Proiectarea Algoritmilor 2009-2010

Laborator 11

Algoritmi euristici de explorare a grafurilor. A*

Cuprins

1. Obiective laborator....1
2. Importanță – aplicații practice....1
3. Descrierea problemei şi a rezolvărilor....1
- 3.1. Prezentarea generala a problemei....2
- 3.2. Greedy Best-First....4
- 3.3. A*....6
- 3.4. Complexitatea algoritmului A*....10
- 3.5. IDA*, RBFS, MA*....10
4. Concluzii....11
5. Referințe....12
6.1-3 Aplicatie 1 – 8 - Puzzle....13

1. Obiective laborator

In cadrul acestui laborator se va discuta despre modelarea problemelor sub forma grafurilor de stari si despre algoritmi specializati in gasirea solutiilor pentru acest tip de grafuri. De asemenea, se vor discuta modalitatile care pot fi folosite in analiza complexitatii si pe baza acestor metode se vor prezenta avantajele si limitarile acestei clase de algoritmi.

2. Importanță - aplicații practice

Algoritmii de cautare euristica sunt folositi in cazurile care implica gasirea unor solutii pentru probleme pentru care fie nu exista un model matematic de rezolvare directa, fie acest model este prea complicat pentru a fi implementat. In acest caz e necesara o explorare a spatiului starilor problemei pentru gasirea unui raspuns. Intrucat o mare parte dintre problemele din viata reala pornesc de la aceste premise gama de aplicatii a algoritmilor euristici este destul de larga. Proiectarea agentilor inteligenti [1], probleme de planificare, proiectare circuitelor VLSI [3], robotica, cautare web, algoritmi de aproximare pentru probleme NP-Complete [10], teoria jocurilor sunt doar cateva dintre domeniile in care cautarea informata este utilizata.

3. Descrierea problemei şi a rezolvărilor

3.1 Prezentare generală a problemei

Primul pas in rezolvarea unei probleme folosind algoritmi euristici de explorare este definirea exacta a problemei. O probleme este bine definita daca cunoastem urmatorii 4 parametri:

- **Starea initiala a problemei** – reprezinta configuratia de plecare.
- **Functie de expandare a nodurilor** – in cazul general este o lista de perechi (actiune, stare_rezultat). Astfel, pentru fiecare stare se enumera toate actiunile posibile precum si starea care va rezulta in urma aplicarii respectivei actiuni.
- **Predicat pentru starea finala** – functie care intoarce *adevarat* daca o stare este stare scop si *fals* altfel
- **Functie de cost** – atribuie o valoare numerica fiecarei cai generate in procesul de explorare. De obicei se foloseste o functie de cost pentru fiecare actiune/tranzitie, atribuind, astfel, o valoare fiecarui arc din graful starilor.

Sarcina algoritmilor de cautare este de a gasi o cale din starea initiala intr-o stare scop. Daca algoritmul gaseste o solutie atunci cand multimea solutiilor este nevida spunem ca algoritmul este **complet**. Daca algoritmul gaseste si calea de cost minim catre starea finala spunem ca algoritmul este **optim**.

In principiu, orice algoritm pe grafuri discutat in laboratoarele si cursurile anterioare poate fi utilizat pentru gasirea solutiei unei probleme astfel definite. In practica, insa, acesti algoritmi nu sunt niciodata utilizati fie pentru ca exploreaza mult prea multe noduri, fie pentru ca nu garanteaza o solutie pentru grafuri definite implicit (prin stare initiala si functie de expandare).

Algoritmii euristici de cautare sunt algoritmi care lucreaza pe grafuri definite ca mai sus si care folosesc o informatie suplimentara, necontinuta in definirea problemei, prin care se accelereaza procesul de gasirea a unei solutii.

In cadrul explorarii starilor fiecare algoritm genereaza un arbore, care in radacina va contine starea initiala. Fiecare nod al arborelui va contine urmatorele informatii:

- **Starea continuta** - stare(nod)
- **Parintele nodului** – π(nod)
- **Cost cale –** costul drumului de la starea initiala pana la nod – g(nod)

De asemenea, pentru fiecare nod definim si o functie de evaluare **f** care indica cat de promitator este un nod in perspectiva gasirii unui drum catre solutie. (De obicei, cu cat f este mai mic, cu atat nodul este mai promitator).

Am mentionat mai sus ca algoritmii de cautare euristica utilizeaza o informatie suplimentara referitoare la gasirea solutiei problemei. Aceasta informatie este reprezentata de o functie **h**, unde h(nod) reprezinta drumul *estimat* de la nod la cea mai apropiata stare solutie. Functia h poate fi definita in orice mod, existand o singura constrangere:

$$h(n)=0 \quad \forall n, solutie(n)=adevarat$$

Intrucat functioneaza prin alegerea, la fiecare pas, a nodului pentru care f(nod) este minim, algoritmii prezentati mai jos fac parte din clasa **Best-First** (nodul cel mai promitator este explorat mereu primul).

