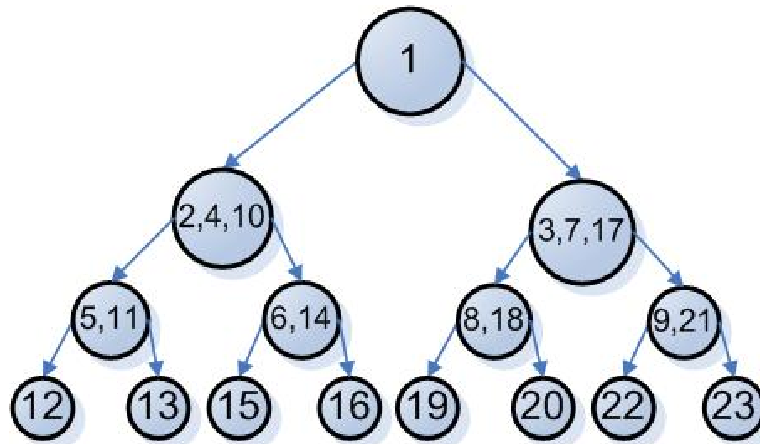


Figura 4.3. *Arborele de căutare în cazul strategiei iterative-deepening*

- ***Completitudine şi optimalitate***

Strategia este completă în arbori finiți la fel ca strategia de căutare pe nivel. De asemenea este şi optimală în aceleaşi condiții cu strategia de căutare pe nivel: costul crescător odată cu adâncimea (lucru adevărat în cazul particular al operatorilor cu costuri egale).

➢ ***Implementare***

Se implemenetază prin apelarea iterativă a strategiei de căutare limitată în adâncime prin iterarea limitei de adâncime *l*. Structura folosită este deci stiva.

➢ ***Avantaje şi dezavantaje***

Avantajul major este faptul că este optimală şi completă la fel ca strategia de căutare pe nivel având însă necesități scăzute de memorie la fel ca strategia de căutare în adâncime. În general, în spații de dimensiune mare este strategia preferată datorită consumului redus de memorie. Dezavantajul este faptul că fiecare nod este parcurs de mai multe ori, dar asimptotic vorbind acest lucru devine irelevant.

4.4 Exerciții

1. Există probleme pentru care căutarea depth-limited este optimală şi completă? Dați 3 exemple şi o regulă generală pentru ca depth-limited este optimală şi completă.

2. Se considera un puzzle 3x3 si un tablou cu leduri de dimensiune 3x3 la care prin apăsarea unui led acesta îşi schimba starea proprie din stins în aprins (şi invers) precum şi a ledurilor aflate în stanga, dreapta, sus, jos față de acesta. Se cere: pentru fiecare dintre strategiile de cautare neinformate cunoscute explorați arborele de cautare până la adâncimea $d=3$ precizand ordinea în care nodurile au fost explorate şi indicând structura utilizată pentru implementare (coadă, stivă etc.)

3. Să se scrie programul C# care rezolvă problema unui puzzle 3x3 folosind strategia depth-first evitând stările repetate şi folosind o interfață apropiată de cea din figura de mai jos. Se recomandă folosirea codului sursă din capitolele anterioare.

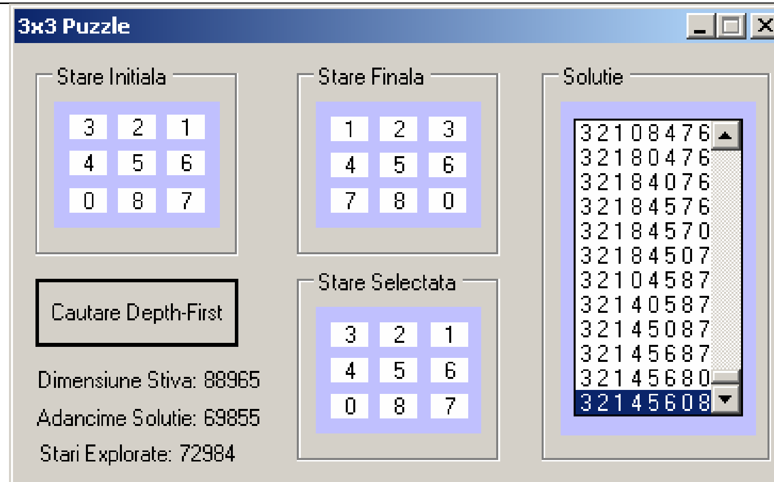


Figura 4.4. *Exemplu de interfață*

Indicații pentru rezolvare:

- Coada de la breadth-first în care se rețin nodurile ce urmează a fi explorate se transformă în cazul lui depth-first în stivă

```
private Stack nodesToExplore = new Stack(); // stiva care
retine nodurile ce vor fi explorate
```

- În funcțiile de adăugare succesori şi de căutare operațiile se vor face cu Push şi Pop în loc de Enqueue şi Dequeue

```
//adaugare succesori ai starii curente in coada (maxim 4
succesori)
        private void AddSuccessors(SearchTreeNode
currentNode){
            SearchTreeNode left, right, up, down;
            int i = 0;
            while ((i < 9) &&
(DecodeState(currentNode.GetState())[i] != 0)) i++;
//determina pozitia spatiului liber, se observa ca i div 3
e linia pe care se afla spatiul liber si i mod 3 e coloana
            if ( (i / 3) != 0) //nu este pe prim linie
deci se poate muta sus
            {
                up = new SearchTreeNode( currentNode,
```

```
EncodeState(GenerateState(DecodeState(currentNode.GetState
()), i, 1, 0, 0, 0)), currentNode.GetDepth() + 1);
                nodesToExplore.Push(up);
            }
            if ((i/3) != 2) //nu este pe linia de jos deci
se poate muta jos
            {
                down = new SearchTreeNode( currentNode,
EncodeState(GenerateState(DecodeState(currentNode.GetState
()), i, 0, 1, 0, 0)), currentNode.GetDepth() + 1);
                nodesToExplore.Push(down);
            }
            if ((i%3) != 0) //nu este pe coloana din
stanga deci se poate muta stanga
            {
                left = new SearchTreeNode(currentNode,
EncodeState(GenerateState(DecodeState(currentNode.GetState
()), i, 0, 0, 1, 0)), currentNode.GetDepth() + 1);
                nodesToExplore.Push(left);
            }
            if ((i%3) != 2) //nu este pe coloana din
dreapta deci se poate muta dreapta
            {
                right = new SearchTreeNode(currentNode,
EncodeState(GenerateState(DecodeState(currentNode.GetState
()), i, 0, 0, 0, 1)), currentNode.GetDepth() + 1);
                nodesToExplore.Push(right);
            }
        }

        //functie search (algoritm de cautare)
        private void Search()
        {
            SearchTreeNode currentNode = new
SearchTreeNode(null, initialState, 0);
            nodesToExplore.Push(currentNode);
            do
            {
                if ( nodesToExplore.Count == 0 )
//verifica daca mai sunt stari de explorat
                {

System.Windows.Forms.MessageBox.Show("Nu exista solutie");
                    break;
```

```
                }
                currentNode = (SearchTreeNode)
nodesToExplore.Pop();
                if ( currentNode.GetState() == finalState)
//verifica daca s-a ajuns la solutie
                {

System.Windows.Forms.MessageBox.Show("S-a gasit solutia");
                    ShowSolution(currentNode);
                    break;
                }
                if
(!exploredStates.ContainsKey(currentNode.GetState()))
//verifica daca starea curenta a mai fost explorata
                {
                    AddSuccessors(currentNode); // adauga
succesorii starii curente

exploredStates.Add(currentNode.GetState(), currentNode);
//adauga starea curenta intre starile explorate
                }
            }while(true);
        }
```

➢ La repornirea căutării trebuie golită acum stiva de nodurile ce urmează a fi explorate

4. Să se rezolve problema anterioară folosind căutarea limitată în adâncime şi căutarea iterativă în adâncime.

5. Să se rezolve problema anterioară folosind căutarea bidirecțională cu breath-first dintr-o parte şi depth-first din cealaltă parte.

6. Să se scrie programul C# care rezolvă problema tabloului cu leduri folosind iterative-deepening. Se poate rezolva această problemă folosind depth-first?

Lucrarea 5 Strategii neinformate de căutare: strategia de căutare cu cost uniform

5.1 Strategia de căutare cu cost uniform (Uniform Cost)

Ceea ce nici una din strategiile de căutare anterior introduse nu putea să rezolve era optimalitatea pentru cazul în care operatorii nu au costuri egale şi 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 de căutare cu cost uniform rezolvă eficient această problemă. Strategia mai este cunoscută şi ca Dijkstra's Single-Source Shortest Path sau simplu algoritmul lui Dijkstra.

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

Un scurt exemplu poate fi util în lămurirea modului de funcționare al strategiei. Fie graful orientat aciclic din figura 5.1, se doreşte găsirea celui mai scurt drum de la A la E. Pentru aceasta nodurile sunt explorate în ordinea costurilor, lucru indicat în figura 5.2 în care se observă că drumul găsit de strategie este A-C-E care este cel mai ieftin.

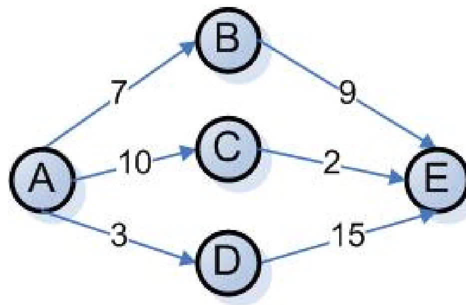


Figura 5.1. *Graf orientat aciclic*

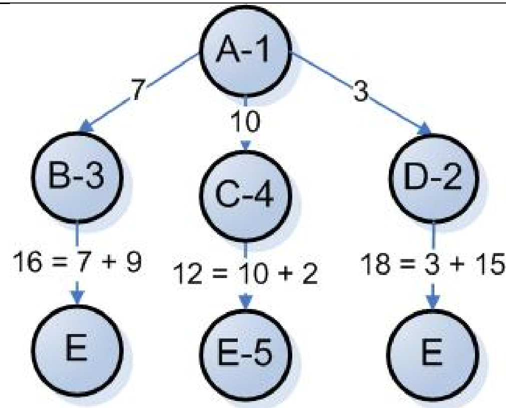


Figura 5.2. *Arborele de căutare în cazul strategiei cu cost uniform*

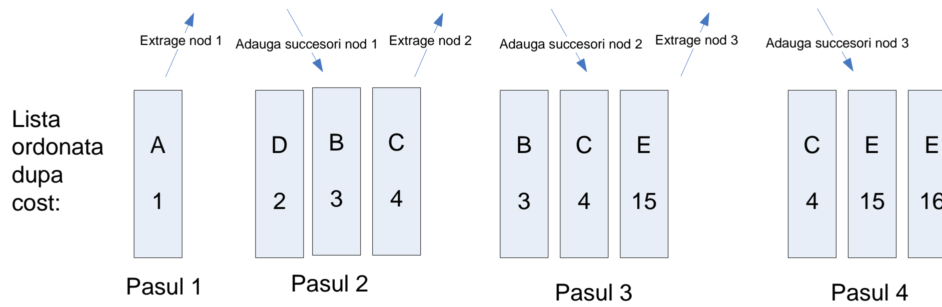


Figura 5.3. *Evoluția listei ordonate după costuri*

➢ *Complexitate de timp şi spațiu*

Complexitatea de timp este $T = O\left(b^d\right)$. O altă valoare frecvent folosită pentru complexitatea de timp este $T = O\left(b^{c_o / c_m}\right)$ unde c_o este costul căii optime în arbore iar c_m este costul minim al operatorilor. Complexitatea de spațiu este $T = O\left(b^d\right)$.

