Inteligență Artificială Tema 1 - Sokoban

Lache Alexandra Florentina Georgiana, 332CD

18.04 - 27.04.2025

1 Descriere

Proiectul de față presupune a unui agent AI în Python care generează o soluție pentru jocul Sokoban. Agentul trebuie să fie realizat folosind următorii algoritmi de căutare în spațiul stărilor: Iterative Deepening A*, respectiv Beam Search.

Una din principalele dificultăți în construirea unui agent care să rezolve jocul de Sokoban provine din factorul mare de ramificare al mutărilor combinat cu adâncimea ridicată a căutării. În varianta studiată, există posibilitatea tragerii cutiilor, asigurând că nu există stări ireversibile. Totuși, mișcările de tragere sunt puțin dezirabile, dorindu-se explorarea de euristici care le minimizează sau chiar le evită.

Scopul acestui document este de a analiza și compara performațele celor doi algoritmi, de a documenta procesul de dezvoltare a soluției și de a evidenția evoluția euristicii utilizate.

2 Jurnal

Obținerea unui timp de execuție tractabil pentru algoritmul IDA* a reprezentat inițial o provocare majoră. Deși a fost primul dintre cei doi algoritmi implementați, din cauza factorului mare de ramificare și a adâncimii considerabile a arborelui de căutare, rezultatele tractabile erau obținute inițial doar pentru primele trei teste.

Ulterior, prin îmbunătățirea funcției euristice — prin introducerea unei constrângeri suplimentare care ținea cont de o permutare optimă a cutiilor — am obținut timpi de execuție tractabili și pentru al patrulea test. Totuși, aceste rezultate erau mult sub așteptări, așa că am început să mă documentez în legătură cu posibilele îmbunătățiri pe care le pot aduce codului.

3 Implementare

Algoritmii implementați de agent sunt IDA* și Beam Search. Amândoi algoritmii se folosesc de euristici pentru a aproxima costul până la starea de final.

Am adăugat următoarele îmbunătățiri pentru ambii algoritmi:

- pruning pentru colțurile care reprezintă un deadlock
- pruning pentru marginile care reprezintă un deadlock
- îmbunătățire performanță calcule folosind funcții specifice din biblioteci
- caching pentru calculul euristicii

Am adăugat următoarea îmbunătățire la IDA* original:

- transposition tables pentru pruning

Am încercat și următoarea optimizare, dar nu am reușit să o implementez eficient:

- calcularea dinamică a pozițiilor de deadlock : în cazul în care o cutie ajunge într-un target aflat într-un colț, e posibil ca numărul de poziții de deadlock să crească considerabil - cum se observă la testul super_hard; ideea pe care am încercat să o implementez consta în recalcularea pozițiilor de deadlock atunci când un copil din arborele de recursie IDA avea mai multe target-uri din colț acoperite decât părintele lui

4 Euristici

Euristica pe care m-am gândit să o abordez are la bază distanța Manhattan. Am început testarea cu o variante din ce în ce mai simpliste, adăugând complexitate. Am ales o euristică care ține cont și de distanța jucătorului, încercând să minimizez numărul de mutări alea acestuia.

Variantele încercate:

- suma dintre distanța Manhattan până la cea mai apropiată cutie și distanța minimă pentru fiecare cutie de la poziția actuală la cel mai apropiat target
- suma dintre distanța Manhattan până la cea mai apropiată cutie și suma minimă a distanțelor pentru fiecare cutie de la poziția actuală la un target astfel încât mai multe cutii să nu se suprapună pe același target (generând toate permutările posibile)
- suma dintre distanța Manhattan până la cea mai apropiată cutie de la poziția actuală a jucătorului și asignarea cutiilor folosind Hungarian Method

5 Rezultate

- Comparație IDA* și Beam Search(fără mutări de pull)

