

Laborator 11 : Algoritmi euristici de explorare a grafurilor. A*

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.

Importanță – aplicații practice

Algoritmii de căutare euristica sunt folosiți în cazurile care implică găsirea unor soluții pentru probleme pentru care fie nu există un model matematic de rezolvare directă, fie acest model este prea complicat pentru a fi implementat. În acest caz este necesară 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 largă. 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 informată este utilizată.

Descrierea problemei și a rezolvărilor

Prezentare generală a problemei

Primul pas în rezolvarea unei probleme folosind algoritmi euristici de explorare este definirea exactă 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 expansiune a nodurilor** – în cazul general este o listă de perechi (acțiune, stare_rezultat). Astfel, pentru fiecare stare se enumerează toate acțiunile posibile precum și starea care va rezulta în urma aplicării respectivei acțiuni.
- **Predicat pentru starea finală** – 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. Dacă algoritmul găsește o soluție atunci când mulțimea soluțiilor este nevidă spunem că algoritmul este complet. Dacă 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 practică, însă, mulți dintre acești algoritmi nu sunt utilizați în acest context, fie pentru că explorează mult prea multe noduri, fie pentru că 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 urmatoarele informatii:

1. **Starea continuta** - stare(nod)
2. **Parintele nodului** - $p(nod)$
3. **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:

$$h(n) = 0 \text{ pt } \forall n, \text{solutie}(n) = \text{adevarat}$$

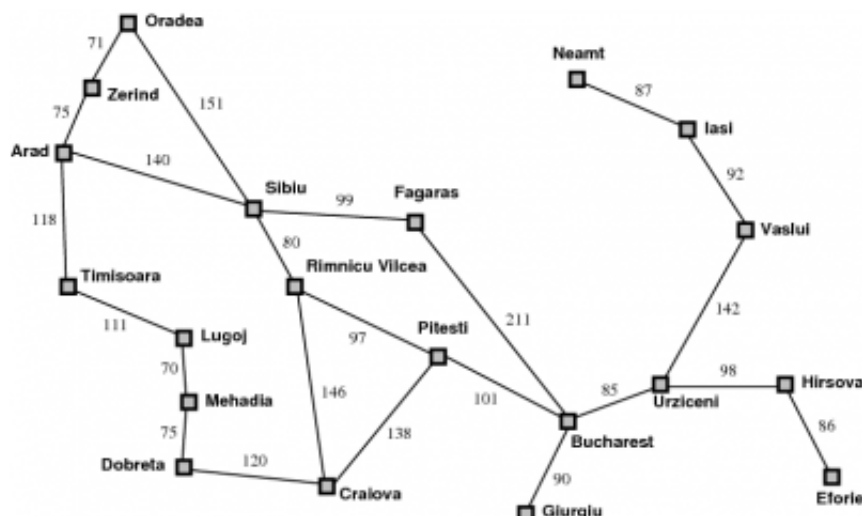
Algoritmi de cautare informata

Intrucat functioneaza prin alegerea, la fiecare pas, a nodului pentru care $f(nod)$ este minim, algoritmii prezentati mai jos fac parte din clasa algoritmilor de cautare informata (nodul cel mai promitator este explorat mereu primul). Best-First tine cont doar de istoric (informatii sigure), pe cand A* estimeaza costul pana la gasirea unei solutii. De notat ca BFS si DFS sunt particularizari ale Best-First:

- pentru BFS: $f = \text{adancime}(S)$
- pentru DFS: $f = -\text{adancime}(S)$

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 indica faptul ca ne aflam in orasul Arad, starea finala este data de predicatul $\text{Oras_curent} == \text{Bucuresti}$, 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.



