Beam Search

Alexandru Sorici și Andrei Olaru

November 1, 2022

1 Introducere

În probleme de căutare ce prezintă un spațiu mare al stărilor, algoritmi de tip best-first (e.g. A*) pot ajunge să consume o cantitate mare de memorie, făcând aplicarea lor dificl de scalat și nepractică. Pentru un algoritm precum A*, consumul de memorie se datorează menținerii frontierei, în care se permite reinserarea unor stări, dacă pentru ele se poate ajunge la un cost mai mic al căii de la starea inițială.

În astfel de instanțe, în practică pot fi folosiți algoritmi care fac un compromis între *lungimea căii găsite* (mai mare decât cea optimă) și consumul de memorie necesar (care se dorește a avea o creștere liniară cu numărul de stări expandate la fiecare pas).

Obiectivul acestei teme este să vă familiarizeze cu astfel de algoritmi, cerând implementarea lor, precum și analiza performanței acestora (în termen de lungime a căii găsite, memorie consumată / număr de stări explorate, timp de rulare) pe seama a două probleme clasice: N-puzzle și Turnurile din Hanoi.

2 Beam Search pentru jocul N-Puzzle (5p)

Metoda ***beam-search*** clasică se bazează pe ideea unei căutări greedy, ca cea folosită în algoritmul *Hill Climbing*, dar permite menținerea unui *cel mai bun set* de stări la fiecare pas, în loc să aleagă de fiecare data expandarea unei singure stări (cea mai promițătoare). Concret, algoritmul folosește o cautare în lațime (breadth-first) pentru a construi arborele de căutare, dar menține cel mult B stări pentru fiecare nivel al arborelui, selectate după criteriul costului minim al unei *funcții euristice*, care estimează costul căii până la starea obiectiv.

B denotă *lățimea bârnei* (beam) și este un parametru fixat la începutul căutării. Cu cât B este mai mic, cu atât numărul maxim de stări explorate și stocate în memorie va fi mai mic, dar căile găsite (secvența de stări de la cea inițială la cea obiectiv) vor fi mai lungi.

Pentru a putea studia influența valorii B asupra performanțelor algoritmului beam-search se propune rezolvarea problemei N-puzzle [1], cu diferite dimensiuni ale laturii N (N=4, 5 și 6).

Sarcinile voastre sunt următoarele:

1. Implementați **două euristici** pentru soluționarea jocului N-puzzle (1p)
2. Implementați algoritmul A* pentru a obține numărul optim de mișcări necesar soluționării unui joc N-puzzle. (0.5p)
3. Implementați algoritmul beam-search (a se vedea pseudo-codul din Figura 1) (1.5p)
4. Efectuați o analiză a rezultatelor celor două implementări, discutând următoarele:
    - Rulați algoritmul A* și Beam-search cu B = 1, 10, 50, 100, 500, 1000 pe fiecare instanță de joc din setul celor furnizate ca test.

[1] N-puzzle: https://en.wikipedia.org/wiki/15_puzzle

```
Beam-Search(start, B, h, limita)
1.  beam = {start}
2.  vizitat = {start}
3.  cât timp beam ≠ ∅ și |vizitat| < limita
4.    succ = ∅
5.    pentru fiecare s ∈ beam
6.      succ = succ ∪ succesori(s)
7.    dacă vreuna dintre dintre stările din succ este stare scop
8.      atunci întoarce SUCCES
9.    selectat = cele mai bune B stări, sortate după h(s)
10.   vizitat = vizitat ∪ selectat
11.   beam = selectat
```

Figure 1: Pseudo-cod pentru Beam search

- Generați un tabel conținând *media* și *varianța* următoarelor metrici, calculate peste numărul dat de jocuri **finalizate** (i.e. în care se ajunge la starea obiectiv în limita de memorie impusă) pentru fiecare dimensiune a laturii N (câte 20 de jocuri per dimensiune de latură): (i) lungimea căii de la starea inițială la cea finală, (ii) numărul de stări stocate în memorie, (iii) timpul de rulare (în secunde). Tabelul va avea ca linii diferitele configurații ale algoritmilor (A*, beam-search cu diferite valori ale lui B), iar pe coloane media si varianța metricilor cerute.
- Discutați rezultatele obținute în termenii compromisului între optimalitatea căii și memoria utilizată pentru a stoca stări explorate.

Notă!

