Proiectarea Algoritmilor 2011-2012

Laborator 11

Algoritmi euristici de explorare a grafurilor. A*

Cuprins

1 Obiective laborator

În cadrul acestui laborator se va discuta despre modelarea problemelor sub forma grafurilor de stări și despre algoritmi specializați în găsirea soluțiilor pentru acest tip de grafuri. De asemenea, se vor discuta modalitățile care pot fi folosite în analiza complexității și pe baza acestor metode se vor prezenta avantajele și limitările acestei clase de algoritmi.

2 Importanţă – aplicaţii practice

Algoritmii de căutare euristica sunt folosiți în cazurile care implica găsirea unor soluții pentru probleme pentru care fie nu exista un model matematic de rezolvare directa, fie acest model este

prea complicat pentru a fi implementat. În acest caz e necesara o explorare a spațiului stărilor problemei pentru găsirea unui răspuns. Întrucât o mare parte dintre problemele din viața reală pornesc de la aceste premise, gama de aplicații a algoritmilor euristici este destul de larga. Proiectarea agenților inteligenți [1], probleme de planificare, proiectare circuitelor VLSI [3], robotica, căutare web, algoritmi de aproximare pentru probleme NP-Complete [10], teoria jocurilor sunt doar câteva dintre domeniile în care căutarea informata este utilizată.

3 Descrierea problemei și a rezolvărilor

3.1 Prezentare generală a problemei

Primul pas în rezolvarea unei probleme folosind algoritmi euristici de explorare este definirea exacta a problemei, prin tuplul (Si, O, Sf) – stare inițială, operatori, stări finale. Încercăm să cunoaștem următorii parametri:

- **Starea inițială a problemei** – reprezintă configurația de plecare.
- **Funcție de expandare a nodurilor** – în cazul general este o lista de perechi (acțiune, stare_rezultat). Astfel, pentru fiecare stare se enumera toate acțiunile posibile precum și starea care va rezulta în urma aplicării respectivei acțiuni.
- **Predicat pentru starea finala** – funcție care întoarce *adevărat* dacă o stare este stare scop și *fals* altfel
- **Funcție de cost** – atribuie o valoare numerică fiecărei căi generate în procesul de explorare. De obicei se folosește o funcție de cost pentru fiecare acțiune/tranziție, atribuind, astfel, o valoare fiecărui arc din graful stărilor.

În funcție de reprezentarea problemei, sarcina algoritmilor de căutare este de a găsi o cale din starea inițială într-o stare scop. Daca algoritmul găsește o soluție atunci când mulțimea soluțiilor este nevidă spunem că algoritmul este complet. Daca algoritmul găsește și calea de cost minim către starea finală spunem că algoritmul este optim.

În principiu, orice algoritm pe grafuri discutat în laboratoarele și cursurile anterioare poate fi utilizat pentru găsirea soluției unei probleme astfel definite. În practica, însă, mulți dintre acești algoritmi nu sunt utilizați în acest context, fie pentru că explorează mult prea multe noduri, fie pentru ca nu garantează o soluție pentru grafuri definite implicit (prin stare inițială și funcție de expandare).

Algoritmii euristici de căutare sunt algoritmi care lucrează pe grafuri definite ca mai sus și care folosesc o informație suplimentară, neconținută în definirea problemei, prin care se accelerează procesul de găsirea a unei soluții.

În cadrul explorării stărilor fiecare algoritm generează un arbore, care în rădăcină va conține starea inițială. Fiecare nod al arborelui va conține următoarele informații:

1. **Starea conținută** - stare(nod)
2. **Părintele nodului** – π(nod)

3. **Cost cale –** costul drumului de la starea inițială până la nod – g(nod)

De asemenea, pentru fiecare nod definim și o funcție de evaluare **f** care indica cat de promițător este un nod în perspectiva găsirii unui drum către soluție. (De obicei, cu cat f este mai mic, cu atât nodul este mai promițător).

