Manea Andrei Iulian, Grupa 331CB

Tema 1 IA: Sokoban Solver

Implementarea algoritmilor și a euristicilor

Am avut de implementat algoritmii IDA* și Simulated Annealing. Pentru ambii algoritmi, am pornit de la o structură clasică, prezentată la curs.

Algoritm IDA*

```
limita = h(s0), s = s0, p[s0] = ⊥
repeta
   r = DF(s, 0, limita)
   daca r = <s0, ..., sf> atunci intoarce r
   daca r = INSUCCES atunci intoarce r
   limita = r

DF(s, g, limita)
   daca g + h(s) > limita atunci intoarce g + h(s)
   daca s este stare finala atunci intoarce cale(s, p)
   min = INSUCCES
   pentru fiecare sj ∈ succesori(s)
      daca sj ∉ cale(s, p)
         p[sj] = s
         r = DF(sj, g + cost_arc(s, sj), limita)
         daca r = <s0, ..., sf> atunci intoarce r
         daca r ≠ INSUCCES si min = INSUCCES sau r < min atunci min = r
   intoarce min
```

La IDA*, în loc de grafuri simple, am lucrat cu stările Sokoban (hărți, pozițiile cutiilor și ale jucătorului). La fiecare iterație:

- Pragul f = g + h pornește de la euristica stării inițiale și crește treptat la cel mai mic overrun găsit
- DFS limitat (`search`) extinde doar sucesorii cu f ≤ prag şi detectează cicluri printr-un set `path_visited` .
- Am optimizat algoritmul folosind **transposition table** (`cost_so_far`), care memorează pentru fiecare stare hash-uită cel mai bun cost g la care a fost deja vizitată. Dacă se revine la aceeași configurație cu g mai mare sau egal, o prunăm instant, evitând reexpansiuni masive.
- Reconstruirea eficientă a căii prin dicționar `path_dict[g] = move` , fără a copia liste la fiecare pas.

Euristicile pe care le-am folosit sunt:

1. **Manhattan Greedy Safe**
   - Pentru fiecare cutie care nu este poziționată pe o țintă, găsește ținta liberă cea mai apropiată (distanță Manhattan) și o „rezervă” pentru acea cutie.
   - Rezultatul este suma distanțelor Manhattan dintre cutii și ținte, un cost rapid care ajută la prune-ul stărilor inutile.
   - Am implementat **deadlock detection** pentru situațiile `corner_deadlock`, `tunnel_deadlock`, `edge_deadlock` și `square_deadlock` (detalii mai jos).
2. **Exact Matching Cost**
   - Filtrează cutiile deja rezolvate și țintele ocupate, apoi, pentru ≤6 cutii, face brute-force pentru toate permutările asocierii cutie→țintă și ia suma minimă de distanțe Manhattan.
   - Adaugă un termen suplimentar de 0.5 × dist (dintre player și cea mai apropiată cutie) pentru a include și costul deplasării jucătorului.
   - Verifică deadlock-urile menționate anterior.

Inițial, am început prin a folosi o distanță Euclidiană și una Manhattan simplă, însă pe măsură ce harta devenea mai complexă, rularea dura din ce în ce mai mult până în punctul în care pe harta `medium_map2` găsirea soluției a durat undeva la *5 ore*. Schimbarea decisivă a fost adăugarea tabelei de transpoziție `cost_so_far`, care a permis filtrarea stărilor mai eficient. **De menționat** este că am mai testat o euristică bazată pe **Hungarian Assignment**, pentru a asigna eficient asocierea cutie-țintă și a face pruning bun în spațiul de căutare.

Algoritm Simulated Annealing

```
Simulated_Annealing(T_init, T_final, cooling, eval, alege_succesor)
   s = s0; T = T_init
   cat timp T > T_final
      s' = alege_succesor(s)
      daca eval(s') < eval(s)
         atunci s = s'
      altfel
         s = s' cu probabilitate pow(e, -((eval(s') - eval(s)) / T))
      T = cooling * T
```

Am adaptat Simulated Annealing pentru Sokoban astfel:

- Am configurat iterațiile folosind hărțile cu pozițiile cutiilor, obstacolelor și cea a jucătorului.
- Am modificat factorul de cooling `decay_rate` pentru a permite mai multe iterații ale algoritmului.
- Am folosit contorul `modes_expanded` pentru numărarea stărilor expandate și `move_path` pentru a urmări mișcările de pull.