- Pentru fiecare dimensiune de problemă, folosiți problemele descrise în fișierul `problems<n>.txt`. O mostră de citire din fișier se află în scheletul oferit.
- Pentru rulările algoritmilor A* și Beam-search veți lua în considerare următoarele limite ale *numărului de stări stocate*:
  - pentru latura N=4 - 100000
  - pentru latura N=5 - 500000
  - pentru latura N=6 - 1000000
- Atenție: rularea algoritmului poate consuma o cantitate însemnată de memorie RAM. Pentru limitarea memoriei RAM folosite, puteți folosi
  ```
  from resource import setrlimit, RLIMIT_AS, RLIMIT_DATA
  setrlimit(RLIMIT_DATA, (3 * 10 ** 9, 3 * 10 ** 9))
  ```
  (pentru o limită de 3 GB, de exemplu)
- Nu orice joc din cele pe care le veți rula se va termina în limita numărului maxim de stări stocate. Raportați metricile din tabelul de mai sus, doar pentru jocurile în care se ajunge la starea obiectiv. Pentru fiecare valoare a laturii de N-puzzle, raportați procentul de jocuri care se finalizează.

3 Generalized Limited Discrepancy Search pentru jocul N-puzzle (5p)

Algoritmul beam-search clasic are avantajul că pune o limită asupra memoriei utilizate în căutarea stării obiectiv. Cu toate acestea, dacă euristica folosită nu este suficient de bună, este posibil ca

```
GLDS(start, h, limita)
1.  vizitat = {start}
2.  discrepanțe = 0
3.  cât timp adevărat
4.    dacă Iterație(start, discrepanțe, h, vizitat, limita) întoarce SUCCES
5.      atunci întoarce SUCCES
6.    discrepanțe = discrepanțe + 1

Iterație(stare, discrepanțe, h, vizitat, limita)
1.  succ = ∅
2.  pentru fiecare s succesor al lui stare
3.    dacă s este stare scop atunci întoarce SUCCES
4.    dacă s ∉ vizitat atunci succ = succ ∪ {s}
5.  dacă succ = ∅  atunci întoarce EȘEC
6.  dacă |vizitat| > limita  atunci întoarce EȘEC
7.  best = cea mai bună stare din succ, după h(s)
8.  dacă discrepanțe = 0  atunci
9.    întoarce Iterație(best, 0, h, vizitat ∪ {best}, limita)
10. altfel
11.   succ = succ \ {best}
12.   cât timp succ ≠ ∅
13.     s = cea mai bună stare din succ, după h(s)
14.     succ = succ \ {s}
15.     dacă Iterație(s, discrepanțe − 1, h, vizitat ∪ {s}, limita) întoarce SUCCES
16.       atunci întoarce SUCCES
17.   întoarce Iterație(best, discrepanțe, h, vizitat ∪ {best}, limita)
```

Figure 2: Pseudo-cod pentru GLDS

beam-search să includă în lista ordonată de explorare de lungime B stări care nu conduc spre starea obiectiv. Pentru a permite "revenirea" asupra unor stări explorate, dorim să studiem un algoritm de tip *backtracking cu branching limitat*, numit Generalized Limited Discrepancy Search (GLDS).

Pentru starea curentă din care începe o căutare, numim *discrepanță (eng. discrepancy)* o cale de explorare în adâncime care nu găsește starea obiectiv *în limita numărului maxim de stări ce pot fi explorate*. Scopul GLDS este să permită explorarea căilor alternative, în limita unui număr maxim de discrepanțe *per nivel al căutarii*. Modul în care se ordonează căile alternative este dat în continuare de funcția euristică. Pseudocodul algoritmului GLDS este dat în Figura 2.

GLDS este un DFS cu o căutare best-first, dar care permite întoarcerea doar limitat. Dacă numărul inițial de discrepanțe este 0, atunci GLDS va avansa în adâncime, alegând mereu cel mai bun succesor, iar când nu mai poate avansa, execuția se termină. Dacă numărul inițial de discrepanțe este 0, atunci din fiecare nod de pe calea dată de best-first se va porni câte o cale alternativă, fiecare cu discrepanță 0. Dacă numărul de discrepanțe inițial este n, din fiecare nod de pe calea dată de best-first se va crea un arbore generat prin același algoritm, dar cu $n-1$ discrepanțe. Cu un număr suficient de mare de discrepanțe permise, algoritmul devine aproape identic cu DFS, mai puțin modalitatea de alegere a ordinii stărilor succesor.

