Tema 1 IA - Pocket Cube

Paunoiu Darius Alexandru - 342C4

8 decembrie 2023

Cuprins

1 Introducere

Tema isi propune analiza algoritmilor A*, BFS-Bidirectional si Monte Carlo Tree Search pentru rezolvarea unui Pocket Cube (Cub rubik 2x2x2), folosind atat eurisitici admisibile, neadmisibile cat si pattern database.

Toate testele pentru Monte Carlo Tree search reprezinta media a 20 de rulari. Algoritmii au fost rulati pe urmatoarea configuratie:

- Procesor: I5-10500h 2.5Ghz Base 4.50 GHz Turbo
- Memorie: 16 GB Ram DDR4 2800MHz

Toate datele au fost salvate folosind pickle, astfel nu este necesarea rerularea algoritmilor pentru cazurile de teste, acestea importandu-se automat daca sunt gasite fisierele. Pentru o rulare completa a fisierului Solution, timpul de executie ar trebui sa fie undeva la 1-2 minute daca fisierele de date sunt prezente.

2 Algoritmul A*

2.1 Euristica h1

Aceasta euristica este una destul de simpla, si admisibila atat prin logica ei cat si prin demonstratia brute-force care incearca sa testeze cat mai multe worst-case-uri. Implementarea acestei demonstratii se regaseste in functiile `test_heuristic_brute_force` si `test_heuristic_random_inputs`. Prima din ele testeaza toate cuburile care sunt la maxim 12 mutari, dar doar din set-ul direct de mutari (U, F, R), iar a 2-a cuburi cate 25000 de cuburi random aflate la distanta de la 1 la 14.

Euristica estimeaza costul pana la cubul rezolvat, numarand cate fete nu sunt la locul lor, si imparte rezultatul la 8, deoarece in cazul cel mai optimist, 8 fete gresite inseamna o mutare (de exemplu, miscarea R strica 8 fete).

2.2 Implementarea algoritmului

Algoritmul este foarte similar cu cel de la laborator, folosind o coada de prioritati pentru a extrage mereu nodul cu costul minim pana la destinatie. Acesta intoarce calea obtinuta pentru a rezolva cubul, si numarul de stari parcurse. Solutiile obtinute au fost verificate pentru corectitudine, fiind astfel corecte.

2.3 Rezultate obtinute

Pentru cele 4 cazuri de test, algoritmul A* intoarce cai de lungime 5, 7, 9, 11 in timpi de aproximativ 0.041, 0.397, 7.41 si 83.1 secunde. Aceste rezultate nu sunt tocmai satisfacatoare pe masura ce complexitatea solutiei creste, dar acest lucru se poate datora euristicii care desi este admisibila, nu este chiar optima.

3 Algoritmul BFS Bidirectional

3.1 Implementare algoritm

Algoritmul BFS Bidirectional incepe 2 cautari incercand sa gaseasca un nod comun, adica o combinatie a cubului comuna, alternand la fiecare pas intre calea $initial \Rightarrow destinatie$ si $destinatie \Rightarrow initial$ Acesta intoarce calea obtinuta pentru a rezolva cubul, si numarul de stari parcurse. Caile intoarse au fost verificate pentru corectitudine, fiind astfel corecte.

3.2 Rezultate obtinute

Pentru cele 4 cazuri de test, algoritmul A* intoarce cai de lungime 5, 7, 9, 11 in timpi de aproximativ 0.063, 0.072, 0.31 si 1.5 secunde. Aceste rezultate sunt foarte satisfacatoare, iar pe masura ce complexitatea solutiei creste timpul este inca unul foarte scazut.

Acest lucru se datoreaza probabil numarului relativ limitat de stari.

4 Analiza A* si BFS Bidirectional

4.1 Analiza timp

Voi incepe mai intai prin a compara cat de mult dureaza pentru fiecare algoritm sa gaseasca o solutie in functie de lungimea iei.

