Readme

Reprezentare cale de rezolvare

În cadrul acestei teme, am avut de implementat jocul Sokoban. Am implementat cei 2 algoritmi ceruţi (Simulated Annealing şi LRTA) fără a folosi mişcări de pull. Soluţia finală la care ajung are următoarea codificare:
-litere mari: L (left),R (right), U (up) şi D (down) pentru cazul în care jucătorul se deplasează într-o direcţie împingând cutia
-litere mici: l, r, u, d pentru cazul în care jucătorul se deplasează fără a folosi cutia.

Astfel, de exemplu, pentru testul simulated annealing, am următorul output:
ddlddrUUUUUUruLLLddRDDurUluurrdL.
Ma deplasez sub cutia (1,3), o împing până la poziţia (7,3), apoi de la (7,3) de 3 ori left (L) pe targetul (7,0). Imping apoi cutia (5,2) la dreapta şi cutia (4,2) 2 poziţii în jos (DD). Cutia (4,2) ajunge pe targetul (2,2), iar boxul care a plecat din (5,2) ajunge în (5,3), apoi îl împing în sus spre (6,3). Playerul se duce în dreapta cutiei (6,4) pentru a împinge ultimul box pe targetul final (6,2).

Soluţia nu este afişată doar ca output la consolă. Creez o fereastră folosind biblioteca Tkinter pusă la dispoziţie de python. În această fereastră, am două butoane prev şi next, pentru a derula pozele din gif-ul creat la final. GIF-ul l-am realizat folosind gif.py pus la dispoziţie în scheletul temei. Astfel, pot să şi văd cum evoluează rezolvarea jocului de către cei 2 algoritmi.

Reprezentare stări

Stările le am reprezentate similar cu cele din map.py. Folosesc Map.from_yaml din map.py pentru a reprezenta stări. Astfel:
- / este reprezentat ca zid la afişare, 1 în forma de matrice
- _ este considerat spaţiu gol, reprezentare cu 0 pe matrice
- B este considerată cutia, notată cu 2 pe matrice
- X reprezintă targetul final, notat cu 3 în matrice
- P este playerul, notat cu 4

Am observat un mic bug la reprezentarea pe stări. Atunci când toate targeturile sunt ocupate (nu neapărat de cutii), codul din gif.py consideră harta rezolvată, chiar dacă de fapt nu este. Ideea este că pentru testul medium_map2, ajung cutiile (1,3) şi (4,2) pe targeturile (7,0) şi (2,2), iar cutia (5,2) trebuie să ajungă la (6,2). Astfel, ultima cutie are calea [(5, 2) (5, 3) (6, 3) (6, 2)]. Problema apare când cutie apare pe (6,3) şi playerul trece prin (6,2), pentru a se poziţiona în dreapta cutiei pentru a o împinge în stânga. Astfel, targeturile (7,0) şi (2,2) sunt acoperite, iar targetul (6,2) este acoperit de cutie şi pare rezolvat dpdv al graficii, deşi în realitate mai are playerul de parcurs 5 paşi.

Idee Simulated Annealing

Iniţial, am avut ideea la simulated annealing să precalculez rutele de la target la destinaţie. Astfel, pentru testele de easy am aplicat iniţial doar un BFS între target şi destinaţie. Am constat că calea oferită de BFS nu era bună, deoarece cutia ajungea pe o margine, iar traseul putea să treacă prin colţuri.

Astfel, am adăugat funcţia is_box_stuck, care avea rolul de a vedea dacă cutia rămâne blocată. Blocajele le-am penalizat foarte rău, cu scor 1.000.000. De la început nu am dorit să folosesc pull-uri, deoarece nu acesta era scopul jocului. Am reuşit astfel la simulated annealing să îmi treacă primele 2 teste. Am denumit funcţia mea de cost pentru o stare energy(state:Map, cai, explored_state_count). Caile sunt cele precalculate de annealing.

Pentru următoarele teste (medium1, hard2 şi large1), am încercat să generez o matrice pătratică, de dimensiune nr_cutii * nr_cutii, în care reţineam căile de la fiecare cutie la fiecare target. Problema a apărut că ajungeam să am o cale pentru o cutie care se suprapunea cu o altă cale. Bfs-ul clasic nu mai făcea faţă, deoarece îmi dădea o cale care conţinea blocaje.
Astfel, am creat funcţia este_cale_push_only(test_map:Map, cale), care pe harta iniţială verifică dacă o cale este validă (dacă la fiecare pas din hartă pot ajunge cu jucătorul în spatele cutiei pentru a o împinge în direcţia aleasă). Folosesc şi funcţia auxiliară este_push_valid(start_x, start_y, end_x, end_y). Aceasta are rolul de a verifica că pasul respectiv din cale este valid. Astfel, noul meu bfs explorează până când găseşte o cale care să respecte toate aceste condiţii. Asocierea box-target o făceam folosind algoritmul Hungarian. Acest algoritm căuta să îmi dea o asociere cutii-targeturi optimă. (Adică căuta să am distanţa minimă per total.) În matricea mea de path-uri, dacă între un target şi o ţintă nu găseşte bfs-ul absolut nici o cale care să respecte este_push_valid, dădeam ca distanţă o valoare mare, default (9999). În acest punct al problemei, îmi treceau 5 teste.

Testul large_map2 reprezenta o problemă, deoarece îmi bloca cutiile între ele în partea de sus. Problema principală se datora faptului că toate căile precalculate treceau prin partea de sus a hărţii spre ţinte. Astfel, apărea un blocaj mare. Funcţia is_box_stuck nu era făcută pentru cutii. Am încercat să o fac şi pentru cutii, însă am constat că îmi găsea blocaje pe medium_1 şi pe large_1, când cutiile ajungeau să fie lipite. Cutiile lipite nu mai puteau fi scoase din acea poziţie, deoarece în spate aveam spaţii goale înconjurate de pereţi (inaccesibile din cauza modului în care erau aşezate cutiile). În acest punct, dacă făceam 2 mişcări de pull, rezolvam harta uşor, însă am încercat pe cât posibil ca rezolvarea să nu conţină mişcări de pull

Ideea salvatoare a fost să creez un dicţionar de coordonate des folosite pe căi. Astfel, prima cale nu are niciun fel de penalizări. Dacă a doua cale se intersectează în vreo coordonată cu prima cale, penalizez cu 5. (valoare standard). Astfel, BFS_ul meu caută să găsească calea box_target fără a suprapune prea mult căile. Cu noile căi, testul a ajuns să treacă consistent.

După ce am creat un bfs funcţional, am creat cu aceleaşi idei şi o euristică de astar, care şi ea ţine cont de obstacole, căi valide prin push şi penalizări de la suprapunerea căilor.
În această etapă, îmi treceau 6 teste pentru annealing.

Testul medium_map1 dădea fail din simplul motiv că precalculam căile, iar algoritmul hungarian dădea greş. Problema cu algoritmul Hungarian era că îmi căuta suma lungimilor căilor să fie minimă. Astfel, îmi găsea potrivirea (1,3) -> (2,2), (4,2) -> (6,2) şi (5,2),(7,0). Această potrivire avea cale 11, dar era nepractică în realitate. Trebuia să împing atât cutia de pe targetul (4,2) în sus, cât şi cea de pe (5,2). Ducea corect cutia (5,2) -> (7,0), însă pentru (4,2) ieşea de pe cale. Am adăugat o recalculare de cale, folosind fie bfs_full, fie a_star_full (cele 2 euristici ale mele), însă calea (1,3) -> (2,2) era invalidă. Problema era că bfs_full calcula calea (1,3),(2,3),(2,2). Calea era considerată validă, deoarece ca să împing cutia de pe (2,3) pe (2,2), aveam liber (2,4), chiar dacă eu nu puteam ajunge niciodată cu jucătorul pe acea poziţie din cauza cutiei. Astfel, am decis să înlocuiesc algoritmul Hungarian cu brute force. Am luat toate permutările între cutii şi ţinte pentru a găsi o cale care să mă ducă la soluţie, chiar dacă este mai lungă. Timpul de rulare a crescut semnificativ pentru testul medium_map2. Restul testelor nu au fost afectate ca timp, deoarece găseam soluţie la prima permutare. (Pentru testele easy, exista doar o singură permutare posibilă). Timpii de

rulare nu au crescut semnificativ, deoarece dacă am foarte multe cutii, în teorie timpul devine factorial. Însă în practică, pentru fiecare permutare fac lista de căi. Dacă euristicile mele (bfs_full sau astar_full) returnează +inf pe distanţă, ştiu că nu am cum să ajung de la o cale la un target, deoarece este blocat. Eu practic pot sa invalidez o permutare în timpul O(nr_cutii), deoarece doar mă uit în matricea de cost la cutie şi la targetul asociat de permutare. Dacă costul este foarte mare (9999), atunci e clar că permutarea este invalidă. Din nefericire, pentru testul medium_map2, toate permutările erau valide. Pentru hard_map1, doar 2 permutări sunt valide. Deci, pentru un test cu multe blocaje, vor fi explorate puţine permutări. Pentru un test cu harta mai largă (mai permisivă) şi multe cutii, va găsi probabil soluţie din puţine permutări. => timpi de rulare buni

La ultimul test, hard_map1, nu funcţiona algoritmul Hungarian corect . De aceea, am păstrat ideea de permutări şi pentru acest test. Problema la acest test era că boxul de pe poziţia (2,5) ajungea la ţintă înainte ca boxul (5,4) să plece de pe poziţie. Cutia (5,4) se bloca aiurea, iar algoritmul doar plimba playerul peste tot fără sa mai poată face nimic.