În pseudocodul din Figura 2, se încearcă algoritmul în mod repetat, pornind de la 0 discrepanțe și crescând treptat numărul de discrepanțe permise. La fiecare apel al funcției *Iterație*, se face o căutare în adâncime, astfel: dacă numărul de discrepanțe este zero, căutarea nu se ramifică, și se caută în starea cea mai bună; dacă sunt permise discrepanțe, se caută întâi în toate stările succesor, *în afară de* cea mai bună stare succesor, fiecare căutare având un număr decrementat de discrepanțe permise, și abia apoi se continuă cu cea mai bună stare succesor. Ciclurile sunt evitate prin folosirea variabilei *vizitat*.

Lungimea maximă a căii (numărul de stări ținute în memorie) este limitată la *limita*. Este important de reținut că variabila *vizitat* contorizează mereu stările *stocate în memorie*. Având în vedere natura recursivă a algoritmului GLDS trebuie menținută atenție la modul în care se actualizează structura la intrarea și *revenirea* din recursivitate. În particular, pentru apeluri precum cele de la liniile 9, 15 și 17 este de reținut că variabilele care se adaugă setului *vizitat* la intrarea în recursivitate (*best* și *s*) sunt implicit scoase din set la revenirea din apelul recursiv.

Sarcinile voastre sunt următoarele:

- Implementați algoritmul GLDS conform pseudocodului din Figura 2 (2p)
- Repetați analiza descrisă în Secțiunea 2 privind statisticile jocurilor finalizate vs. nefinalizate, timp de rulare, număr de stări vizitate, lungime a căilor găsite. Efectuați o comparație între rezultatele obținute de beam-search și cele obținute de GLDS, *generând* și *discutând* pe seama următoarelor tipuri de grafice: (3p)
  - Pe același grafic generați un *grouped bar plot* [2] afișând procentul de jocuri finalizate și nefinalizate pentru GLDS si Beam-Search
  - Pe aceleași grafice generați diagrame *box plot* [3] care să prezinte vizual diferențele de medie și variantă (raportată la cele 5 instanțe de problemă per dimensiune a laturii de puzzle) între GLDS și Beam-Search pentru metricile: număr de stări vizitate, lungime a căilor de căutare găsite, timp de rulare. Se va genera câte un set de grafice pentru fiecare dimensiune (N=4, 5 și 6) a problemei.
- **Notă!** Pentru jocurile nefinalizate, raportați ce procent se termină din cauza depășirii numărului maxim de stări vizitate și ce procent se termină din cauza lipsei oricărui succesor.

4 Beam Search cu Limited Discrepancy Backtracking (BLDS) pentru jocurile N-puzzle și Turnurile din Hanoi

Algoritmii prezentați în Secțiunile 2 și 3 pot fi îmbinați într-o procedură ce utilizează părțile esențiale (BFS cu limitare dată de lățimea bârnei pentru beam-search, DFS prin backtracking cu reveniri limitate pentru GLDS) ale fiecărei abordări. Pseudo-codul din Figura 3 ilustrează această îmbinare. Diferența esențială față de GLDS este că BLDS contorizează discrepanțele la nivelul secvențelor (eng. slice), de lungime maxim B, de stări succesor pe fiecare nivel de căutare (nivel al unei căutări BFS).

Pentru fiecare nivel d de căutare, plecând de la setul de noduri conținut în slice-ul curent (variabila *nivel* în pseudo-cod), BLDS va genera toate stările succesor *încă nevizitate* de la nivelul *d+1* și le va *ordona* conform euristicii h. Dacă nu există succesori sau numărul de succesori care ar fi generați pe nivelul *d+1* întrece limita de stări stocate, algoritmul va întoarce eșec. (liniile 2-8 din pseudocod).
BLDS continua prin a considera slice-urile de pe nivelul *d+1*, luând stările (ordonate dupa h) de pe acest nivel și introducându-le într-o secvență de lungime maxim B. Alegerea slice-ului care să constituie mulțimea stărilor de pe acest nivel (variabla *nivel_urm*) se face în funcție de numărul *discrepanțelor*. Similar cu GLDS, dacă numărul curent de discrepanțe este 0, atunci pentru slice-ul *nivel_urm* se va continua expandarea BFS (la nivelul $d+2$) cu discrepanță 0. Dacă numărul de discrepanțe curent este n, atunci pentru fiecare slice de pe nivelul $d+1$ (i.e. fiecare alternativa pentru variabila *nivel_urm* - linia 16 din pseudocod) se va continua explorarea BFS cu $n-1$ discrepanțe. Interpretarea variabilei *vizitat* este similară cu cea din GLDS, astfel că e nevoie de atenție în actualizarea variabilei cu stările din slice-ul dat de *nivel_urm* la intrarea și revenirea din apeluri recursive.