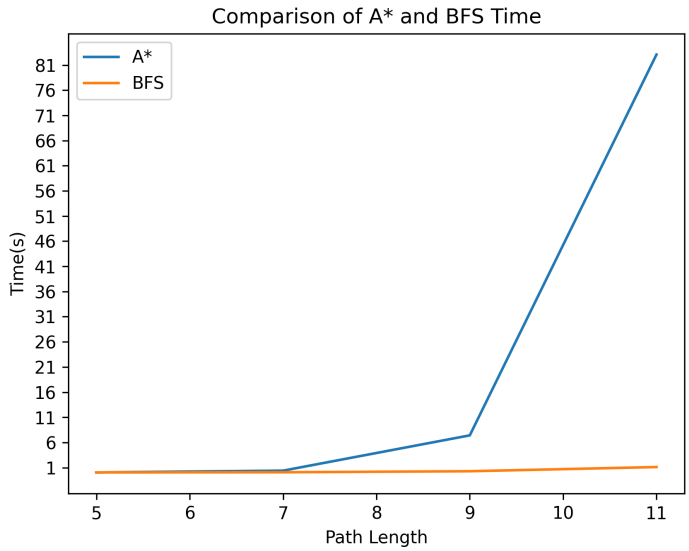

Figura 1: Comparatie timp executie A* si BFS Bidirectional.

Din graficul de mai sus, se poate vedea cum timpul pt BFS este mult mai aproape de un liniar, in timp ce pt A* tinde spre un tind exponential cu panta foarte mare.

Deoarece prima portiune a graficului pare lipita datorita scalarii, vom include mai jos un grafic care compara primele 3 cazuri.

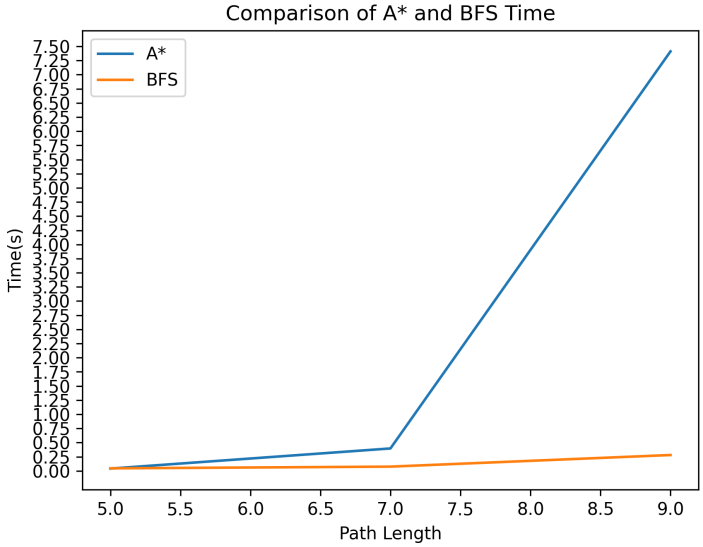

Figura 2: Comparatie timp executie A* si BFS Bidirectional pe primele 3 cazuri.

Din acest grafic se poate observa cum inca de la inceput panta de la A* este mult mai mare decat panta de la BFS. Astfel inca de la cazuri care au distante relativ mici de 5-7 mutari se poate observa ca BFS scaleaza mai bine decat A*.

Vom intra si mai in detaliu, comparand fiecare caz individual.

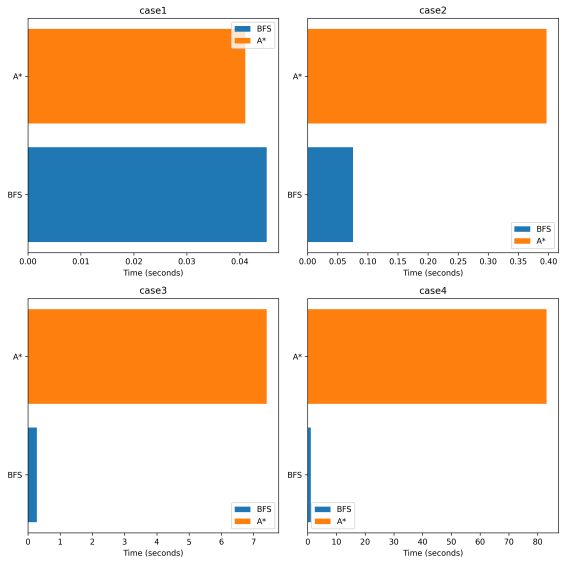

Figura 3: Comparatie timp executie A* si BFS Bidirectional per caz.