Am menționat mai sus ca algoritmii de căutare euristica utilizează o informație suplimentară referitoare la găsirea soluției problemei. Această informație este reprezentata de o funcție **h**, unde h(nod) reprezintă drumul *estimat* de la nod la cea mai apropiată stare soluție. Funcția h poate fi definită în orice mod, existând o singura constrângere:

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

3.2 Algoritmi de căutare informată

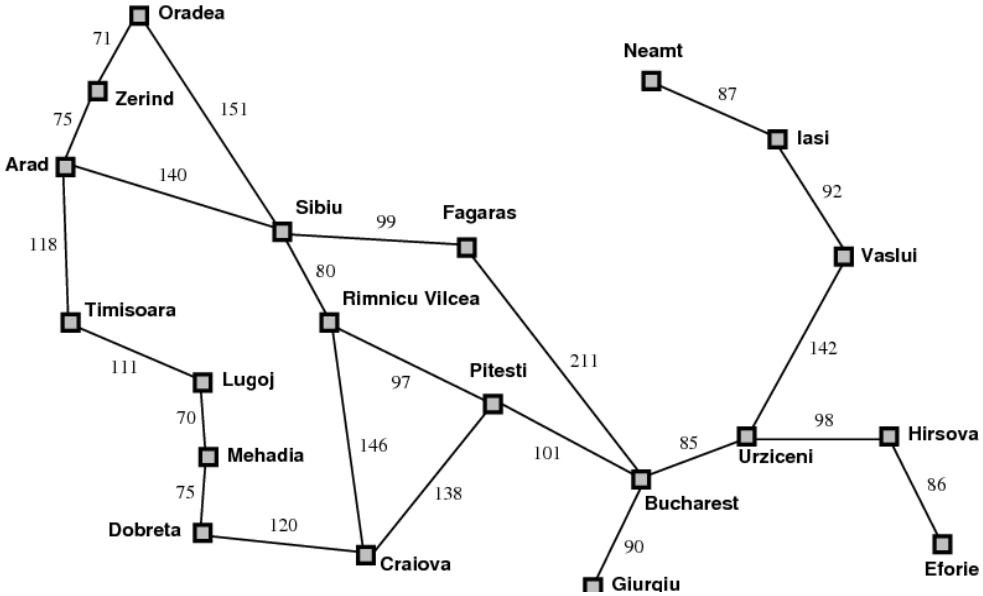
Întrucât funcționează prin alegerea, la fiecare pas, a nodului pentru care f(nod) este minim, algoritmii prezentați mai jos fac parte din clasa algoritmilor de căutare informată (nodul cel mai promițător este explorat mereu primul). Best-First ține cont doar de istoric (informații sigure), pe când A* estimează costul până la găsirea unei soluții.

De notat ca BFS și DFS sunt particularizări ale Best-First:

- pentru BFS: f = adâncime (S)
- pentru DFS: f = - adâncime (S)

Vom prezenta în continuare câțiva dintre cei mai importanți algoritmi de căutare euristica. Vom folosi pentru exemplificare, următoarea problema: data fiind o harta rutiera a României, sa se găsească o cale (de preferință de cost minim) intre Arad și București.

Pentru aceasta problema starea inițială indică că ne aflam în orașul Arad, starea finală este dată de predicatul *Oras_curent == Bucuresti*, funcția de expandare întoarce toate orașele în care putem ajunge dintr-un oraș dat, iar funcția de cost indică numărul de km al fiecărui drum intre doua orașe, presupunând că viteza de deplasare este constanta. Ca euristica vom utiliza pentru fiecare oraș distanta geometrica(în linie dreapta) până la București.



