Belu Andreea-Daniela
333 CB

Tema 1 Inteligență Artificială
~ Sokoban Puzzle folosind IDA* și Simulated Annealing ~

Am început implementarea temei prin implementarea algoritmului IDA*. Apoi, am continuat cu implementarea și testarea diferitelor euristici până când am ajuns la cele finale *`deadlock_hungarian_heuristic`*, *`efficient_heuristic`* și *`deadlock_efficient_heuristic`* care au reușit să rezolve toate hărțile puse la dispoziție în temă. Dintre cele trei euristici, consider că cea mai calitativă este *`deadlock_efficient_heuristic`*, conducând la un număr mai mic de mișcări de tip pull pentru jumătate din teste. Apoi, am implementat algoritmul Simulated Annealing, considerând anumite optimizări și analizând diferite comportamente pe toate euristicile implementate pentru IDA*. Scopul final a fost perfecționarea acestui algoritm pentru a se putea folosi de aceeași euristică finală în rezolvarea eficientă a tuturor tipurilor de hărți.

I. IDA* (*`ida_star.py`*)

Am folosit un constructor în care am inițializat toate variabilele pe care le-am considerat utile: cele care fac referire la păstrarea unei soluții curente și cele care îmi salvează cea mai bună soluție întâlnită până la pasul curent, o funcție ajutătoare care returnează pozițiile cutiilor care trebuie mutate și poziția jucătorului, precum și o funcție care realizează implementarea propriu-zisă a algoritmului (*`ida_star`*) și funcția de solve moștenită din clasa Solver. În continuare, voi detalia implementarea funcțiilor *`ida_star`* și *`solve`*.

def ida_star(self, path, moves, g, threshold, current_pull_moves):

Aceasta este o funcție recursivă care se bazează pe backtraking. Se extrage ultima stare din cale, crescând numărul de stări explorate. Se verifică dacă aceasta este o stare finală (dacă toate cutiile au ajuns la un target). În acest caz, soluția descoperită până acum este considerată cea mai bună. Altfel, se verifică dacă starea curentă a mai fost vizitată anterior pentru a evita posibilele cicluri care pot apărea pe parcurs. În cazul în care calea curentă este considerată în continuare (nu avem stări care să conducă la cicluri), starea curentă este marcată ca vizitată. Se aplică euristica aleasă pe starea curentă, iar în cazul în care rezultatul este unul infinit, înseamnă că starea curentă nu ne poate conduce la o soluție bună. Se calculează valoarea totală a funcției estimate pentru a verifica dacă căutarea informată pe ramura în explorare poate continua sau nu. Se consideră fiecare mișcare posibilă din starea curentă care poate fi obținută prin mutare. Se verifică dacă aceasta este promițătoare, caz în care este adăugată în cale și se continuă recursivitatea. Dacă am găsit o soluție, aceasta este returnată, altfel ținem cont de cel mai mic cost întâlnit până acum și continuăm backtracking-ul.

def solve(self):

Se pornește din starea inițială care este adăugată în cale, cu 0 mutări și stări explorate, neexistând o soluție bună descoperită până acum. Pe această stare se aplică euristica primită ca parametru. Anumite euristici pot întoarce un rezultat infinit, caz în care pentru starea respectivă nu vom avea o soluție. Am inițializat threshold-ul cu valoarea euristicii inițiale. Se aplică *`ida_star`* pentru un număr de pași,

verificând, la fiecare pas, dacă o soluție a fost găsită. Algoritmul se oprește când numărul de pași a fost atins sau cand am găsit o soluție. Dacă am ajuns la un threshold infinit, înseamnă că nu avem soluție, altfel acesta este actualizat și se continuă cu următoarea iterație.

Implementarea acestui algoritm a fost destul de straightforward chiar dacă nu am avut o implementare standard la laborator pentru aceasta. Singura problemă pe care am avut-o a fost legată de considerarea unui număr prea mic de iterații, algoritmul blocându-se pe testele mari destul de repede, fapt ce a fost rezolvat prin setarea unei limite mai mari.

Optimizările aduse în plus față de pseudocodul prezentat la curs sunt: considerarea și minimizarea numărului de mișcări de tip pull posibile, urmărirea constantă a celei mai bune soluții, folosirea unui dicționar pentru detecția ciclurilor posibile.

Resursele utile în implementarea acestui algoritm au fost documentația de la GeeksForGeeks pentru înțelegerea algoritmului, discuțiile pe diverse forumuri despre Sokoban și modalități de îmbunătățire a căutării în spațiul jocului (Stackoverflow, Computer Science Stack Exchange).

II. Simulated Annealing *(`simulated_annealing.py`)*

Similar ca la IDA* am folosit un constructor în care am inițializat toate variabilele utile, respectiv cele care salvează soluția curentă și cele care salvează cea mai bună soluție întâlnită până la pasul actual. Am ales pentru temperatura inițială o valoare mare pentru a permite o explorare mai largă a spațiului și o rată de răcire extrem de lentă pentru o convergență treptată. De asemenea, și aici am implementat o funcție ajutătoare care să returneze pozițiile cutiilor și a jucătorului pentru o stare specifică, precum și funcția *`solve`* moștenită din clasa Solver. În plus, aici am implementat funcția *`softmax`* care are ca scop convertirea energiilor în probabilități, valorile mai mari conducând la probabilități mari. Am considerat necesară și implementarea unei funcții care calculează energia pentru o stare curentă astfel considerând minimizarea mișcărilor de tip pull, folosirea funcției euristice pentru estimarea distanței până la soluție, precum și o penalizare adusă pentru numărul total de mișcări, scopul fiind favorizarea soluțiilor mai scurte.