➢ *Completitudine şi optimalitate*

Strategia de căutare cu cost uniform este completă, costul strict crescător odată cu adâncimea în arbore făcând ca ciclurile să fie evitate. Strategia este optimală în funcție de cost cu excepția situației când apar costuri negative în arbore.

- ***Implementare***

Se implementează folosind o listă sortată crescător după costuri.

- ***Avantaje şi dezavantaje***

Avantajul strategiei este că este completă şi optimală chiar şi atunci când costul nu este strict crescător cu nivelul. Dezavantajul însă este că are necesități de memorie ridicate, la fel ca în cazul lui breath-first.

5.2 Exerciții

1. Poate fi văzută căutarea breadth-first ca un caz particular de uniform-cost?

2. Complexitatea strategiei uniform-cost este $T=O(b^d)$. Explicați de ce este corectă şi relația $T=O(b^{co/cm})$ unde c_o este costul căii optime în arbore iar c_m este costul minim al operatorilor.

3. Se consideră un puzzle 3x3 şi un tablou cu leduri de dimensiune 3x3 la care prin apasarea unui led acesta îşi schimbă starea proprie din stins în aprins (şi invers) precum şi a ledurilor aflate în stânga, dreapta, sus, jos față de acesta. Pentru ficare dintre următoarele strategii stabiliți dacă sunt optimale şi complete precum şi complexitatea efectivă de timp şi spațiu la o adâncime a soluției d=10: a) breadth-first b) depth-first c) depth-limited d) iterative-deepening e) uniform-cost f) bidirectional search.

4. Se consideră un spațiu cu obstacole de dimensiune 100x100 careuri şi un agent inteligent care doreste să ajungă din *SI(x,y)* în *SF(x',y')*. Fiecare careu are asociată o cotă care are asociată o valoare aleatoare şi se defineşte un prag care reprezintă valoarea numerică a cotei maxime peste care agentul poate să treacă (orice careu a cărui cotă depăşeşte acest prag se consideră obstacol). Se consideră că deplasarea în linie dreaptă are costul 100 iar cea în diagonală are costul 141. Să se implementeze programul C# care rezolvă această problemă. Se va folosi o interfață apropiată de cea din figura 5.4.

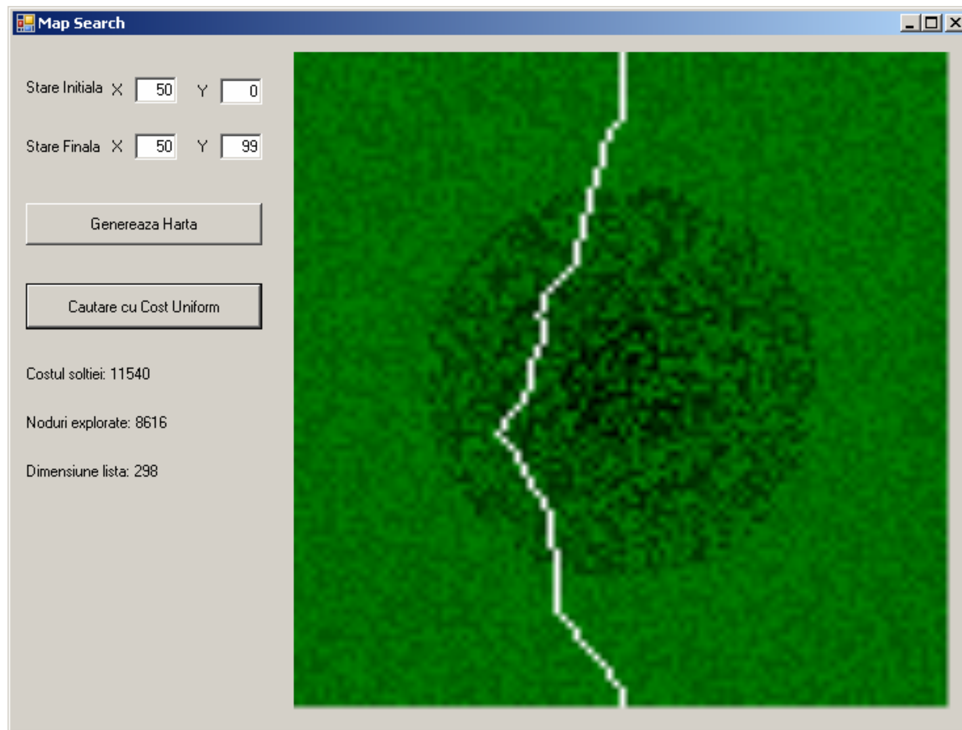


Figura 5.4 *Exemplu de interfață pentru problema spațiului cu obstacole*

Indicații pentru rezolvare:

➢ Se defineşte o clasă pentru nodul arborelui de căutare

```
class SearchTreeNode
    {
        public SearchTreeNode parrent = null;
        public int cost;
        public int level;
        public long id = 0;
        public int coordX = 0;
        public int coordY = 0;
        public int nodeOperator = 0; //degreess from 90 to
360
    }
```

- Se definesc costurile pentru deplasarea în linie dreaptă şi în diagonală

```
// costul miscarii in linie dreapta
private const int straigthMovement = 100;
// costul miscarii in diagonala sqrt(2) =
1.4142135623730950488016887242097
private const int diagonalMovement = 141;
```

- Se defineşte matricea pentru memorarea hărții şi înălțimea treptei pentru obstacole (ulterior aceasta se va citi dintr-un text-box)

```
// matrice utilizata pentru memorarea hartii 100x100
private int[,] m = new int[100,100];
private int obstacleHeight = 50;
```

- Se definesc listele cu noduri deja explorate respectiv cu noduri ce urmează a fi explorate. Deoarece lista cu noduri ce urmează a fi explorate trebuie sortată se defineşte şi un obiect de tip IComparer

```
Hashtable ExploredStates;
ArrayList StatesToExplore;
private IComparer myComparer = null;
```

- Clasa care implementeaza IComparer pentru costuri este următoarea

```
class CostComparer : IComparer
    {

        int IComparer.Compare(Object x, Object y)
        {
            SearchTreeNode n1 = (SearchTreeNode)x;
            SearchTreeNode n2 = (SearchTreeNode)y;
            if (n1.cost > n2.cost)
            {
                return 1;
            }
            else
            {
                if (n1.cost < n2.cost)
                {
                    return -1;
```

```
                }
                else
                {
                    return 0;
                }
            }
        }
}
```

➢ Se generează aleator harta şi se afişează în PictureBox. Pentru a face problema mai interesantă harta a fost generată aleator cu cote care evoluează de la centrul spre marginea hărții astfel încât în centru să fie densitate de obstacole mai mare (pentru simplitate se poate renunța la acest lucru).

```
// harta este generata aleator cu obstacole a caror
densitate creste spre centrul hartii
private void GenerateMap()
        {
            int i,j;
            Random r = new Random();
            for (i = 0; i < 100; i++)
            {
                for (j = 0; j < 100; j++)
                {

                    if ((Math.Pow(i - 50, 2) + Math.Pow(j
- 50, 2)) < Math.Pow(10, 2))
                    {
                        m[i, j] = r.Next(128);
                    }
                    else
                    {
                        if ((Math.Pow(i - 50, 2) +
Math.Pow(j - 50, 2)) < Math.Pow(20, 2))
                        {
                            m[i, j] = r.Next(100);
                        }
                        else
                        {
                            if ((Math.Pow(i - 50, 2) +
Math.Pow(j - 50, 2)) < Math.Pow(30, 2))
                            {
                                m[i, j] = r.Next(80);
                            }
```

```
                                else
                                {
                                    m[i, j] = r.Next(40);
                                }
                            }
                        }
                    }
                }
        }

        // harta din matrice este desenata in PictureBox
        private void DisplayMapOnPictureBox()
        {
            Bitmap bmp = new Bitmap(100, 100);
            int i, j;
            Color col = Color.Green;
            for (i = 0; i < 100; i++)
            {
                for (j = 0; j < 100; j++)
                {
                    bmp.SetPixel(i, j,
Color.FromArgb(col.R , col.G - m[i, j], col.B));
                }
            }
            pictureBoxMap.Image = bmp;
            pictureBoxMap.SizeMode =
PictureBoxSizeMode.StretchImage;
        }
```

- Funcția search cu evitarea stărilor repetate

```
//algoritmul de cautare (functia Search)
private void Search(SearchTreeNode startNode,
SearchTreeNode targetNode)
        {

            bool solutionFound = false;
            SearchTreeNode currentNode;
            StatesToExplore.Add(startNode);
            // cautarea continua pana cand se gaseste o
solutie
            while (solutionFound == false)
            {
                if (StatesToExplore.Count == 0)
```

```
                {
                    // daca lista e goala nu mai sunt
succesori si deci nu exista solutie

System.Windows.Forms.MessageBox.Show("Nu exista solutie");
                    break;
                }
                currentNode =
GetNextSuccessor(StatesToExplore);
                if (Solution(currentNode, targetNode))
                {

System.Windows.Forms.MessageBox.Show("Solutie Gasita" );
                    ShowSolution(currentNode);
                    solutionFound = true;
                }
                else
                {
                    // verifica daca nodul current nu a
fost deja explorat
                    if
(!(ExploredStates.ContainsKey(currentNode.id)))
                    {
                        AddSuccessors(currentNode,
StatesToExplore, targetNode);
                        ExploredStates.Add(currentNode.id,
currentNode);
                    }
                }
            }
        }
```