Ideea mea pentru a soluţiona această problemă a fost să văd dacă am cutiile pe targeturi, mai puţin una, să văd dacă o pot împinge. Am o funcţie clone_state_with_boxes, care are rolul de a îmi clona harta de nr de boxuri. Apoi pe fiecare hartă păstrează, pe rând, câte o cutie pe poziţie şi restul pe targeturi. Apelez funcţia este_cutie_blocata(pos, test_map:Map, cale_cutie, cai). În această funcţie, verific dacă pot ajunge în spatele cutiei (calculez o direcţie generală), folosind bfs_simple. Aplic bfs_simple pe mapele clonate din funcţia clone_state_with_boxes(state,cai,poz). Astfel, determin ce cutie trebuie să mut ultima oară. De exemplu, pentru testul hard_map1, obţin cutia (2,5) că trebuie să rămână pe loc până la final.

Funcţia simulated_annealing:

În această funcţie, am ca parametri:

- initial_map,
-tfinal=1
-alpha=0.02
-max_iter=1000000
-initial_temp=1000
-cooling_rate=0.9995

Am observat că pot să îi dau un comportament mai de lrta, dacă scad temperatura iniţială şi cea finală mult şi scad şi alpha (tfinal=0.001, alpha=0.01, max_iter=1000000, initial_temp=1, cooling_rate=0.9995) -> comportament aproape similar cu lrta, rată de succes mai mare -> creştere de la 30% la 60% rată de succes pe hard_map1.

În cadrul algoritmului, înainte de iteraţii şi de tot, generez toate căile de la boxuri la destinaţie în matrice. Trec prin toate permutările cu algoritmul simulated_annealing. Dacă există căi invalide în permutare, elimin acea alegere din prima.

Apoi iterez cât timp nu am ajuns la nr maxim de iteraţii şi temperatura iniţială mai mare decât cea finală. Dacă harta este rezolvată (toate cutiile se află pe targeturi), returnez calea de stări care a dus la rezultat şi numărul de stări explorate.

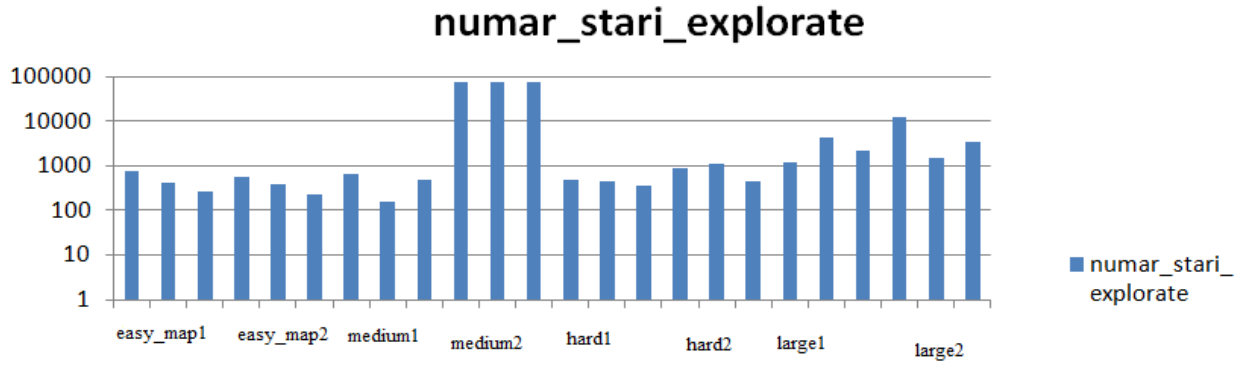
Altfel, dacă nu mă aflu în starea finală, explorez vecinii. Pentru toţi vecinii, calculez energia lor, cu funcţia energy. Calculez probabilităţile folosind funcţia softmax din laborator, apoi aleg aleator un vecin cu probabilitatea dată (dacă un vecin are energie mult mai mică decât restul, va avea şanse semnificativ mai mari să fie ales). Dacă noua stare are o energie mai bună, o salvez în mutări_bune şi, cu o anumită probabilitate, voi trece în starea următoare (şansă de 100% dacă noua stare are valoare mai mică dpdv energetic).

Dacă la final nu găsesc soluţie, afişez un mesaj şi returnez un rezultat parţial.

Optimizări aduse la simulated annealing:
-pot alege să caut căi cu bfs (euristică mai ineficientă ca timp, dar ajunge la acelaşi rezultat) sau astar, care explorează mai inteligent căile. La astar folosesc ca metrică de distanţă Manhattan.
-recalculez de la început drumurile -> matricea de drumuri. Astfel, voi avea mult mai puţin de calculat în cadrul algortimului de simulated annealing.
-permutări între cutii şi ţinte -> dacă o cale este invalidă, invalidez din start permutarea => optimizare foarte utilă dpdv computaţional
-factor de relaxare -> relax_factor. Pot la început să am o abordare mai nepermisivă (pentru a nu bloca cutii de la început) şi mai greedy mai încolo
-funcţii pentru cutii blocate -> dacă blochez cutia la următoarea mutare, este clar că nu mai pot să merg mai departe (reduc mult spaţiul de căutare)
-la final, când optimizez calea, păstrez doar stările unice.
-annealing funcţionează cu o rată de 100% pe testele uşoare (easy_map1 şi 2, medium_map1, large_map1, hard_map2), cu o rată de 75-80% pe large_map2 şi cu o rată de 35-40 % pe medium_map2 şi hard_map1.

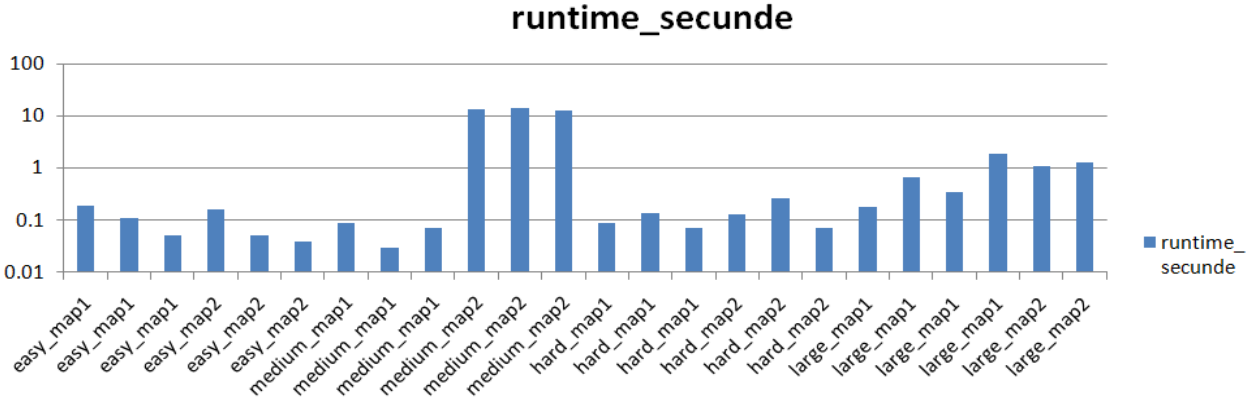
Grafice Simulated_Annealing

Mai jos, este graficul cu numărul de stări explorate pentru fiecare test cu euristica bfs. Din punct de vedere al stărilor explorate, Astar este la fel de bun. Astar este mai eficient dpdv al returnării rezultatului (durează rularea mai puţin timp cu acelaşi număr de stări explorate).

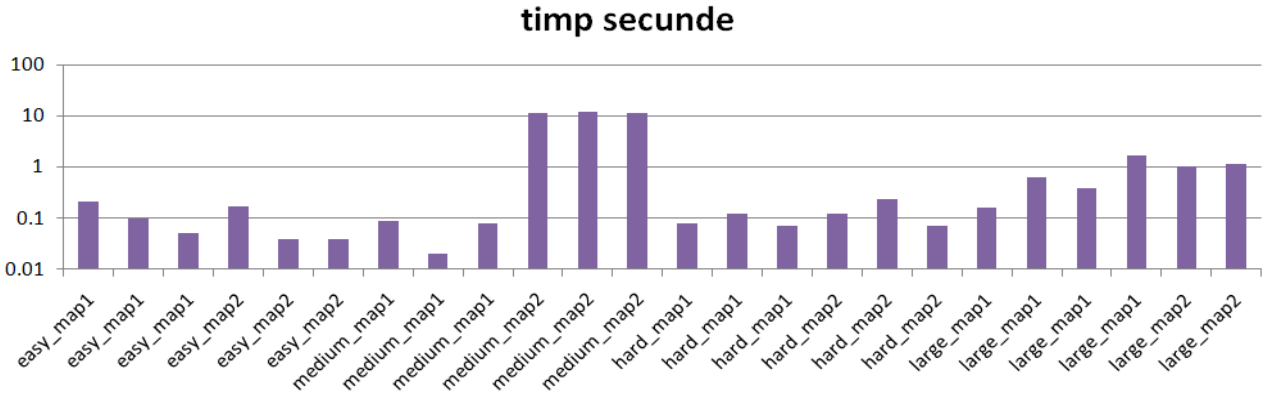


Deoarece Simulated Annealing are componente stocastice, am realizat câte 3 teste pentru fiecare test dat în tests.py. Astfel, apar teste care nu sunt rezolvate corect. Testele hard_map1 şi medium_map2 sunt cele mai problematice.