Vom prezenta in continuare cativa dintre cei mai importanti algoritmi de cautare euristica. Vom folosi pentru exemplificare, urmatoarea problema: data fiind o harta rutiera a Romaniei, sa se gaseasca o cale (de preferinta de cost minim) intre Arad si Bucuresti. Pentru aceasta problema starea initiala inidica ca ne aflam in orasul Arad, starea finala este data de predicatul Oras_curent == Bucuresti, functia de expandare intoarce toate orasele in care putem ajunge dintr-un oras dat, iar functia de cost indica numarul de km al fiecarui drum intre doua orase, presupunand ca viteza de deplasare este constanta. Ca euristica vom utiliza pentru fiecare oras distanta geometrica(in linie dreapta) pana la Bucuresti.

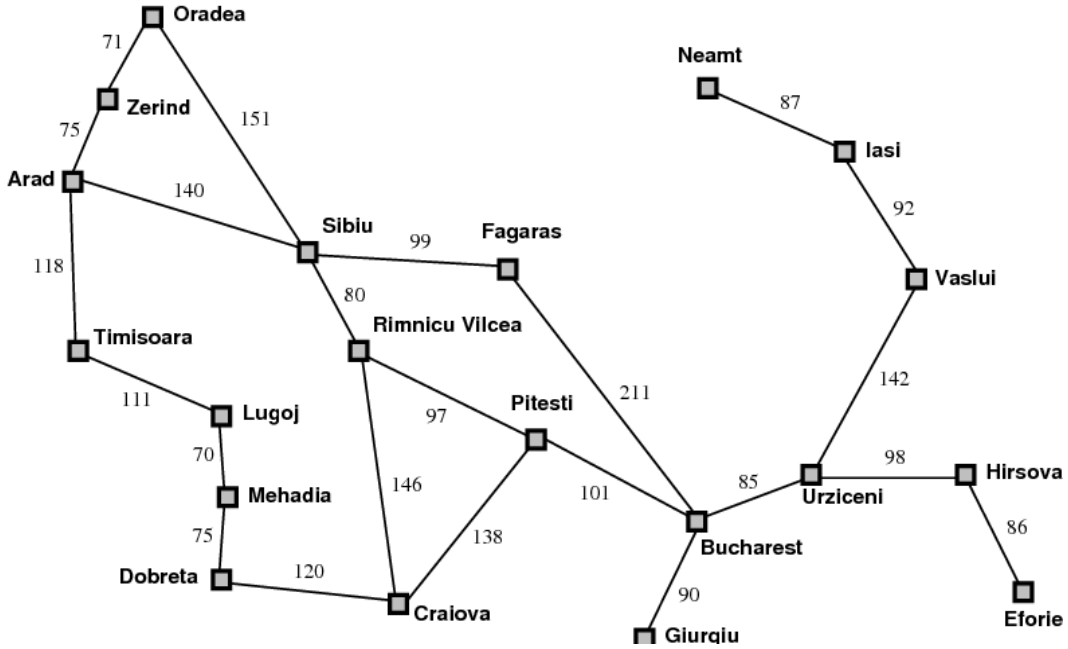


Fig 1: Graful starilor

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

Fig 2 : Functia euristica

3.2 Greedy Best-First

In cazul acestui algoritm se considera ca nodul care merita sa fie expandat in pasul urmator este cel mai apropiat de solutie. Deci, in acest caz, avem:

$$f(n)=h(n),\ \ \forall n$$

Pseudocodul acestui algoritm:

```
Greedy Best-First(sinitial , expand, h, solution)
      closed ← {}
      n ← new-node()
      state(n) ← sinitial
      π(n) ← nil
      open ← { n }
```

```
// Bucla principala
repeat
      if open = Ø then return failure
      n ← get_best(open) with f(n) = h(n) = min
      open ← open - {n}
      if solution(state(n)) then return build-path(n)
      else if n not in closed then
            closed ← closed U {n}
            for each s in expand(n)
                  n' ← new-node()
                  state(n') ← s
                  π(n') ← n
                  open ← open U {n'}
            end-for
end-repeat
```

In cadrul algoritmului se folosesc doua multimi: closed – indica nodurile deja explorate si expandate si open – nodurile descoperite dar neexpandate. Open este intializata cu nodul corespunzator starii intiale. La fiecare pas al algoritmului este ales din open nodul cu valoarea f(n) = h(n) cea mai mica (din acest motiv e de preferat ca open sa fie implementata ca o coada cu prioritati). Daca nodul se dovedeste a fi o solutie a problemei atunci este intoarsa ca rezultat calea de la starea initiala pana la nod (mergand recursiv din parinte in parinte). Daca nodul nu a fost deja explorat atunci se expandeaza iar nodurile corespunzatoare starilor rezultate sunt introduse in multimea open. Daca multimea open ramane fara elemente atunci nu exista niciun drum catre solutie si algoritmul intoarce esec.