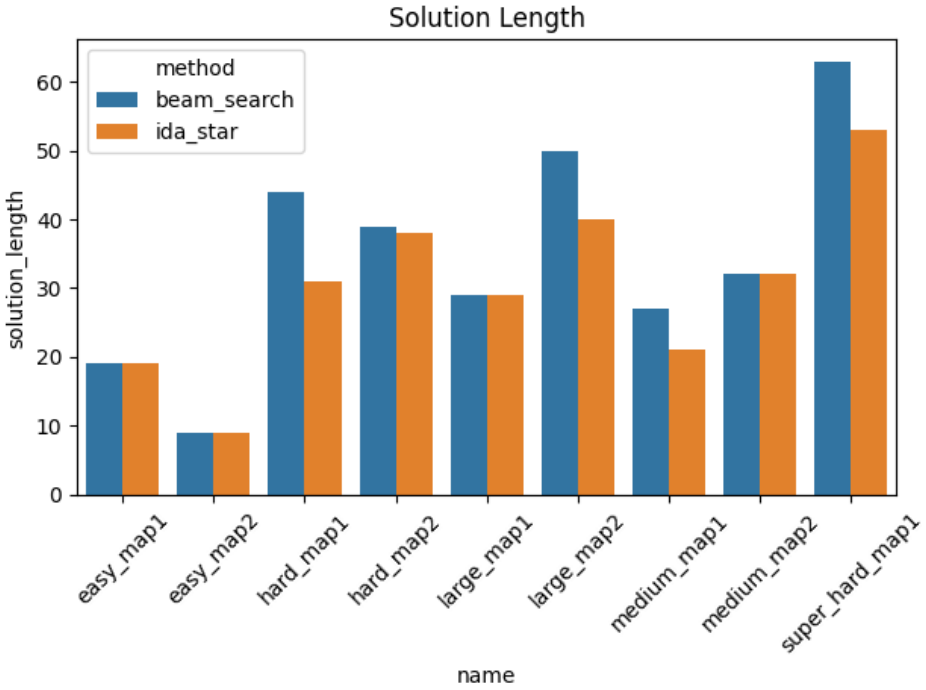

Figure 1: Lungimea soluției

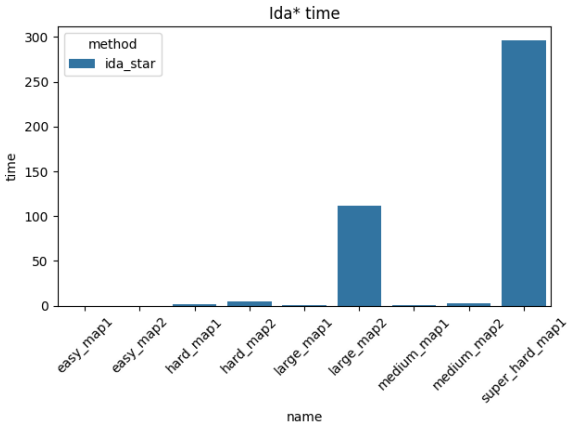

Figure 2: Timp rulare IDA*

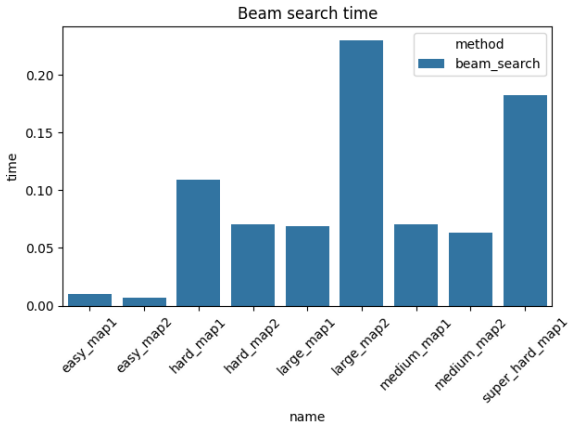

Figure 3: Timp rulare Beam Search

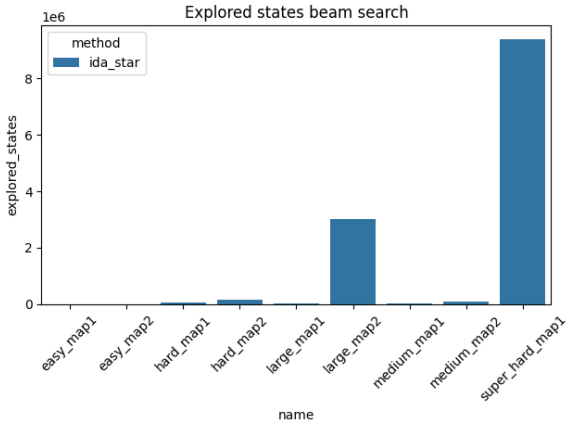

Figure 4: Stări explorate IDA*

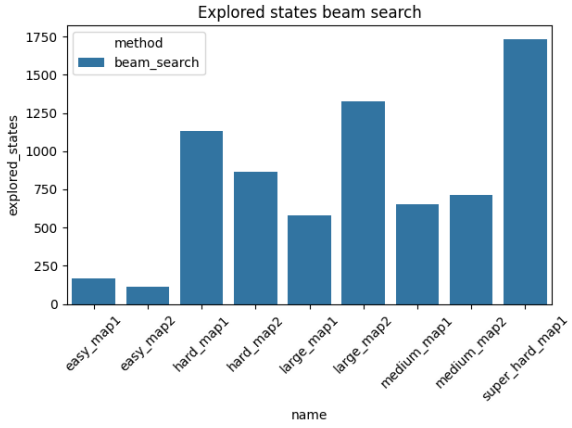

Figure 5: Stări explorate Beam Search

- Evoluție euristică Beam Search și IDA*