3.3 *Greedy Best-First*

În cazul acestui algoritm se considera ca nodul care merita sa fie expandat în pasul următor este cel mai apropiat de soluție. Deci, în acest caz, avem:

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

Pseudocodul acestui algoritm:

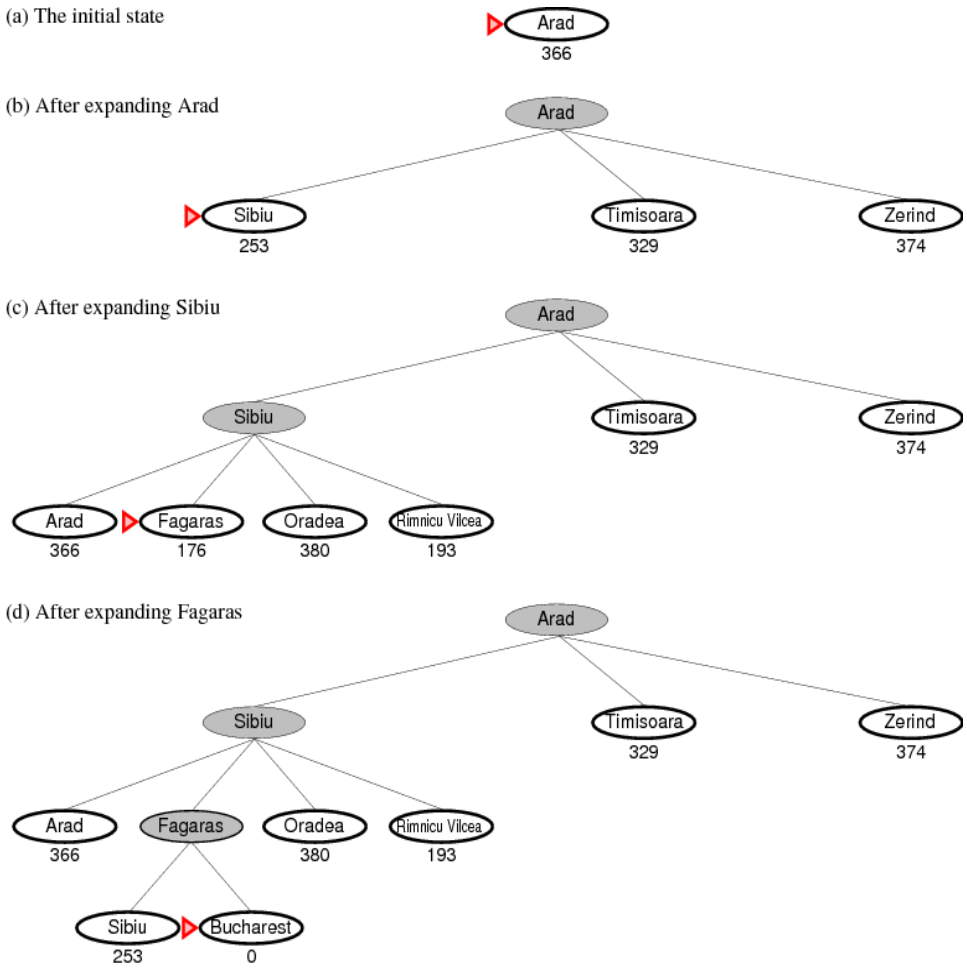
```
Greedy Best-First(sinitial , expand, h, solution)
      closed ← {}
      n ← new-node()
      state(n) ← sinitial
      π(n) ← nil
      open ← { n }
      // Bucla principala
      repeat
            if open = Ø then return failure
            n ← get_best(open) with f(n) = h(n) = min
        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

În cadrul algoritmului se folosesc două mulțimi: *closed* – indica nodurile deja explorate și expandate și *open* – nodurile descoperite dar neexpandate. Open este inițializată cu nodul corespunzător stării inițiale. La fiecare pas al algoritmului este ales din open nodul cu valoarea

f(n) = h(n) cea mai mică (din acest motiv e de preferat ca open să fie implementată ca o coadă de priorități). Daca nodul se dovedește a fi o soluție a problemei atunci este întoarsă ca rezultat calea de la starea inițială până la nod (mergând recursiv din părinte în părinte). Daca nodul nu a fost deja explorat atunci se expandează iar nodurile corespunzătoare stărilor rezultate sunt introduse în mulțimea open. Dacă mulțimea open rămâne fără elemente atunci nu exista niciun drum către soluție și algoritmul întoarce eșec.

Greedy Best-First urmărește mereu soluția care pare cea mai aproape de sursă. Din acest motiv nu se vor analiza stări care deși par mai depărtate de soluție produc o cale către soluție mai scurtă (vezi exemplul de rulare). De asemenea, întrucât nodurile din closed nu sunt niciodată reexplorate se va găsi calea cea mai scurta către scop doar dacă se întâmpla ca această cale să fie analizată înaintea altor cai către aceiași stare scop. Din acest motiv, algoritmul nu este optim. De asemenea, pentru grafuri infinite e posibil ca algoritmul sa ruleze la infinit chiar dacă exista o soluție. Rezultă că algoritmul nu îndeplinește nici condiția de completitudine.

În figura de mai jos se prezintă rularea algoritmului Greedy Best-First pe exemplul dat mai sus. În primul pas algoritmul expandeaza nodul Arad, iar ca nod următor de explorat se alege Sibiu, întrucât are valoarea h(n) minimă. Se alege în continuare Făgăraș după care urmează București, care este un nod final. Se observa însă că acest drum nu este minimal. Deși Făgăraș este mai aproape ca distanta geometrica de București, în momentul în care starea curenta este Sibiu alegerea optimala este Râmnicu-Vâlcea. În continuare ar fi urmat Pitești și apoi București obținându-se un drum cu 32 km mai scurt.