A* pare sa aiba un avantaj pentru primul caz, ceea ce este posibil sa fie adevarat si nu doar un hazard de rulare, deoarece solutia este foarte aproape de cubul initial si atunci euristica are sanse mari sa fie mai precisa. Lucrurile se schimba in momentul in care complexitatea solutiei creste, timpul devenind mult mai mare la A*. Acest lucru se poate datora faptului ca desi sunt la 7 mutari distanta de solutie, euristica ne va spune mereu ca suntem la 3.

Din punct de vedere al timpului de executie, este clar ca BFS scaleaza mult mai bine, cel putin atunci cand avem o euristica nu tocmai optima.

4.2 Analiza lungimea solutiei

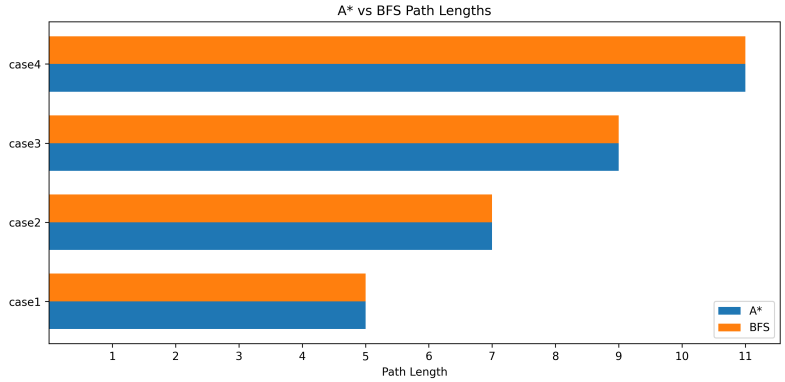

Figura 4: Comparatie lungime solutie A* si BFS Bidirectional.

Exact cum am mentionat in sectiunile rezultate, lungimile solutiei sunt aceleasi, deci putem trage concluzia ca ambii algoritmi ajung la o solutie de lungime optima, dar in afara de cazul 1, A* dureaza mult mai mult sa ajunga la aceasta solutie.