Greedy Best-First urmareste mereu solutia care pare cea mai aproape de sursa. Din acest motiv nu se vor analiza stari care desi par mai departate de solutie produc o cale catre solutie mai scurta (vezi exemplul de rulare). De asemenea, intrucat nodurile din closed nu sunt niciodata reexplorate se va gasi calea cea mai scurta catre scop doar daca se intampla ca aceasta cale sa fie analizata inaintea altor cai catre aceiasi stare scop. Din acest motiv, algoritmul nu este optim. De asemenea, pentru grafuri infinite e posibil ca algoritmul sa ruleze la infinit chiar daca exista o solutie. Rezulta ca algoritmul nu indeplineste nici conditia de completitudine.

In figura de mai jos se prezinta rularea algoritmului Greedy Best-First pe exemplul dat mai sus. In primul pas algoritmul expandeaza nodul Arad, iar ca nod urmator de explorat se alege Sibiu, intrucat are valoarea h(n) minima. Se alege in contiuare Fagaras dupa care urmeaza Bucuresti, care este un nod final. Se observa insa ca acest drum nu este minimal. Desi Fagaras este mai aproape ca distanta geometrica de Bucuresti, in momentul in care starea curenta este Sibiu alegerea optimala este Ramnicu-Valcea. In continuare ar fi urmat Pitesti si apoi Bucuresti obtinandu-se un drum cu 32 km mai scurt.

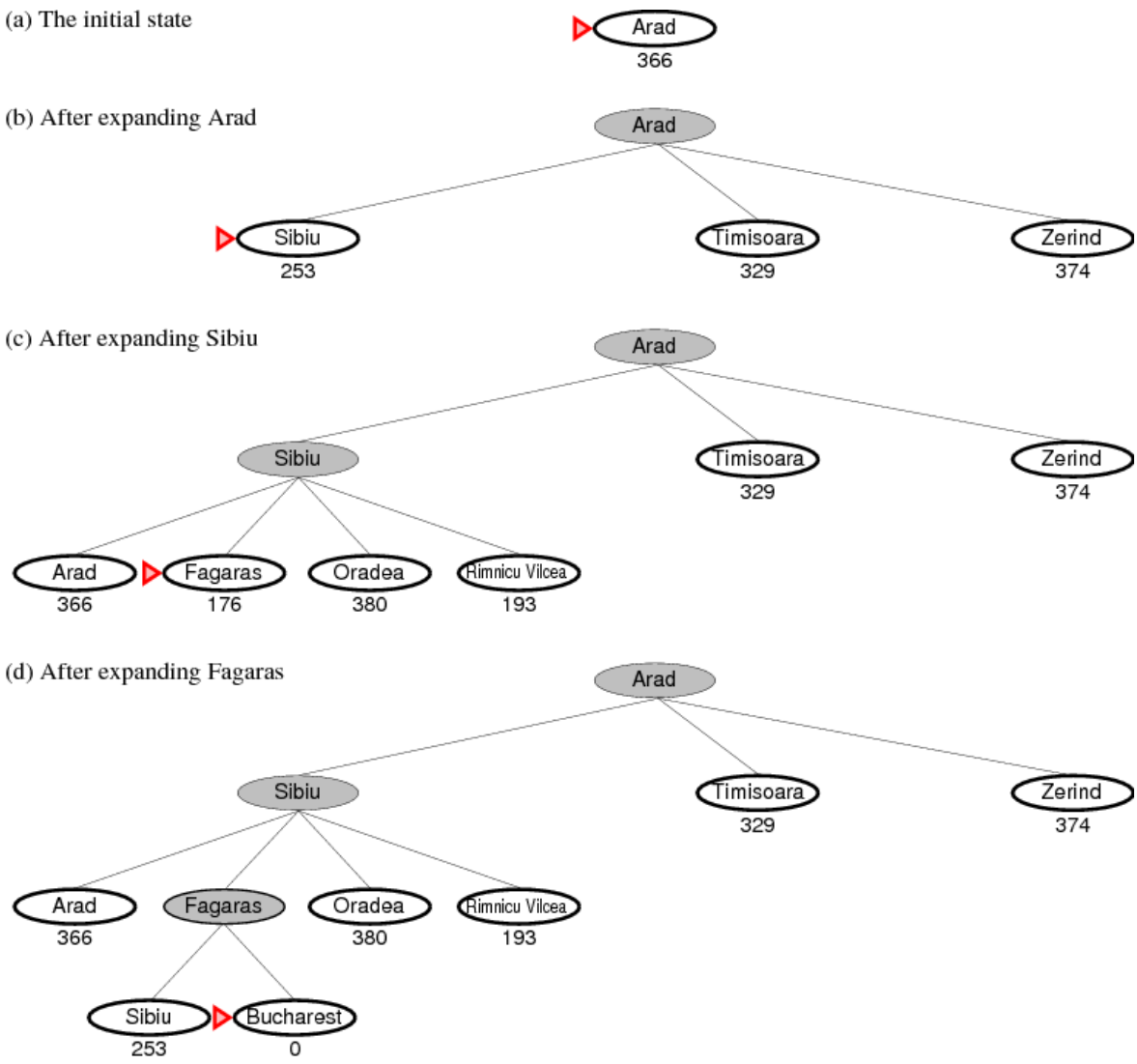


Fig 3 : Rulare Greedy Best-First

3.3 A* (A Star)

A* reprezinta cel mai cunoscut algoritm de cautare euristica. El foloseste, de asemenea o politica Best-First, insa nu sufera de aceleasi defecte pe care le are Greedy Best-First definit mai sus. Acest lucru este realizat prin definirea functiei de evaluare astfel:

$$f(n)=g(n)+h(n), \forall\, n \in Noduri$$

A* evalueaza nodurile combinand distanta deja parcursa pana la nod cu distanta estimata pana la cea mai apropiata stare scop. Cu alte cuvinte, pentru un nod n oarecare, **f(*n*) reprezinta costul estimat al celei mai bune solutii care trece prin *n***. Aceasta strategie se dovedeste a fi completa si optimala daca euristica h(n) este admisibila:

$$0 \le h(n) \le h^*(n), \forall\, n \in Nodes$$

unde $h^*(n)$ este distanta exacta de la nodul n la cea mai apropiata solutie. Cu alte cuvinte A* intoarce mereu solutia optima daca o solutie exista atat timp cat ramanem optimisti si nu supraestimam distanta pana la solutie. Daca h(n) nu este admisibila o solutie va fi in continuare gasita, dar nu se garanteaza optimalitatea. De asemenea, pentru a ne asigura ca vom gasi drumul optim catre o solutie chiar daca acest drum nu este analizat primul, A* permite scoaterea nodurilor din closed si reintroducerea lor in open daca o cale mai buna pentru un nod din closed (g(n) mai mic) a fost gasita.

Algoritmul evolueaza in felul urmator: initial se introduce in multimea open (organizata ca o coada de prioritati dupa f(n)) nodul corespunzator starii initiale. La fiecare pas se extrage din open nodul cu f(n) minim. Daca se dovedeste ca nodul n contine chiar o stare scop atunci se intoarce calea de la starea initiala pana la nodul n. Altfel, daca nodul nu a fost explorat deja se expandeaza. Pentru fiecare stare rezultata, daca nu a fost generata de alt nod inca (nu este nici in open nici in closed) atunci se introduce in open. Daca exista un nod corespunzator starii generate in open sau closed se verifica daca nu cumva nodul curent produce o cale mai scurta catre s. Daca acest lucru se intampla se seteaza nodul curent ca parinte al nodului starii s si se corecteaza distanta g. Aceasta corectare implica reevaluarea tuturor cailor care trec prin nodul lui s, deci acest nod va trebui reintrodus in open in cazul in care era inclus in closed.

Pseudocodul pentru A* este prezentat in continuare:

A-Star(s $_{initial}$, expand, h, solution)

```
//initializari
closed ← {}
n ← new-node()
state(n) ← s initial
g(n) ← 0
π(n) ← nil
open ← { n }


//Bucla principala
repeat
      if open = Ø then return failure
      n ← get_best(open) with f(n) = g(n)+h(n) = min
      open ← open - {n}
```

```
        if solution(state(n)) then return build-path(n)
        else if n not in closed then
              closed ← closed U {n}
              for each s in expand(n)
                    cost_from_n ← g(n) + cost(state(n), s)
                    if not (s in closed U open) then
                          n' ← new-node()
                          state(n') ← s
                          π(n') ← n
                          g(n') ← cost_from_n
                          open ← open U { n'}
                    else
                          n' ← get(closed U open, s)
                          if cost_from_n < g(n') then
                                π(n') ← n
                                g(n') ← cost_from_n
                                if n' in closed then
                                      closed ← closed – { n'}
                                      open ← open U { n'}
              end-for
  end-repeat
```

Algoritmul prezentat mai sus va intoarce calea optima catre solutie, daca o solutie exista. Singurul incovenient fata de Greedy Best-First este ca sunt necesare reevaluarile nodurilor din closed. Si aceasta problema poate fi rezolvata daca impunem o conditie mai tare asupra euristicii h, si anume ca euristica sa fie **consistenta** (sau **monotona**):

$$h(n) \leq h(n') + cost(n, n'), \forall n' \in expand(n)$$

Daca o functie este consistenta atunci ea este si admisibila. Daca euristica h indeplineste si aceasta conditie atunci algoritmul A* este asemanator cu Greedy Best-First cu modificarea ca functia de evaluare este f = g + h, in loc de f = h.

In imaginea de mai jos se prezinta rularea algoritmului pe exemplul laboratorului. Se observa ca euristica aleasa (distanta in linie dreapta) este consistenta si deci admisibila. Se observa ca in pasul (e), dupa expandarea nodului Fagaras desi exista o solutie in multimea open aceasta nu este aleasa pentru explorare. Se va alege Pitesti, intrucat f(nod(Bucuresti)) = 450 > f(nod(Pitesti)) = 417, semnificatia acestei inegalitati fiind ca e posibil sa existe prin Pitesti un drum mai bun catre Bucuresti decat cel descoperit pana acum.