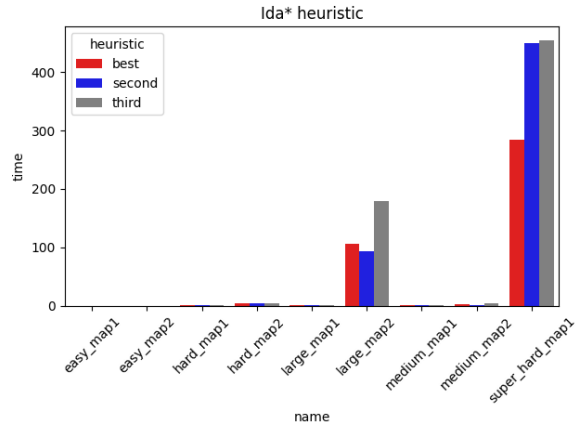

Figure 6: Euristică IDA*

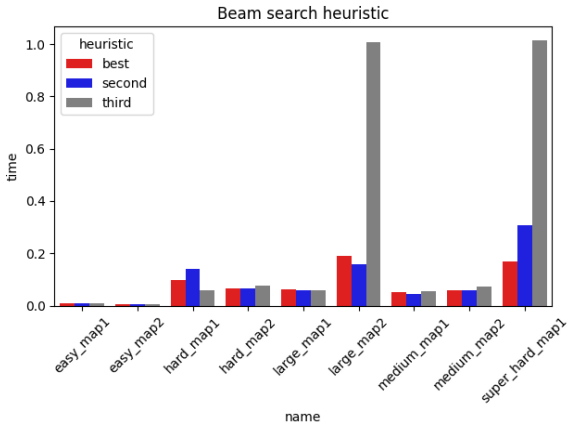

Figure 7: Euristică Beam Search

– Pentru rulările prezentate, sunt prezente celelalte îmbunătățiri(transposition tables, caching

de euristică, precalcularea pozițiilor statice de deadlock).

– Diferența mare pentru a treia euristică la Beam Search este dată de nevoia creșterii beam-ului pentru a ajunge la o soluție validă.