Mai jos, există un grafic cu timpii pentru bfs pe simulated annealing.

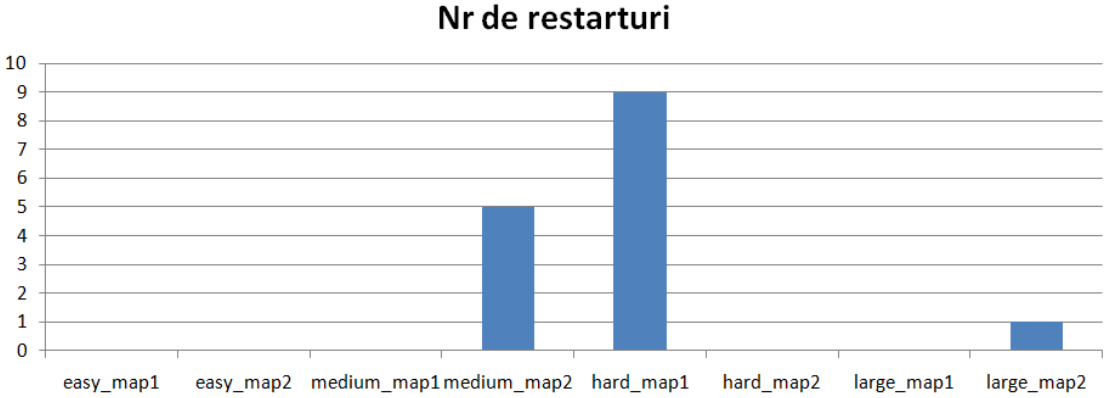


După cum se poate observa, timpii par a fi direct proporţionali cu numărul de stări explorate. Sunt explorate în medie 5600 de stări în fiecare secundă, împreună cu toate procesările aferente. Mai jos, voi pune un grafic cu timpii pentru euristica cu astar:



După cum se observă între cele 2 grafice, timpii folosind euristica astar sunt cu 7-8% mai buni faţă de bfs. Pentru testele mici, nu se observă o diferenţă semnificativă. Cea mai mare diferenţă este pe testul medium_map2, care se reduce de la 13.46 la 11.58 secunde. (Reducere de 13.96%).

Având în vedere că am realizat un simulated annealing fără pull-uri, îmi pun întrebarea câte restarturi am realizat. Răspunsul este că nr. de restarturi este egal cu numărul de permutări explorate. Un restart este foarte costisitor, totuşi pentru o permutare greşită îmi dau repede seama că nu există soluţie (bfs şi astar returnează o distanţă infinită, deci ştiu că nu există căi valide până la cutii). Mai jos, voi avea un grafic cu numărul de restarturi folosit:



Am restart pe testul medium_map2 (găseşte permutarea 2,1,0) ca fiind validă. Există 6 permutări pentru acest test şi abia la ultima găseşte soluţie. În schimb, la hard_map1, din 24 de permutări, găseşte soluţie după a 8-a, ceea ce este mai bine.
La large_map2, uneori găseşte din prima permutare, alteori face un restart. Există cazuri rare când face 2 sau chiar 3 restarturi.

Idee Lrta

Algoritm LRTA efectiv:

În cadrul algoritmului de lrta, am folosit un dicţionar H, unde reţin valorile returnate de euristică (A*,BFS sau manhattan). Pentru fiecare stare care nu este în dicţionar o adaug. Caut vecinii care au cel mai mic scor (un scor de 0 înseamnă că toate cutiile sunt pe poziţie). Pentru fiecare vecin, calculez funcţia de cost, salvez starea în dicţionar împreună cu scorul.

Caut să minimizez scorul, iar în situaţie de tiebreaker, mă duc spre cea mai apropiată stare de cutie. Algoritmul are 3 posibile căi de a se încheia:

-funcţia isSolved returnează True şi salvez calea, pe care o voi da mai departe spre optimizare

-algoritmul ajunge cu iteraţiile la final fără a găsi soluţie

-algoritmul se blochează într-o stare în care, orice ar face, scorul devine +inf. (situaţie rară, dar se poate întâmpla).

Pentru LRTA, am folosit ca euristici pentru funcţia mea bfs, a_star, manhattan şi smart_manhattan. Bfs_distance calculează distanţa de la target la ţintă. A_Star face acelaşi lucru un pic mai eficient. Manhattan este cel mai simplist, doar face distanţa absolută în formă de L. Smart_manhattan ia în calcul şi obstacole şi le penalizează. A_star şi Bfs fac ambele cam acelaşi lucru şi obţin cam acelaşi număr de paşi. Diferenţa este la timp, unde A_star este cu aproximativ 20% mai eficient. Si Manhattan este destul de eficient, însă pică testele large_map1, large_map2 şi easy_map1. Motivul pentru care pică aceste teste este că între target şi destinaţie există obstacole (nu există o cale directă decât ocolind acel obstacol). Astar şi BFS lucrează exact la fel dpdv al stărilor expandate. Diferenţa este doar de timp.

La lrta, am început prin a îmi crea o funcţie care se uită dacă este cutia blocată (is_deadlocked_box). Funcţia aceasta am gândit-o progresiv să devină complexă. Aici, eu practic testez dacă cutia este prinsă în colţuri. Fac un bfs prin care verific că jucătorul are acces la acea cutie (adică poate ajunge la ea). De asemenea, verific dacă cutia se află pe marginea hărţii. Dacă cutia este pe o margine a hărţii şi are targetul ei asociat pe aceeaşi margine, fără obstacole între, atunci returneaza False (cutia nu este blocată). Funcţia is_obstacle_between face parte din testarea cutiei pe marginea hărţii.

În euristică, fac un for pe cutii şi aplic euristica (iniţial aveam un manhattan care trecea pe 4 teste, mai apoi 5), însă manhattan nu este o soluţie viabilă dacă am obstacole.
În forul din funcţia heuristic, aplic funcţia de cost (manhattan -> 5 teste, astar şi bfs -> 8 teste). Aplic euristica aleasă pentru a afla distanţa dintre cutie şi target. Dacă cutia nu este în target, aplic un nou bfs2 (doar distanţa) între locul unde ar trebui să vină playerul (în spatele cutiei) şi targetul final. Dacă nu pot ajunge în spatele cutiei, înseamnă că mutarea nu este bună. Get_general_direction este funcţia care îmi returnează poziţia pe care trebuie să o ia playerul faţă de cutie. Am făcut aceste funcpoziţia pe care trebuie să o ia playerul faţă de cutie. Am făcut aceste funcţii, deoarece nu îmi treceau testele pe medium_2 şi hard_1.
În cadrul funcţiei heuristic, returnez scorul total de la fiecare box la fiecare target alocat.

Deoarece am făcut algoritmul fără niciun fel de pull, iar algoritmul Hungarian nu dădea rezultate bune pentru alocare de targeturi pe testele medium2 şi hard1, am folosit toate permutările cutie-target, ca şi la Simulated Annealing. Până la calcul de targeturi pe bază de permutări, îmi treceau testele easy, medium1, hard2 şi testele large. Motivul pentru care cred că nu treceau şi medium2 şi hard1 este că pe toate testele pe care rula corect, hărţile erau oarecum largi. La easy, aveam doar o cutie şi faptul că aveam funcţie de blocare cutie bine făcută îmi garanta că voi ajunge la destinaţie. Celelalte teste treceau, deoarece aveam multe căi până la destinaţie. Astfel, puteam duce cutiile prin căi diferite fără să le blochez. De asemenea, nu conta ce cutie duceam la ce target, mereu găseam o cale validă.

Testul medium_2 a trecut uşor după ce am implementat sistemul de permutări. Durează mult acest test (6.5s), deoarece găseşte toate permutările valide, dar permutarea cu soluţia (2,1,0) este ultima luată. Deci face câte 1000 de paşi pe primele 5, iar la ultima găseşte soluţie în 351 de paşi, ducând la un total de 5351).

La testul hard_1, m-am lovit de aceeaşi problemă ca şi la annealing: muta cutia de pe (2,5) pe (6,5) foarte devreme, blocând cutia (5,4). Funcţia no_boxes_around are rolul de a îmi garanta că o cutie nu are alte cutii în jur. Dacă găsesc o cutie la care nu pot ajunge şi are cutii

în jur, înseamnă că este blocată de acea cutie. Deci returnez +inf în euristică. Am observat apoi că algoritmul meu îşi dă seama că cutia (5,4) va ajunge să fie blocată de (2,5).

