Sokoban
Tema 1 - Inteligență Artificială

Alexandru Sima (332CA)

24 aprilie 2025

Rezumat

Analiza a 2 strategii de rezolvare a jocului de Sokoban prin explorarea spațiului stărilor: **Beam Search** și **Learning Real-Time A***. Prezentarea unei soluții de rezolvare și evaluarea performanțelor, comparând cele 2 strategii, cât și prezentarea raționamentului care a condus la această soluții și comparații cu idei anterioare.

Cuprins

1 Soluție propusă

1.1 Beam Search

Algoritmul funcționează după cum urmează: se pleacă de la o stare inițială, care va reprezenta *beam-ul* inițial; la fiecare pas, se expandează vecinii fiecărei stări din *beam*, se elimină duplicatele și se calculează costul fiecărei stări, conform euristicii, apoi se aleg următoarele k stări pentru a forma noul *beam*. Pentru a evita anumite situații în care algoritmul se blochează în niște minime locale ale funcției de cost, am ales o abordare stocastică: nu se aleg cele mai bune k stări, ci k stări cu o probabilitate care ia în calcul costul stării (aplicând softmax[1] pe costurile obținute).

Algoritmul poate fi ajustat prin:

- *heuristic*: funcție care estimează costul unei stări;
- *state_generator*: funcție care generează vecinii unei stări;
- *max_iters*: numărul maxim de iterații ale *beam-ului* până la care se încearcă rezolvarea;
- k: dimensiunea *beam-ului*.

1.2 Learning Real-Time A*

Algoritmul, ca și cel precedent, pornește dintr-o stare inițială, și, la fiecare pas, evaluează vecinii stării în care se află și avansează înspre cel de cost minim. Pentru a învăța costurile (și a converge spre costurile reale), se reține într-o tabelă asocieri (*stare* $\rightarrow$ *cost*). Aceasta se populează cu valori când se evaluează o stare nemaiîntâlnită (se introduce costul estimat de euristică) și la fiecare tranziție între stări (fiind cel mai bun cost cunoscut, se consideră că este mai aproape de adevăr). Deoarece, când nu este permis ca jucătorul să tragă cutii, pot exista stări de deadlock (în care jocul este pierdut), dacă, după un număr stabilit de pași, algoritmul nu găsește o stare cu un cost mai bun, acesta repornește din starea inițială (reținând tabela de estimări, pentru a o îmbunătăți mai departe).

Algoritmul poate fi ajustat prin:

- *heuristic*: funcție care estimează costul unei stări;
- *state_generator*: funcție care generează vecinii unei stări;
- *max_iters*: numărul maxim de mișcări din starea inițială până la care se încearcă rezolvarea;
- *max_restarts*: numărul maxim de resetări în starea inițială;
- *steps_before_restart*: numărul de stări vizitate succesiv fără găsirea unui cost mai bun la care se resetează algoritmul în starea inițială.

1.3 Euristica propusă

Funcția de cost aleasă pentru algoritmii de căutare este suma distanțelor de la cutii la pătrățelele țintă. Pentru fiecare cutie, se calculează distanța (în împingeri) către fiecare pătrățel de pe hartă (și, implicit, către fiecare țintă), apoi se asociază fiecărei cutii o țintă, astfel încât suma distanțelor să fie minimă. Împerechearea se face cu ajutorul *Hungarian method*[2], implementată deja în *Python* în biblioteca *scipy* (*scipy.optimize.linear_sum_assignment*).

[1] `https://en.wikipedia.org/wiki/Softmax_function`
[2] `https://en.wikipedia.org/wiki/Hungarian_algorithm`

1.4 Analiza performanțelor

Parametri utilizați:

- Parametri comuni:
  - *heuristic*: Manhattan;
  - *state_generator*: mutări fără trageri;
  - *max_iters*: 200;
- Beam Search:
  - k: 80;
- Learning Real-Time A*:
  - *max_restarts*: 500;
  - *steps_before_restart*: 50;

Rulând algoritmii pe setul de teste furnizat, se observă următoarele tipare:

Timpul de execuție

Timpul de execuție pentru algoritmul Beam Search este semnificativ mai mare decât cel pentru Learning Real-Time A*, datorită nevoii de a evalua vecinii tuturor stărilor din beam la fiecare pas ($lungime\_beam \cdot număr\_vecini \approx k \cdot 4$); LRTA*, pe de altă parte, consideră doar starea curentă și vecinii acesteia. Se poate observa în (1) că LRTA* este de ≈ 3 ori mai rapid decât Beam Search pentru o bună parte din teste.