Euristicile pe care le-am folosit sunt:

1. **Hungarian Assignment**
   - Combină asocierea optimă cutie-țintă cu ridicarea la pătrat a distanțelor, ceea ce accentuează diferențele între stări foarte bune și foarte slabe.
   - Penalizarea `undo_moves` descurajează oscilațiile inutile (bucle push-pull) și ajută algoritmul să iasă din minime locale.
2. **Exact Matching Cost**
   - Oferă un bound precis (prin brute-force la câte ≤6 cutii) al costului real de rezolvare.
   - Pe măsură ce temperatura scade, landscape-ul de cost devine din ce în ce mai „neted” și orientat spre soluție, permițând selecții și tranziții mai eficiente spre stări de cost mai mic.

Implementarea de la Simulated Annealing a fost destul de straight-forward, s-a îmbunătățit în momentul în care am adăugat **deadlock detection** prin faptul că s-a dat prune la stări mai eficient.

Deadlock Detection

Funcțiile de deadlock detection pe care le-am implementat sunt:

- `is_corner_deadlock` - Detectează cazurile în care o cutie care nu este plasată pe o ţintă este blocată într-un unghi drept (perete vertical + perete orizontal), fără cale de ieşire.
- `is_tunnel_deadlock` - Identifică cutiile aflate într-un tunel îngust (perete de ambele părţi pe direcţie orizontală) care nu pot ajunge la nicio ţintă din interiorul acelui tunel.
- `is_edge_deadlock` - Marchează ca deadlock cutiile poziţionate pe marginea hărţii care nu au nicio ţintă aliniată pe aceeaşi margine, deci nu pot fi împinse spre destinaţie.
- `is_2x2_deadlock` - Semnalează situaţia în care patru cutii formează un bloc 2×2 complet, fără ca vreo cutie să fie plasată pe o ţintă, blocându-se reciproc.

Adițional, funcția `configure_deadlocks` primeşte lista de tipuri de deadlock-uri active și returnează lista funcțiilor corespunzătoare pentru verificarea fiecărui criteriu, folosită de euristici pentru pruning.

Rulare

Inițial, rulam testele manual, selectând în cod testul dorit. În prezent, am optimizat acest proces prin crearea unor argumente în linia de comandă care să personalizeze rularea unui anumit test cu anumite opțiuni. De exemplu:

1. Dacă vreau să rulez toate testele cu ambii algoritmi și fiecare euristică, rulez comanda

```
> python3 main.py tests/*.yaml --benchmark
```

2. Dacă vreau să rulez doar testul easy_map1 pe algoritmul Simulated Annealing cu euristica Exact Matching Cost fără deadlock detection și să salvez soluția în format GIF, rulez comanda:

```
> python3 main.py tests/easy_map1.yaml --a simulated_annealing --H exact_matching --no-deadlocks
```

3. Dacă vreau să rulez doar testul hard_map1 pe algoritmul IDA* cu euristica Manhattan Greedy Safe cu deadlock detection pentru corner și 2x2, rulez comanda:

```
> python3 main.py tests/hard_map1.yaml --a ida_star --H manhattan --corner --square
```

Comparația celor 2 algoritmi

Timp de execuție

În graficele de la pagina 6 se poate observa cum timpul de rulare este mult mai mare la algoritmul IDA, *deoarece IDA* efectuează o căutare exhaustivă, iterând prag după prag și extinzând adesea zeci sau chiar sute de mii de stări până la găsirea soluţiei, în timp ce Simulated Annealing, prin sampling aleator şi acceptare probabilistică, rămâne de ordinul secundelor chiar şi pe hărţi complexe. Mai mult, observăm că IDA* cu euristica Manhattan Greedy Safe reduce semnificativ timpul faţă de varianta cu Exact Matching Cost, dar tot nu poate concura cu viteza lui SA, care rezolvă aceeaşi hartă în sub 3 secunde atunci când foloseşte aceeaşi euristică de matching.

Număr de stări construite

În graficele de la pagina 7 se poate observa cum numărul de stări expandate este cu mult mai mare la algoritmul Simulated Annealing decât la IDA*, datorită naturii sale stocastice şi lipsei prune-ului agresiv bazat pe prag (f = g + h). În timp ce IDA* taie ferm sub-arborele de căutare ori de câte ori suma cost-euristică depăşeşte pragul curent şi evită deadlock-urile şi ciclurile, Simulated Annealing probează în mod repetat numeroase succesiuni aleatorii de mutări, adesea reintrând în regiuni deja explorate pentru a scăpa din optimum-uri locale.