Am observat-o la lrta este că se plimbă foarte mult pe hartă fără să deplaseze cutii. O optimizare foarte importantă pe care am realizat-o a fost să adaug în algoritmul de lrta un tiebreaker, adică ce se întâmplă dacă întâlnesc 2 sau mai mulţi vecini buni (cu acelaşi scor). Dacă am 2 vecini cu acelaşi scor, folosesc funcţia closest_box_distance(state: Map), care îmi caută cea mai apropiată cutie. Astfel, voi aproape în permanenţă aproape de cutii. Acest lucru reduce foarte mult numărul de paşi realizaţi de algoritm în căutarea unei soluţii. Dacă înainte stătea mai bine de 5 secunde pentru o singură permutare a testului medium_map2, acum îi ia în jur de 6.5 secunde cu totul, explorând 5351 de stări. Cea mai notabilă diferenţă a fost pe testul large_map2, care şi-a redus numărul de paşi de la 952 la 196.

Pentru testul hard_1, deşi în aparenţă sunt multe stări de explorat (1000 stări/permutare * 24 permutări), în realitate nu găseşte căi valide de la cutia de pe poziţia (1,1) decât pe targetul (1,0). Astfel, din start rămân doar 3 cutii de asociat cu 6 posibile variante. Algoritmul lrta are avantajul că pentru o permutare greşită, în medie, dă fail rapid.
Algoritmul îmi găseşte soluţii în timp tractabil (sub 14 secunde) pe toate hărţile (mai puţin super_hard_map1).

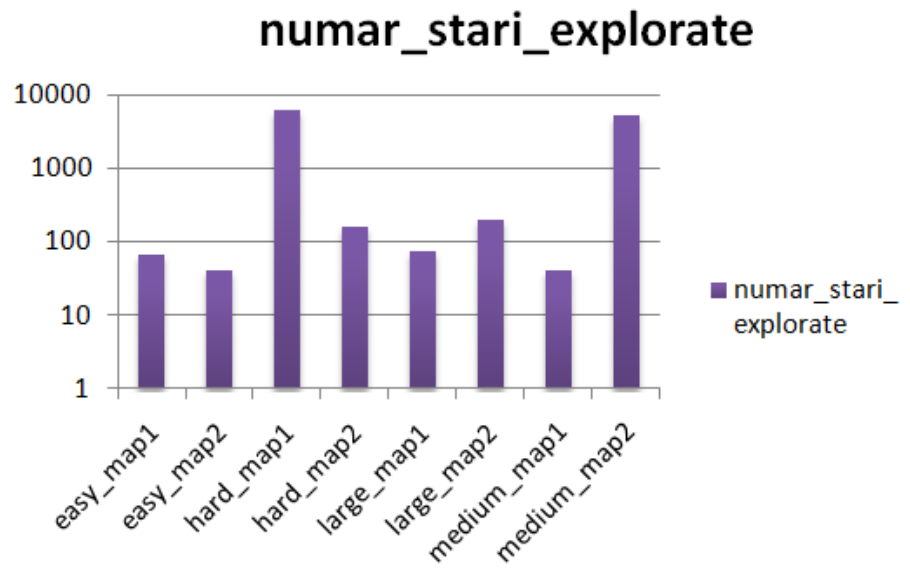
Funcţia interfaţă_grafică are rolul de a pune în evidenţă rezolvarea obţinută pe acel test. Folosesc gif.py pentru a salva atât imaginile, cât şi gif-ul complet. Pe ecran, apare o fereastră cu 2 butoane (prev şi next). Astfel, pot vedea cum rezolvă algoritmii mei jocul de sokoban. Mă folosesc de ceea ce am în schelet.

O altă optimizare importantă pe care am făcut-o are de-a face cu optimizarea căii finale. La simulated_annealing, primesc doar lista cu stările care aduc o îmbunătăţire. Eu practic trebuie doar să aplic un bfs pe player între poziţia din stare curentă şi starea următoare. Astfel, obţin o cale de o lungime acceptabilă.

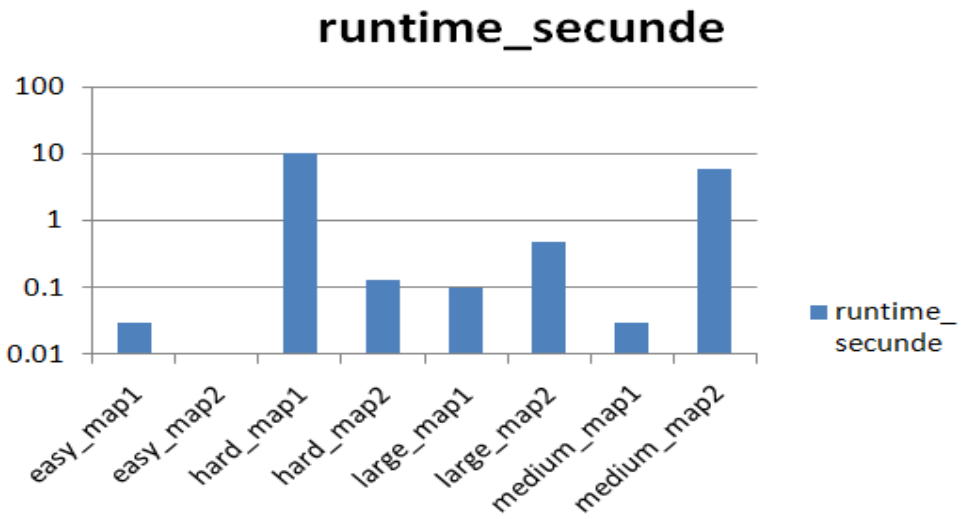
La LRTA, însă, nu funcţionează aşa. Chiar dacă îmi returnează o cale de lungime doar 196, este nepractic să am 196 de poze. Astfel, am încercat pe cât posibil să optimizez calea.
Ideea din spate este că mă uit la stările importante (cele care plimbă cutii). La lrta, salvez mereu în cale starea prin care trec pentru a putea returna o listă completă de stări la final.
Mă uit de fiecare dată când mişc o cutie. Mă uit la starea anterioară în care mişc o cutie. Pot aplica astar sau bfs pentru a afla calea. (este de regulă o cale mult mai scurtă decât ceea ce imi returnează algoritmul). Astfel, păstrând stările importante, completez între ele cu stările în care se deplasează playerul efectiv fără a împinge cutii. Creez o cale mult mai scurtă decât calea iniţială:

Nume test	Cale originală	Cale optimizată	Reducere nr. paşi
Easy_map1	65	19	70.76 %
Easy_map2	41	9	78.04 %
Medium_map1	41	21	48.7 %
Medium_map2	351	44	87.46 %
Hard_map1	105	38	63.8 %
Hard_map2	156	35	77.5 %
Large_map1	73	35	52.05 %
Large_map2	196	54	72.44 %

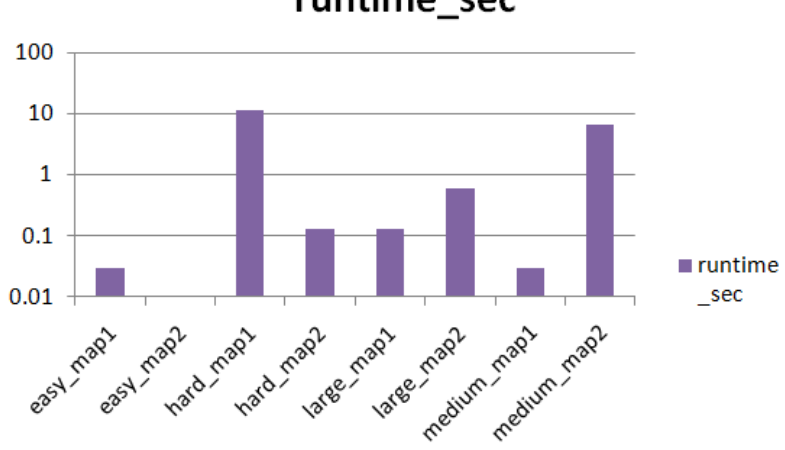
Grafic pentru număr de stări explorate cu LRTA realizat cu euristică astar



Grafic pentru timpul de rulare cu LRTA realizat cu euristică astar

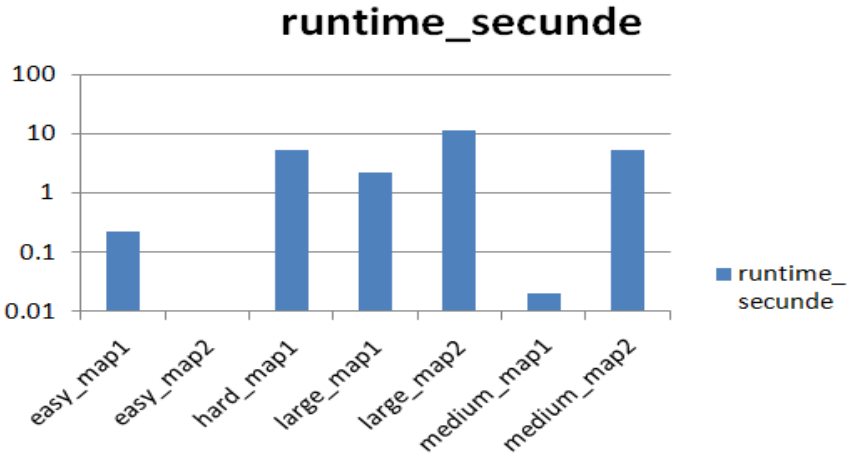


Grafic pentru timpul de rulare cu LRTA realizat cu euristică bfs
(ia un pic mai mult timp faţă de astar, cu 15-20%)