3.4 A* (A star)

A* reprezintă cel mai cunoscut algoritm de căutare euristica. El folosește, de asemenea o politica Best-First, însă nu suferă de aceleași defecte pe care le are Greedy Best-First definit mai sus. Acest lucru este realizat prin definirea funcției de evaluare astfel:

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

A* evaluează nodurile combinând distanta deja parcursa până la nod cu distanta estimata până la cea mai apropiata stare scop. Cu alte cuvinte, pentru un nod n oarecare, **f(*n*) reprezintă costul estimat al celei mai bune soluții care trece prin *n***. Aceasta strategie se dovedește a fi completă și optimală dacă euristica h(n) este admisibilă:

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

unde $h^*(n)$ este distanța exactă de la nodul n la cea mai apropiata soluție. Cu alte cuvinte A* întoarce mereu soluția optimă dacă o soluție există atât timp cât rămânem optimiști și nu supraestimam distanta până la soluție. Daca h(n) nu este admisibilă o soluție va fi în continuare găsită, dar nu se garantează optimalitatea. De asemenea, pentru a ne asigura ca vom găsi drumul optim către o soluție chiar dacă acest drum nu este analizat primul, A* permite scoaterea nodurilor din closed și reintroducerea lor în open dacă o cale mai bună pentru un nod din closed (g(n) mai mic) a fost găsită.

Algoritmul evoluează în felul următor: inițial se introduce în mulțimea open (organizata ca o coada de priorități după f(n)) nodul corespunzător stării inițiale. La fiecare pas **se extrage din open nodul cu f(n) minim**. Daca se dovedește ca nodul n conține chiar o stare scop atunci se întoarce calea de la starea inițială până la nodul n. Altfel, dacă nodul nu a fost explorat deja se expandează. Pentru fiecare stare rezultata, dacă nu a fost generata de alt nod încă (nu este nici în open nici în closed) atunci se introduce în open. Daca exista un nod corespunzător stării generate în open sau closed se verifica dacă nu cumva nodul curent produce o cale mai scurtă către s. Daca acest lucru se întâmplă se setează nodul curent ca părinte al nodului stării s și **se corectează distanta g.** Aceasta corectare implica reevaluarea tuturor cailor care trec prin nodul lui s, deci acest nod va trebui reintrodus în open în cazul în care era inclus în closed.

Pseudocodul pentru A* este prezentat în 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)
```