Numărul de stări vizitate

Din aceleași considerente ca mai sus, numărul de stări vizitate (incluse în beam în cazul Beam Search) este mai mic în cazul LRTA*, deoarece acesta consideră doar starea curentă. Diferența este însă mai mică, mai ales în cazul testelor complexe și cu multe "stări capcană", după cum se poate observa în (2), deoarece LRTA* trebuie să revină în starea inițială pentru a evita *deadlock*-urile.

Optimalitatea soluției

Conform rezultatelor din (3), Beam Search găsește soluții mai scurte decât LRTA*, deoarece cel din urmă poate să se întoarcă în stări anterioare pentru că învață în timp real costurile stărilor. Exploatând acest comportament, putem reporni algoritmul LRTA* de mai multe ori, ținând cont de estimările anterioare de cost, pentru a găsi soluții mai bune, deoarece algoritmul converge către costurile reale. Se poate vedea în (4) că se obțin soluții în general mai bune, dar, pentru anumite teste, nu este suficient, chiar dacă se repornește de 100 de ori!

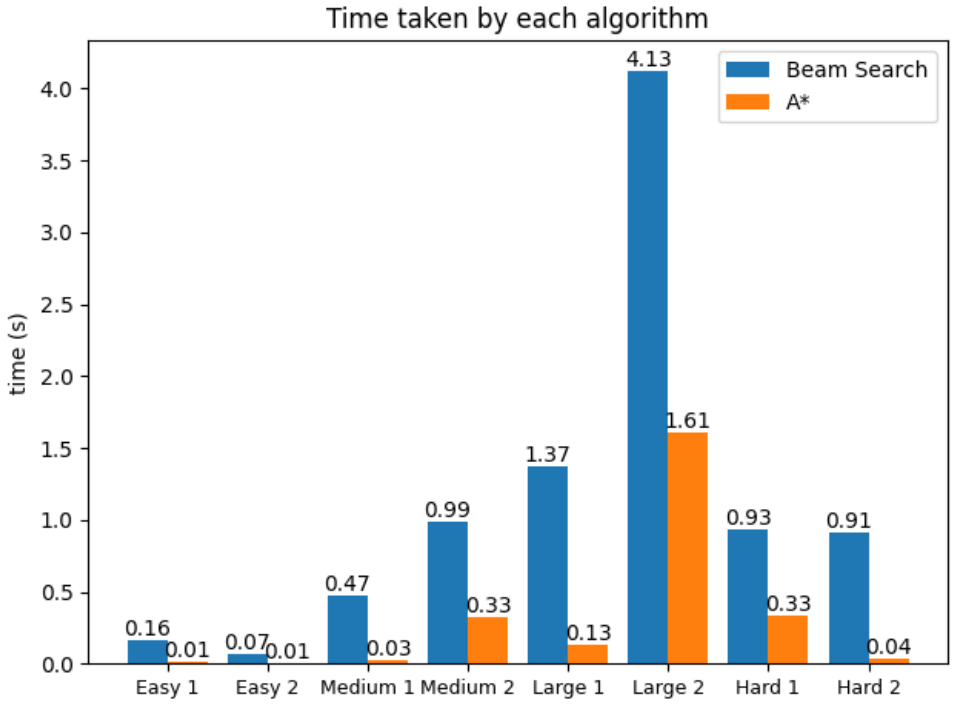

Figura 1: Timpul de execuție al algoritmilor pe diferite teste

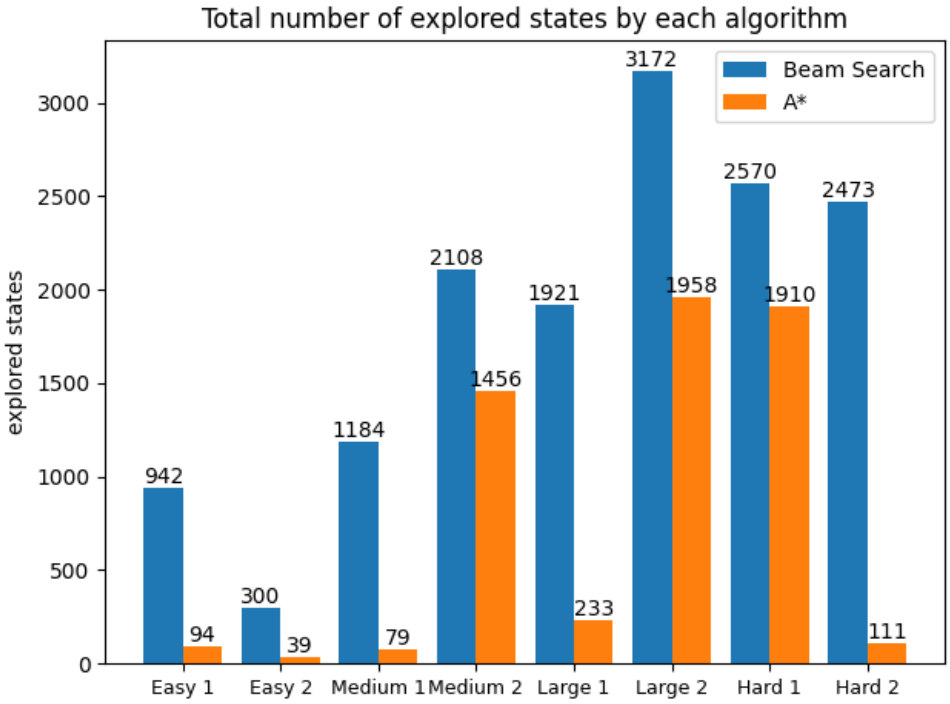

Figura 2: Numărul de stări vizitate de algoritmi în cazul fiecărui test

2 Raționament

2.1 Distanța Manhattan