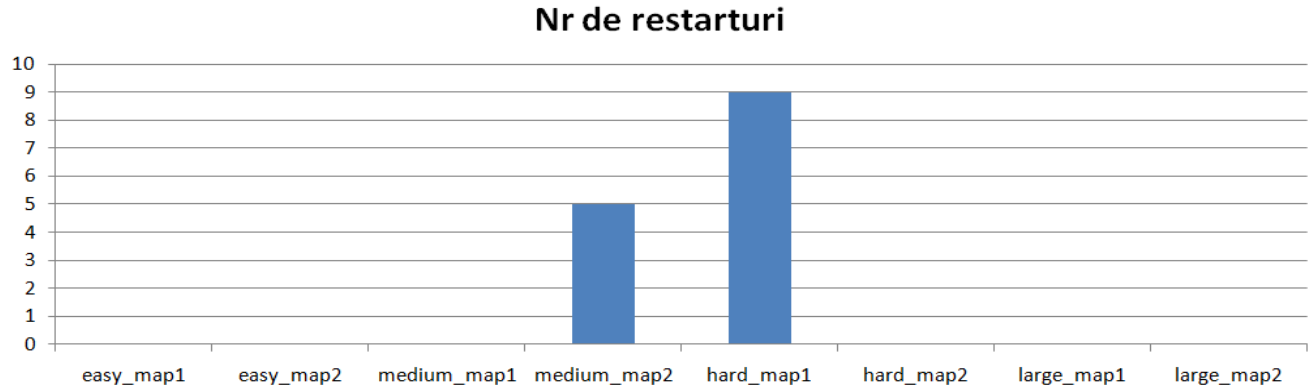


Grafic pentru manhattan

De menţionat că rulează până se opreşte fără soluţie pe testele easy_map1, large_map1 şi large_map2. Pe easy_map1 face 1000 de paşi, iar pe hard_map1 24000.



Deoarece la lrta nu am mişcări de pull, voi numără de câte ori dă algoritmul restart (o metrică destul de utilă). Mai jos, se află graficul cu nr. de restarturi pentru fiecare test în parte:



Spre deosebire de annealing, găseşte soluţie din prima permutare la large_map2. Pentru medium1 şi hard2, se comportă similar cu annealing dpdv al restarturilor. Singura diferenţă este că lrta îmi găseşte soluţii consistente pe toate testele (fără super_hard1). Annealing nu are rată de 100% pe toate testele. Acest lucru se datorează componentei stocastice (în algoritmul de annealing, îmi aleg următoarea stare aleator pe baza unor probabilităţi). Orice alegere are o probabilitate, deci se pot alege şi soluţii slabe local.

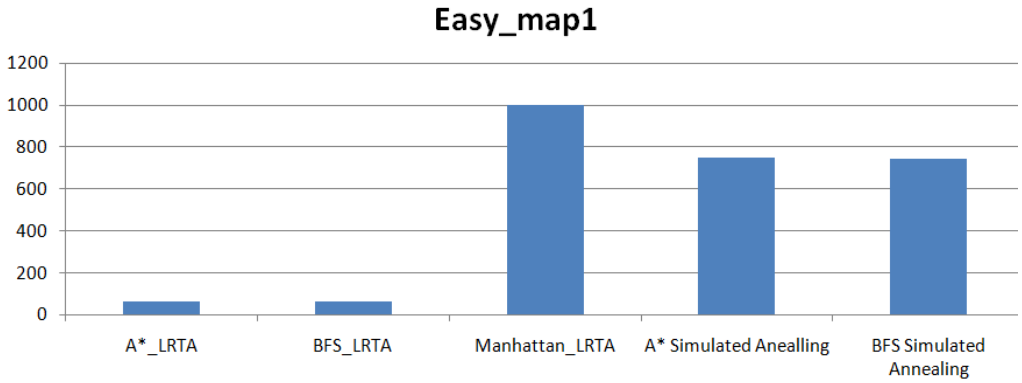
Comparaţie Simulated Annealing vs LRTA

Simulated Annealing găseşte soluţia optimă în timp. Am ales factorul de răcire la 0.995 (aproape 1) pentru a avea o şansă mai mare ca algoritmul să conveargă spre soluţia optimă. (nu e obligatoriu să o şi găsească). Din acest punct de vedere, LRTA obţine o soluţie neoptimă într-un timp mai scurt. Simulated Annealing este mai lent pentru teste mai mari şi obţine o soluţie pe teste cu o probabilitate variabilă. LRTA este mai rapid şi, chiar dacă nu obţine o soluţie optimă, ajunge la un rezultat corect care rezolvă problema (pe teste are rată de 100% succes), pe când SA are rată de succes de 35-40% pe testele medium2 şi hard1.

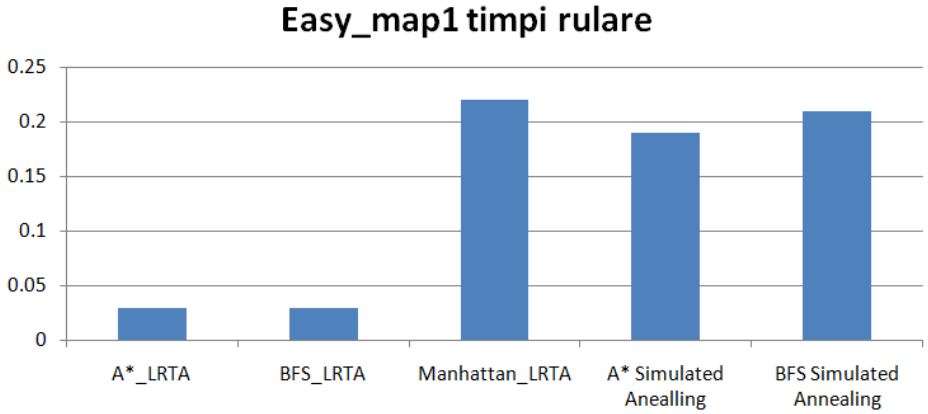
LRTA este stabil şi adaptativ pe măsură ce descoperă harta, pe când celălalt algoritm este foarte sensibil la parametrii primiţi (t_iniţial, t_final, cooling rate).

Per total, LRTA lucrează mai bine, deoarece găseşte soluţia cu o probabilitate de 100% pe testele date şi are timpi de rulare mai scurţi. Dacă testele ar fi fost şi mai mari (hărţi 20x20), ar fi existat o diferenţă mult mai mare dpdv a timpului între cei 2 algoritmi (în favoarea LRTA). Mai departe, voi realiza grafice cu toate euristicile de pe fiecare algoritm pentru a vedea diferenţa de stări explorate şi de timpi pe fiecare teste.

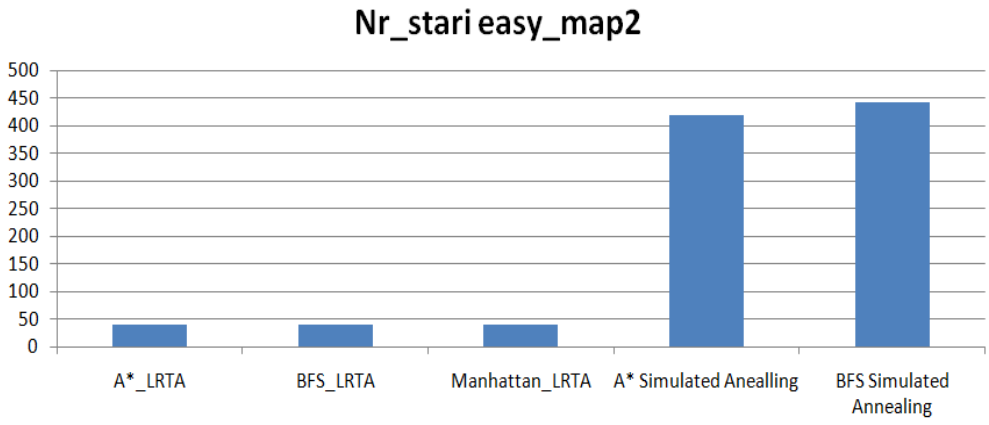
Teste easy

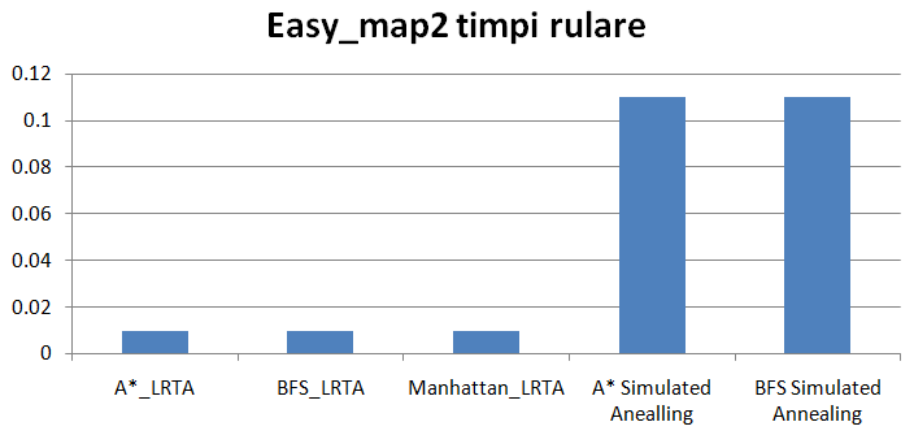


Se observă că LRTA rezolvă rapid cu A* şi BFS, pe când Manhattan nu găseşte soluţie din cauza obstacolelor. Este o diferenţă mare de eficienţă între LRTA (65 de stări) şi Simulated Annealing (743-750) de stări.