Număr de mișcări de pull

În graficele de la pagina 8 se poate observa că IDA* folosește mult mai puține mișcări de pull. IDA* sacrifică timpul de calcul (şi un număr moderat de expansiuni) pentru a garanta o soluţie cu puţine pull-uri şi fără back-tracking inutil, în timp ce Simulated Annealing prioritizează viteza de găsire a unei soluţii suficient de bune şi robustetea în faţa deadlock-urilor, dar plăteşte acest lucru prin explorări redundante şi un număr crescut de pull-uri.

Concluzii

În elaborarea acestei teme am explorat două paradigme fundamentale de căutare în spaţiul de stări pentru jocul Sokoban: **exhaustivă** (IDA*) şi **stocastică** (Simulated Annealing). IDA* a demonstrat că, printr-o combinaţie riguroasă de prune pe baza pragului f = g + h, transposition table şi detectare de deadlock-uri, poate găsi soluţii cu un număr minim de „pull”-uri, garantând calitatea optimă a traiectoriei. În schimb, Simulated Annealing, deşi nu oferă garanţia optimului global, s-a dovedit mult mai rapid pe hărţi complexe, reuşind să scape din capcanele locale prin acceptarea controlată a mutărilor cu cost mai mare.

Prin compararea celor două algoritmi pe seturi variate de teste, am constatat că:

- **IDA*** excelează la minimizarea numărului de back-tracking-uri şi la obţinerea de soluţii robuste, însă plăteşte acest avantaj printr-un număr ridicat de expansiuni şi timp de execuţie substanţial.
- **Simulated Annealing**, deşi produce de multe ori mai multe „pull”-uri şi explorări redundante, este capabil să găsească rapid soluţii rezonabile, fiind potrivit când timpul de răspuns este critic.

Bibliografie

https://algorithmsinsight.wordpress.com/graph-theory-2/ida-star-algorithm-in-general/

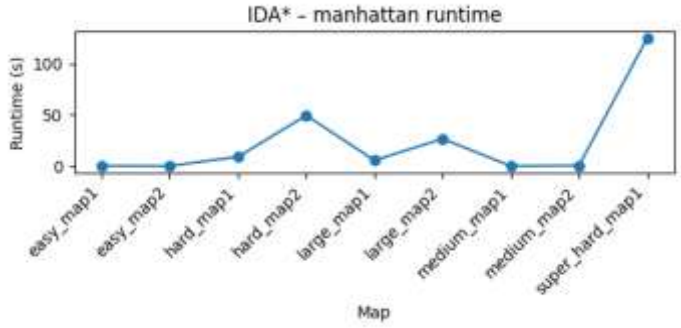
https://python.plainenglish.io/hungarian-algorithm-introduction-python-implementation-93e7c0890e15

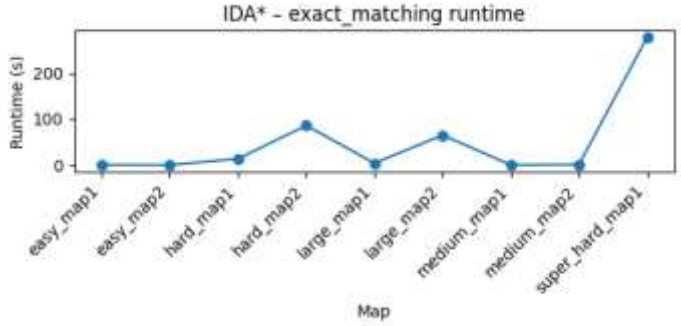
https://doc.neuro.tu-berlin.de/bachelor/2023-BA-NiklasPeters.pdf

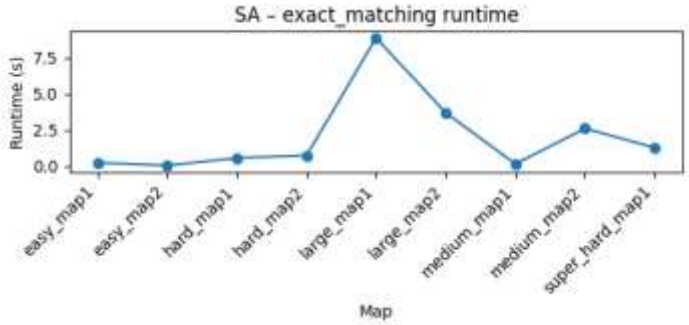
https://verificationglasses.wordpress.com/2021/01/17/a-star-sokoban-planning/