- Crearea, adăugarea şi extragerea succesorilor. Pentru o mai bună vizualizare a mişcărilor au fost codificate după unghiul la care se face mişcarea 0, 45, 90 etc.

```
// se adauga succesorii nodului curent in lista dupa cele
8 directii de miscare
        private void AddSuccessors(SearchTreeNode node,
ArrayList StatesToExplore, SearchTreeNode target)
        {
            int i, newX, newY;
            SearchTreeNode n;
            int cost = 0;
```

```
            for (i = 0; i < 8; i++)
            {
                newX = -1; newY = -1;
                switch ( i*45 )
                {
                    case 0: newX = node.coordX; newY =
node.coordY + 1; cost = straigthMovement;  break;
                    case 45: newX = node.coordX + 1; newY
= node.coordY + 1; cost = diagonalMovement; break;
                    case 90: newX = node.coordX + 1; newY
= node.coordY; cost = straigthMovement; break;
                    case 135: newX = node.coordX + 1; newY
= node.coordY - 1; cost = diagonalMovement; break;
                    case 180: newX = node.coordX; newY =
node.coordY - 1; cost = straigthMovement; break;
                    case 225: newX = node.coordX - 1; newY
= node.coordY - 1; cost = diagonalMovement; break;
                    case 270: newX = node.coordX - 1; newY
= node.coordY; cost = straigthMovement; break;
                    case 315: newX = node.coordX - 1; newY
= node.coordY + 1; cost = diagonalMovement; break;
                }
                // creeaza succesorul doar daca nodul nu
este in afara hartii si nivelul este mai mic decat nivelul
obstacolului
                if (CoordinatesInsideBounds(newX, newY))
                {
                    if (m[newX, newY] < obstacleHeight)
                    {
                    n = new SearchTreeNode();
                    n.parrent = node;
                    n.Accessible = true;
                    n.nodeOperator = i * 45;
                    n.coordX = newX;
                    n.coordY = newY;
                    n.id = n.coordX + n.coordY * 10000;
                    n.level = node.level + 1;
                    n.heuristic = DiagonalDistance(n,
target);
                    n.cost = node.cost + cost;
                    node.succesors[i] = n;
                    StatesToExplore.Add(n);
                    }
                }
```

```
            }
            StatesToExplore.Sort(myComparer);
        }

        // verifica daca coordonatele sunt in interiorul
hartii
        private bool CoordinatesInsideBounds(int x, int y)
        {
            if ((x >= 0) && (x < 100) && (y >= 0) && (y <
100))
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        // returneaza urmatorul succesor, adica starea cea
mai apropiata de starea finala
        private SearchTreeNode GetNextSuccessor(ArrayList
StatesToExplore)
        {
            SearchTreeNode state =
(SearchTreeNode)StatesToExplore[0];
            StatesToExplore.RemoveAt(0);
            return state;
        }
```

➢ Afişarea soluției şi testarea dacă un anume nod conține sau nu starea finală.

```
private void ShowSolution(SearchTreeNode position)
        {
            Bitmap bmp = new Bitmap(pictureBoxMap.Image);
            SearchTreeNode n = position;
            while (n != null)
            {
                bmp.SetPixel(n.coordX, n.coordY,
Color.Red);
                n = n.parrent;
            }
            pictureBoxMap.Image = bmp;
            pictureBoxMap.SizeMode =
```

```
PictureBoxSizeMode.StretchImage;
            labelNoduri.Text = "Noduri explorate: " +
ExploredStates.Count.ToString();
            labelDimensiune.Text = "Dimensiune lista: " +
StatesToExplore.Count.ToString();
            labelCost.Text = "Costul soltiei: " +
position.cost.ToString();
        }

        // verifica daca nodul curent este chiar nodul
tinta
        private bool Solution(SearchTreeNode node,
SearchTreeNode targetnode)
        {
            if ((node.coordX == targetnode.coordX) &&
(node.coordY == targetnode.coordY))
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

➢ Funcția care calculează distanța diagonală.

```
// calculeaza distanta diagonala intre doua puncte
        private int DiagonalDistance(SearchTreeNode
current, SearchTreeNode target)
        {
            int xd = Math.Abs(current.coordX -
target.coordX);
            int yd = Math.Abs(current.coordY -
target.coordY);
            return straigthMovement * (Math.Max(xd, yd) -
Math.Min(xd, yd)) + diagonalMovement * Math.Min(xd, yd);
        }
```

➢ Funcțiile asociate diverselor evenimente.

```
private void Form1_Load(object sender, EventArgs e)
        {
```

```
            GenerateMap();
            DisplayMapOnPictureBox();
        }

        private void buttonGenerateMap_Click(object
sender, EventArgs e)
        {
            GenerateMap();
            DisplayMapOnPictureBox();
        }

        private void buttonCostUniform_Click(object
sender, EventArgs e)
        {
            SearchTreeNode x = new SearchTreeNode();
            SearchTreeNode y = new SearchTreeNode();
            y.coordX =
Convert.ToInt32(textBoxFinalStateX.Text);
            y.coordY =
Convert.ToInt32(textBoxFinalStateY.Text);
            y.heuristic = 0;
            y.id = y.coordX + 10000 * y.coordY;
            x.coordX =
Convert.ToInt32(textBoxInitialStateX.Text);
            x.coordY =
Convert.ToInt32(textBoxInitialStateY.Text);
            x.heuristic = DiagonalDistance(x, y);
            x.id = x.coordX + 10000 * x.coordY;
            x.level = 0;
            x.parrent = null;
            x.cost = 0;
            m[x.coordX, x.coordY] = 0;
            m[y.coordX, y.coordY] = 0;
            ExploredStates = new Hashtable();
            StatesToExplore = new ArrayList();
            myComparer = new CostComparer();
            Search(x, y);
        }
```

5. Se va scrie programul pentru rezolvarea aceleiaşi probleme de la punctul anterior dar de această dată fiecare cotă reprezintă costul de acces al careului în cauză (se renunță deci la costul deplasării în linie dreaptă şi în diagonală).

Lucrarea 6 Strategii informate de căutare: strategiile de căutare best first şi greedy

6.1 Ce este o euristică

Conform dicționarului explicativ al limbii române euristic,-ă,euristici cu referire la procedee şi metodologie înseamnă ceva care serveşte la descoperirea unor cunoştinte noi. În ceea ce priveşte strategiile de căutare, euristica este o informație care ajută la luarea unei decizii cu privre la noul nod ce urmează a fi explorat, informație care ar trebui să conducă mai repede la atingerea stării finale. Russell şi Norvig spun în lucrarea lor de referință "… information about the state space can prevent algorithms from blundering about in the dark". Acesta este rolul eurisiticilor, de a preveni ca o strategie să caute "în întuneric" încercând să deschidă o cale mai bună, mai scurtă, către starea finală. Strategiile euristice de căutare se mai numesc şi strategii informate de căutare.

Euristica este valoarea estimată a distanței de la starea curentă la starea finală, iar funcția care evaluează acest lucru o notăm în general cu *h*, i.e.

h(n) = costul estimat al celei mai ieftine căi de la nodul dat la nodul soluție

Deci euristica este o funcție care se aplică unui nod dintr-un arbore de căutare, mai exact stării asociate lui, şi se foloseşte la ordonarea nodurilor în lista de noduri ce urmează a fi explorate. Astfel, lista este ordonată după euristică în cazul căutării Greedy respectiv după cost plus euristică în căutarea cazul A*, aşa cum se va vedea în cele ce urmează.

Pentru ilustrarea mai clară a conceptului de euristică iată câteva exemple de euristici:

- Pentru explorarea unei hărti funcția *h(n)* poate fi distanța în linie dreaptă de la punctul current la punctul final (numită şi Straight Line Distance Heuristic h_{SLD}).
- Pentru umplerea unui rucsac cu obiecte: *h(n)* poate fi spațiul liber care rămîne în rucsac (sau invers volumul/greutatea obiectului ales).
- Pentru un puzzle *h(n)* poate fi numărul de piese aflate în pozitii greşite sau suma distanțelor pieselor până la poziția lor finală, sumă evaluată ca suma distanțelor Manhattan a fiecărei piese până la poziția ei finală (distanța Manhattan se mai numeşte şi "city block distance" şi este distanța în linie dreaptă fără a merge în diagonală).

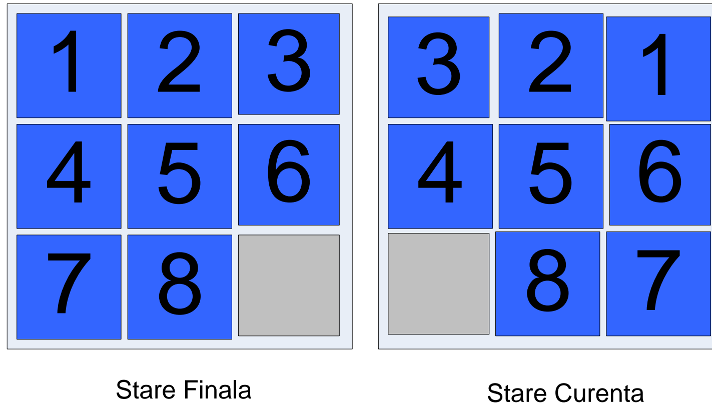


Figura 6.1 *Stare a unui puzzle 3x3 cu valorile celor două euristici*

Se face relevantă următoarea observație. Căutarea cu cost uniform utilizează o informație pe baza căreia se decide care este nodul cel mai bun dar atenție aceasta nu este o căutare euristică deoarece această măsură este un simplu cost care facilitează găsirea soluției optimale ca şi cost dar nu împinge strategia spre starea finală. Este extrem de relevant să putem distinge între ce înseamnă un cost şi ce înseamnă o euristică, dacă un cost ajută strategia la a găsi soluția de cost optim euristica aduce avantaje în necesitățile de timp şi necesitățile de spațiu.

6.2 Strategia de căutare Best First

Strategia Best-First este un simplu model teoretic şi idealizat de strategie de căutare euristică. Aceasta presupune explorarea nodului şi alegerea ca successor a celui mai bun nod. Sigur acest lucru este imposibil de realizat în general pentru că este necesară o funcție care evaluează care este nodul cel mai bun, iar dacă o astfel de funcție ar exista nici nu mai am avea nevoie de un arbore de căutare pentru că strategia de căutare ar merge direct către soluție având complexitate de timp şi spațiu liniară. Deci noțiunea de căutare best-first rămâne doar ca model idealizat al strategiilor euristice.

6.3 Strategia de căutare Greedy

Strategia greedy alege spre explorare întotdeauna nodul care este cel mai apropiat de starea finală, deci cel cu cea mai bună euristică. În acest sens se poate face o analogie cu strategia cu cost uniform care alegea întotdeauna nodul cu costul cel mai bun. Dacă strategia de căutare cu cost uniform păstra o listă sortată după valoarea costului, Greedy va păstra o listă sortată după valoarea euristicii.

➢ ***Complexitate de timp şi spațiu***

Complexitatea de timp a strategiei Greedy este tot $T = O\left(b^d\right)$ în cel mai defavorabil caz. La fel şi complexitatea de spațiu $S = O\left(b^d\right)$. În practică însă, de cele mai multe ori nu ne confruntăm cu cazuri defavorabile pentru Greedy şi timpul de calcul respectiv spațiul sunt mult mai mici.

➢ ***Completitudine şi optimalitate***

Strategia greedy nu este completă în arbori infiniți, ciclurile conducând evident la blocaje. Dacă se folosesc mecanisme de evitare a ciclurilor, strategia este completă. Totodată, strategia nu este nici optimală deoarece nu ține cont de costuri.

➢ ***Implementare***

Greedy se implementeaza identic cu strategia cu cost uniform, doar că nodurile sunt păstrate în listă ordonate după euristică şi nu după cost. Se poate însă implementa şi pe paradigma de la căutarea în adâncime, folosindu-se o stivă şi păstrându-se un consum mai mic memorie dar conducând mai încet spre soluție.

➢ ***Avantaje şi dezavantaje***

Avantajul major este că strategia greedy oferă rapid soluții şi în practică greedy se dovedeşte a fi o alegere foarte bună atunci când nu se doreşte găsirea unei soluții optimale. Dezavantajul este că strategia nu este completă şi nici optimală, deasemenea când se implementează identic cu uniform-cost poate conduce la necesități de memorie ridicate.

6.4 Exerciții

1. Să se scrie programul pentru problema puzzleului 3x3 din capitolele anterioare folosind strategia informată Greedy. Se va rezolva pentru ambele euristici anterior introduse şi se vor compara rezultatele obținute: adâncime soluție, număr de noduri explorate etc.

2. Se consideră spațiul cu obstacole din capitolul anterior. Să se scrie programul care rezolvă această problemă folosind strategia Greedy. Se va folosi interfața din figura 6.2.

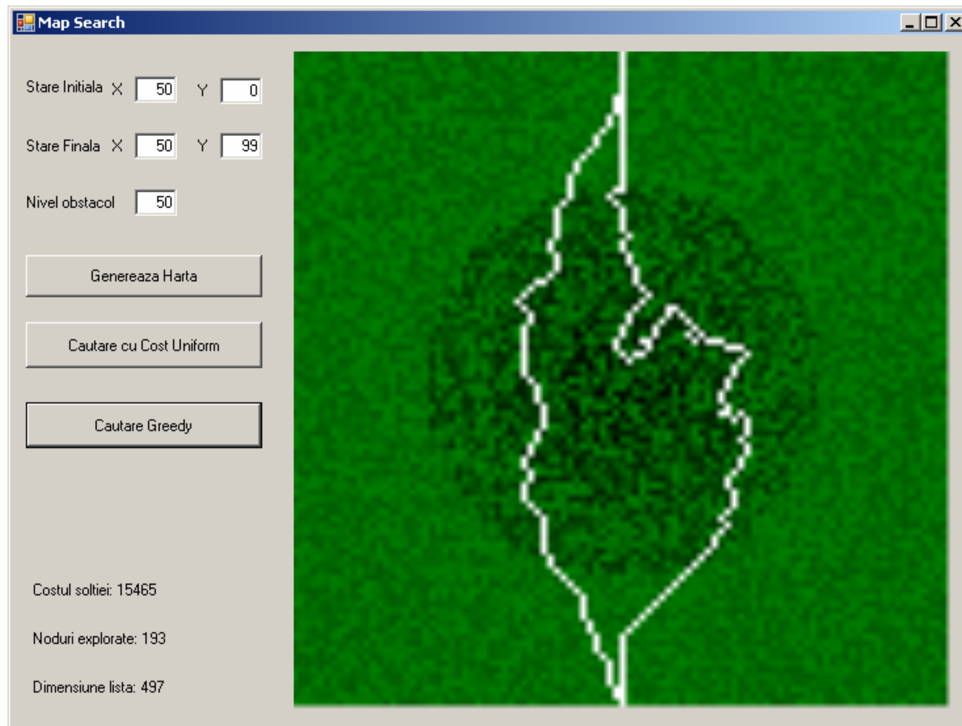


Figura 6.2 *Exemplu de interfață pentru problema spațiului cu obstacole*

Inidicații pentru rezolvare

➢ În clasa pentru nodul arborelui de căutare se adaugă euristica