4.3 Analiza numar stari explorate

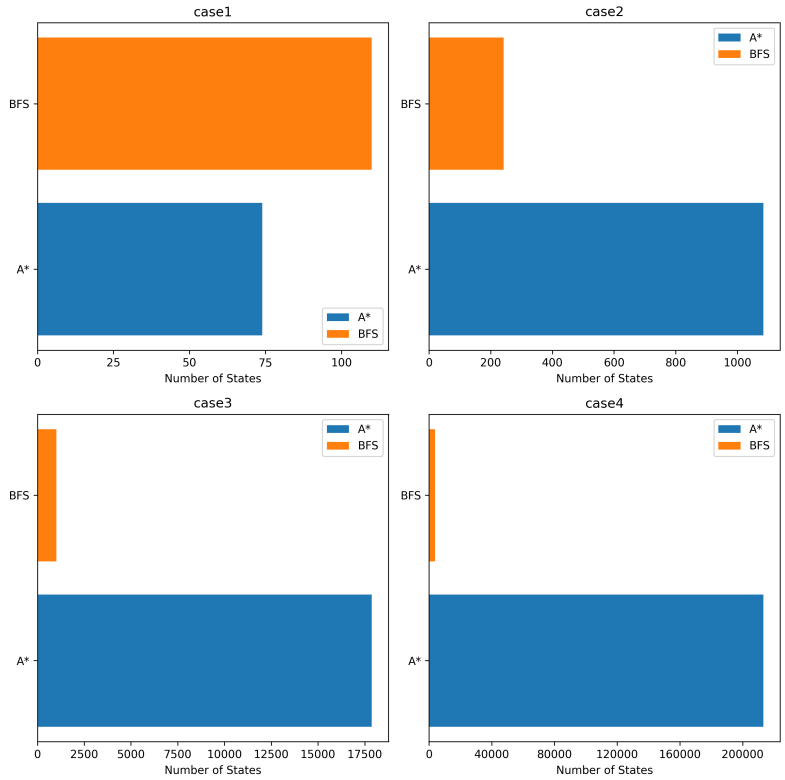

Figura 5: Comparatie lungime solutie A* si BFS Bidirectional.

Se poate observa, exact cum am mentionat mai sus, pentru primul caz, numarul de stari este mai mic pentru A*, de aici si timpul mai scurt de executie. Cand lungimea solutiei creste, numarul de stari creste si el exponential, deoarece nu avem o euristica suficient de buna sa ne spuna exact care este cea mai buna cale, si ajungem sa exploram foarte multe stari. Un lucru interesant totusi, durata de expansiune a unei stari este relativ aceeasi, 0.00043 secunde pentru A* si 0.00039 secunde pentru BFS, acest lucru indicand ca o euristica mai buna ar putea scala A* la performante mai bune decat BFS.

5 Algoritmul Monte Carlo Tree Search

5.1 Implementare algoritm

Algoritmul Monte Carlo Tree Search (MCTS) porneste de la nodul initial, si incearca sa contruiasca un arbore prin cele 4 faze ale sale:

1. SELECTIE: se alege nodul care maximizeaza formula pentru Upper Bound Confidence pana cand gasim un nod cu actiuni neexplorate sau solutia arborelui.

2. EXPANSIUNE: se creaza un nod nou, alegand una din miscarile neincercate inca. Nu se vor alege randuri de miscari inutile pentru a eficientiza algoritmul (De exemplu, daca nodul curent a rezultat din parinte prin miscarea R, nu se va face miscarea R').

3. ROLLOUT: Se joaca jocul pentru maxim 14 mutari, si alege starea cea mai buna. Similar, nu se vor alege miscari inutile. Reward-ul pentru un cub finalizat este foarte mare comparativ cu celelalte reward-uri.

4. Propagare: Propagam rezultatul catre radacina.

5.2 Euristica H2

A doua euristica se bazeaza tot pe numarul de fete gresite, dar incearca sa fie putin mai inteligenta:

- Daca avem 8 fete gresite, va intoarce mereu 1, sperand sa fim la o mutare distanta de destanatie.
- Daca avem 15 fete gresite, va intoarce mereu 3, deoarece majoritatea gruparilor de 3 miscari vor duce la 15 fete gresite. Evident, nu oricare 3 miscari duc la 15 fete gresite, incearca sa fie optimista din acest punct de vedere.
- Daca sunt mai putin de 8 fete gresite, va intoarce mereu 2.
- Altfel, va intoarce numarul de fete gresite pe 2.

Euristica nu este admisibila din cauza ultimei conditii, dar am sperat ca aceasta impartire pe categorii sa duca la expansiune unor noduri mai calitative decat in cazul euristii h1.

5.3 Rezultate obtinute

Rezultatele obtinute pentru ambele euristici sunt foarte dezamagitoare. Intr-adevar numarul de stari este mult mai mic decat la A*, dar durata de expansiune a unei stari este mult mai mare, mai exact 0.001056 secunde, aproximativ de 2.4 ori mai mare decat la A*. Cel mai nesatisfacator este rata de gasire a solutiei.

Toate analizele pentru MCTS sunt media a 20 de rulari.