Acest algoritm a necesitat implementarea treptată a mai multor **strategii avansate** pentru a ajunge la comportamentul dorit. Am plecat de la implementarea de bază prezentată în laborator care implică următorii pași standard: inițializare, verificare dacă starea inițială este rezolvabilă, alegerea parametrilor pentru probabilitatea de acceptare, temperatura finală (aleasă una foarte mică pentru a favoriza explorarea spațiului), numărul maxim de iterații, precum și un contor pentru numărul de mișcări consecutive în care a fost aleasă o mișcare proastă din cauza lipsei uneia mai bune. Am observat o blocare în optime locale, motiv pentru care am ales sa reinițializez starea în care mă aflu cu parametrii celei mai bune stări, considerând o temperatură mai mare. Tot o reinițializare are loc și în cazul în care toate mișcările posibile duc la energii infinite. Modul de acceptare a rămas cel utilizat în laborator, adică acceptarea mișcărilor mai bune / probabilistic a celor mai proaste, contorizând și numărul de stări proaste alese consecutive, întrucât dacă am acceptat prea multe mișcări proaste să aibă loc o reinițializare a parametrilor. Testarea tuturor stărilor posibile cu euristica primită ca parametru, urmărirea și salvarea celei mai bune soluții, trecerea într-o altă stare mai bună sau mai proastă, verificarea găsirii unei soluții, precum și răcirea treptată au rămas precum în algoritmul de bază. O altă optimizare care a fost necesară se referă la alegerea probabilistică a stării. Am observat că alpha nu ajută suficient de mult atunci când harta este una foarte mare și poate sa aleagă orice în mod

random, algoritmul fiind depășit în acest caz. Astfel, am ales să integrez tehnica softmax din laborator pentru a ghida puțin alegerea stării. Acum în 50% din cazuri algoritmul revine în stările cunoscute deja ca bune, iar în restul cazurilor preferă să exploreze, cu scopul descoperirii unor posibile stări mult mai bune decât cele cunoscute până acum. Comportamentul obținut acum este cel mai bun, considerând reinițializările și alegerea stărilor prin care algoritmul nu mai rămâne blocat pentru mult timp, întrucât la un moment dat în mod sigur va reveni la o stare bună.

Implementare: spre deosebire de IDA* unde înțelegerea și implementarea algoritmului a fost mult mai directă, aici implementarea acestui algoritm consider că a fost una bazată foarte mult pe experimente. Spun asta pentru că implementarea unei noi optimizări conducea la rezolvarea unei probleme, însă o altă problemă era detectată.

Resurse utile au fost laboratorul unde am implementat Simulated Annealing și de unde am găsit soluția ce vizează alegerea probabilistică condiționată, informațiile predate la curs unde am descoperit necesitatea reinițializării în cazul blocării în minime locale, dar și diferite surse de pe Internet (GeeksForGeeks / YouTube) ce vizează înțelegerea vizuală și implementarea algoritmului în mai multe situații de jocuri pentru a observa diferite comportamente.

III. Euristici (`*heuristics.py*`)

Prima euristică la care m-am gândit pentru IDA* a fost detectarea unor posibile drumuri directe. Raționamentul pe care l-am avut a fost că dacă există un drum direct înseamnă că trebuie ca o cutie să se afle pe aceeași linie / coloană cu un target încă disponibil (nu este nicio cutie amplasată pe acesta) și să nu fie blocată de alte cutii, respectiv ziduri. Dacă am găsit un target pentru o cutie, acesta devine indisponibil. Aceasta este o abordare Greedy banală, foarte ineficientă, complexitatea acesteia fiind $\Theta(n^3)$. Rulând testele puse la dispoziție am descoperit că nu există hartă care să fie rezolvată de acest algoritm, dată fiind configurația inițială a hărților. Funcțiile utilizate în acest caz au fost: *`direct_path_heuristic (state)`* ce se folosește de *`is_direct_path_column (target_pos, box_pos, state)`* și *`is_direct_path_row (target_pos, box_pos, state)`*. Apoi, m-am întrebat dacă această abordare ar putea fi îmbunătățită prin detectarea timpurie a unor blocaje: atunci când o cutie este blocată de alte cutii și de un zid (blocaj orizontal sau vertical) (*`is_box_deadlock(state)`*) sau când o cutie este blocată pentru că este într-un colț al hărții (*`is_corner_deadlock(state)`*) din care nu se mai poate mișca. Am observat că acest raționament nu conduce la o îmbunătățire a euristicii, motivul fiind tot configurația hărților. Cu toate acestea, am decis să păstrez acest raționament în implementarea euristicilor ulterioare. În acest caz, funcția principală folosită a fost *`deadlock_direct_path_heuristic (state)`* ce se folosește de cele două funcții menționate anterior pentru blocaje și de o funcție ajutătoare *`is_there_a_box (box_pos_x,box_pos_y, boxes_not_on_targets)`*.

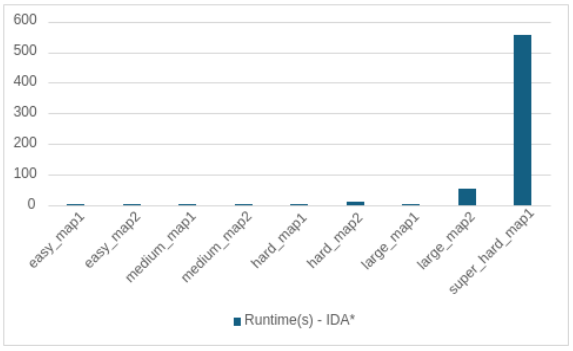
Ulterior, după ce am implementat și algoritmul Simulated Annealing am observat că output-ul este același, fapt deloc surprinzător considerând felul în care arată hărțile. Nu consider imperioasă realizarea unor grafice în acest caz, când euristica implementată este una nefuncțională, deci nu poate exista o comparație între algoritmi.

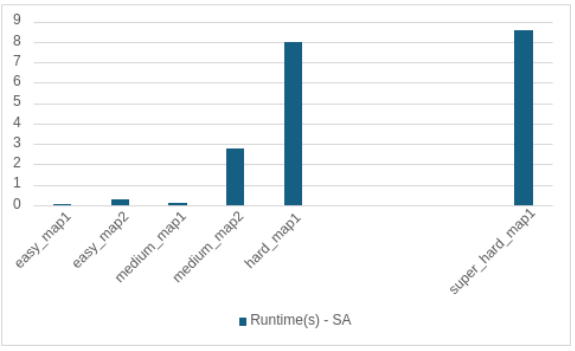
Apoi, analizând mai bine configurația hărților și euristicile pe care algoritmul A* le folosește în laborator, anume distanța Manhattan și distanța Euclidiană, am decis să implementez aceste două euristici considerând faptul că IDA* nu este decât o îmbunătățire a lui A* din punct de vedere al complexității spațiale, beneficiind de avantajele algoritmului ID-DFS.