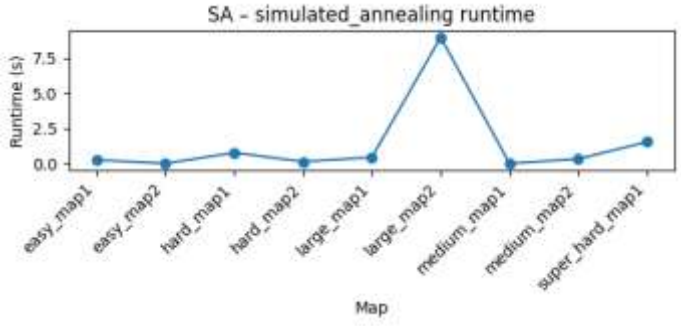
https://www.youtube.com/watch?v=BUHc8p5Mpdo

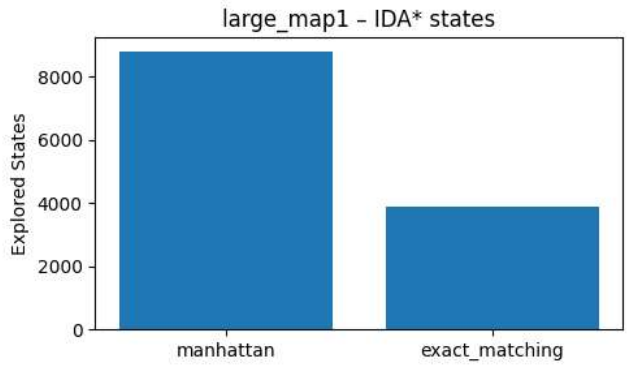
https://curs.upb.ro/2024/course/view.php?id=10346#section-6

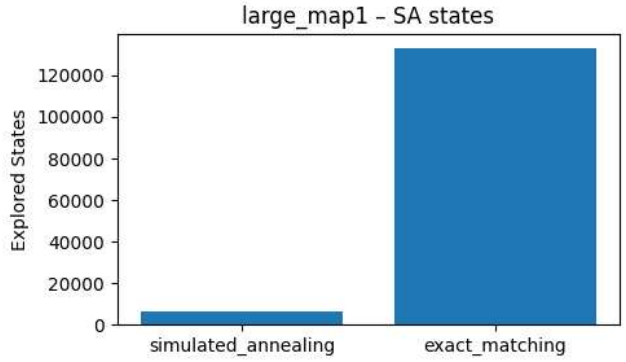

IDA* – manhattan runtime
Runtime (s)
100
50
0
easy_map1
easy_map2
hard_map1
hard_map2
large_map1
large_map2
medium_map1
medium_map2
super_hard_map1
Map

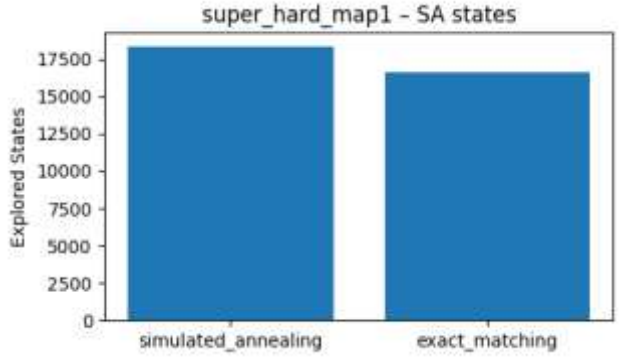

IDA* – exact_matching runtime
Runtime (s)
200
100
0
easy_map1
easy_map2
hard_map1
hard_map2
large_map1
large_map2
medium_map1
medium_map2
super_hard_map1
Map

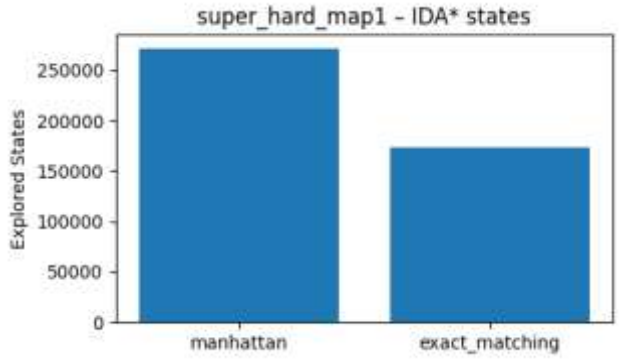

SA – exact_matching runtime
Runtime (s)
7.5
5.0
2.5
0.0
easy_map1
easy_map2
hard_map1
hard_map2
large_map1
large_map2
medium_map1
medium_map2
super_hard_map1
Map