Greedy Best-First

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

$$f(n) = h(n), \text{ pt. oricare } n$$

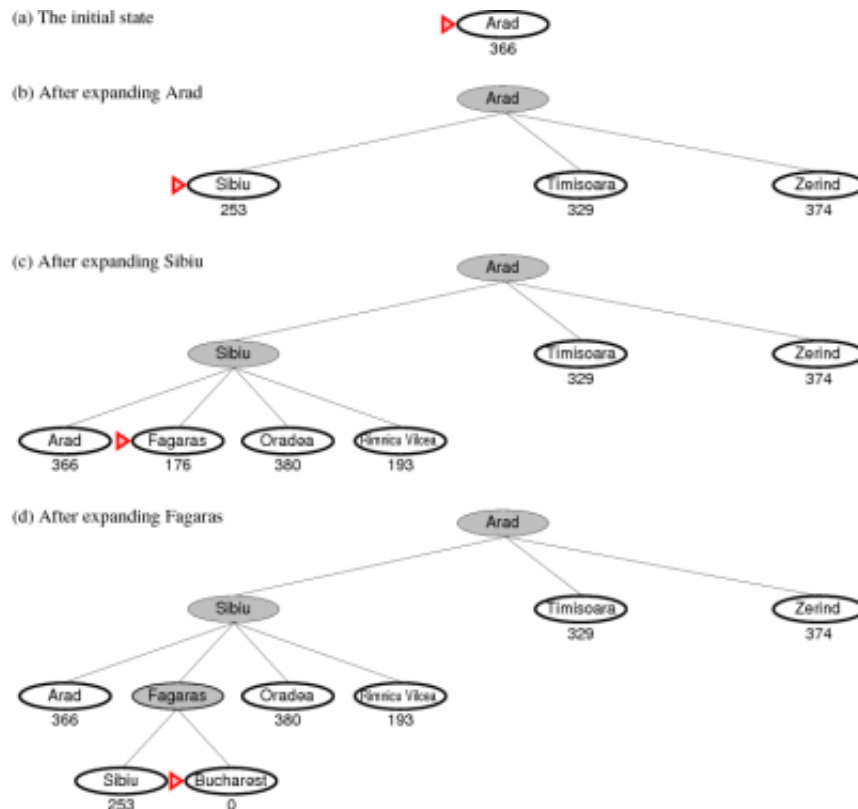
Pseudocodul acestui algoritm:

```
Greedy Best-First(sinitial , expand, h, solution)
  closed ← {}
  n ← new-node()
  state(n) ← sinitial
  π(n) ← nil
  open ← { n }
  // Bucla principala
  repeat
    if open = ∅ 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 ∪ {n}
      for each s in expand(n)
        n' ← new-node()
        state(n') ← s
        π(n') ← n
        open ← open ∪ {n'}
      end-for
    end-repeat
```

În cadrul algoritmului se folosesc două mulțimi: closed – indică 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 cu priorități). Dacă 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). Dacă 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 există 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 desigur par mai departate 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 scurtă către scop doar dacă se întâmplă ca aceasta cale să fie analizată înaintea altor căi către aceeași stare scop. Din acest motiv, algoritmul nu este optim. De asemenea, pentru grafuri infinite e posibil ca algoritmul să ruleze la infinit chiar dacă există o soluție. Rezultă ca 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 expandează 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 observă însă că acest drum nu este minimal. Deși Făgăraș este mai aproape ca distanță geometrică de București, în momentul în care starea curentă este Sibiu alegerea optimă este Râmnicu-Valcea. În continuare ar fi urmat Pitesti și apoi București obținându-se un drum cu 32 km mai scurt.



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:

$$f(n) = g(n) + h(n) \text{ , pt. oricare } n \in \text{Noduri}$$

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 optima daca euristica $h(n)$ este admisibila:

$$0 \leq h(n) \leq h^*(n) \text{ , pt. oricare } n \in \text{Noduri}$$

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 urmatoare: 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 }