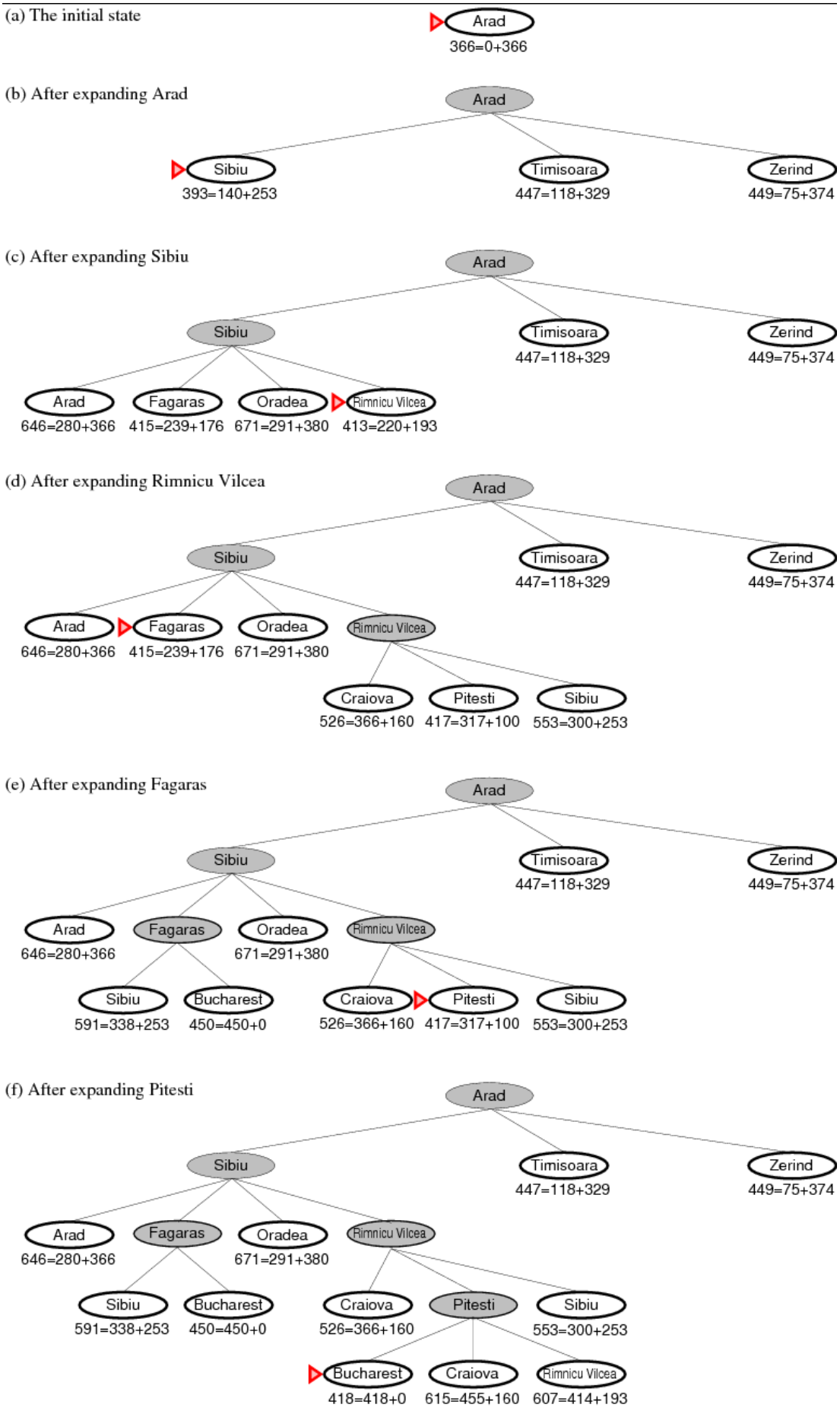
```
            else if n not în closed then
                  closed ← closed U {n}
                  for each s în expand(n)
                        cost_from_n ← g(n) + cost(state(n), s)
                        if not (s în closed U open) then
                              n' ← new-node()
                              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' în closed then
                                          closed ← closed – { n'}
                                          open ← open U { n'}
                  end-for
      end-repeat
```