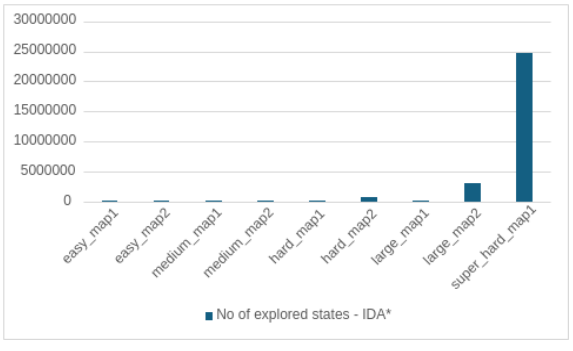
manhattan_dist_heuristic(state) calculează pentru fiecare cutie distanța Manhattan minimă până la cea mai apropiată țintă posibilă, însumând aceste distanțe pentru a obține distanța totală folosită ca valoarea de comparație a euristicii pe diferite stări. În acest caz nu se ține cont de obstacole sau de felul în care cutiile pot interacționa între ele. Conform graficelor prezentate mai jos ce plotează timpul de execuție, numărul de stări explorate și calitatea euristicii reprezentată de numărul de mișcări de tip pull pentru fiecare hartă rezolvată cu succes, am observat următorul comportament: pentru hărțile ușoare, algoritmul găsește eficient soluția, în schimb pentru hărțile mai dificile, care implică mai multe cutii, algoritmul devine confuz în ceea ce privește ordinea optimă de amplasare a cutiilor, numărul de stări explorate și timpii de execuție crescând semnificativ. Consider că aceasta se datorează faptului că euristica implementată nu ține cont de blocajele care pot exista. Am încercat să rezolv acest lucru prin integrarea celor două funcții: cea care detectează dacă o cutie este blocată într-un colț și cea care detectează dacă o cutie este blocată de alte cutii și de un zid. Astfel, am ajuns la *deadlock_manhattan_heuristic(state)*. Această implementare a condus la următorul comportament: explorarea unui număr semnificativ mai mic de stări, în special pentru hărțile medii și dificile, timpii sunt mai buni față de Manhattan de bază și prin evitarea situațiilor de blocaj, algoritmul tinde să găsească soluții care necesită mai puține mișcări de tip pull pe anumite teste, care sunt considerate costisitoare.

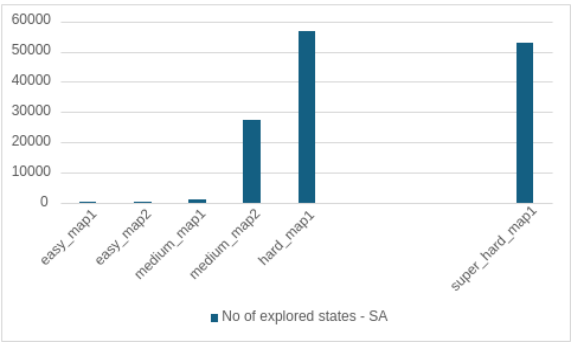
În ceea ce privește diferențele dintre cei doi algoritmi am observat următorul aspect: IDA* combinat cu Manhattan explorează foarte multe posibilități, devenind ineficient pe hărți foarte complexe, pe de altă parte Simulated Annealing, ce beneficiază de capacitatea de a scăpa din minime locale prin reinițializări, pare a fi mai eficient, deși nu reușește să găsească o soluție pe toate tipurile de hărți. În plus, aici numărul de mișcări de tip pull este unul mult mai redus, întrucât IDA* deja explorează suficient de mult spațiul, neavând nevoie de un număr semnificativ de mișcări de tip pull. Integrarea detectării blocajelor aduce îmbunătățiri în ambele cazuri, dar consider ca o îmbunătățire mai vizibilă este în cazul lui IDA* unde se elimină explorarea unor stări inutile.