```
class SearchTreeNode
    {
        public SearchTreeNode parrent = null;
        public int cost;
        public int heuristic;
        public int level;
        public long id = 0;
        public int coordX = 0;
        public int coordY = 0;
```

```
        public int nodeOperator = 0; //degreess from 90 to
360
    }
```

- Se foloseşte un IComparer pe bază de euristică

```
class HeuristicComparer : IComparer
    {

        int IComparer.Compare(Object x, Object y)
        {
            SearchTreeNode n1 = (SearchTreeNode) x ;
            SearchTreeNode n2 = (SearchTreeNode) y ;
            if (n1.heuristic > n2.heuristic)
            {
                return 1;
            }
            else
            {
                if (n1.heuristic < n2.heuristic)
                {
                    return -1;
                }
                else
                {
                    return 0;
                }
            }
        }
```

- Restul programului este identic cu cel din lucrarea anterioară, doar sortarea se va face după euristică aşa cum rezultă din următorul cod sursă

```
private void buttonGreedy_Click(object sender, EventArgs
e)
        {
            SearchTreeNode x = new SearchTreeNode();
            SearchTreeNode y = new SearchTreeNode();
            y.coordX =
Convert.ToInt32(textBoxFinalStateX.Text);
            y.coordY =
Convert.ToInt32(textBoxFinalStateY.Text);
```