[2] Bar plot folosind biblioteca Python matplotlib
[3] Box plot folosind biblioteca Python matplotlib

```
BLDS(start, h, B, limita)
1.  vizitat = {start}
2.  discrepanțe = 0
3.  cât timp adevărat
4.    dacă Iter({start}, discrepanțe, h, vizitat, limita) întoarce SUCCES
5.      atunci întoarce SUCCES
6.    discrepanțe = discrepanțe + 1

Iter(nivel, discrepanțe, B, h, vizitat, limita)
1.  succ = ∅
2.  pentru fiecare s ∈ nivel
3.    pentru fiecare s' ∈ succesori(s)
4.      dacă s' este stare scop atunci întoarce SUCCES
5.      dacă s' ∉ vizitat atunci succ = succ ∪ {s'}
6.  dacă succ = ∅ atunci întoarce EȘEC
7.  dacă |vizitat| + min(B, |succ|) > limita atunci întoarce EȘEC
8.  sortează succ după h
9.  dacă discrepanțe = 0 atunci
10.   nivel_urm = primele B stări din succ sortat după h
11.   întoarce Iter(nivel_urm, 0, B, h, vizitat ∪ nivel_urm, limita)
12. altfel
13.   deja_explorate = B
14.   cât timp deja_explorate < |succ|
15.     n = min(|succ| − deja_explorate, B)
16.     nivel_urm = următoarele n stări din succ, după deja_explorate
17.     val = Iter(nivel_urm, discrepanțe − 1, B, h, vizitat ∪ nivel_urm, limita)
18.     dacă val = SUCCES atunci întoarce SUCCES
19.     deja_explorate = deja_explorate + |nivel_urm|
20.   nivel_urm = primele B stări din succ sortat după h
21.   întoarce Iter(nivel_urm, discrepanțe, B, h, vizitat ∪ nivel_urm, limita)
```

Figure 3: Pseudo-cod pentru Beam + GLDS

Pentru a testa flexibilitatea soluției BLDS, se propune implementarea unui joc suplimentar față de N-puzzle, anume **Turnurile din Hanoi** [4]. În mod particular, se cere implementarea mecanismului de joc pentru *4 turnuri* și un număr de discuri *N*.

Lista completă de sarcini pentru BLDS este:

1. Implementați algoritmul BLDS (1.5p)
2. Implementați mecanismul de joc pentru jocul *Turnurile din Hanoi*, precum și cel puțin o euristică pentru soluționarea acestuia prin algoritmul BLDS. (2p)
   - Se considera 4 turnuri
   - Testați cu 5, 8, 10 și 15 discuri
   - Inițial, toate discurile sunt plasate pe cel mai din stânga turn
   - la final, toate discurile trebuie să fie plasate pe cel mai din dreapta turn
3. Realizați următorul set de analize (1.5p)

[4] Turnurile din Hanoi: https://en.wikipedia.org/wiki/Tower_of_Hanoi

- Comparați rezultatele obținute de BLDS cu cele ale GLDS și Beam-Search pentru problema N-puzzle. Discutați pe seama metricilor: procent de jocuri rezolvate, număr de stări vizitate, lungime a căii de căutare, timp de rulare. Realizați comparația în mod vizual, folosindu-vă de tipul de grafice indicat în Secțiunea 3. Discutați diferențele observate.
- Rulați algorimtul BLDS pe fiecare dimensiune de problemă și creați un raport similar cu cel pentru N-puzzle, indicând procent de jocuri rezolvate, număr de stări vizitate, lungime a căii de căutare, timp de rulare. Folosiți aceleași limite maxime (pentru memorie și număr maxim de stări vizitate) ca și în cazul jocului N-puzzle.

5 Cerințe upload

Ca rezultat al temei trebuie să furnizați două fișiere:

1. O arhiva conținând codul ce rezolvă tema data. Pentru cei ce rezolvă tema în Python, se acceptă și un notebook de python (fișier .ipynb)
2. Un fișier PDF ce cuprinde toate analizele și comparațile cerute în temă. Atenție! Punctajul maxim se va acorda doar dacă tabelele și graficele incluse în raport sunt însoțite de o *interpretare* a acestora (i.e. o explicare a rezultatelor obținute).