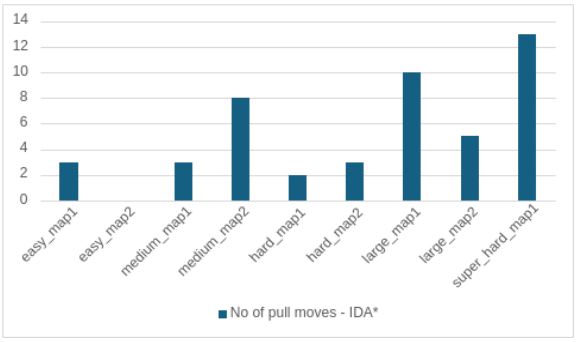
manhattan_dist_heuristic

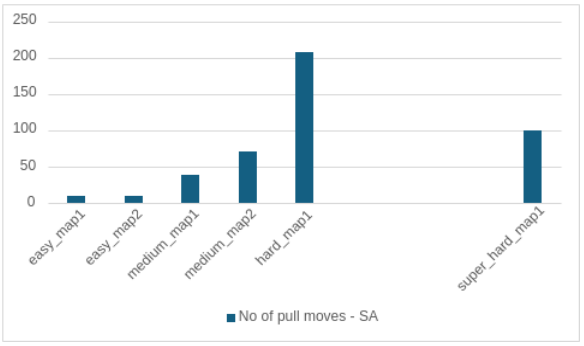




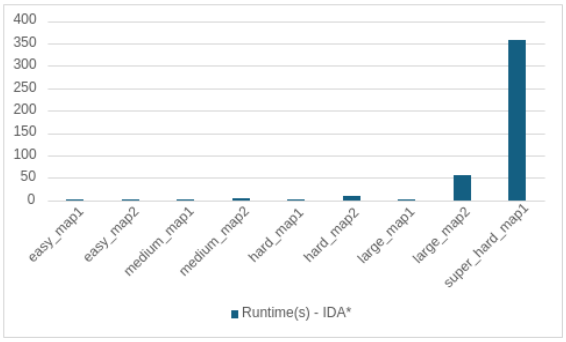


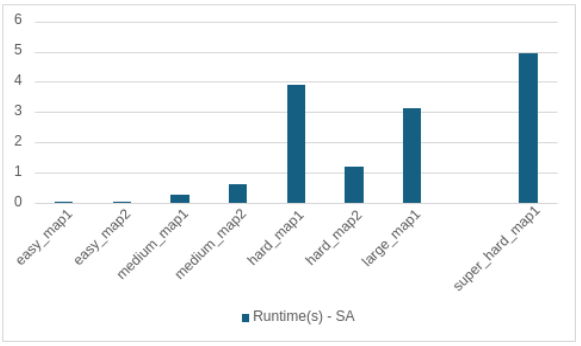


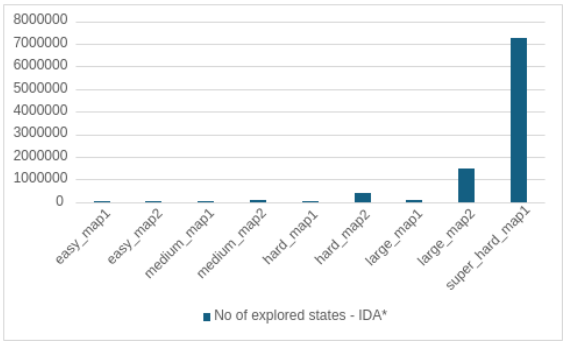


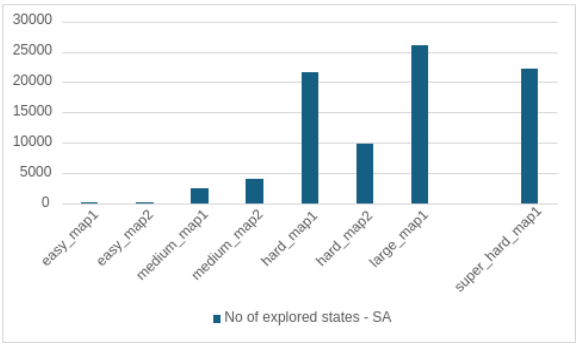


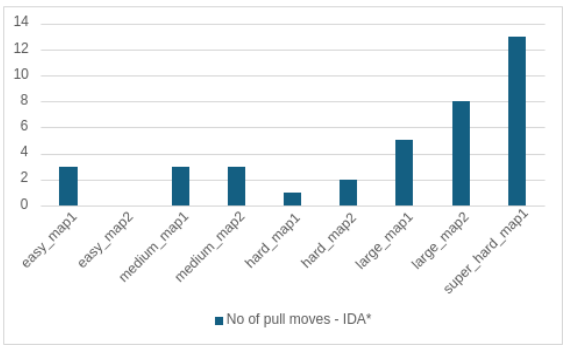
deadlock_manhattan_heuristic

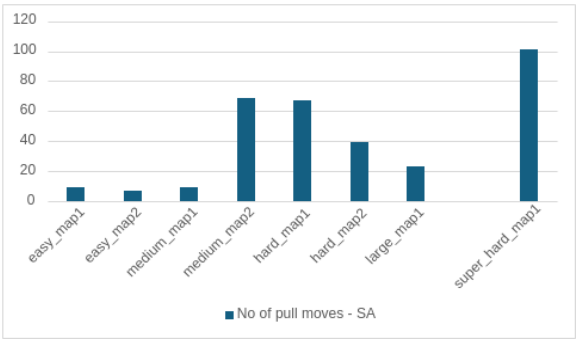








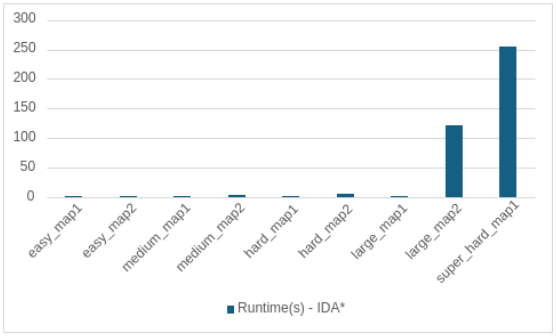


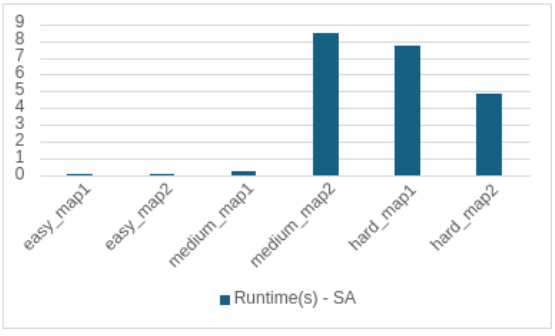


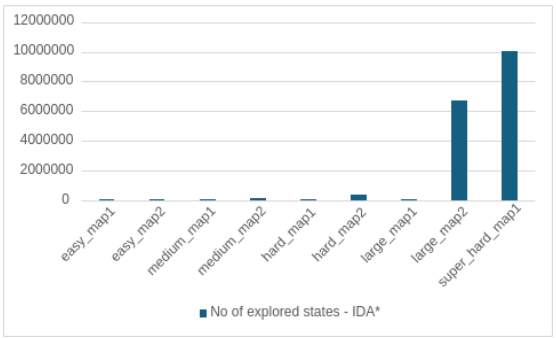
euclidian_dist_heuristic(state) calculează pentru fiecare cutie distanța Euclidiană minimă (în linie dreaptă) până la cea mai apropiată țintă, însumând toate aceste distanțe pentru a afla valoarea totală, ce va fi folosită ca reper de comparație între diferitele stări pe care se aplică euristica. Am ales să rotunjesc rezultatul în sus întrucât, lucrând cu algoritmi de căutare informată, ce au proprietatea de admisibilitate, euristica trebuie sa fie una optimistă. La fel ca la euristica Manhattan de bază nici aceasta nu ia în considerare obstacolele sau interacțiunile dintre cutii. Însă, distanța Euclidiană este mereu mai mică sau egală cu distanța Manhattan, fapt ce am considerat că ar putea duce la o îmbunătățire. Conform graficelor prezentate mai jos ce plotează timpul de execuție, numărul de stări explorate și calitatea euristicii reprezentată de numărul de mișcări de tip pull pentru fiecare hartă rezolvată cu succes, am observat următorul comportament: explorarea mai puținor stări comparativ cu Manhattan, pe testele mari. Pe testele mici și medii rezultatele par îmbunătățite. Considerând și de această dată faptul ca interacțiunea dintre cutii este ignorată, am ales să integrez cele două funcții de blocaje pentru a observa care ar fi noul comportament adus de acestea. Astfel, am implementat funcția *deadlock_euclidian_heuristic(state)*. Am ajuns la concluzia că prin abandonarea rapidă a căilor blocate, algoritmul poate explora mai eficient spațiul stărilor valide. Am reușit să aduc un progres, dar nu unul suficient de semnificativ.