5.4 Analiza H1 si H2

5.4.1 Analiza numar de stari

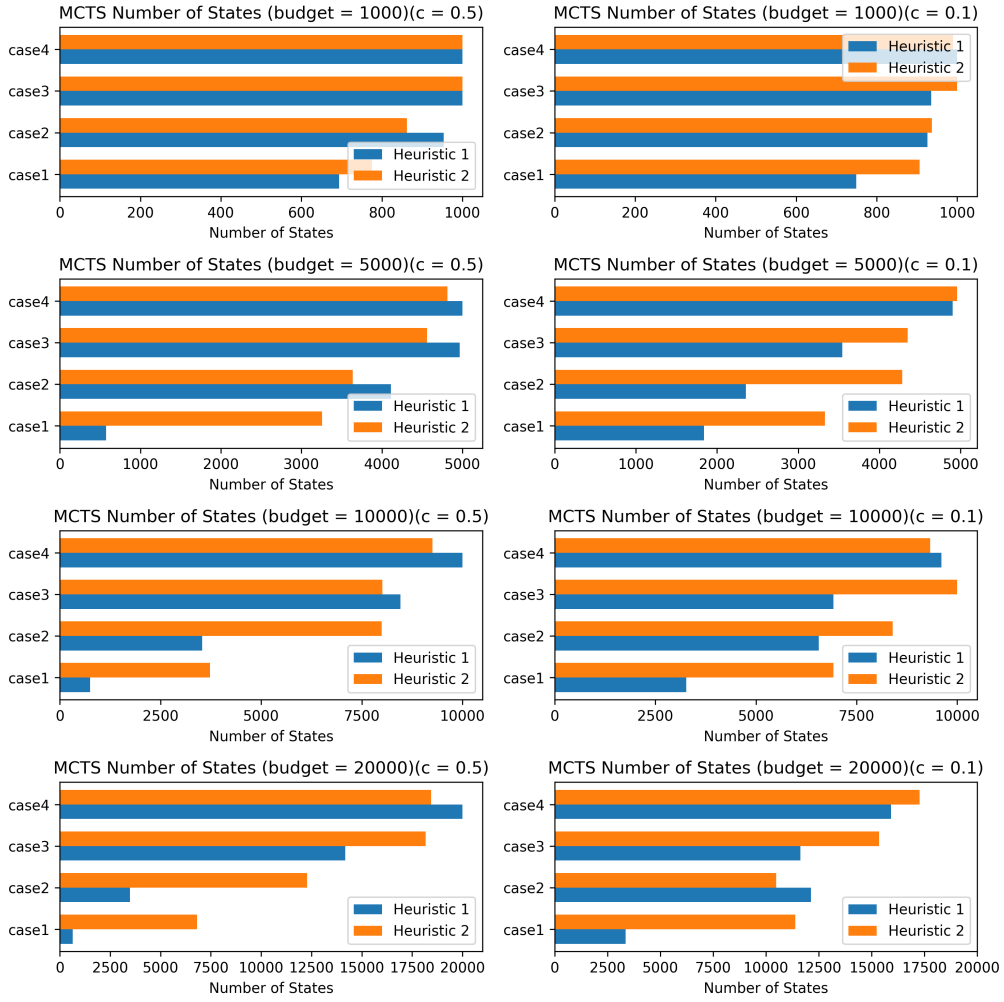

Figura 6: Comparatie numar de stari MCTS pentru cele 2 euristici.

Cea mai interesanta analiza este cea asupra numarului de stari, deoarece din ea se poate deduce timpul de executie (fiind direct proportionale), dar si daca s-a gasit solutie(daca nr de stari este egal cu bugetul, atunci nu s-a gasit solutie).

Uitandu-ne la cele 2 grafice, putem trage urmatoarele concluzii:

- Euristica h1, in general construieste mai putine stari.
- Pentru sistemul de reward-uri ales, c = 0.1 are rezultate mai bune decat c = 0.5 pe cazurile mari, iar pe cazurile mici c = 0.5 are rezultate mai bune (in special pentru euristica h1).
- Daca consideram c-ul satisfacator pentru euristica h1 (0.5 pentru cazurile 1 si 2, 0.1 altfel), aceasta are in mare parte rezultate mai bune decat h2. Exista si exceptii, dar am sa presupun ca acestea pot fi datorate sistemului de alegeri aleatoare, chiar daca au fost 20 de rulari.