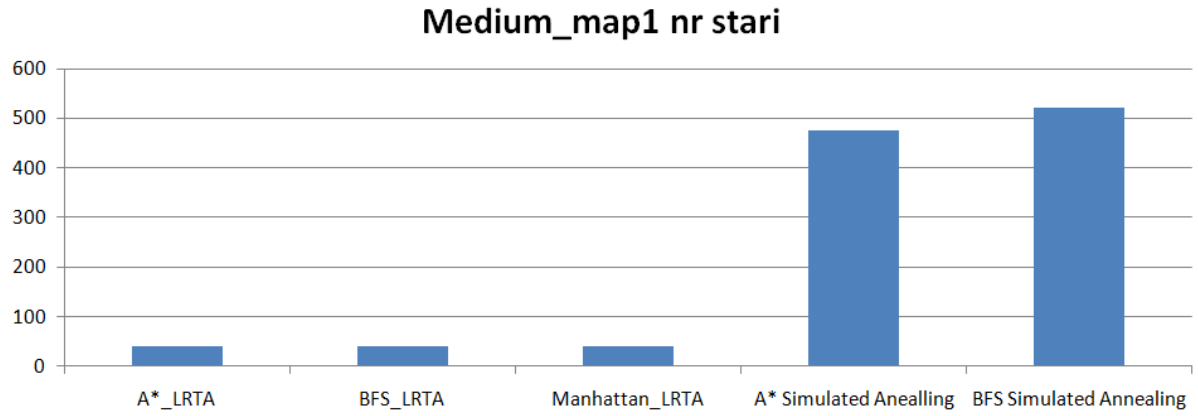
Timpii de rulare sunt şi ei direct proporţionali cu numărul de stări explorate. De menţionat că Manhattan lrta atinge limita de paşi fără a găsi soluţia.

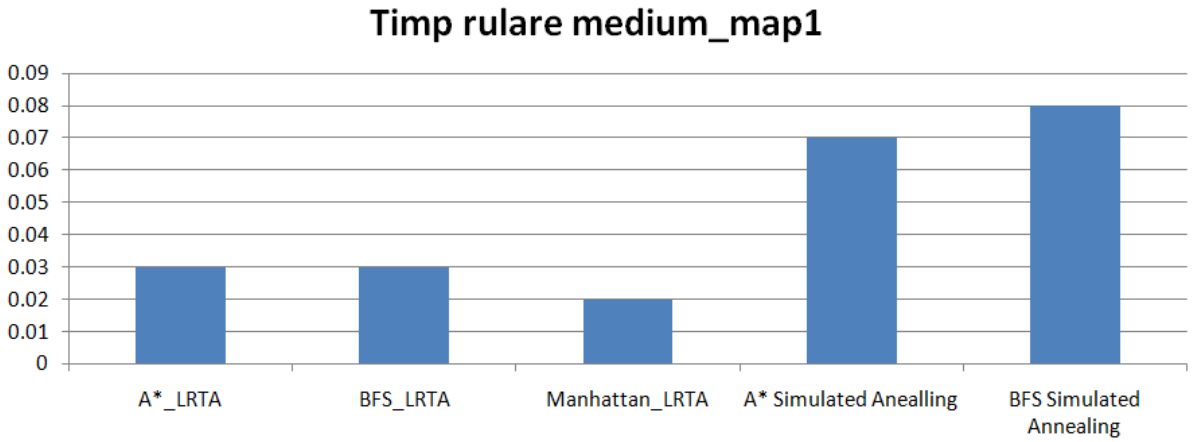




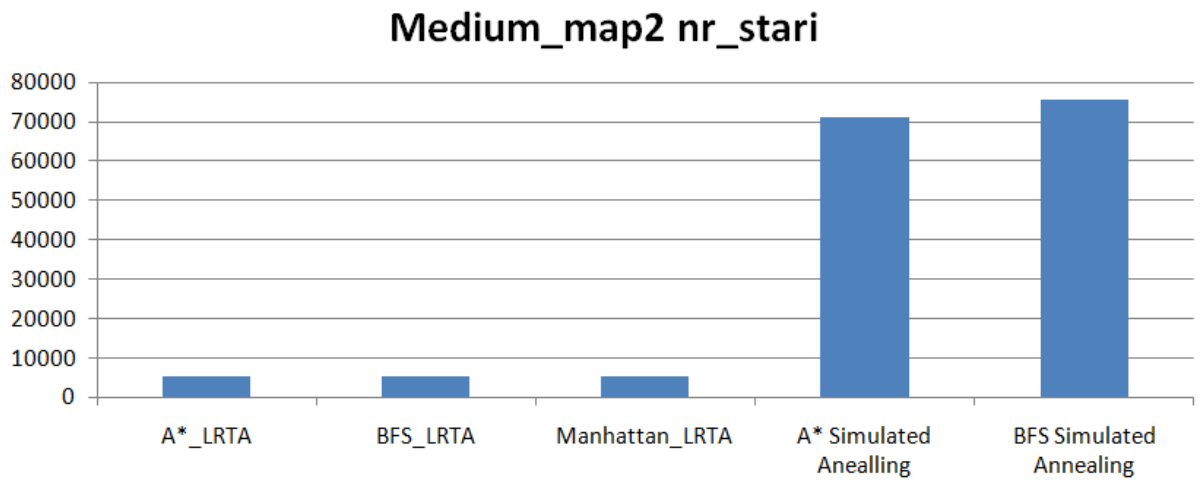
Testul easy_map2 este cel mai simplu dintre toate. Poate fi rezolvat uşor şi cu Manhattan, deoarece există cale directă de la cutie la destinaţie. Se observă faptul că A* obţine foarte rapid soluţia (41 de paşi şi 0.01 secunde), pe când la Simulated Annealing dureaza 0.11 secunde, cu 420, respectiv 442 stări explorate.