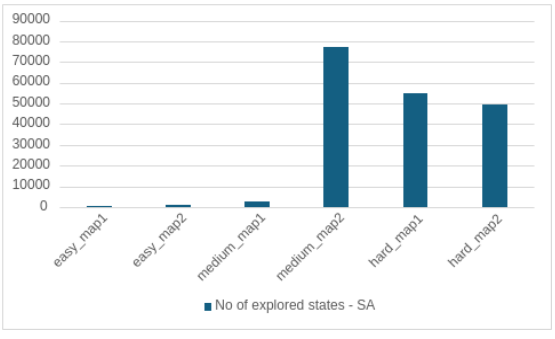
Comparând cei doi algoritmi, Simulated Annealing realizează o explorare mai amplă a spațiului stărilor posibile și considerând capacitatea acestuia de a alege aleator o stare următoare, aceasta putând fi în 50% din cazuri una deja cunoscută ca fiind eficientă pentru rezultat, am observat un progres pe testele dificile. Detectarea blocajelor conduce la detectarea timpurie a stărilor invalide, reducând numărul de stări explorate, deci o îmbunătățire și pentru Simulated Annealing. În ceea ce privește numărul de mișcări de tip pull, euristica Euclidiană fiind optimistă generează, în general, mai puține mișcări de tip pull decât Manhattan, însă acest fapt devine mai vizibil pentru Simulated Annealing în cazul ambelor euristici, motivul datorându-se tot explorării spațiului.