5.4.2 Analiza timp

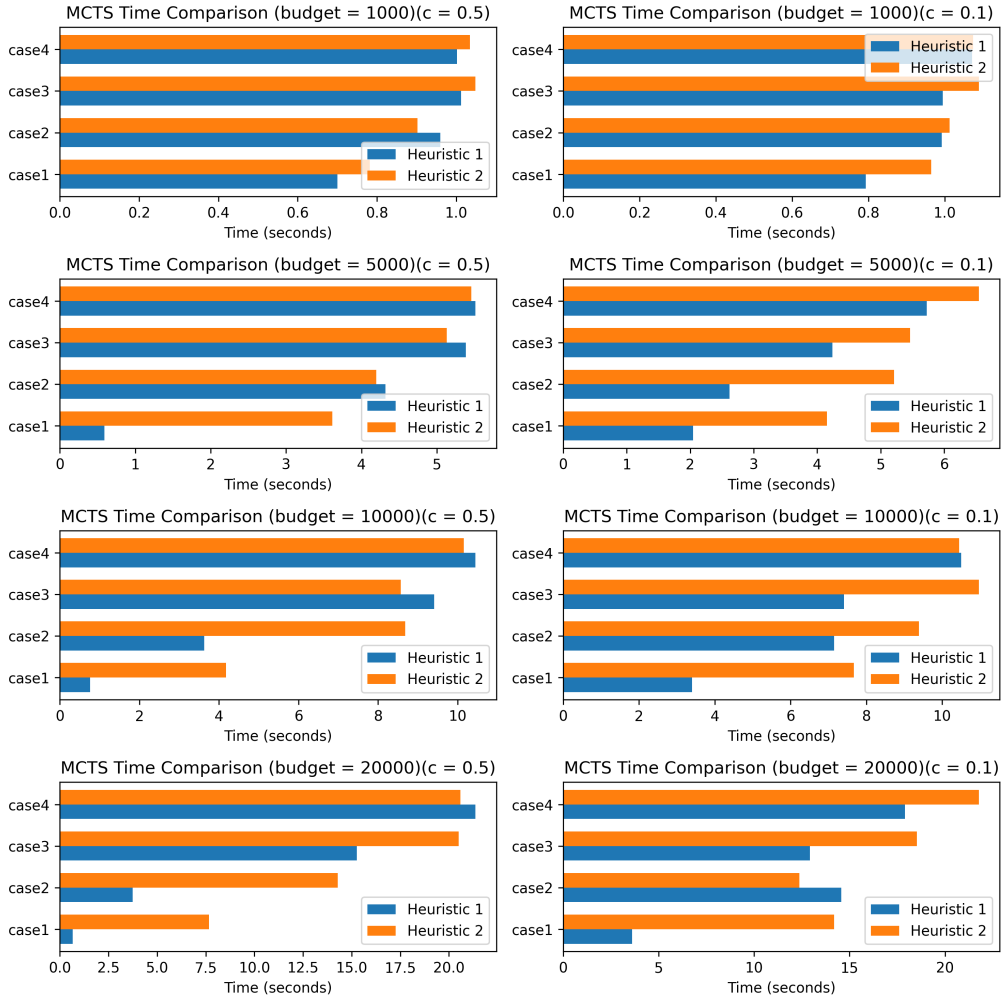

Figura 7: Comparatie timp executie MCTS pentru cele 2 euristici.

Dupa cum am mentionat anterior, graficele pentru timpul de executie si numarul de stari sunt foarte asemanatoare, acestea doua fiind direct corelate. Deci, teoria s-a dovedit a fi adevarata, asadar concluziile anterioare se respecta si aici:

- Euristica h1, in general are un timp mai scurt de rulare.
- Pentru sistemul de reward-uri ales, c = 0.1 are rezultate mai bune decat c = 0.5 pe cazurile mari, iar pe cazurile mici c = 0.5 are rezultate mai bune (in special pentru euristica h1).
- Daca luam c-ul satisfacator pentru euristica h1 (0.5 pentru cazurile 1 si 2, 0.1 altfel), aceasta are in mare parte rezultate mai bune decat h2.