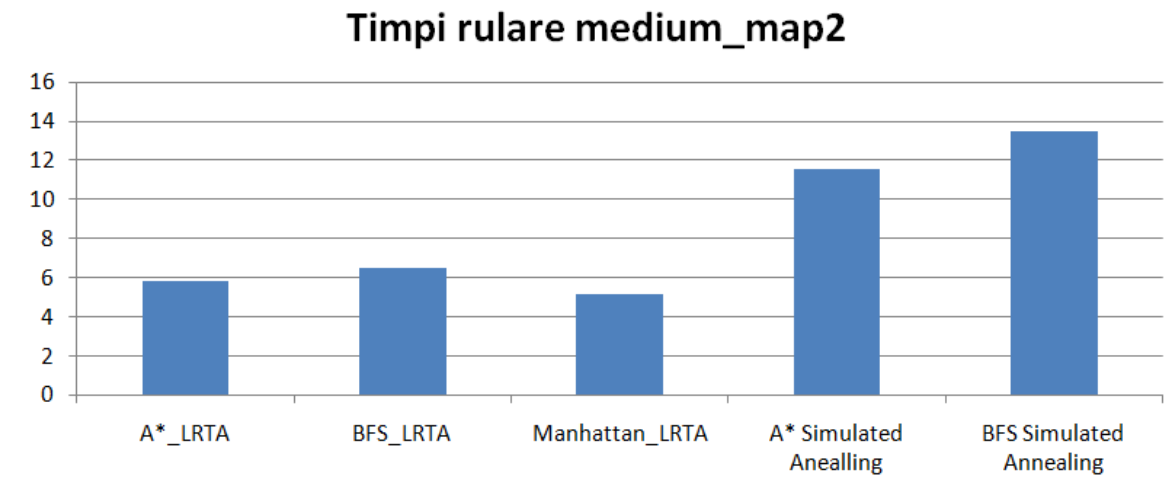
Teste medium





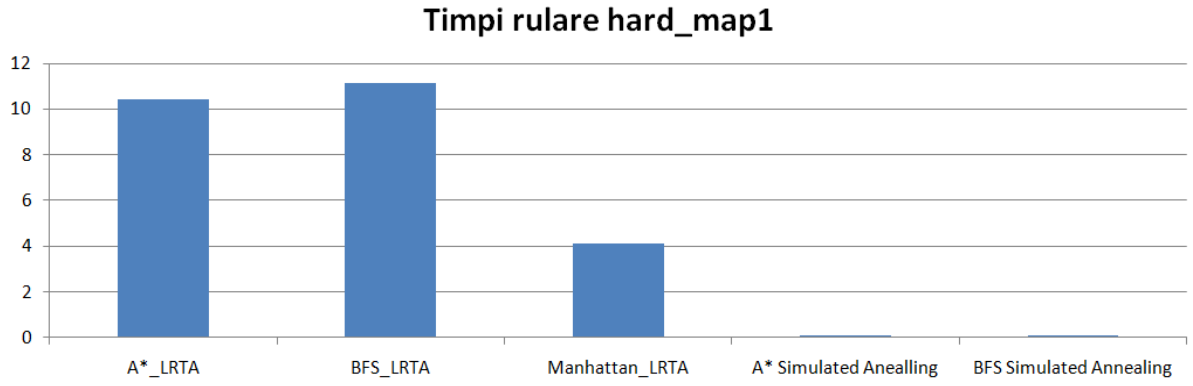
Se observă şi la medium1 acelaşi trend, în care LRTA explorează puţine stări, pe când anealling explorează foarte multe. Motivul principal este natura stocastică a algoritmului SA. Optimizare la lrta în care în situaţia de egalitate de energie între 2 stări se merge spre cea mai apropiată cutie reduce foarte mult numărul de stări explorate. În acest caz particular, Manhattan pe LRTA scoate timp mai bun decat A* şi BFS, datorită faptului că au loc mai puţine procesări(funcţia Manhattan este mult mai simplă dpdv computaţional).

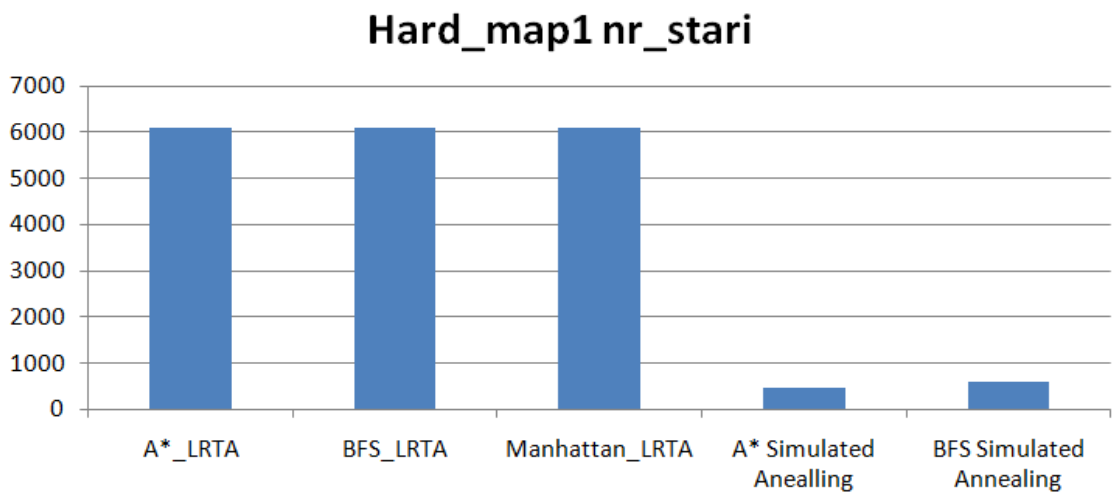




Se observă cel mai bine pe acest test că A* este mai eficient decât bfs cu aproximativ 10-12% atât pe LRTA, cât şi pe annealing. Pentru LRTA, cel mai eficient este Manhattan, datorită faptului că funcţia propriu-zisă este foarte ieftină dpdv computaţional. Astfel, are cel mai bun timp de 5.18 secunde. Testele de annealing trec mai greu ca timp, deoarece explorez mult mai stări decât în cazul LRTA. În timp ce la LRTA explorez doar 5351 de stări, la Annealing explorez peste 70.000. De aici apare diferenţa mare de timpi.

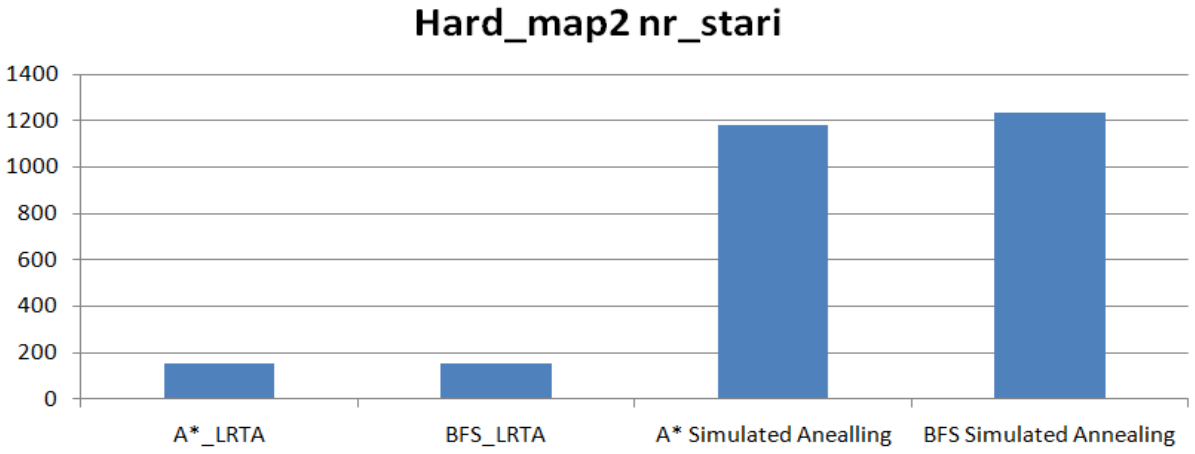
Teste hard

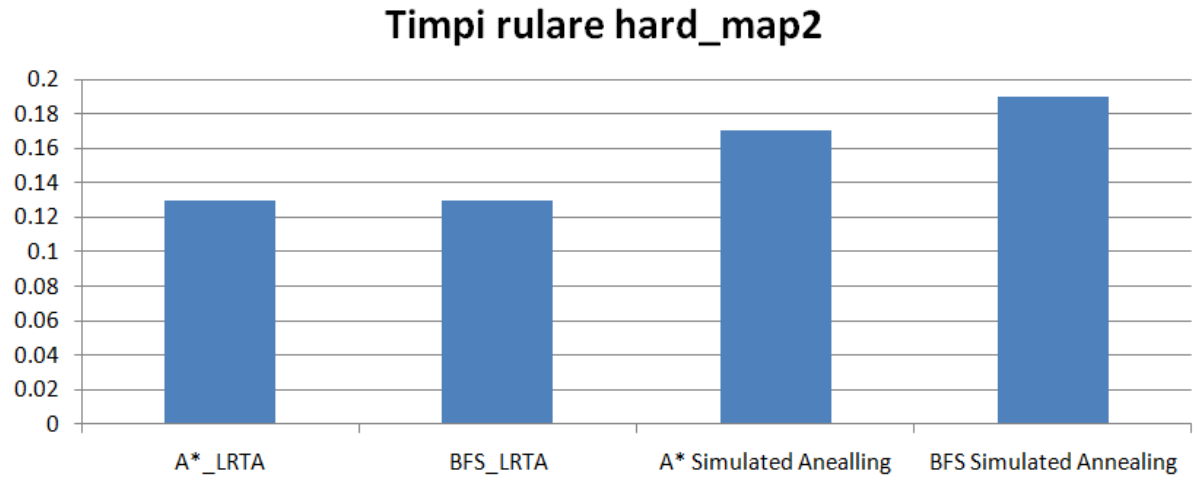




Pe testul hard_map1, algoritmul meu de annealing ori dă rezultatul direct în număr foarte mic de stări, ori dă fail. Trece foarte repede annealing, deoarece este o zonă mică de explorat. LRTA încearcă 9 permutări proaste până să găsească soluţia finală. Aici, funcţionează foarte rapid simulated annealing, deoarece lucrează la temperatură mare pe stările iniţiale şi poate accepta soluţii local slabe, care de fapt sunt foarte bune la nivel global.