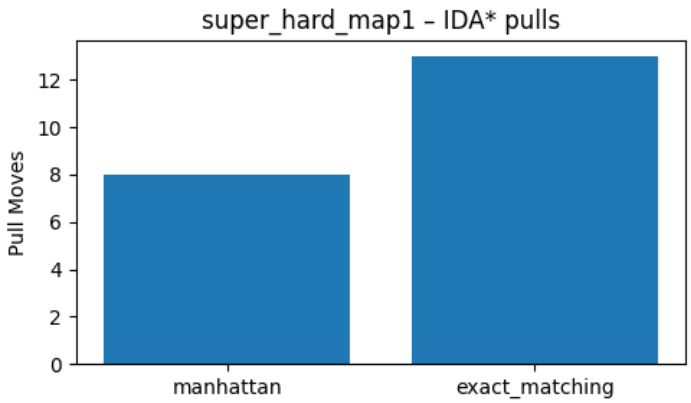

SA – simulated_annealing runtime
Runtime (s)
7.5
5.0
2.5
0.0
easy_map1
easy_map2
hard_map1
hard_map2
large_map1
large_map2
medium_map1
medium_map2
super_hard_map1
Map

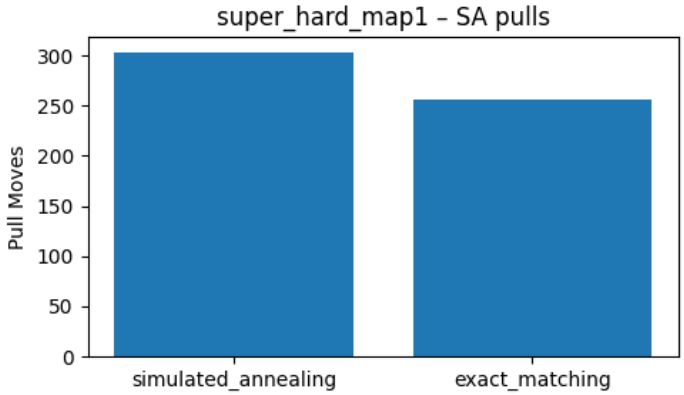

large_map1 – IDA* states
Explored States
8000
6000
4000
2000
0
manhattan
exact_matching

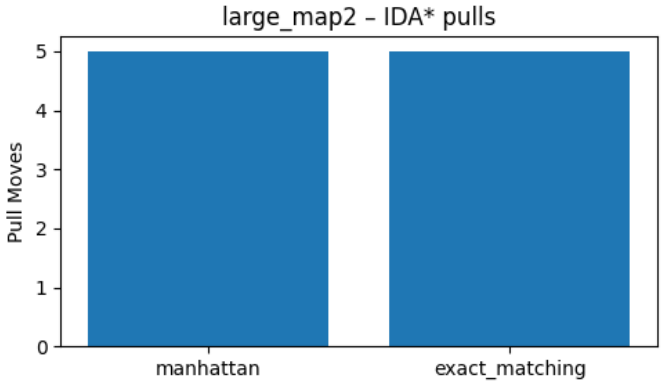

large_map1 – SA states
Explored States
120000
100000
80000
60000
40000
20000
0
simulated_annealing
exact_matching

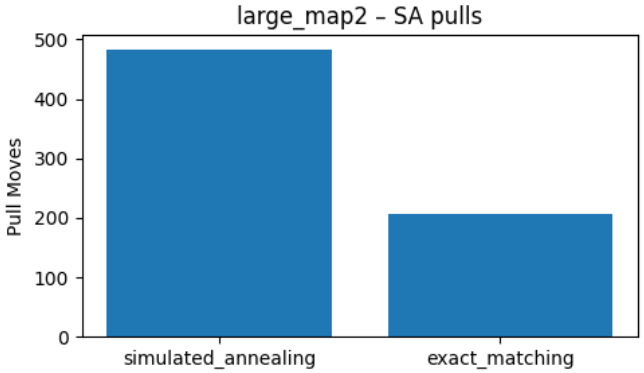

super_hard_map1 – SA states
Explored States
17500
15000
12500
10000
7500
5000
2500
0
simulated_annealing
exact_matching


super_hard_map1 – IDA* states
Explored States
250000
200000
150000
100000
50000
0
manhattan
exact_matching


super_hard_map1 – IDA* pulls
Pull Moves
0
2
4
6
8
10
12
manhattan
exact_matching


super_hard_map1 – SA pulls
Pull Moves
0
50
100
150
200
250
300
simulated_annealing
exact_matching


large_map2 – IDA* pulls
Pull Moves
0
1
2
3
4
5
manhattan
exact_matching


large_map2 – SA pulls
Pull Moves
0
100
200
300
400
500
simulated_annealing
exact_matching