5.4.3 Analiza lungime solutie

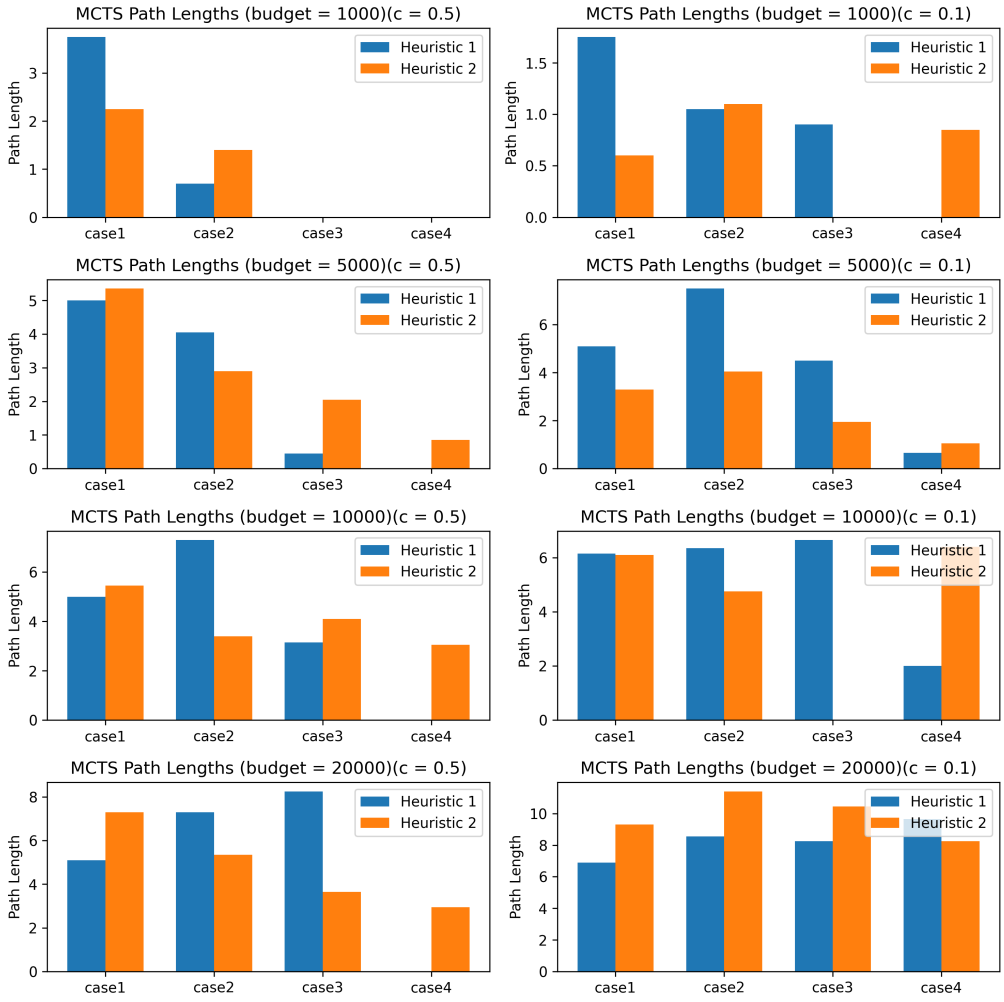

Figura 8: Comparatie timp executie MCTS pentru cele 2 euristici.