- Îmbunătățire timp odată cu eliminarea pozițiilor de deadlock statice

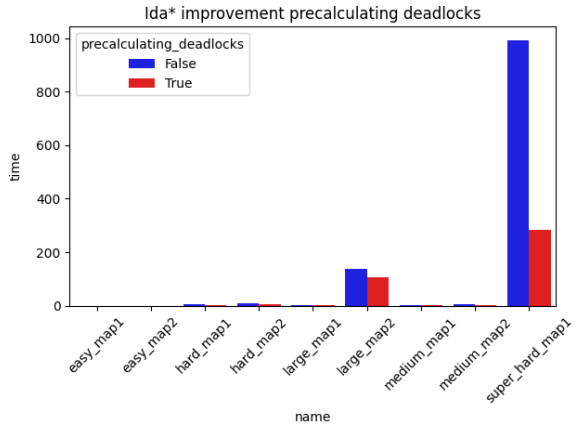

Figure 8: Precalculare deadlock IDA*

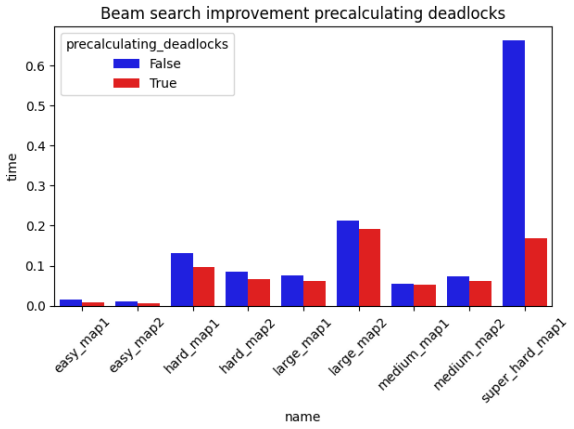

Figure 9: Precalculare deadlock Beam Search

– Pentru anumite configurații ale hărții, în care o cutie se află deja pe margine sau în colț, nu are rost să o luăm pe acea ramură în arborele de recursivitate, întrucât nu putem obține o soluție care nu necesită mișcări de tragere. Am încercat să precalculez o parte din aceste poziții, pentru a îmbunătăți performanța algoritmului prin reducerea numărului de stări explorate.

- Comparație soluții care folosesc pull vs soluții care nu folosesc pull

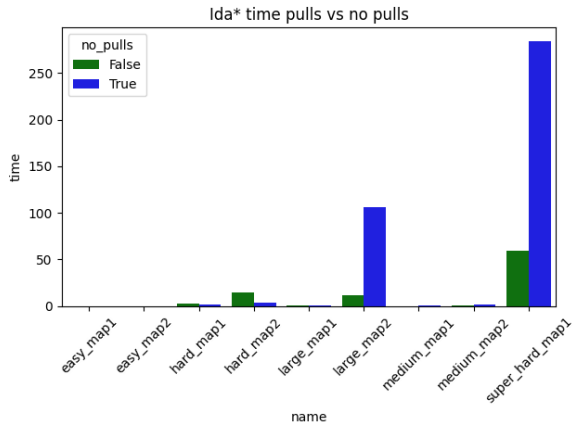

Figure 10: Timp IDA*

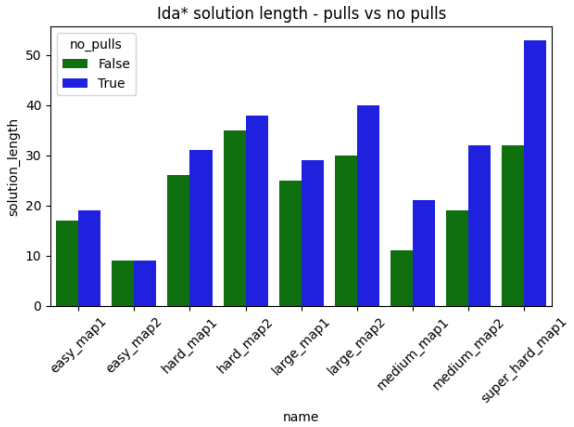

Figure 11: Lungime soluție IDA*

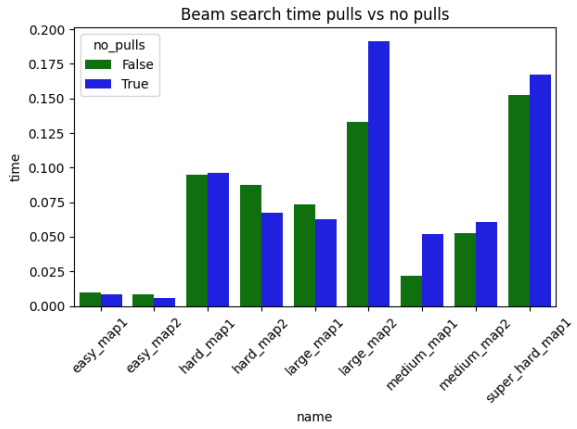

Figure 12: Timp Beam Search

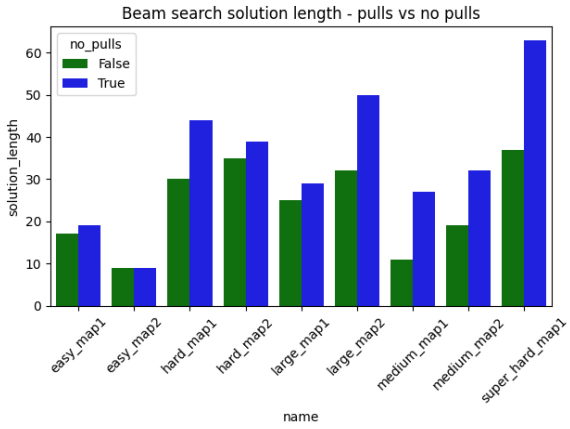

Figure 13: Lungime soluție Beam Search

  - Se observă că atunci când se folosesc și mișcări de tragere, soluțiile au o dimensiune mult mai mică la testele complexe, chiar dacă gradul de ramificare este mai mare.
  - Aceasta este o problemă mai ales la IDA*, întrucât soluția trebuie căutată într-un arbore cu înălțime mai mare.