```
            y.heuristic = 0;
            y.id = y.coordX + 10000 * y.coordY;
            x.coordX =
Convert.ToInt32(textBoxInitialStateX.Text);
            x.coordY =
Convert.ToInt32(textBoxInitialStateY.Text);
            x.heuristic = DiagonalDistance(x, y);
            x.id = x.coordX + 10000 * x.coordY;
            x.level = 0;
            x.parrent = null;
            x.cost = 0;
            m[x.coordX, x.coordY] = 0;
            m[y.coordX, y.coordY] = 0;
            ExploredStates = new Hashtable();
            StatesToExplore = new ArrayList();
            myComparer = new HeuristicComparer();
            Search(x, y);
        }
```

3. Se va scrie programul pentru rezolvarea aceleiaşi probleme de la punctul anterior dar de această dată fiecare cotă reprezintă costul de acces al careului în cauză (se renunță deci la costul deplasării în linie dreaptă şi în diagonală).

Lucrarea 7 Alegerea unei euristici

Pentru problemele discutate în acest material, dar şi pentru multe alte probleme din practică, este uşor să inventăm euristici. Nu în utimul rând există şi programe dedicate pentru aceasta. Problema care se pune acum este de a decide din mai multe euristici care este mai bună. Astfel, dacă pentru două euristici h_1, h_2 avem $h_1(n)>h_2(n)$ atunci spunem că h_1 domină pe h_2 şi utilizarea euristicii dominatoare duce întotdeauna la un rezultat mai bun, adică vor fi explorate mai puține noduri pentru a găsi soluția. La modul general, dacă se cunosc mai multe euristici monotone $h_1,h_2,...,h_m$ şi nu se ştie care euristică domină se poate utiliza $h(n)=max(h_1,h_2,...,h_m)$.

7.1 *Probleme cu constrângeri*

O gamă aparte de probleme decizionale sunt problemele cu constrângeri, în care un set de variabile care pot lua diverse valori trebuie să satisfacă anumite constrângeri pentru ca starea finală să fie atinsă. Între cele mai cunoscute probleme cu constrângeri, studiate în liceu sau în primii ani de facultate, sunt: aşezarea a 8 regine pe o tablă de şah, colorarea unei hărți etc. În acest caz sunt în general preferate urmatoarele trei euristici:

- Euristica celei mai constrânse variabile – variabila care poate lua cele mai puține valori este aleasă.
- Euristica variabilei cu cea mai mare putere de constrângere – variabila care constrânge cele mai multe variabile este aleasă.
- Euristica valorii cu cea mai mică putere de constrângere – valoarea care oferă cea mai mare libertate în alegerile ulterioare este aleasă.

Pentru evitarea stărilor fără rezolvare se foloseşte mecanismul numit forward-checking. De exemplu: la problema aşezării celor 8 regine, se încearcă aşezarea unei noi regine doar pe pozițiile pe care nu este deja atacată.

7.2 *Rolul unei euristici*

Rolul unei euristici este de a aduce factorul efectiv de ramificație către 1 deoarece în acest caz indiferent de adâncimea soluției necesitățile de timp şi spațiu nu vor înregistra o creştere exponențială. Sigur, pentru marea parte a problemelor o euristică care să realizeze acest lucru nu se poate găsi, dar se poate constata că euristicile reuşesc în general să reducă semnificativ factorul de ramificație. Pentru evaluarea eficienței unei euristici se poate calcula factorul efectiv de ramificație aşa cum a fost el definit în capitolul 2.

7.3 Exerciții

1. Scrieți programul pentru rezolvarea problemei aranjării a 8 regine pe tabla de şah folosind diverse constrângeri şi comparați rezultatele obținute.

2. Scrieți programul pentru rezolvarea problemei colorării unei hărți folosind diverse constrângeri şi comparați rezultatele obținute.

3. Scrieți programul pentru rezolvarea problemei unui puzzle 3x3 cu cele două euristici din secțiunea anterioară şi comparați rezultatele obținute. Care dintre cele două euristici este cea dominatoare.

Lucrarea 8 Strategii informate de căutare: strategia de căutare A*

Strategia A* poate fi văzută ca o combinație între strategia de căutare greedy şi strategia de căutare cu cost uniform, A* combinând cele mai bune caracteristici ale acestora. Pentru aceasta A* utilizează funcția de evaluare a nodului: $f(n)=g(n)+h(n)$ unde $g(n)$ este costul de la starea inițială la starea curentă (funcția de la căutarea cu cost uniform) iar $h(n)$ este costul celui ieftin drum de la starea curentă la starea finală (euristica de la căutarea Greedy).

- ***Complexitate de timp şi spațiu***

Complexitatea de spațiu în cel mai rău caz este exponențială şi la A* dacă nu este respectată condiția: $|h(n)-h^*(n)|<O(lg(h^*(n)))$ *($h^*(n)$ este costul real până la starea finală)*. Totuşi pentru marea parte a situațiilor practice A* solicită resurse mai mici decât ceilalți algoritmi de complexitate exponențială.

- ***Completitudine şi optimalitate***

A* este completă şi este optimală cu condiția ca euristica $h(n)$ să fie admisibilă. O euristică se numeşte admisibilă dacă nu supraestimează costul atingerii stării finale plecând din starea curentă. Eurisiticile admisibile se mai numesc şi euristici optimiste deoarece întodeauna estimează că este mai uşor de a ajunge la final decât este în realitate. În baza discuției din capitolul anterior se poate utiliza $h(n)=max(h_1,h_2,\dots,h_m)$ iar dacă fiecare din euristicile h_i este admisibilă atunci şi $h(n)$ este adimisibilă.

- ***Implementare***

A* se implementează folosind o listă sortată crescător după funcția *f*.

- ***Avantaje şi dezavantaje***

A* este cea mai bună strategie de căutare, se poate demostra chiar că este optimal-eficientă, adică orice altă strategie care explorează mai puține noduri decât A* riscă să nu găsească cea mai bună soluție. Dezavantajul este că necesită resurse de memorie relative ridicate. Pentru a înlătura acest dezavantaj o potențială soluție este combinarea lui A* cu Iterative Deepening combinație cunoscută sub numele de Iterative Deepening A* sau IDA*.

8.1 Demonstrație asupra optimalității lui A*

Este evident că A* este completă, costul uniform conducând inevitabil la explorarea nodurilor până la găsirea unei soluții. Pentru a demonstra că A* este optimală recurgem la reducere la absurd şi presupunem că A* nu este optimală. Fie în acest caz c_o costul optim al căii către soluție şi fie x o stare finală sub-optimală returnată de A*, avem deci:

$g(x) > c_o$ (1)

Fie un nod n care aparține căii optimale, deoarece h este o euristica admisibilă:

$c_o >= f(n)$ (2)

Deoarece A* nu a deschis starea n avem:

$f(n) >= f(x)$ (3)

Din (2) si (3) rezultă

$c_o >= f(x)$ (4)

Dar din moment ce x este starea finală avem:

$h(x) = 0$ (5)

Acum din (4) şi (5) rezultă:

$c_o >= g(x)$ (6)

Şi deci x nu poate fi suboptimal deoarece costul său este mai mic sau egal cu costul optimal, ceea ce este o contradicție cu ipoteza (1) şi deci A* este optimal.

8.2 Exerciții

1. Se consideră harta din figură şi distanțele în linie dreaptă (h_{SLD}) din tabel. Se cere:

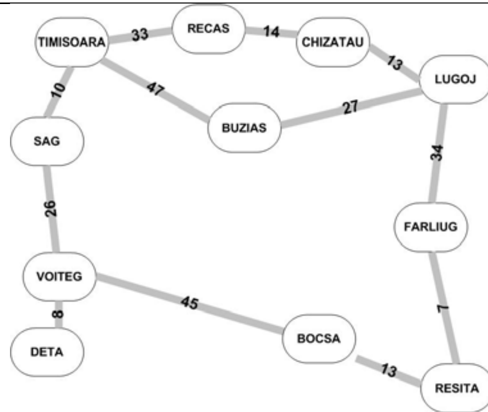


Figura 8.1 *Hartă oarecare.*

Oras	h_{SLD} (Resita)
Timisora	100
Recas	90
Lugoj	30
Buzias	65
Farliug	30
Bocsa	10
Sag	95
Voiteg	40
Deta	70

Figura 8.2 *Euristica distanței în linie dreaptă pentru punctele de pe hartă.*

a) explicați căutarea greedy şi ilustrați ordinea în care nodurile sunt explorate pentru găsirea unui drum Timişoara-Reşita

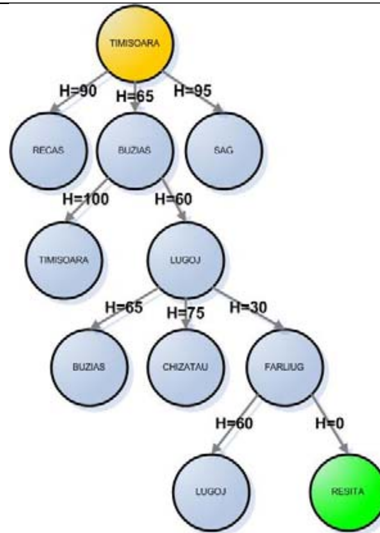


Figura 8.3 *Arborele de căutare pentru strategia Greedy*

b) Explicați căutarea A* şi ilustrați ordinea în care nodurile sunt explorate pentru găsirea unui drum Timişoara-Reşița

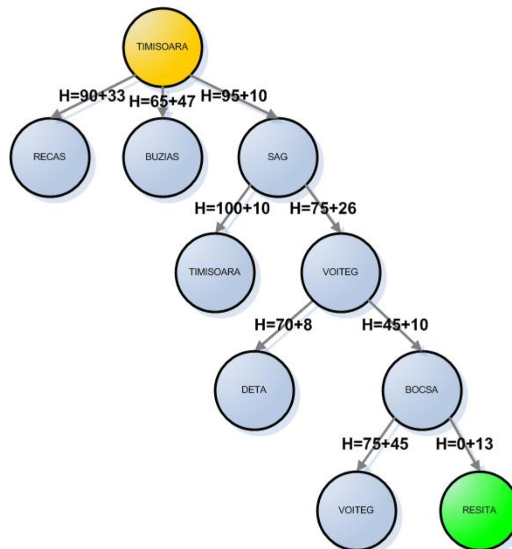


Figura 8.4 *Arborele de căutare pentru strategia* A*

c) Care dintre cele două căutări a ajuns mai rapid la solutie şi care a găsit soluția optimală? Observați că arborele de căutare are aceeaşi adâncime pentru ambele strategii, însă doar A* a găsit soluția optimală (Greedy a parcurs 115km iar A* 94km)

2. Se consideră spațiul cu obstacole din capitolul anterior. Să se scrie programul care rezolvă această problemă folosind strategia A*. Se va folosi interfața din figura 8.5.

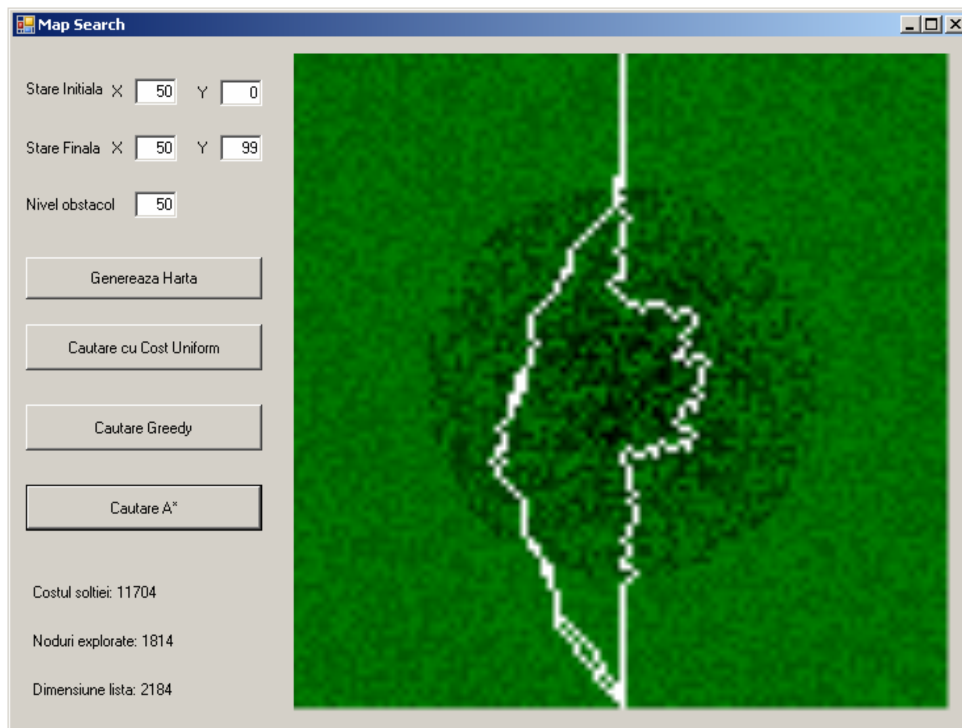


Figura 8.5 *Exemplu de interfață pentru problema spațiului cu obstacole*

```
        private void buttonAstar_Click(object sender,
EventArgs e)
        {
            SearchTreeNode x = new SearchTreeNode();
            SearchTreeNode y = new SearchTreeNode();
            y.coordX =
Convert.ToInt32(textBoxFinalStateX.Text);
```

```
            y.coordY =
Convert.ToInt32(textBoxFinalStateY.Text);
            y.heuristic = 0;
            y.id = y.coordX + 10000 * y.coordY;
            x.coordX =
Convert.ToInt32(textBoxInitialStateX.Text);
            x.coordY =
Convert.ToInt32(textBoxInitialStateY.Text);
            x.heuristic = DiagonalDistance(x, y);
            x.id = x.coordX + 10000 * x.coordY;
            x.level = 0;
            x.parrent = null;
            x.cost = 0;
            m[x.coordX, x.coordY] = 0;
            m[y.coordX, y.coordY] = 0;
            ExploredStates = new Hashtable();
            StatesToExplore = new ArrayList();
            myComparer = new CostPlusHeuristicComparer();
            Search(x, y);
        }
```

3. Se va scrie programul pentru rezolvarea aceleiaşi probleme de la punctul anterior dar de această dată fiecare cotă reprezintă costul de acces al careului în cauză (se renunță deci la costul deplasării în linie dreaptă şi în diagonală). Se mai poate folosi euristica distanței diagonale în acest caz? Explicați implicațiile, propuneți o altă euristică.

4. Acelaşi program de la punctele 3 şi 4 dar care desenează cu culori diferite traseele urmate de Uniform-cost, Greedy, A* şi afişează casete comparative cu parametrii fiecărei strategii (costul drumului, timp, memorie etc.).

Lucrarea 9 Strategii de căutare în spații cu incertitudini

Un caz aparte sunt scenariile cu incertitudini, în acest caz agentul inteligent nu este omniscient cu privire la spațiul stărilor. De exemplu în cadrul problemei găsirii drumului optim pe o hartă, harta era generată anterior şi cunoscută de către agent. Dar, putem la fel de bine trata şi cazul în care agentul nu cunoaşte harta şi trebuie să o descopere singur. Un astfel de scenariu nu este simplu de tratat din punct de vedere al optimalității şi completitudinii, din fericire putem construi o soluție relativ eficientă bazată pe noțiunile introduse până acum. În cele ce urmează vom trece direct la rezolvarea problemei găsirii drumului pe harta 100x100 din capitolele anterioare la care adăugăm faptul că harta nu este cunoscută în prealabil şi agentul trebuie să o descopere. Recurgem la simplificarea în baza căreia punctele de pe hartă au valori binare 1 sau 0 după cum sunt accesibile sau nu. Se va folosi o interfață apropiată de cea din figura 9.1. Sintetizăm indicațiile pentru rezolvare după cum urmează:

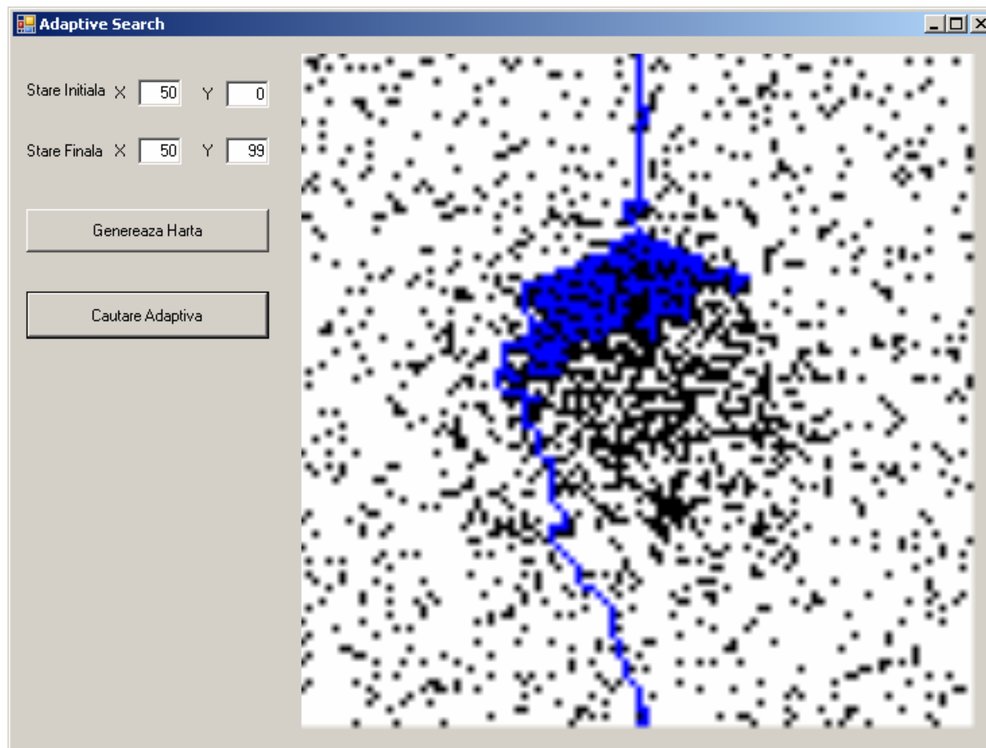


Figura 9.1 *Drumul găsit de agentul inteligent folosind strategia de căutare adaptivă*

- Se definesc costurile deplasării şi se generează harta într-o manieră similară cu cea din capitolele aterioare