Prima idee de implementare a fost însumarea distanței Manhattan de la fiecare cutie la cea mai apropiată țintă. Această euristică se calculează foarte ușor și este *consistentă*, dar nu este foarte eficientă, tendința jucătorului fiind să împingă cutiile în direcția țintelor, ignorând existența pereților. Pentru a compensa acest fapt, am mărit *beamul* și am ales o abordare stocastică în alegerea stărilor care să constituie *beam-ul*. Inițial am decis să se aleagă k stări cu o probabilitate proporțională cu costul fiecăreia (de exemplu, având 3 stări cu costurile 1, 2 și 3, acestea să fie alese cu probabilitățile $\frac{3}{6}$, $\frac{2}{6}$, respectiv $\frac{1}{6}$). Această abordare nu a funcționat, deoarece stărilor cu costuri mari le erau atribuite probabilități mult prea mari. Soluția a fost transformarea costurilor în probabilități cu ajutorul funcției *softmax*, care oferă probabilități care scad exponențial cu cât costul este la o distanță mai mare de maximum. Funcția *softmax* folosită (6) a suferit mici modificări față de cea originală, deoarece am dorit să atribui probabilități mari stărilor cu costuri mici, am normalizat valorile x_i în apropierea lui 0 și nu am adăugat un parametru de temperatură (care controlează cât de brusc scad probabilitățile odată cu distanța, comportându-se similar cu cea din algoritmul *Simulated Annealing*).

În parametri aleși la **1.4**, algoritmii rezolvă în medie 6/8 teste.