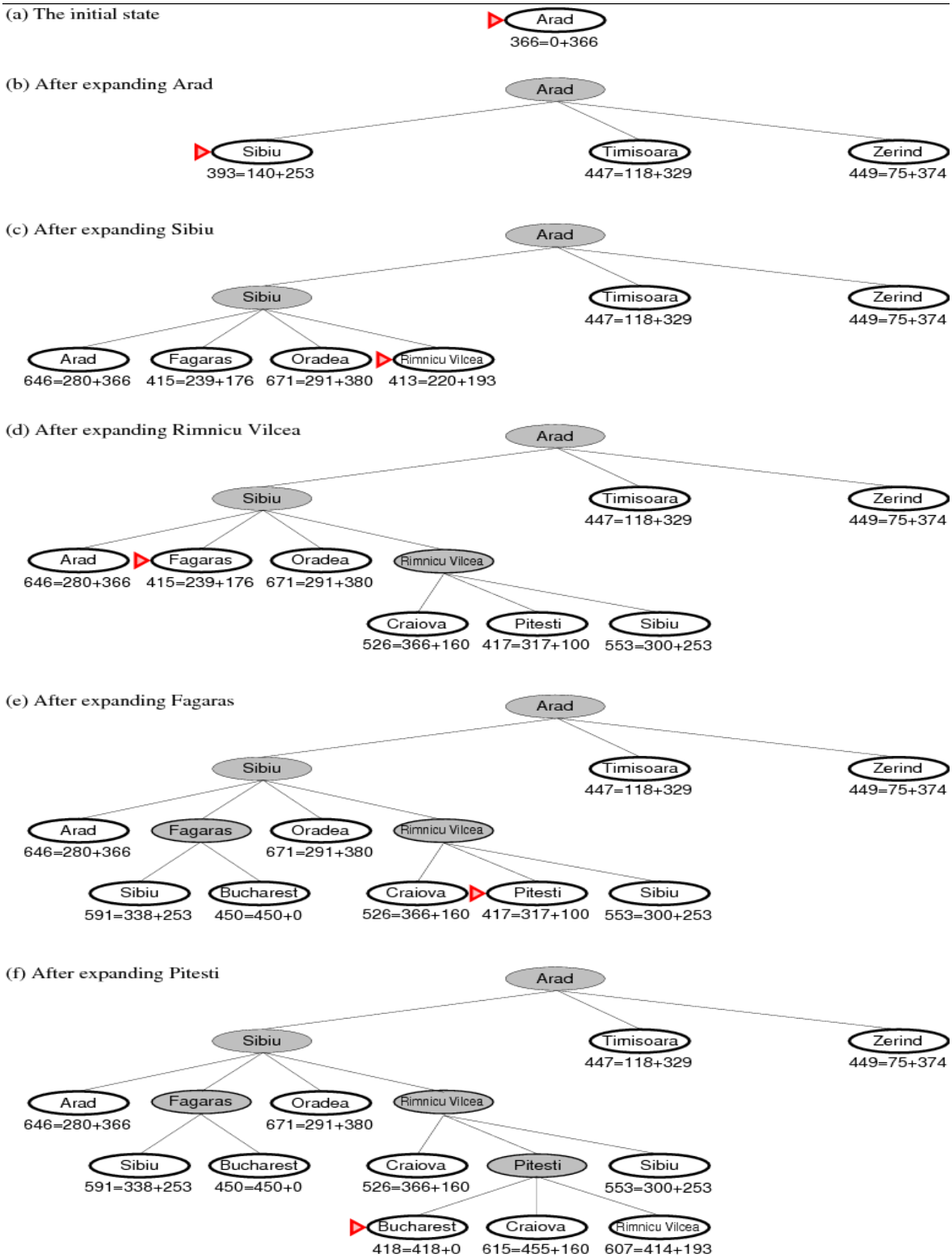


*Fig 4 : Rulare A**

3.4 [EXTRA] Complexitatea algoritmului A*

Pe langa proprietatile de completitudine si optimalitate A* mai are o calitate care il face atragator: pentru o euristica data orice algoritm de cautare complet si optim va explora cel putin la

fel de multe noduri ca A*, ceea ce inseamna ca A* este **optimal din punctul de vedere al eficientei**. In practica, insa, A* poate fi de multe ori imposibil de rulat datorita dimensiunii prea mari a spatiului de cautare. Singura modalitate prin care spatiul de cautare poate fi redus este prin gasirea unei euristici foarte bune – cu cat euristica este mai apropiata de distanta reala fata de stare solutie cu atat spatiul de cautare este mai strans (vezi figura de mai jos). S-a demonstrat ca spatiul de cautare incepe sa creasca exponential daca eroarea euristicii fata de distanta reala pana la solutie nu are o crestere subexponentiala:

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

Din pacate, in majoritatea cazurilor, eroarea creste liniar cu distanta pana la solutie ceea ce face ca A* sa devina un algoritm mare consumator de timp, dar mai ales de memorie. Intrucat in procesul de cautare se retin toate nodurile deja explorate (multimea closed), in cazul unei dimensiuni mari a spatiului de cautare cantitatea de memorie alocata cautarii este in cele din urma epuizata. Pentru acest incovenient au fost gasite mai multe solutii. Una dintre acestea este utilizarea unor euristici care sunt mai stranse de distanta reala pana la starea scop, desi nu sunt admisibile. Se obtin solutii mai rapid, dar nu se mai garanteaza optimalitatea acestora. Folosim aceasta metoda cand ne intereseaza mai mult sa gasim o solutie repede, indiferent de optimalitatea ei. Alte abordari presupun sacrificarea timpului de executie pentru a margini spatiul de memorie utilizat [3]. Aceste alternative sunt prezentate in continuare:

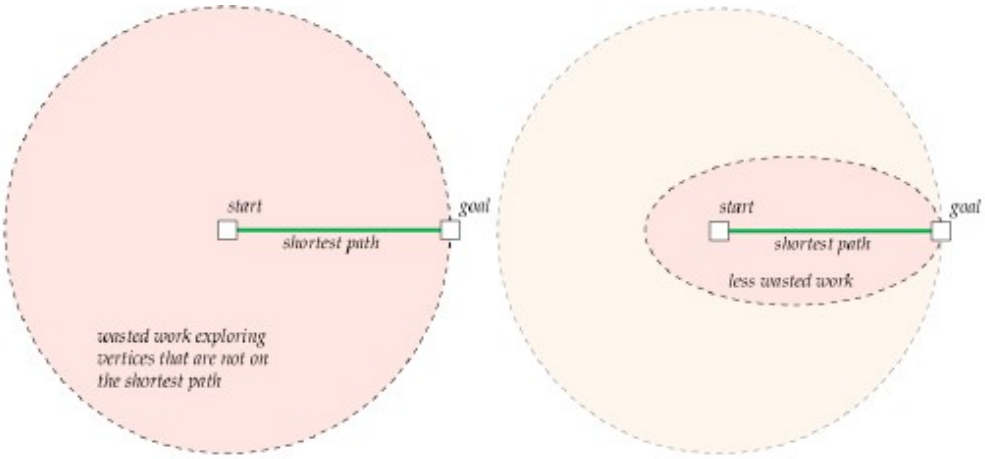


Fig 5 : Volumul spatiului de cautare in functie de euristica aleasa

3.5 [EXTRA] IDA*, RBFS, MA*

IDA* (Iterative deepening A*) [5] utilizeaza conceptul de adancire iterativa[1][8] in procesul de explorare a nodurilor – la un anumit pas se vor explora doar noduri pentru care functia de evaluare are o valoare mai mica decat o limita data, limita care este incrementata treptat pana se gaseste o solutie. IDA* nu mai necesita utilizarea unor multimi pentru retinerea nodurilor explorate si se comporta bine in cazul in care toate actiunile au acelasi cost. Din pacate, devine ineficient in momentul in care costurile sunt variabile.

RBFS (Recursive Best-First Search) [6] functioneaza intr-un mod asemanator cu DFS, explorand la fiecare pas nodul cel mai promitator fara a retine informatii despre nodurile deja explorate. Spre deosebire de DFS, se retine in orice moment cea mai buna alternativa sub forma

unui pointer catre un nod neexplorat. Daca valoarea functiei de evaluare (f = g + h) pentru nodul curent devine mai mare decat valoarea caii alternative, drumul curent este abandonat si se incepe explorarea nodului retinut ca alternativa. RBFS are avantajul ca spatiul de memorie creste doar liniar in raport cu lungimea caii analizate. Marele dezavantaj este ca se ajunge de cele mai multe ori la reexpandari si reexplorari repetate ale acelorasi noduri, lucru care poate fi dezavantajos mai ales daca exista multe cai prin care se ajunge la aceiasi stare sau functia *expand* este costisitoare computational (vezi problema deplasarii unui robot intr-un mediu real). RBFS este optimal daca euristica folosita este admisibila.

MA* (Memory-bounded A*) este o varianta a A* in care se limiteaza cantitatea de memorie folosita pentru retinerea nodurilor. Exista doua versiuni – MA* [7]si SMA* [4] (Simple Memory-Bounded A*), ambele bazandu-se pe acelasi principiu. SMA* ruleaza similar cu A* pana in momentul in care cantitatea de memorie devine insuficienta. In acest moment spatiul de memorie necesar adaugarii unui nod nou este obtinut prin stergerea celui mai putin promitator nod deja explorat. In cazul in care exista o egalitate in privinta valorii functiei de evaluare se sterge nodul cel mai vechi. Pentru a evita posibilitatea in care nodul sters este totusi un nod care conduce la o cale optimala catre solutie valoarea f a nodului sters este retinuta la nodul parinte intr-un mod asemanator felului in care in RBFS se retine cea mai buna alternativa la nodul curent. Si in acest caz vor exista reexplorari de noduri, dar acest lucru se va intampla doar cand toate caile mai promitatoare vor esua. Desi exista probleme in care aceste regenerari de noduri au o frecventa care face ca algoritmul sa devina intractabil, MA* si SMA* asigura un compromis bun intre timpul de executie si limitarile de memorie.