```
private const int straigthMovement = 100; // costul
miscarii in linie dreapta
        private const int diagonalMovement = 141; //
costul miscarii in diagonala sqrt(2) =
1.4142135623730950488016887242097

        private int[,] m = new int[100,100]; // matrice
utilizata pentru memorarea hartii 100x100

        // harta este generata aleator cu obstacole a
caror densitate creste spre centrul hartii
        private void GenerateMap()
        {
            int i,j;
            int treshold;
            Random r = new Random();
            for (i = 0; i < 100; i++)
            {
                for (j = 0; j < 100; j++)
                {

                    if ((Math.Pow(i - 50, 2) + Math.Pow(j
- 50, 2)) < Math.Pow(10, 2))
                    {
                        treshold = 4;
                    }
                    else
                    {
                        if ((Math.Pow(i - 50, 2) +
Math.Pow(j - 50, 2)) < Math.Pow(20, 2))
                        {
                            treshold = 6;
                        }
                        else
                        {
                            if ((Math.Pow(i - 50, 2) +
Math.Pow(j - 50, 2)) < Math.Pow(30, 2))
                            {
                                treshold = 8;
                            }
                            else
```

```
                            {
                                treshold = 9;
                            }
                        }
                    }
                    if (r.Next(10) < treshold)
                    {
                        m[i, j] = 0;
                    }
                    else
                    {
                        m[i, j] = 1;
                    }
                }
            }
        }

        // harta din matrice este desenata in PictureBox
        private void DisplayMapOnPictureBox()
        {
            Bitmap bmp = new Bitmap(100, 100);
            int i, j;
            for (i = 0; i < 100; i++)
            {
                for (j = 0; j < 100; j++)
                {
                    if (m[i , j ] == 0)
                    {
                        bmp.SetPixel(i, j, Color.White);
                    }
                    if (m[i , j ] == 1)
                    {
                        bmp.SetPixel(i, j, Color.Black);
                    }
                }
            }
            pictureBoxMap.Image = bmp;
            pictureBoxMap.SizeMode =
PictureBoxSizeMode.StretchImage;
        }
```

- În funcția search are loc următoarea modificare, deoarece nu este sigur că agentul va putea fi mutat în succesorul generat de funcție (ar putea fi un

obstacol) se apelează `roboPosition = MoveBetweenTreeNodes(roboPosition, currentNode)`. După acest apel, către funcția care încearcă să pună agentul în succesor, dacă poziția este egală cu nodul succesor atunci înseamnă că succesorul a fost accesibil şi deci nu era un obstacol.

```
        //algoritmul de cautare (functia Search)
        private void PathFind(SearchTreeNode startNode,
SearchTreeNode targetNode)
        {
            int movements = 0;
            Hashtable ExploredStates = new Hashtable();
            ArrayList StatesToExplore = new ArrayList();
            bool solutionFound = false;
            SearchTreeNode currentNode;
            SearchTreeNode roboPosition;
            SearchTreeNode rootNode;
            roboPosition = startNode;
            rootNode = startNode;

            StatesToExplore.Add(rootNode);
            // search continues until a solution is found
or no states to explore axists
            while ((StatesToExplore.Count != 0) &&
(solutionFound != true))
            {
                currentNode =
GetNextSuccessor(StatesToExplore);


                    if
(!(ExploredStates.ContainsKey(currentNode.id))) // current
node was not explored
                    {
                        // inca nu se stie daca pozitia
din nodul curent este sau nu accesibila
                        // se incearca mutarea robotului
din pozitia curenta in pozitia din nodul curent
                        roboPosition =
MoveBetweenTreeNodes(roboPosition, currentNode);
                        // se marcheaza pe harta noua
pozitie a robotului
                        Bitmap bmp = new
Bitmap(pictureBoxMap.Image);
```

```
bmp.SetPixel(roboPosition.coordX, roboPosition.coordY,
Color.Blue); pictureBoxMap.Image = bmp;
                        // se verifica daca pozitia
robotului este identica cu pozitia din nodul curent
                        if (roboPosition.id ==
currentNode.id)
                        {
                            // daca da se adauga
succesorii nodului curent in lista
                            AddSuccessors(currentNode,
StatesToExplore, targetNode);

ExploredStates.Add(currentNode.id, currentNode);
                        }
                        // daca nu, pozitia din nodul
curent nu este accesibila
                        else
                        {
                            currentNode.Accessible =
false;
                        }
                    }
                // se verifica daca pozitia curenta este
chiar solutia
                if (Solution(roboPosition, targetNode))
                {

System.Windows.Forms.MessageBox.Show("Target Reached" +
movements.ToString());
                    solutionFound = true;
                }
            }
            if (solutionFound == false) {
System.Windows.Forms.MessageBox.Show("There is no
solution"); }
        }
```

➢ Funcțiile de adăugare a succesorilor, extragere a noului nod de explorat şi de verificare a coordonatelor sunt similare cu cele din capitolele anterioare.

```
// se adauga succesorii nodului curent in lista
        // atentie, nu toti succesori sunt accesibili,
putand fi si obstacole, lucru care se va afla cand robotul
```

```
incearca sa miste in acel nod
        // dupa ce au fost adaugati, lista de succesori
este sortata
        private void AddSuccessors(SearchTreeNode node,
ArrayList StatesToExplore, SearchTreeNode target)
        {
            int i, newX, newY;
            SearchTreeNode n;
            for (i = 0; i < 8; i++)
            {
                newX = -1; newY = -1;
                switch ( i*45 )
                {
                    case 0: newX = node.coordX; newY =
node.coordY + 1; break;
                    case 45: newX = node.coordX + 1; newY
= node.coordY + 1; break;
                    case 90: newX = node.coordX + 1; newY
= node.coordY; break;
                    case 135: newX = node.coordX + 1; newY
= node.coordY - 1; break;
                    case 180: newX = node.coordX; newY =
node.coordY - 1; break;
                    case 225: newX = node.coordX - 1; newY
= node.coordY - 1; break;
                    case 270: newX = node.coordX - 1; newY
= node.coordY; break;
                    case 315: newX = node.coordX - 1; newY
= node.coordY + 1; break;
                }
                if (CoordinatesInsideBounds(newX, newY))
                {
                    n = new SearchTreeNode();
                    n.parrent = node;
                    n.Accessible = true;
                    n.nodeOperator = i * 45;
                    n.coordX = newX;
                    n.coordY = newY;
                    n.id = n.coordX + n.coordY * 10000;
                    n.level = node.level + 1;
                    n.heuristic = DiagonalDistance(n,
target);
                    node.succesors[i] = n;
                    StatesToExplore.Add(n);
```

```
                }
            }
            IComparer myComparer = new
HeuristicComparer();
            StatesToExplore.Sort(myComparer);
        }

        // verifica daca coordonatele sunt in interiorul
hartii
        private bool CoordinatesInsideBounds(int x, int y)
        {
            if ((x >= 0) && (x < 100) && (y >= 0) && (y <
100))
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        // returneaza urmatorul succesor, adica starea cea
mai apropiata de starea finala
        private SearchTreeNode GetNextSuccessor(ArrayList
StatesToExplore)
        {
            SearchTreeNode state =
(SearchTreeNode)StatesToExplore[0];
            StatesToExplore.RemoveAt(0);
            return state;
        }
```

- Avem nevoie acum de câteva funcții care nu au mai fost întâlnite în cazul implementărilor anterioare, este vorba de funcțiile care încearcă mutarea robotului între nodurile arborelui de căutare. Se disting aici două cazuri distincte: cazul în care se încearcă mutarea dintr-un nod fiu într-un nod părinte sau într-un nod care nu este terminal respectiv cazul în care se încearcă mutarea într-un nod terminal. În primul caz drumul este accesibil iar în cel de-al doilea caz trebuie verificat dacă mutarea poate fi făcută. Funcțiile `MoveRobotFromChildToParrentNode`, `MoveRobotFromParrentToChildNode` adresează primul caz iar funcția `MoveRobotFromParrentToChildNodeIfPossible` adresează cel de-

al doilea caz. Funcția `MoveBetweenTreeNodes` foloseşte cele trei funcții pentru a deplasa robotul între nodurile arborelui de căutare.