Algoritmul prezentat mai sus va întoarce calea optima către soluție, dacă o soluție există. Singurul inconvenient fata de Greedy Best-First este ca sunt necesare reevaluările nodurilor din closed. Si aceasta problema poate fi rezolvata dacă impunem o condiție mai tare asupra euristicii h, și anume ca euristica să fie **consistentă** (sau **monotonă**):

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

Daca o funcție este consistentă atunci ea este și admisibilă. Daca euristica h îndeplinește și această condiție atunci algoritmul A* este asemănător cu Greedy Best-First cu modificarea că funcția de evaluare este f = g + h, în loc de f = h.

În imaginea de mai jos se prezintă rularea algoritmului pe exemplul laboratorului. Se observă că euristica aleasă (distanța în linie dreaptă) este consistentă și deci admisibilă. Se observă că în pasul (e), după expandarea nodului Făgăraș deși există o soluție în mulțimea open aceasta nu este aleasă pentru explorare. Se va alege Pitești, întrucât f(nod(București)) = 450 > f(nod(Pitești)) = 417, semnificația acestei inegalități fiindcă e posibil sa existe prin Pitești un drum mai bun către București decât cel descoperit până acum.

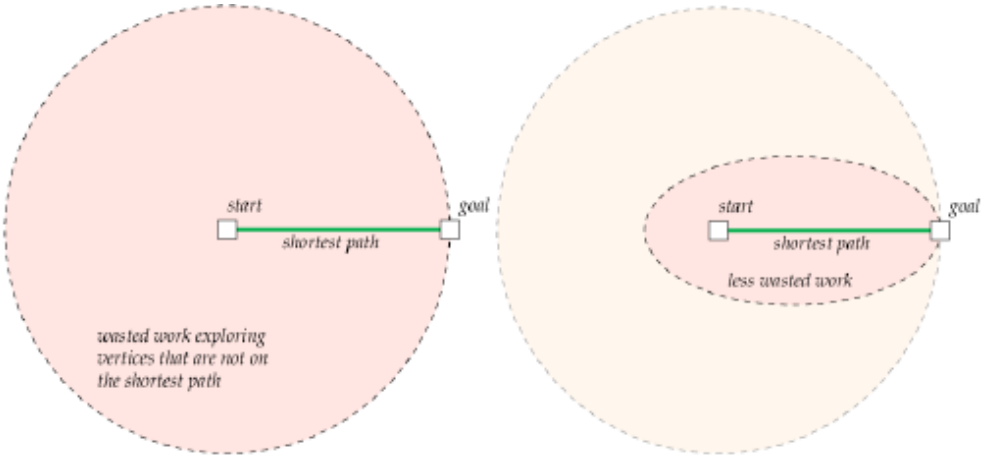

(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

3.5 Complexitatea algoritmului A*

Pe lângă proprietățile de completitudine și optimalitate A* mai are o calitate care îl face atrăgător: pentru o euristică dată orice algoritm de căutare complet și optim va explora cel puțin la fel de multe noduri ca A*, ceea ce înseamnă că A* este **optimal din punctul de vedere al eficientei**. În practica, însă, A* poate fi de multe ori imposibil de rulat datorita dimensiunii prea mari a spațiului de căutare. Singura modalitate prin care spațiul de căutare poate fi redus este prin găsirea unei euristici foarte bune – cu cat euristica este mai apropiata de distanta reala fata de stare soluție cu atât spațiul de căutare este mai strâns (vezi figura de mai jos). S-a demonstrat ca spațiul de căutare începe să crească exponențial dacă eroarea euristicii fata de distanta reală până la soluție nu are o creștere sub-exponențială:

$$\left|h(n)-h^*(n)\right| \le O\left(\log h^*(n)\right)$$

Din păcate, în majoritatea cazurilor, eroarea creste liniar cu distanta până la soluție ceea ce face ca A* sa devină un algoritm mare consumator de timp, dar mai ales de memorie. Întrucât în procesul de căutare se rețin toate nodurile deja explorate (mulțimea closed), în cazul unei dimensiuni mari a spațiului de căutare cantitatea de memorie alocata catarii este în cele din urma epuizată. Pentru acest inconvenient au fost găsite mai multe soluții. Una dintre acestea este utilizarea unor euristici care sunt mai strânse de distanța reală până la starea scop, deși nu sunt admisibile. Se obțin soluții mai rapid, dar nu se mai garantează optimalitatea acestora. Folosim aceasta metoda când ne interesează mai mult să găsim o soluție repede, indiferent de optimalitatea ei.