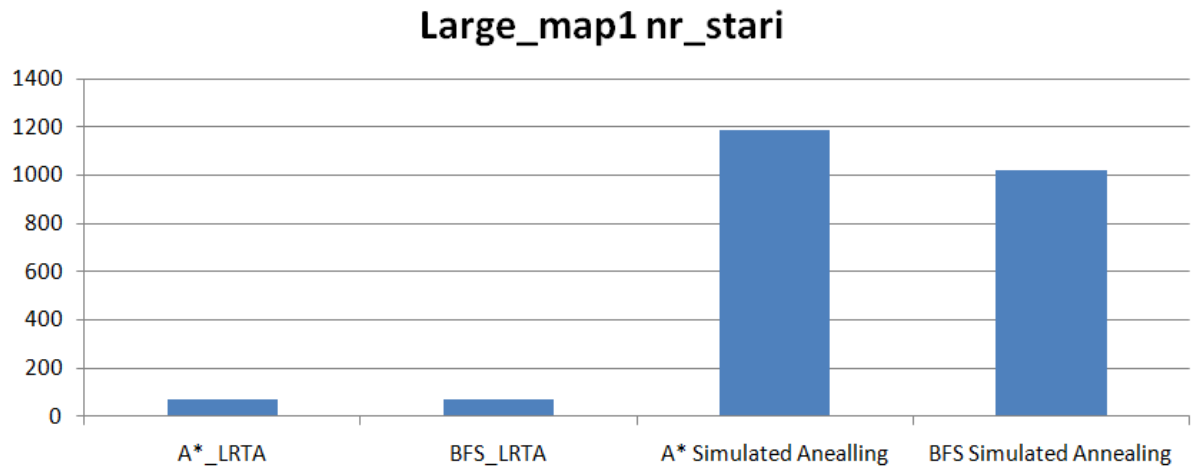
De asemenea, la LRTA, este foarte este foarte eficient dpdv al timpului datorită faptului că este o euristică foarte simplă. Manhattan pică pe toate testele la care ajung cu cutia în linie cu targetul, iar între box şi ţintă există un obstacol (easy1, large1,large2,hard2).

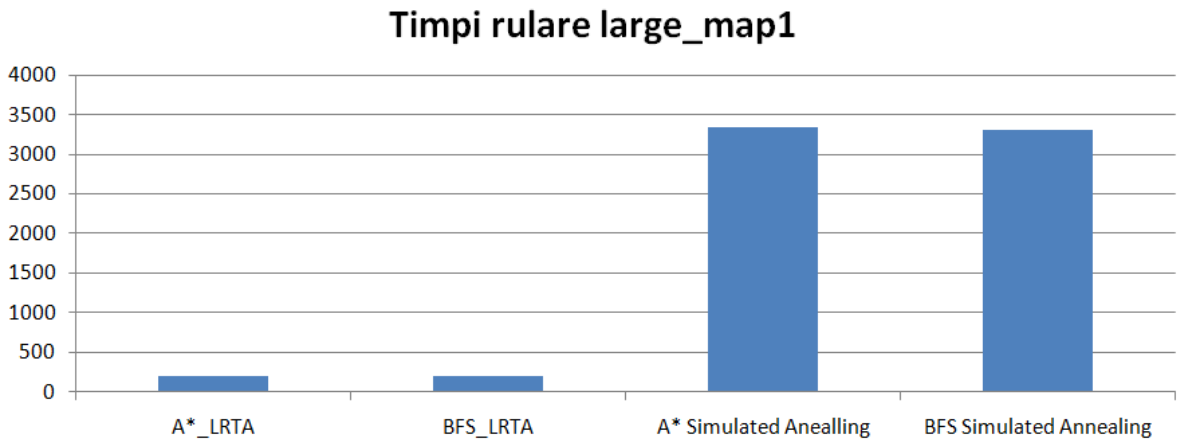




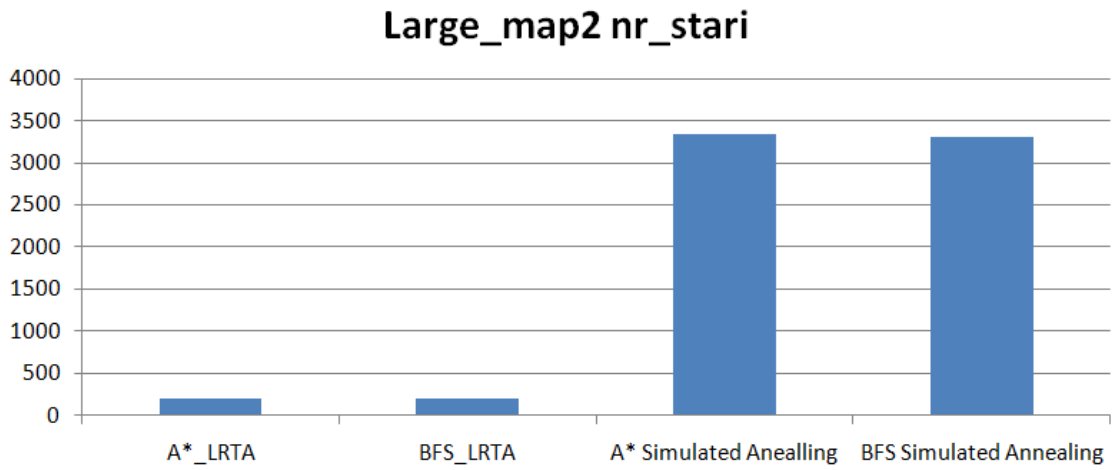
Hard_map2 este o hartă simplă. Este similară cu easy_map1, doar că este în oglindă şi cu 2 cutii. Pe acest test apar diferenţe mici de timpi între lrta şi simulated annealing. Pe această hartă, nu mai funcţionează distanţe Manhattan, deoarece cutiile ajung să aibă obstacol între target şi destinaţie. Este un test rapid şi este foarte simplu, deoarece nu am interferenţe între cutii, iar calea este ca la easy_map1 pentru ambele cutii.

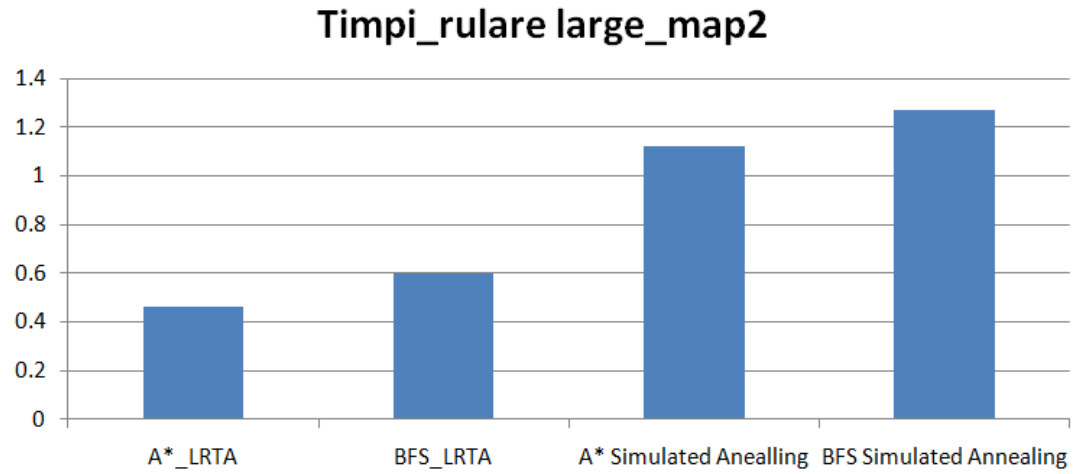
Teste Large





Testul large_map1 este similar cu medium_map1, doar că toate căile spre ţinte sunt mai lungi. De asemenea, există un obstacol mare care separă harta pe din 2, de aceea Manhattan pe LRTA nu funcţionează corect. Este o hartă care este rulată rapid atât pe lrta, cât şi pe simulated annealing. Se observă faptul că simulated annealing are nevoie de mai mult timp decât lrta. Se observă şi că A* este mai bun decât bfs dpdv computaţional. A* are un manhattan în spate, care poate aproxima costuri spre ţintă mult mai rapid decât bfs. BFS explorează în lăţime până când găseşte că targetul este inclus în vectorul de stări vizitate. Hărțile cu obstacole mari și drumuri ocolitoare afectează grav algoritmii care folosesc euristici naive (ca Manhattan fără obstacole). De asemenea, se poate observa că Simulated Annealing se apropie foarte mult ca timp de LRTA atunci când vine vorba de hărţi simple. Simulated Annealing ia decizii aleatoare local, însă LRTA are o viziune mult mai bună asupra hărţii ca întreg.





Şi în cadrul testului large_map2, ca şi la large_map1, nu funcţionează Manhattan pe LRTA. Se observă clar că A* este mai eficient atât pentru LRTA, cât şi pentru annealing.
În vreme ce lrta explorează doar 196 de stări, Simulated Annealing are peste 3000 de stări de explorat. Am observat un comportament ciudat la cel de-al doilea algoritm pe acest test. Uneori, îmi trece numai din 1000 de paşi, alteori face peste 6000. Aici, componenta stocastică are un efect mult mai pronunţat.

Rulare algoritmi:

Algoritmii pot fi rulaţi din main.py pus la dispoziţie în scheletul temei. Rulez python3 main.py [nume_algoritm] [nume_test], unde numele algoritmului poate fi lrta sau simulated_annealing, iar [nume_test] poate fi easy_map1, easy_map2, medium_map1, medium_map2, hard_map1, hard_map2, super_hard_map1. De menţionat că în linia de comandă dau denumirea fără .yaml. De asemenea, toate testele rulează pe ambii algoritmi (pe simulated_annealing nu rulează din prima hard_map1 şi medium_map2). Testul super_hard_map1 nu rulează pe niciunul dintre algoritmi. Menţionez că nu am rulat niciodată din Visual Studio Code, deoarece nu am interfaţa grafică unde să văd rezolvarea. Am scris tot codul în pycharm şi rulez main cu parametrii [lrta/simulated_annealing] [nume_test], fără extensia .yaml.