Concluzii

Deseori singura modalitate de rezolvare a problemelor dificile este de e explora graful starilor acelei probleme. Algoritmii clasici pe grafuri nu sunt potriviti pentru cautarea in aceste grafuri fie pentru ca nu garanteaza un rezultat fie pentru ca sunt ineficienti.

Algoritmii euristici sunt algoritmi care exploreaza astfel de grafuri folosindu-se de o informatie suplimentara despre modul in care se poate ajunge la o stare scop mai rapid. A* este, teoretic, cel mai eficient algoritm de explorare euristica. In practica, insa, pentru probleme foarte dificile, A* implica un consum prea mare de memorie. In acest caz se folosesc variante de algoritmi care incearca sa minimizeze consumul de memorie in defavoarea timpului de executie.

Responsabil laborator: Marius-Andrei Danila
(marius.danila@cti.pub.ro)

Referințe

[1] S. Russel, P. Norvig - Artificial Intelligence: A Modern Approach - Prentice Hall, 2nd Edition – cap. 4

[2] C.A Giumale – Introducere in Analiza Algoritmilor, cap. 7

[3] Mehul Shah - Algorithms in the real world (Course Notes) - Introduction to VLSI routing

[4] S. Russel - Efficient memory-bounded search methods

[5] R. Korf - Depth-first iterative-deepening: an optimal admissible tree search

[6] Recursive Best-First Search - Prezentare

[7] P. Chakrabati, S. Ghosh, A. Acharya, S. DeSarkar – Heuristic search in restricted memory

[8] Wikipedia - Iterative deepening

[9] A. E. Prieditis - Machine Discovery of Effective Admissible Heuristics

[10] Wikipedia - Travelling salesman problem - Heuristic and approximation algorithms

[11] Wikipedia – A*

[12] Tutorial A*

[13] H. Kaindl, A. Khorsand - Memory Bounded Bidirectional Search, AAAI-94 Proceedings

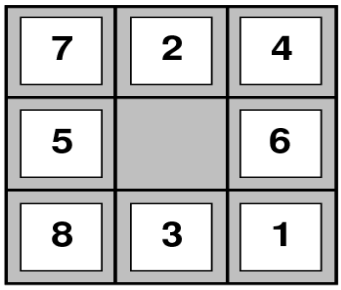
Proiectarea Algoritmilor 2009-2010

Laborator 11

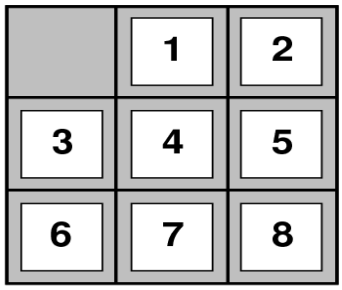
Algoritmi euristici de explorare. A*

6.1 Problema 1 (7 + 1 pct.)

Se considera urmatorul joc (8-puzzle): 8 piese sunt asezate pe un grid cu 9 pozitii. O piesa poate fi mutata in orice moment in stanga, dreapta, sus sau jos atat timp cat spatiul in care este mutata este spatiul gol. Scopul jocului este ca plecand de la o configuratie initiala sa se ajunga la o configuratie predefinita:



Start State



Goal State

- **(7 pct)** Implementati algoritmul A* pentru problema 8-puzzle. Tineti seama, la testare, ca in cazul 8-puzzle nu se poate ajunge dintr-o configuratie data in orice configuratie posibila. (http://en.wikipedia.org/wiki/Fifteen_puzzle#Solvability). Implementarea se va face in fisierele AStar.cpp / Astar.java. Functa de expandare, predicatul de stare finala si euristica sunt deja implementate in modulul de testare PuzzleSolver.cpp/ PuzzleSolver.java
- **(1 pct bonus)** Implementati o varianta simplificata de A* in care sa plecati de la presupunerea ca euristica folosita este consistenta.

6.2 Problema 2 (1 + 1 pct.)

O modalitate importanta de construire a euristicilor este asa numita tehnica a **problemei relaxate** [1]. Spre exemplu sa consideram problema 8 – puzzle. Modul in care putem muta piesele este guvernat de 3 reguli:

- O piesa se poate muta din pozitia A in pozitia B
- O piesa se poate muta din pozitia A in pozitia B daca pozitiile A si B sunt adiacente
- O piesa se poate muta din pozitia A in pozitia B daca B este goala

O versiune relaxata a unei probleme este o varianta a acestei probleme in care dispar un numar de reguli. Spre exemplu, ne putem imagina o varianta relaxata a 8-puzzle in care e valabila doar prima regula. Se observa ca solutia pentru problema originala este in acelasi timp solutie si pentru problema relaxata. Dar pentru ca problema relaxata are mai putine reguli, solutia optima va avea un numar mai mic de pasi, deci un drum mai scurt de la starea initiala la cea tinta. Cu alte cuvinte, **costul unei solutii optimale pentru problema relaxata este o euristica admisibila pentru problema originala**. Mai mult, euristica astfel construita este si consistenta.