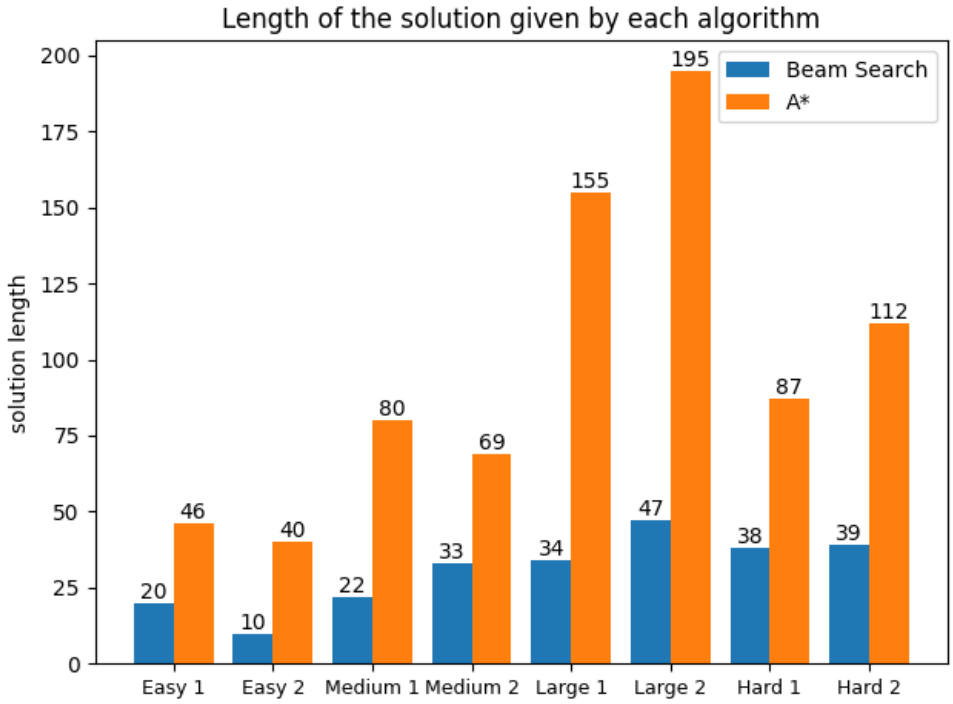

Figura 3: Lungimea soluției (numărul de mișcări) găsite de algoritmi pentru teste

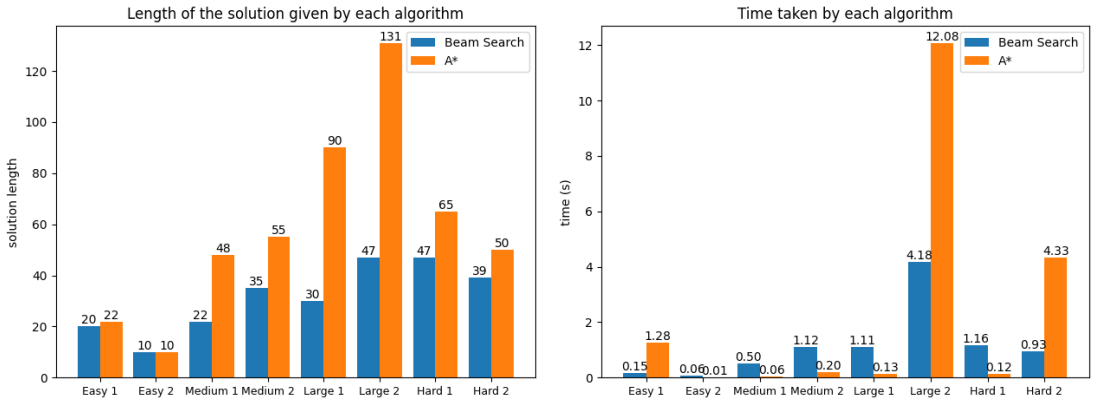

Figura 4: Lungimile soluțiilor și durata execuției pentru Beam Search și LRTA* cu 100 de reporniri după găsirea soluției

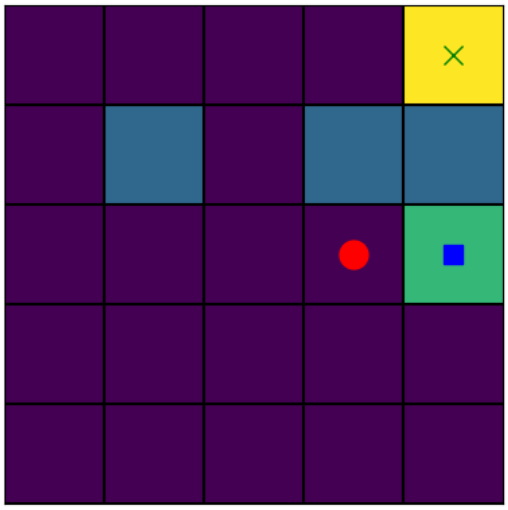

Figura 5: Situație în care Beam Search se blochează într-o stare de minim local, caz în care k trebuie mărit

$$P(x_i) = \frac{e^{x_i - \max(x_k)}}{\sum_{j=1}^{n} e^{x_j - \max(x_k)}}$$

Figura 6: Funcția softmax folosită

2.2 Numărul minim de mutări

Următoarea idee a fost similară cu cea de mai sus, dar cu un calcul mai fidel al distanțelor (în loc de distanță Manhattan, pentru fiecare cutie se calculează numărul de mișcări până la ținte, folosind algoritmul lui Lee[3]), apoi se însumează distanțele pentru fiecare cutie. În plus, am adăugat un termen care să țină cont de distanța jucătorului față de cutiiile care nu sunt pe ținte, dar, pentru un k mic și cu pondere egala cu cea a distanței cutiilor, jucătorul putea ajunge "să se teamă" să împingă cutiile pe ținte, deoarece în acest mod se "îndepărta" de cutie. Acest impediment a putut fi reglat acordând o pondere mai mare distanței cutiilor.

2.3 Corespondența cutie - țintă de cost minim

3 Concluzii

[3] https://www.pbinfo.ro/articole/18589/algoritmul-lui-lee