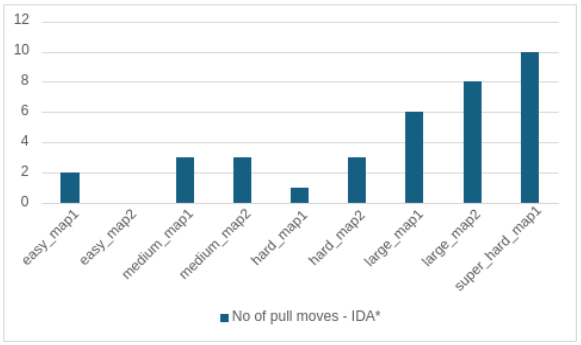
euclidian_dist_heuristic

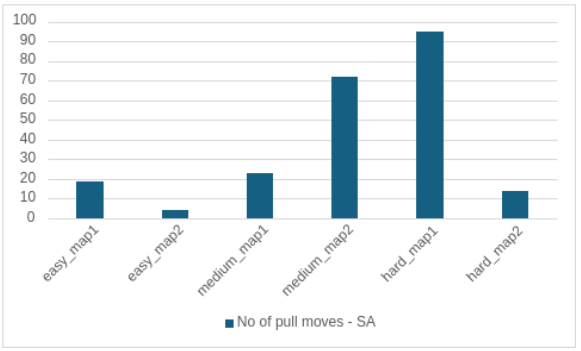




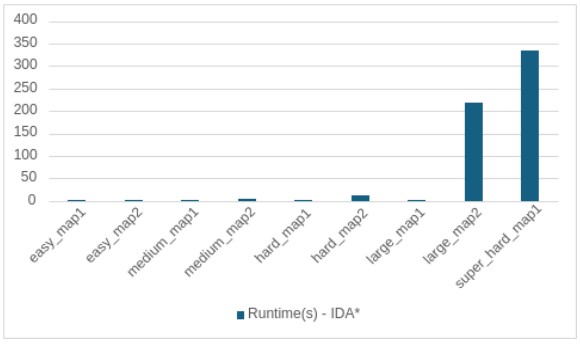


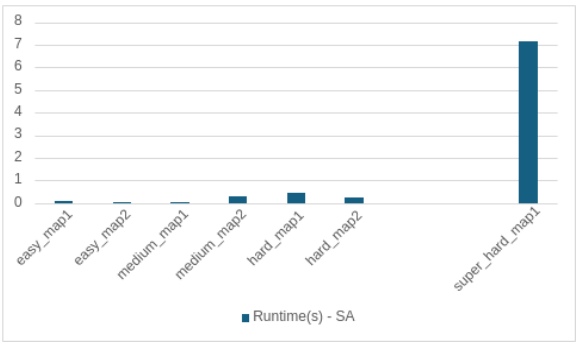


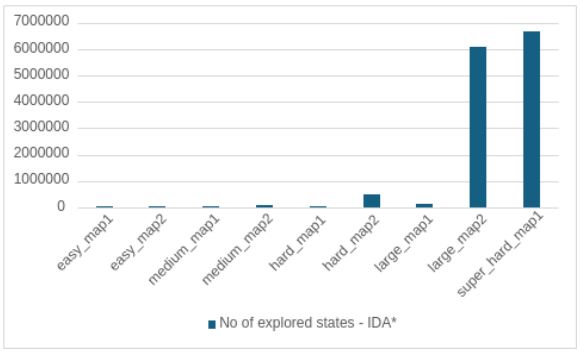


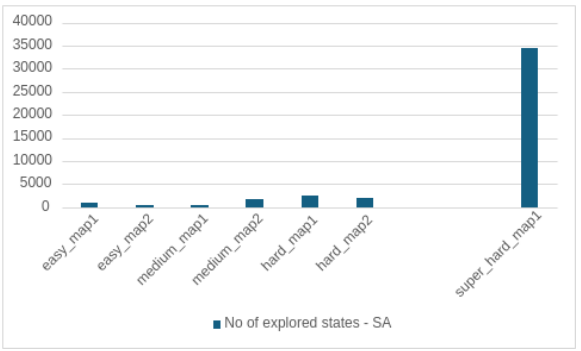


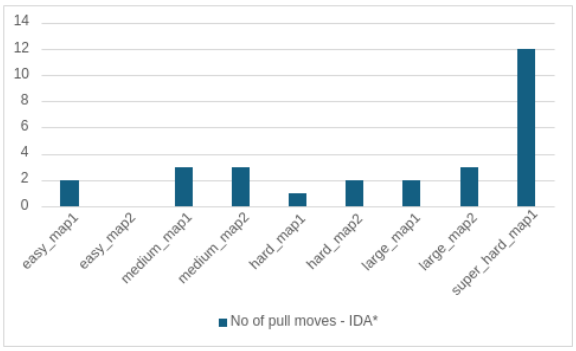
deadlock_euclidian_heuristic

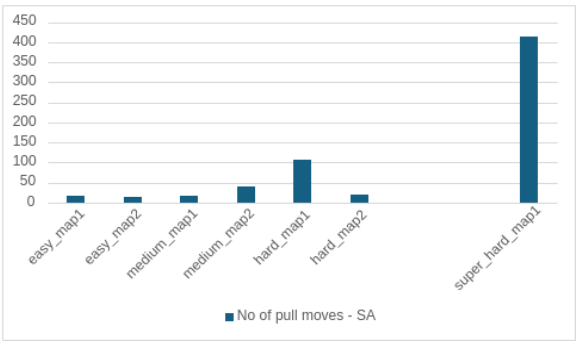










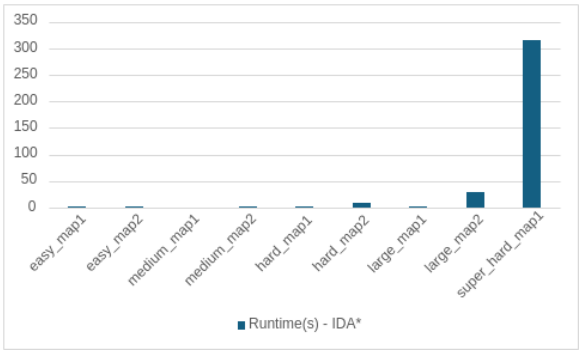


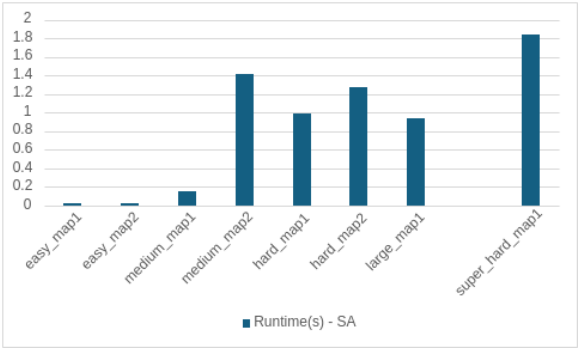
hungarian_dist_heuristic(state) - după mai mult research legat de ce alte euristici pot fi folosite în detectarea drumurilor optimale între obiecte și ținte, am descoperit Hungarian distance. Spre deosebire de cele două tipuri de distanțe explorate până acum, Manhattan și Euclidiană, care doar găsesc cutia cea mai apropiată de ele, ignorând faptul că pot fi mai multe cutii cu aceeași distanță minimă, fapt ce subestimează costul real, Hungarian distance pune accent pe asignarea cutiilor către ținte care minimizează distanța totală. Am ales să sortez cutiile după distanțele minime și să asignez fiecare cutie la cel mai apropiat target disponibil. Conform graficelor prezentate mai jos ce plotează timpul de execuție, numărul de stări explorate și calitatea euristicii reprezentată de numărul de mișcări de tip pull pentru fiecare hartă rezolvată cu succes, am observat următorul comportament: prin asignarea optimă, costul real este estimat mai bine conform mutărilor, deci am considerat o euristică mai informată; pentru IDA* numărul de stări explorate este considerabil mai mic, iar timpul de execuție este îmbunătățit pentru majoritatea hărților. *deadlock_hungarian_heuristic(state)* - combinarea cu detectarea blocajelor, pe care am realizat-o de această dată din curiozitate dacă aduce un beneficiu în plus, a condus la o eficiență mai mare și la o abandonare mai rapidă a căilor fără succes.

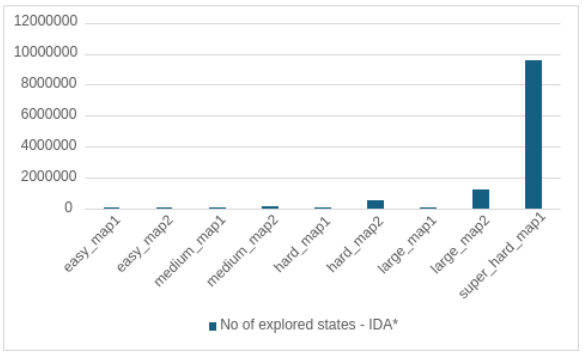
Comparând cei doi algoritmi și considerând ambele euristici, în ceea ce privește Simulated Annealing am observat ca acesta găsește mai ușor soluțiile în spații mai complexe, neavând nevoie de atât de multe reinițializări. Un progres a fost realizat și pentru acest algoritm în ceea ce privește detecția și abandonarea căilor imposibile. Cu toate acestea, am observat că numărul de stări explorate este unul în continuare mare. Referitor la numărul de mișcări de tip pull, combinația dintre asignarea optimă și eliminarea căilor care duc la blocaje permite găsirea unor soluții mult mai eficiente din perspectiva mișcărilor de tip pull în cazul ambilor algoritmi.

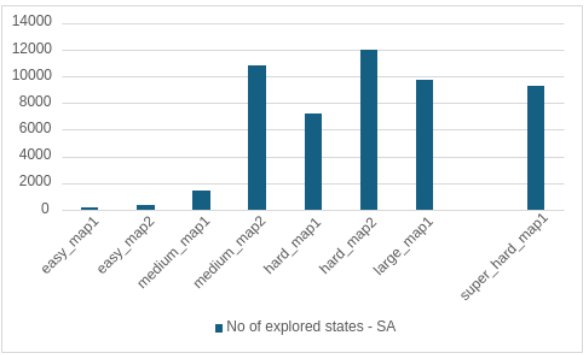
Întrucât, conform statisticilor rezultate, am observat că Hungarian distance oferă cea mai bună performanță dintre cele trei euristici analizate (Manhattan, Euclidiană, Hungarian), iar versiunea cu detecție de deadlock aduce îmbunătățiri substanțiale, în special pentru Simulated Annealing, am ales să continui cu acest algoritm, gândindu-mă cum aș putea să-l îmbunătățesc pentru a ajunge la rezultatul dorit.