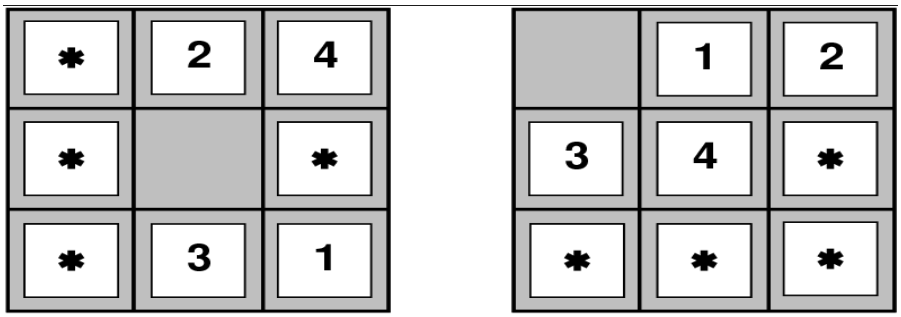
In PuzzleSolver.cpp/PuzzleSolver.java aveti definita euristica obtinuta din relaxarea care contine doar prima regula (numita Missplaced Tiles Heuristic). Euristica intoarce pentru o configuratie de puzzle numarul de piese care nu sunt pe pozitiile in care trebuie sa fie. Motivatia este ca daca e valabila doar prima regula, pentru a rezolva problema mutam piesele care nu sunt pe pozitiile lor la locul potrivit si am termiant. Deci, numarul de mutari este egal cu numarul de piese nearanjate.

Construiti eficient euristica obtinuta din relaxarea care contine doar a doua regula (Manhattan Distance Heuristic) in fisierul PuzzleSolver (ganditi-va cum si care e numarul minim de pasi in care se rezolva aceasta problema relaxata). Observati cate noduri sunt expandate fata de prima euristica.

(Bonus 1p.) - Implementati euristica care foloseste doar a III-a regula (Gaschnig's Heuristic)

6.4 Problema 3 (2 pct.)

(2 pct) Multe euristici puternice pot fi create folosindu-ne de rezolvari complete ale unei probleme partiale, mai mici, derivata din problema originala. Sa consideram 8-puzzle. O problema partiala este, de exemplu, cea in care ne intereseaza sa asezam doar piesele 1, 2, 3, 4 la locurile lor:



Start State **Goal State**

Intrucat aceasta este o problema mai simpla, ea poate fi rezolvata direct si putem sti numarul exact de pasi pana la o stare finala din orice stare data pentru problema partiala. De asemenea, se observa usor ca aceasta euristica exacta pentru problema partiala este o euristica admisibla si consistenta pentru problema originala si poate fi folosita ca atare in algoritmul A*.

Putem alege si alte piese in afara de 1,2,3,4. Spre exemplu un pattern cu 5,6,7,8. Mai mult, pentru o stare s pentru care dorim sa calculam euristica putem lua maximul dintre euristica corespunzatoare problemei partiale 1,2,3,4 si euristica problemei partiale 5,6,7,8, profitand de urmatoarea proprietate:

$$Daca\ h_1, h_2, h_3, \ldots, h_k - admisibile \Rightarrow h(n) = \max_{i=1}^{k} h_i(n) \quad este\ admisibila.$$

Plecand de la acest principiu se construiesc asa numitele pattern databases – se pleaca de la starea scop si se merge in sens invers, expandand nodurile pana cand toate posibilitatile de pattern-uri au fost explorate. Intrucat ne intereseaza distanta minima de la un anumit pattern pana la solutie, aceasta explorare a pattern-urilor trebuie sa fie o explorare Breadth-First. Folosind aceasta parcurgere vom asocia fiecarui pattern costul solutiei pana la starea tinta intr-o structura de tip hash map.

In momentul in care vom dori sa aflam euristica unei stari a puzzle-ului vom face lookup pe pattern-ul corespunzator acestui pattern in tabela hash si vom obtine valoarea euristicii. Daca avem mai multe tabele pentru mai multe tipuri de pattern-uri vom face maximul intre valorile corespunzatoare fiecarui pattern. Superioritatea acestor euristici a fost dovedita experimental [1].

Implementati o euristica pentru problema 8-puzzle folosind solutii pentru problemele partiale corespunzatoare pattern-urilor {1,2,3,4} si {5, 6, 7, 8} si observati cate noduri sunt expandate comparativ cu celelalte euristici implementate. Rezolvarea se va face in Pattern.cpp/Database.java unde veti implementa modul in care baza de date este generata.

[Extra] Tema de gandire

Particularizati algoritmul A* pentru a rezolva problema cubului Rubik.

Marea dificultatea in cazul acestei probleme este numarul mare de posibilitati de expandare, ceea ce necesita o euristica foarte buna pentru a preveni epuizarea rapida a memoriei. Prima euristica utila pentru aceasta problema nu a fost inventata de un om, ci de un program – ABSOLVER[9] care foloseste tehnica problemei relaxate. Tot ABSOLVER a gasit cea mai eficienta euristica pentru problema 8-puzzle si 15-puzzle.