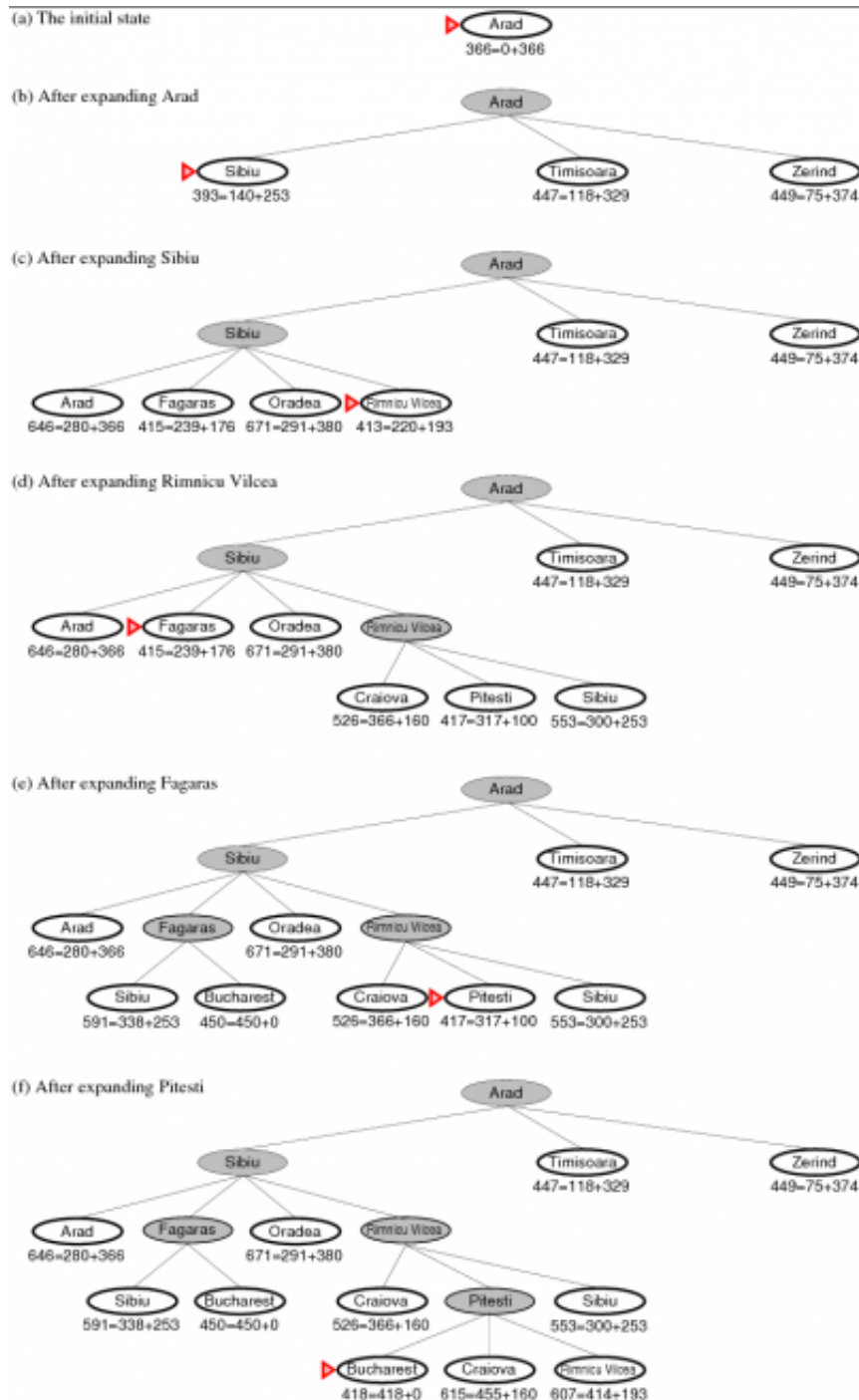
  //Bucula principala
  repeat
    if open = ∅ then return failure
    n ← get_best(open) with f(n) = g(n)+h(n) = min
    open ← open - {n}
    if solution(state(n)) then return build-path(n)
    else if n not in closed then
      closed ← closed U {n}
      for each s in expand(n)
        cost_from_n ← 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
  end-repeat

```

Algoritmul prezentat mai sus va intoarce calea optima catre solutie, daca o solutie exista. Singurul inconvenient 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**):

$$h(n) \leq h(n') + \text{cost}(n, n') \text{ , pt. oricare } n' \in \text{expand}(n)$$

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 rulara 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 $f(\text{nod}(\text{Bucuresti})) = 450 > f(\text{nod}(\text{Pitesti})) = 417$, semnificatia acestei inegalitati fiindca e posibil sa existe prin Pitesti un drum mai bun catre Bucuresti decat cel descoperit pana acum.

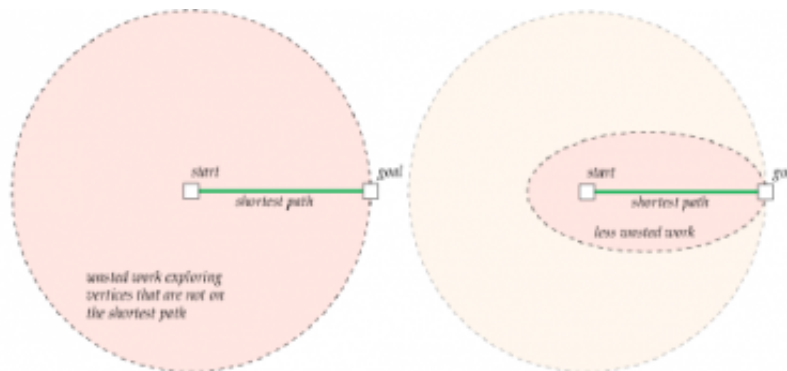


Complexitate algoritmului A*

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 linear 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 inconvenient 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.



Volumul spatiului de cautare in functie de euristica aleasa

Alte abordari presupun sacrificarea timpului de executie pentru a margini spatiul de memorie utilizat [3]. Aceste alternative sunt prezentate in continuare.

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 caili alternative, drumul curent este abandonat si se incepe explorarea nodului retinut ca alternativa. RBFS are avantajul ca spatiul de memorie creste doar linear in raport cu lungimea caili 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 computationally (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 optima la 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 intructabil, MA* si SMA* asigura un compromis bun intre timpul de executie si limitarile de memorie.

Concluzii și observații

Deseori singura modalitate de rezolvare a problemelor dificile este de a 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 euristica. 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 să minimizeze consumul de memorie în defavoarea timpului de execuție.

Referinte