- Îmbunătățire caching euristică

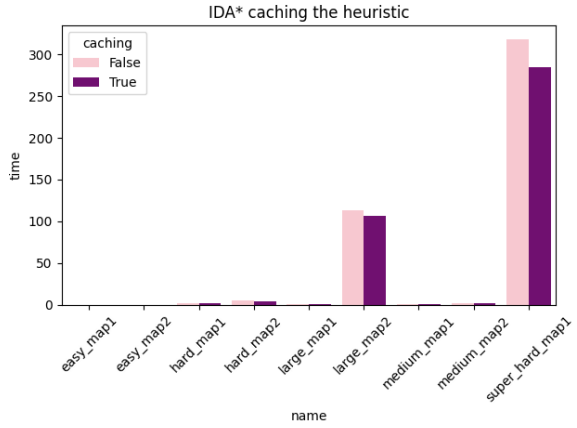

Figure 14: IDA* caching

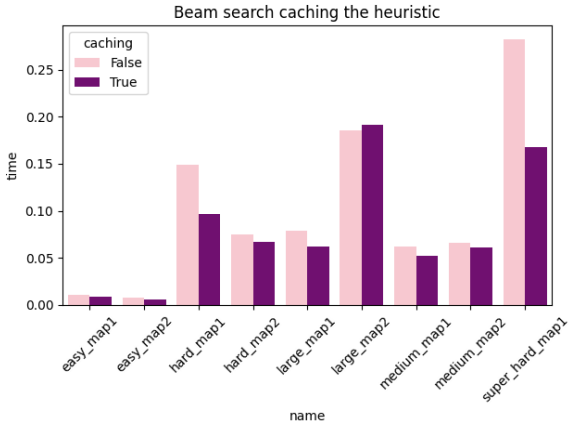

Figure 15: Beam Search caching

  - Ideea de a memora valorile anterioare ale funcției euristice a apărut din dorința de a evita timpul pierdut prin recalcularea repetată a distanței Manhattan către fiecare țintă pentru aceeași poziție, operație realizată frecvent de-a lungul execuției programului.

6 Concluzii

- IDA* excelează la a găsi soluția optimă ca număr de mutări minime. Totuși, recursivitatea poate fi costisitoare, mai ales în cazul unui arbore de căutare adânc. A găsi și testa îmbunătățiri astfel încât timpul de rulare al algoritmului pentru toate testele să fie sub 5 minute a fost o provocare. Din fericire, am reușit acest aspect, obținând 4 min 45 secunde la ultimul test :).
- Mărimea buffer-ului de căutare la Beam Search variază invers proporțional cu eficiența euristicii de a aproxima costul real. Pentru o euristică mai puternică, această dimensiune este mai mică. De exemplu, pentru testul super_hard, pentru a găsi o soluție am avut nevoie de un beam cu dimensiunea 28 folosind cea mai bună euristică, în timp ce pentru cea mai proastă euristică testată am avut nevoie de un beam cu dimensiunea cuprinsă între 150 și 200.

References

[1] A. Junghanns, J. Schaffer, *Sokoban: Enhancing General Single-Agent Search Methods using Domain Knowledge*, 2000.
[2] L. Rei, R. Teixeira, *Willy: A Sokoban Solving Agent*, 2009.
[3] A. Venkatesan, A. Jain, R. Grewal, *AI in Game Playing: Sokoban Solver*, 2018.
[4] N. Sandhu Peters, *Solving Sokoban efficiently: Search tree pruning techniques and other enhancements*, 2023.
[5] https://www.algorithms-and-technologies.com/iterative_deepening_a_star/python/
[6] https://hussainwali.medium.com/simple-implementation-of-beam-search-in-python-64b2d3e2fd7e
[7] https://timallanwheeler.com/blog/2022/01/19/basic-search-algorithms-on-sokoban/
[8] https://github.com/csdankim/MCTS_Sokoban
[9] https://github.com/mpSchrader/gym-sokoban