```
// se incearca mutarea robotul intre doua noduri ale
arborelui de cautare
        // atentie, pozitia noua este posibil sa nu poata
fi accesata daca este obstacol
        private SearchTreeNode
MoveBetweenTreeNodes(SearchTreeNode position,
SearchTreeNode newposition)
        {
            SearchTreeNode n;
            Hashtable pList = new Hashtable();
            n = newposition;
            // se construieste o lista a parintilor
nodului tinta
            while (n != null) { pList.Add(n.id, n); n =
n.parrent; }
            n = position;
            // robotul este mutat din parinte in parinte
pana cand se gaseste parintele comun
            while (!(pList.ContainsKey(n.id))) { position
= MoveRobotFromChildToParrentNode(n); n = n.parrent; }
            // acum robotul este in parintele comun
            if (position.id == newposition.id)
            {
                // daca este chiar pozitia noua ne putem
opri
                return position;
            }
            else
            {
                // altfel continuam deplasarea din parinte
in nodul tinta
                Stack sList = new Stack();
                SearchTreeNode s;
                s = newposition.parrent;
                // adaugam intr-o stiva toti parintii
nodului tinta
                while (s != n) { sList.Push(s); s =
s.parrent; }
                // robotul se muta din parinte in fiu pana
cand ajunge la parintele nodului tinta
                while (sList.Count != 0) { s =
```

```
(SearchTreeNode)sList.Pop(); position =
MoveRobotFromParrentToChildNode(s); }
                // acum robotul este in parintele nodului
tinta
                // se efectueaza mutarea daca este
posibila
                position =
MoveRobotFromParrentToChildNodeIfPossible(newposition);
                return position;
            }
        }

        // robotul este mutat din nodul fiu in nodul
parinte
        // atentie, aceasta mutare este tot timpul
posibila, deoarece intr-un nod fiu intotdeauna s-a ajuns
dintr-un nod parinte
        private SearchTreeNode
MoveRobotFromChildToParrentNode(SearchTreeNode roboNode)
        {
            return roboNode.parrent;
        }

        // robotul este mutat din nodul parinte in nodul
fiu
        // atentie, aceasta functie este doar pentru cazul
cand nodul fiu este deja explorat si deci accesibil
        private SearchTreeNode
MoveRobotFromParrentToChildNode(SearchTreeNode roboNode)
        {
            return roboNode;
        }

        // robotul este mutat din nodul fiu in nodul
parinte
        // atentie, aceasta mutare nu este tot timpul
posibila, functia testeaza asadar daca nodul fiu este
accesibil
        private SearchTreeNode
MoveRobotFromParrentToChildNodeIfPossible(SearchTreeNode
roboNode)
        {
            SearchTreeNode n = null;
            if (m[roboNode.coordX, roboNode.coordY] == 0)
```

```
// daca pe harta in acest punct nu este un obstacol
            {
                if ((roboNode.nodeOperator == 0) ||
(roboNode.nodeOperator == 90) || (roboNode.nodeOperator ==
180) || (roboNode.nodeOperator == 270))
                {
                    n = roboNode;
                }
                else // se interzice pe ramura else ca
mutarea sa se faca in diagonala pe langa un obstacol
                {
                    switch (roboNode.nodeOperator)
                    {
                        case 45:
                            if
((m[roboNode.parrent.coordX, roboNode.parrent.coordY + 1]
== 0) && (m[roboNode.parrent.coordX + 1,
roboNode.parrent.coordY ] == 0))
                            {
                                n = roboNode;
                            }
                            else
                            {
                                n = roboNode.parrent;
                            }
                            break;
                        case 135:
                            if
((m[roboNode.parrent.coordX, roboNode.parrent.coordY - 1]
== 0) && (m[roboNode.parrent.coordX + 1,
roboNode.parrent.coordY] == 0))
                            {
                                n = roboNode;
                            }
                            else
                            {
                                n = roboNode.parrent;
                            }
                            break;
                        case 225:
                            if
((m[roboNode.parrent.coordX, roboNode.parrent.coordY - 1]
== 0) && (m[roboNode.parrent.coordX - 1,
roboNode.parrent.coordY] == 0))
```

```
                                    {
                                        n = roboNode;
                                    }
                                    else
                                    {
                                        n = roboNode.parrent;
                                    }
                                    break;
                                case 315:
                                    if ((m[roboNode.parrent.coordX
- 1, roboNode.parrent.coordY] == 0) &&
(m[roboNode.parrent.coordX, roboNode.parrent.coordY + 1]
== 0))
                                    {
                                        n = roboNode;
                                    }
                                    else
                                    {
                                        n = roboNode.parrent;
                                    }
                                    break;
                        }
                    }
                }
                else // daca pe harta in acest punct este un
obstacol atunci robotul se va afla tot in nodul parinte
                {
                    n = roboNode.parrent;
                }
                return n;
            }
```

➢ Următoarele funcțiile sunt din nou similare cu cele din capitolele anterioare.

```
        // verifica daca nodul curent nu este chiar nodul
tinta
        private bool Solution(SearchTreeNode node,
SearchTreeNode targetnode)
        {
            if ((node.coordX == targetnode.coordX) &&
(node.coordY == targetnode.coordY))
            {
```

```
                return true;
            }
            else
            {
                return false;
            }
        }

        // calculeaza distanta diagonala intre doua puncte
        private int DiagonalDistance(SearchTreeNode
current, SearchTreeNode target)
        {
            int xd = Math.Abs(current.coordX -
target.coordX);
            int yd = Math.Abs(current.coordY -
target.coordY);
            return straigthMovement * (Math.Max(xd, yd) -
Math.Min(xd, yd)) + diagonalMovement * Math.Min(xd, yd);
        }

        private void Form1_Load(object sender, EventArgs
e)
        {
            GenerateMap();
            DisplayMapOnPictureBox();
        }

        private void buttonGenerateMap_Click(object
sender, EventArgs e)
        {
            GenerateMap();
            DisplayMapOnPictureBox();
        }

        private void buttonSearch_Click(object sender,
EventArgs e)
        {
            SearchTreeNode x = new SearchTreeNode();
            SearchTreeNode y = new SearchTreeNode();
            y.coordX =
Convert.ToInt32(textBoxFinalStateX.Text);
            y.coordY =
Convert.ToInt32(textBoxFinalStateY.Text);
            y.heuristic = 0;
```