- [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 in the real world (Course Notes) - Introduction to VLSI routing [<http://www.cs.cmu.edu/afs/cs/project/pscico-guyb/294/class-notes/all/18.ps>]
- [4] S. Russel - Efficient memory-bounded search methods [<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.6489&rep=rep1&type=pdf>]
- [5] R. Korf - Depth-first iterative-deepening: an optimal admissible tree search
- [6] Recursive Best-First Search - Prezentare [<http://www.ailab.si/gregorl/Students/mui/Recursive%20Best-First%20Search.ppt>]
- [7] P. Chakrabati, S. Ghosh, A. Acharya, S. DeSarkar – Heuristic search in restricted memory
- [8] Wikipedia - Iterative deepening [http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search]
- [9] A. E. Prieditis - Machine Discovery of Effective Admissible Heuristics [<http://dli.iit.ac.in/ijcai/IJCAI-91-VOL2/PDF/017.pdf>]
- [10] Wikipedia - Travelling salesman problem - Heuristic and approximation algorithms [http://en.wikipedia.org/wiki/Travelling_salesman_problem#Heuristic_and_approximation_algorithms]
- [11] Wikipedia – A* [http://en.wikipedia.org/wiki/A*_search_algorithm]
- [12] Tutorial A* [<http://www.policyalmanac.org/games/aStarTutorial.htm>]
- [13] H. Kaindl, A. Khorsand - Memory Bounded Bidirectional Search, AAAI-94 Proceedings [<http://aaai.org/Papers/AAAI/1994/AAAI94-209.pdf>]

Probleme

Labirint

Fie un labirint dat sub forma unei matrici în care 0 înseamnă obstacol și 1 înseamnă liber. Se cere să se determine un drum între două poziții din labirint folosind algoritmi de căutare informată. Să se implementeze algoritmul A* pentru rezolvarea problemei folosind o euristică admisibilă. [4 pct]

Problema misionarilor și a canibalilor

Trei misionari și trei canibali ajung la malul estic al unui râu. Aici se află o barcă cu două locuri cu care se poate traversa râul (râul nu se poate traversa înot). Dacă pe unul dintre maluri numărul de canibali este mai mare decât numărul de misionari, atunci misionarii de pe acel mal vor fi mâncați de canibali. Se cere să determinați cum pot trece toți râul fără ca misionarii să fie mâncați de canibali.

Să se rezolve această problemă folosind algoritmul A* cu:

- a) O euristică care nu este admisibilă [3 pct]
- b) O euristică admisibilă și monotona [3 pct]

pa/laboratoare/laborator-11.txt · Last modified: 2013/05/14 12:06 by cristian.condurache