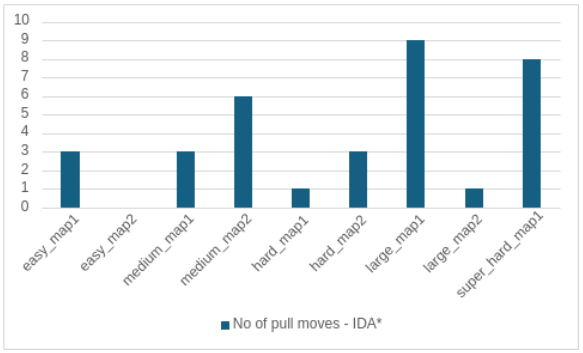
hungarian_dist_heuristic

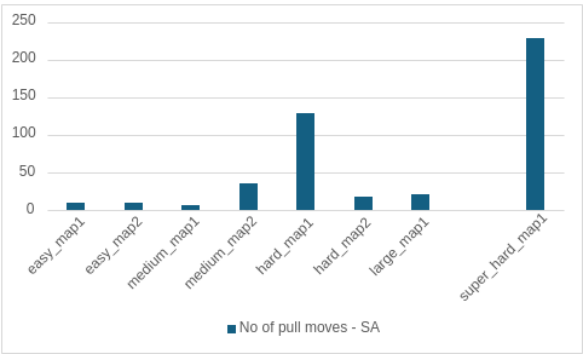




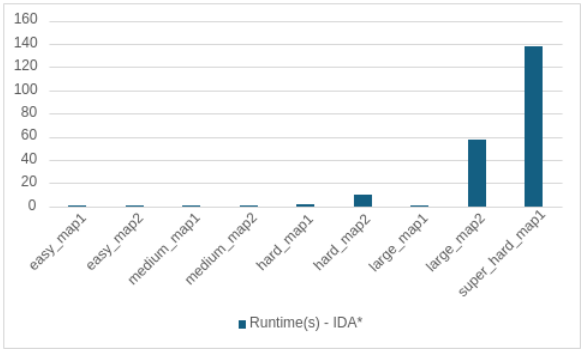


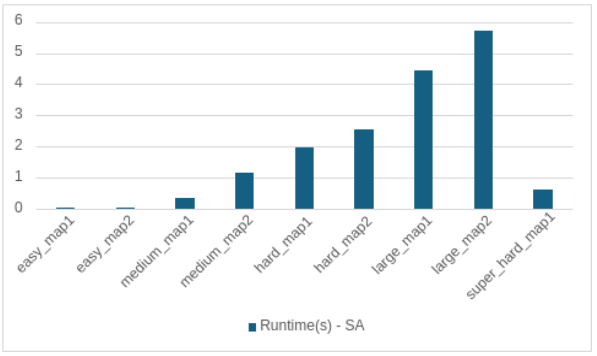


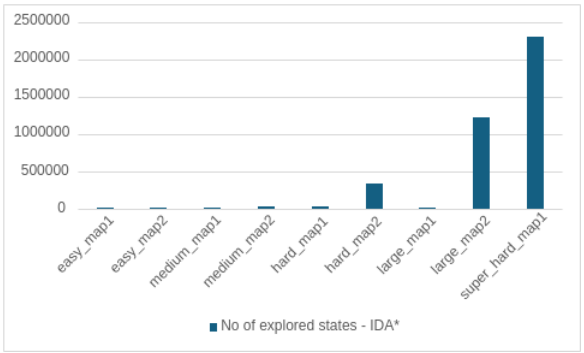


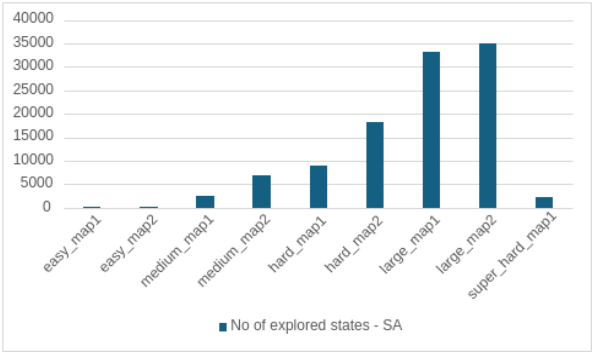


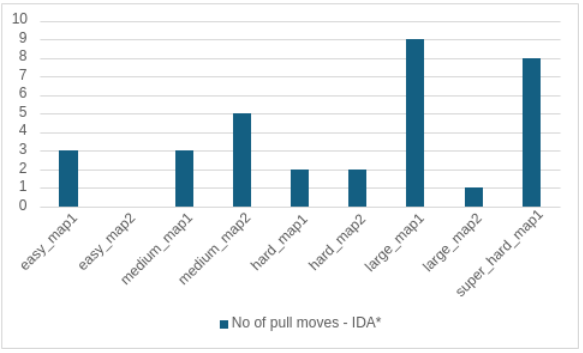
deadlock_hungarian_heuristic

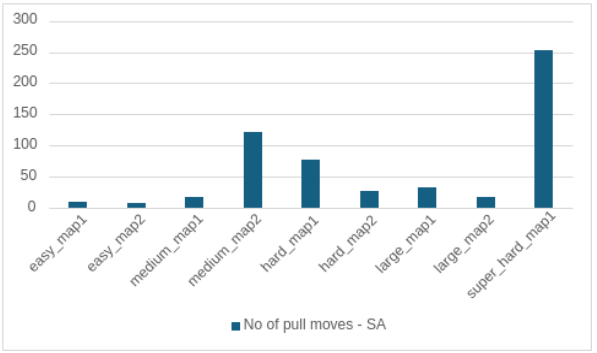






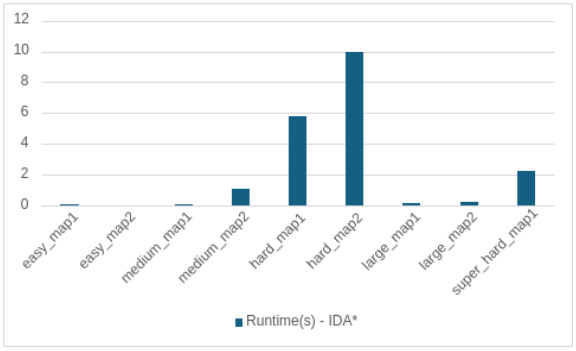


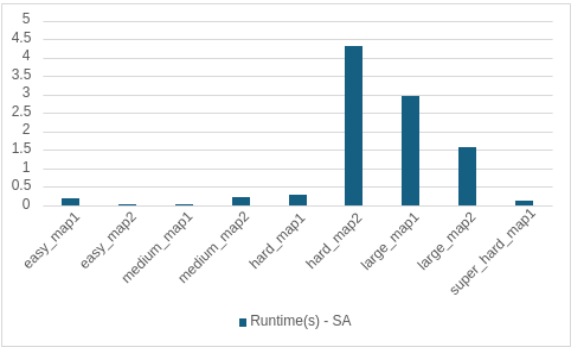


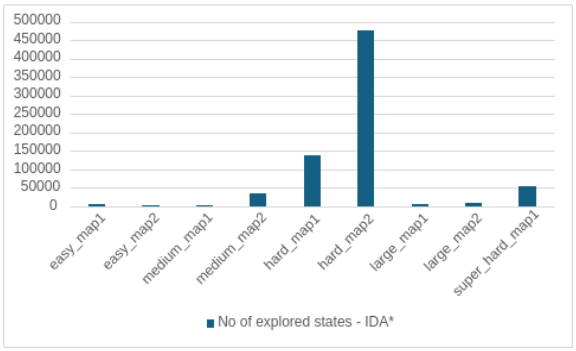


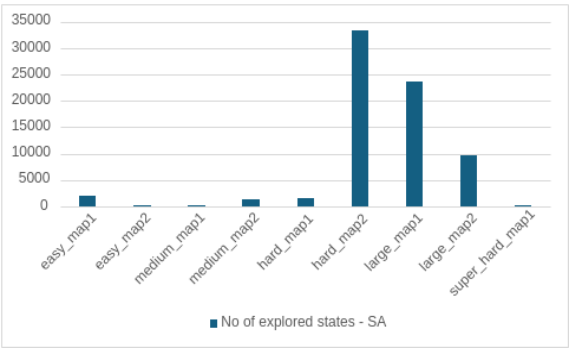
efficient_heuristic(state) - soluția la care am ajuns a fost să consider o penalizare pentru cutiile care sunt blocate de alte cutii și / sau ziduri. Pentru aceasta am implementat o funcție ajutătoare (*`compute_penalties_mobility(state, boxes_positions)`*) care calculează numărul de direcții blocate și calculează o penalizare, bazându-se pe numărul acestora. Valoarea returnată este suma dintre 2 * costul obținut aplicând Hungarian distance și penalizarea obținută. Am înmulțit cu 2 pentru a acorda o mai mare influență distanțelor cutie-target care sunt, de fapt, ținta jocului (ducerea cutiilor în poziții finale). Felul în care IDA* este influențat de această euristică este acela în care cutiile care devin greu de manevrat sunt evitate pe cât posibil și sunt alese cutiile care mențin mobilitatea celorlalte cutii. Un progres a fost adus și de aceasta dată pentru IDA*. Conform graficelor prezentate mai jos ce plotează timpul de execuție, numărul de stări explorate și calitatea euristicii reprezentată de numărul de mișcări