```
            y.id = y.coordX + 10000 * y.coordY;
            x.coordX =
Convert.ToInt32(textBoxInitialStateX.Text);
            x.coordY =
Convert.ToInt32(textBoxInitialStateY.Text);
            x.heuristic = DiagonalDistance(x, y);
            x.id = x.coordX + 10000 * x.coordY;
            x.level = 0;
            x.parrent = null;
            m[x.coordX, x.coordY] = 0;
            m[y.coordX, y.coordY] = 0;
            PathFind(x, y);
        }
```

9.1 Exerciții

1. Este algoritmul de căutare descris anterior complet? Dar optimal? Modificați programul astfel încât agentul să găsească drumul de cost minim.

Lucrarea 10 Întrebări recapitulative şi probleme

1. Se consideră o tablă de şah de dimensiune *n*x*n* şi se doreşte parcurgerea celor *n*x*n* careuri o singură dată folosind săritura calului. Se cere: a) Care este complexitatea spațiului stărilor pentru această problemă b) Propuneți două strategii neinformate de căutare care să fie optimale c) Pentru strategiile propuse în funcție de factorul de ramificație aferent problemei şi de adâncimea căutarii precizați complexitatea de timp şi spațiu d) Este posibilă folosirea căutării limitate în adâncime pe această problema şi dacă da atunci este această strategie completă şi optimală?

Indicații pentru rezolvare

a) Complexitatea spațiului stărilor este $O\left(2^{n^2}\right)$.

b) Strategiile breadth-first şi iterative deepening sunt optimale deoacrece costul operatorilor este egal.

c) Factorul de ramificație este 8 iar adancimea este $n \times n = n^2$ deoarece calul trebuie să treacă prin fiecare careu o singura data. Pentru *breadth-first* avem $T_{BF} = S_{BF} = O\left(b^d\right) = O\left(8^{n^2}\right)$ iar pentru iterative-deepening avem $T_{ID} = O\left(b^d\right) = O\left(8^{n^2}\right)$, $S_{ID} = O\left(b \cdot d\right) = O\left(8 \cdot n^2\right)$.

d) Da, căutarea limitată în adâncime este completă deoarece se cunoaşte adâncimea soluției n^2 şi optimală deoarece toate soluțiile se află pe limita de adâncime.

2. Se consideră spațiul cu obstacole din figura 10.1 şi se doreşte găsirea unui drum de la B la A. Numerele supraunitare din careuri reprezintă costurile trecerii în poziția respectivă. Se cere: a) Demonstrați dacă euristica "distanța diagonală" este o euristică admisibilă sau nu. b) Propuneți două strategii (una neinformată şi una informată) complete şi două strategii optimale de căutare c) Propuneți o euristică care să fie admisibilă d) Explorați câte 3 noduri folosind cele 4 strategii propuse e) Este căutarea iterativă în adâncime optimală pe această problemă şi în caz contrar cum poate fi făcută optimală?

	6	8	5	3	5	
3	2	6		3	3	4
4	A 1	1		3	1 B	5
7	5			4	9	1
	9	9	9	9	1	

Figura 10.1 *Spațiu cu obstacole*

Indicații pentru rezolvare

a) Distanța diagonală nu este o euristică admisibilă deoarece fiecare careu are asociat un cost aleator (distanța diagonală era o euristică definită pentru spațiul cu obstacole în cazul când diferența de cost apărea doar la deplasarea în linie dreaptă față de diagonală)
b) Strategii complete: breadth first, bidirectional search, strategii optimale: uniform cost, A*
c) Pentru a fi admisibilă o euristică trebuie să nu supraestimeze costul drumului între două puncte, astfel ținând cont că avem costuri supraunitare, distanța între două puncte este mai mare sau egală cu diferența maximă dintre coordonatele lor. Această euristică se defineşte astfel: $P(x_P, y_P)$, $B(x_B, y_B)$, $h(P,B) = \max\{\Delta_x, \Delta_y\}$, $\Delta_x = |x_P - x_B|$, $\Delta_y = |y_P - y_B|$.
e) Căutarea iterativă în adâncime nu este optimală deoarece costul operatorilor este diferit. Ea poate fi făcută optimală dacă în loc de limita de adancime se utilizează o limită de cost iterativă.

3. a) Explicați cum se calculează complexitatea de timp şi spațiu a strategiei Iterative Deepening b) Dați un exemplu de spațiu al stărilor pentru care strategia Depth-First este mai eficientă decât Iterative Deepening c) În ce ordine sunt parcurse nodurile dacă la strategia Greedy se foloseşte euristica *h(n)=-g(n)*, care ar fi caracteristica esențială la nivelul timpului de calcul (*g(n)* reprezintă costul uniform) d) În ce situații este A* optimală, demonstrați si exemplificați.

4. a) Dați un exemplu de problemă cu constrângeri (CSP) şi justificați alegerea a minim două euristici pe această problemă b) Ce intelegeți prin euristică admisibilă, ce se întâmplă dacă folosim o euristică ne-admisibilă la A* (explicați aspectele pozitive şi negative care există). Dați un exemplu de spațiu al stărilor pentru care strategia Depth-Limited este mai eficientă decât Iterative Deepening c) Construiți o euristică pentru strategia Greedy astfel încât să fie explorate întâi nodurile cele mai scumpe d) Este o euristică admisibilă monotonă (exemplificați)? e) Explicați de ce A* nu intră în cicluri infinite f) Dați un exemplu de spațiu al stărilor pentru care strategia Iterative Deepening este mai eficientă decât A* g) Este o euristică monotonă admisibilă (exemplificați)? h) Explicați de ce Iterative Deepening nu intră in cicluri infinite.

6. Se consideră un spațiu bidimensional reținut într-o matrice de dimensiune $m \times n$ şi un agent (entitate) aflat în punctul $A(x_A, y_A)$ ce doreşte să ajungă în puctul $B(x_B, y_B)$; $x_A, x_B \in \{1,2,...,m\}$, $y_A, y_B \in \{1,2,...,n\}$. Fiecare punct din spațiu $P(x_P, y_P)$ are asociat un cost supraunitar $C(P(x_P, y_P)) \in \{1,2,...,\mu\}$ care reprezintă costul de acces al locației

respective şi orice cost $c > \lambda$, $\lambda \le \mu$ se consideră obstacol şi nu poate fi trecut de către agent. Euristica folosită va fi $h(P,B) = \max\{\Delta_x, \Delta_y\}$, $\Delta_x = |x_P - x_B|$, $\Delta_y = |y_P - y_B|$ iar pentru costul uniform se utilizează costul de acces al fiecărei locații, deci $g(P(x_P, y_P)) = C(P(x_P, y_P)) + g(Parinte(P))$ unde $g(Parinte(P))$ este costul nodului părinte al lui $P(x_P, y_P)$ (deci la costul fiului se adaugă costul uniform al tatălui). Valorile $m, n, x_A, y_A, x_B, y_B, \lambda, \mu$ sunt variabile şi se introduc de la tastatură iar costurile se distribuie aleator pe harta. Se cere:

a) Să se implementeze pe această problemă strategiile de căutare: a) breadth-first b) depth-first c) depth limited d) iterative deepening e) uniform-cost f) bidirectional search (breadth first din ambele părți) g) bidirectional search (breadth first dintr-o parte şi depth-first din alta) h) bidirectional search (breadth first dintr-o parte şi iterative deepening din alta) i) bidirectional search (uniform cost din ambele părți) j) greedy k) A-star l).

b) Se consideră că agentul are de parcurs circuitul $\Omega = \{P_0, P_1, ..., P_k\}$ şi trebuie să depună obiecte în fiecare punct $P_i \subset \Omega$ respectând un set de priorități $\Pi : \Omega \times \Omega \to \{0,1\}$ unde $\Pi(P_i, P_j) = 1$ dacă depunerea de obiecte în P_i este condiționată de depunerea prealabilă în P_j respectiv $\Pi(P_i, P_j) = 0$ dacă nu există nici o condiționare între P_i şi P_j. Mulțimea Ω şi funcția Π (definită sub forma unei matrici) se citesc de la tastatură. Folosind algoritmul (algoritmii) de la a) dați un traseu pe care agentul poate să îl parcurgă astfel încât circuitul să fie parcurs în mod optimal. Observație: este evident că mulțimea Ω şi relația Π formează un graf orientat aciclic – în cazul în care graful ar contine cicluri problema nu ar mai avea soluție.

Programul va afişa si rezultate referitoare la timpul de calcul, memoria utilizată, numărul de stări atise, etc.

Bibliografie

[1] Konar, A., *Artificial Intelligence and Soft Computing Behavioral and Cognitive Modeling of the Human Brain*, CRC Press, 1999.

[2] Korf, R.E., *Artificial Intelligence Search Algorithms*, Computer Science Department University of California, Los Angeles, 1998. http://citeseer.ist.psu.edu/493289.html .

[3] Russell, S.J., Norvig, P., *Artificial Intelligence A Modern Approach*, Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[4] Exemple de Jocuri, *Learn4Good*, (recomandate in scop didactic) - http://www.learn4good.com/games/.

[5] Exemplu A-star http://www.policyalmanac.org/games/aStarTutorial.htm.

Anexe

Anexa 1. Elemente de teoria complexității algoritmilor

O problemă este o mulțime nevidă de întrebări între care există o relație, pot fi una sau mai multe întrebări şi este obligatoriu ca ele să aibă o dimensiune finită. De exemplu factorizarea unui întreg este o problemă cu următorul enunț: având un întreg *n* să se găsească numerele prime al căror produs este. Un algoritm este un set bine definit de paşi pentru rezolvarea unei probleme. Altfel spus, un algoritm este ansamblul de paşi prin care un set de date de intrare este transformat într-un set de date de ieşire în scopul rezolvării unei probleme. De exemplu pentru rezolvarea problemei factorizării întregului de mai sus se poate folosi următorul algoritm: pentru toți întregii *i* de la 2 la radical din *n* verifică dacă *n* este divizibil cu *i*.

Este evident că eficiența unui algoritm este o funcție care depinde de dimensiunea datelor de intrare şi totodată eficiența unui algoritm trebuie să fie o caracteristică intrinsecă a algoritmului care să nu depindă de un anume model al maşinii de calcul. În acest context este impropriu să numim un algoritm ca fiind eficient în baza faptului că pe un anumit procesor a avut un timp de rulare oarecare, şi mai mult, faptul că pe un procesor a avut un anume timp de rulare nu va spune nimic cu privire la timpul de rulare în momentul în care dimensiunea datelor de intrare se dublează. Să presupunem ca exemplu naiv doi algoritmi care caută un element într-un şir ordonat crescător, algoritmul A1 implementează o căutare naivă în care şirul este parcurs de la un capăt la altul element cu element în scopul identificării elementului căutat iar A2 implementează o căutare binară, prin înjumătățirea succesivă a şirului în care se face căutarea. Să presupunem că timpul de calcul pentru un tablou cu 1.000.000 de elemente este pentru primul algoritm 5 milisecunde şi pentru al doilea 5 microsecunde. La dublarea dimensiunii datelor de intrare timpul de calcul pentru primul algoritm se va dubla în timp ce pentru al doilea va creşte nesemnificativ. Aceasta deoarece, pentru găsirea unui element într-un şir de *n* elemente, primul algoritm efectuează cel mult *n* paşi iar cel de-al doilea cel mult $log_2 n$ paşi. În consecință performanța unui algoritm nu trebuie descrisă în funcție de timpul de rulare al acestuia ci în funcție de numărul de paşi pe care algoritmul îi necesită. Aici intră în joc teoria complexității care este domeniul care ne oferă un răspuns cu privire la numărul de paşi necesari ca un algoritm să ofere în rezultat.

În general cu privire la un algoritm, sub raportul complexității, ne interesează atât timpul (despre care s-a spus că se măsoară în număr de paşi) cât şi spațiul (care înseamnă cantitate de memorie necesară) – de aici şi noțiunile de complexitate de timp şi complexitate de spațiu. Pentru măsurarea complexității folosim următorii indicatori de performanță:

i) **limita asimptotică superioară**:

$$f(n)=O(g(n)) \Leftrightarrow \exists c, n_o \ a.i. 0 \le f(n) \le c \cdot g(n) \forall n \ge n_0$$

ii) **limita asimptotică inferioară**:

$$f(n)=\Omega(g(n)) \Leftrightarrow \exists c, n_o \ a.i. 0 \le c \cdot g(n) \le f(n), \forall n \ge n_0$$

iii) **limita asimptotică restrânsă**:

$$f(n)=\Theta(g(n)) \Leftrightarrow \exists c_1, c_2, n_o \ a.i. c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \forall n \ge n_0$$

iv) **limitele asimptotice relaxate**:

$$f(n)=o(g(n)) \Leftrightarrow \exists n_o \ a.i. \forall c \ 0 \le f(n) < c \cdot g(n) \forall n \ge n_0$$

$$f(n)=\omega(g(n)) \Leftrightarrow \exists n_o \ a.i. \forall c \ 0 \le c \cdot g(n) < f(n) \forall n \ge n_0$$

Este important de spus că $f = F\left(g\left(n\right)\right)$, unde F este oricare din indicatorii Ω, Θ, O, o, nu se citeşte: “f egal F de g(n)”, ci corect este “f este de ordinul F al lui g(n)” sau mai simplu “f este F(g(n))”. Intuitiv semnul “=“ nu are semnificația semnului egal, în acest caz este echivalent cu $\in$.

Astfel în funcție de timpul de calcul algoritmii se împart în clase după cum urmează: constant $O(1)$, logaritmic $O(\lg(n))$ (se observă că $\log_c n = \Theta(\lg n), \forall c > 0$), poli-logaritmic $O\left(\lg^c\left(n\right)\right)$, fracțional $O\left(n^c\right), 0 < c < 1$, liniar $O(n)$, liniar logaritmic $O\left(n \log_2 n\right)$, pătratic (sau cvadratic) $O(n^2)$, cubic $O(n^3)$, polinomial $O(n^m)$ (cu observația că: liniar, pătratic, cubic sunt timpi polinomiali), super-polinomial $O(c^{f(n)})$ (unde c este o constantă iar $f(n)$ nu este constant dar este mai mic decât $O(n)$), exponențial $O(c^{f(n)})$ (unde c este o constantă iar $f(n)$ este un polinom de gradul 1), factorial (sau Combinatorial) $O\left(n!\right)$, dublu exponențial $O\left(2^{c^n}\right)$.

Următoarele proprietăți intuitive decurg direct din definiția acestor indicatori:

i) $f(n)=O(g(n)) \Leftrightarrow g(n)=\Omega(f(n))$

ii) $f(n)=\Theta(g(n)) \Leftrightarrow f(n)=O(g(n)) \wedge f(n)=\Omega(g(n))$

iii) $f(n)=O(h(n)) \wedge g(n)=O(h(n)) \Rightarrow (f+g)(n)=O(h(n))$

iv) $f(n)=O(h(n)) \wedge g(n)=O(i(n)) \Rightarrow (f \cdot g)(n)=O(h(n) \cdot i(n))$

v) $f(n)=O(f(n))-\textit{reflexivitate}$

vi) $f(n)=O(g(n))\wedge g(n)=O(h(n))\Rightarrow f(n)=O(h(n))-\textit{tranzitivitate}$

Reținem de asemenea următoarele aproximări utile:

i) $f(n)=a_k\cdot n^k+a_{k-1}\cdot n^{k-1}+...+a_1\cdot n+a_0\Rightarrow f(n)=\Theta(n^k)$

ii) $n!=o(n^n)\wedge n!=\Omega(2^n)$

iii) $1<\ln\ln n<\ln n<e^{\sqrt{(\ln n)(\ln\ln n)}}<n^{\varepsilon}<n^{c}<n^{\ln n}<c^{n}<n^{n}<c^{c^{n}}$

Pentru a ilustra importanța cunoaşterii complexității prezentăm următorul tabel al unor magnitudini uzuale şi de asemenea vom considera 4 algoritmi A_1, A_2, A_3, A_4 având complexitățile $O(n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$ precum şi un sistem capabil să execute 10^{10} operații/secundă, în scopul unei comparații vom evalua timpul necesar rezolvării algoritmilor pentru $n=10^6$. Tabelul 1 sintetizează aceste rezultate.

Secunde într-un an	3×10^{7}
Vârsta sistemului solar în secunde	2×10^{17}
Electroni în univers	8.37×10^{77}
Timpul pentru a rezolva A_1	1×10^{-3}
Timpul pentru a rezolva A_2	1×10^{2}
Timpul pentru a rezolva A_3	1×10^{8}
Timpul pentru a rezolva A_4	1×10^{303020}

Tabel 1. Câteva magnitudini uzuale.