Volumul spațiului de căutare în funcție de euristica aleasa

Alte abordări presupun sacrificarea timpului de execuție pentru a margini spațiul de memorie utilizat [3]. Aceste alternative sunt prezentate în continuare.

3.6 IDA*, RBFS, MA*

3.6.1 IDA*

Iterative deepening A* [5] utilizează conceptul de adâncire iterativă [1][8] în procesul de explorare a nodurilor – la un anumit pas se vor explora doar noduri pentru care funcția de evaluare are o valoare mai mica decât o limită dată, limită care este incrementată treptat până se găsește o soluție. IDA* nu mai necesită utilizarea unor mulțimi pentru reținerea nodurilor explorate și se comportă

bine în cazul în care toate acțiunile au același cost. Din păcate, devine ineficient în momentul în care costurile sunt variabile.

3.6.2 RBFS

Recursive Best-First Search [6] funcționează intr-un mod asemănător cu DFS, explorând la fiecare pas nodul cel mai promițător fără a reține informații despre nodurile deja explorate. Spre deosebire de DFS, **se retine în orice moment cea mai bună alternativă sub forma unui pointer către un nod neexplorat.** Daca valoarea funcției de evaluare (f = g + h) pentru nodul curent devine mai mare decât valoarea căii alternative, drumul curent este abandonat și se începe explorarea nodului reținut ca alternativă. RBFS are avantajul ca spațiul de memorie crește doar liniar în raport cu lungimea căii analizate. Marele dezavantaj este ca se ajunge de cele mai multe ori la reexpăndări și reexplorări repetate ale acelorași noduri, lucru care poate fi dezavantajos mai ales dacă exista multe cai prin care se ajunge la aceiași stare sau funcția *expand* este costisitoare computațional (vezi problema deplasării unui robot intr-un mediu real). RBFS este optimal dacă euristica folosită este admisibilă.

3.6.3 MA*

Memory-bounded A* este o varianta a A* în care se limitează cantitatea de memorie folosită pentru reținerea nodurilor. Exista doua versiuni – MA* [7]și SMA* [4] (Simple Memory-Bounded A*), ambele bazându-se pe același principiu. SMA* rulează similar cu A* până în momentul în care cantitatea de memorie devine insuficientă. În acest moment spațiul de memorie necesar adăugării unui nod nou este obținut prin ștergerea celui mai puțin promițător nod deja explorat. În cazul în care exista o egalitate în privința valorii funcției de evaluare se șterge nodul cel mai vechi. Pentru a evita posibilitatea în care nodul șters este totuși un nod care conduce la o cale optimala către soluție valoarea f a nodului șters este reținută la nodul părinte intr-un mod asemănător felului în care în RBFS se reține cea mai bună alternativă la nodul curent. Și în acest caz vor exista reexplorări de noduri, dar acest lucru se va întâmpla doar când toate căile mai promițătoare vor eșua. Deși există probleme în care aceste regenerări de noduri au o frecvență care face ca algoritmul să devină intractabil, MA* și SMA* asigura un compromis bun între timpul de execuție și limitările de memorie.

4 Concluzii și observații

Deseori singura modalitate de rezolvare a problemelor dificile este de e explora graful stărilor acelei probleme. Algoritmii clasici pe grafuri nu sunt întotdeauna potriviți pentru căutarea în aceste grafuri fie pentru că nu garantează un rezultat fie pentru că sunt ineficienți.

Algoritmii euristici sunt algoritmi care explorează astfel de grafuri folosindu-se de o informație suplimentară despre modul în care se poate ajunge la o stare scop mai rapid. A* este, teoretic, cel mai eficient algoritm de explorare euristică. În practică, însă, pentru probleme foarte dificile, A* implică un consum prea mare de memorie. În acest caz se folosesc variante de algoritmi care încearcă sa minimizeze consumul de memorie în defavoarea timpului de execuție.

5 Referințe

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

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

[3] Mehul Shah - Algorithms în 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 în 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