de tip pull pentru fiecare hartă rezolvată cu succes, am observat următorul comportament: diminuarea drastică a numărului de stări explorate, timpul de execuție îmbunătățit pentru majoritatea hărților și eficiența soluțiilor determinată de numărul redus de mișcări totale. Analog, pentru Simulated Annealing calculul costului devine unul foarte informat, algoritmul identificând rapid căile potențial bune și eliminând rapid căile înșelătoare. Diminuarea numărului de mișcări de tip pull a fost realizată cu succes, conducând la cea mai bună calitate obținută până acum, deoarece prin penalizarea cutiilor cu mobilitate redusă, algoritmul este ghidat spre soluții care mențin cutiile în poziții cu mobilitate bună, reducând necesitatea mișcărilor de tip pull pentru a corecta poziții problematice. *deadlock_efficient_heuristic(state)* - și de această dată am vrut să observ comportamentul celor doi algoritmi atunci cand este luată în considerare și detectarea blocajelor. Astfel, am ajuns la eficiența maximă, întrucât numărul de stări explorate a fost redus semnificativ și pentru problemele anterior intractabile a fost găsită o rezolvare, micșorând totodată și numărul de mișcări de tip pull. Însă, întrucât numărul de mișcări de tip pull este unul mare pentru Simulated Annealing și calitatea temei se referă la încercarea diminuării cât mai mult a numărul de mișcări de tip pull, consider cea mai calitativă euristică - *efficient_heuristic(state)*.

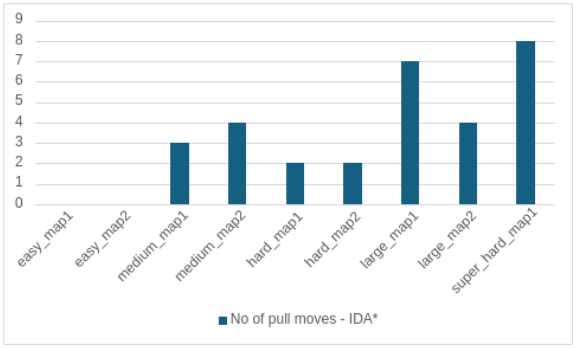
efficient_heuristic

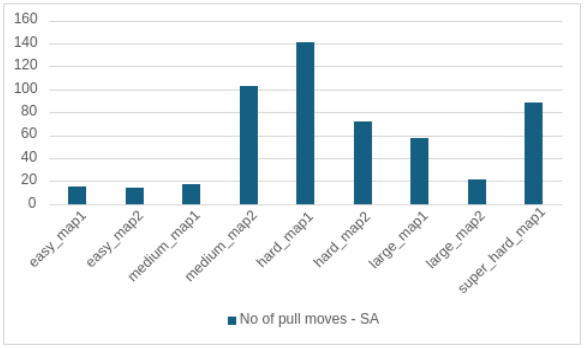












deadlock_efficient_heuristic