De mentionat este ca pentru aceasta analiza, cand algoritmul nu a gasit solutie, acesta a returnat o cale goala, deci lungimi mai scurte ale solutiei nu inseamna neaparat mai bine in acest caz. Acestea fiind luate in considerare, putem trage urmatoarele concluzii:

- Euristica h1, in general gaseste solutia mai des.
- Pentru c = 0.5, euristica H2 gaseste solutia pentru cazuri mai mari, in special pe bugete mici, unde H1 nu reuseste sa o gaseasca. Deci pare ca acele valori statice intoarse reusesc sa obtina un rezultat mai bun, descoperind mai devreme niste noduri mai bune.
- Pentru sistemul de reward-uri ales, c = 0.1 are rezultate mai bune decat c = 0.5 pe cazurile mari, iar pe cazurile mici c = 0.5 are rezultate mai bune (in special pentru euristica h1).
- Daca luam c-ul satisfacator pentru euristica h1 (0.5 pentru cazurile 1 si 2, 0.1 altfel), aceasta are in mare parte rezultate mai bune decat h2.
- Evident, se poate observa ca pe masura ce creste bugetul, creste si sansa de a gasi o solutie.

5.4.4 Concluzii analiza euristici

Desi euristica h2 are rezultate foarte bune in anumite cazuri, voi alege mai departe sa analizez euristica h1, deoarece pare a fi mai consistenta, iar alternand intre constante in functie de dimensiunea cazului de intrare (0.5 pentru cazurile 1 si 2, 0.1 pentru cazurile 3 si 4) putem obtine un comportament de multe ori chiar mai bun decat in cazul lui h2. Daca aveam la dispozitie doar constanta c = 0.5, as fi ales euristica h2.

Asadar, voi continua analiza cu euristica h1, dar voi continua analiza pentru ambele constante, alegand mereu varianta mai buna. Pentru buget, voi alege bugetul maxim de 20000, deoarece algoritmul se opreste din construit in momentul in care gaseste solutie, deci numarul de stari descoperite este doar limitat de budget, nu si afectat.

5.5 Rezultate obtinute

Toate valorile de mai jos sunt rezultatele mediei a 20 de rulari consecutive. Aceste numere pot fi deduse si din grafice, dar am vrut sa le mentionez si in format scris pentru a justifica si mai bine alegerile facute.

Pentru cazul 1 cu buget 20000 si euristica 1:

c = 0.5:

- Sansa gasire solutie: 1.0
- Numar stari : 846.5
- Numar stari (solutie):: 846.5
- Lungime medie solutie: 5.4
- Timp rulare: 1.38s
- Timp rulare (solutie):: 1.38s

c = 0.1:

- Sansa gasire solutie: 0.95
- Numar stari : 2579
- Numar stari (solutie):: 1662
- Lungime medie solutie: 4.95
- Timp rulare: 2.71s
- Timp rulare (solutie):: 1.77s

Pentru cazul 2 cu buget 20000 si euristica 1:

c = 0.5:

- Sansa gasire solutie: 1.0
- Numar stari : 3748.5
- Numar stari (solutie):: 3748.5
- Lungime medie solutie: 7.3
- Timp rulare: 3.98s
- Timp rulare (solutie):: 3.98s

c = 0.1:

- Sansa gasire solutie: 0.75
- Numar stari : 8637
- Numar stari (solutie):: 4850
- Lungime medie solutie: 8.25
- Timp rulare: 9.07s
- Timp rulare (solutie):: 5.02s

Pentru cazurile mari, comparatia pentru cele 2 constante nu are rost, deoarece pt c = 0.5, de multe ori nu se gaseste solutie. Datele pentru cazurile 3 si 4 cu c = 0.1 sunt:

cazul 3:

- Sansa gasire solutie: 0.7
- Numar stari : 11033.55
- Numar stari (solutie): 7190.78
- Lungime medie solutie: 10.7
- Timp rulare: 11.52s
- Timp rulare (solutie):: 7.41s

cazul 4:

- Sansa gasire solutie: 0.2
- Numar stari : 18452.5
- Numar stari (solutie): 12262.5
- Lungime medie solutie: 5.4
- Timp rulare: 19.58s
- Timp rulare (solutie): 13.22s

6 Analiza Monte Carlo Tree Search si A*

6.1 Analiza timp de executie

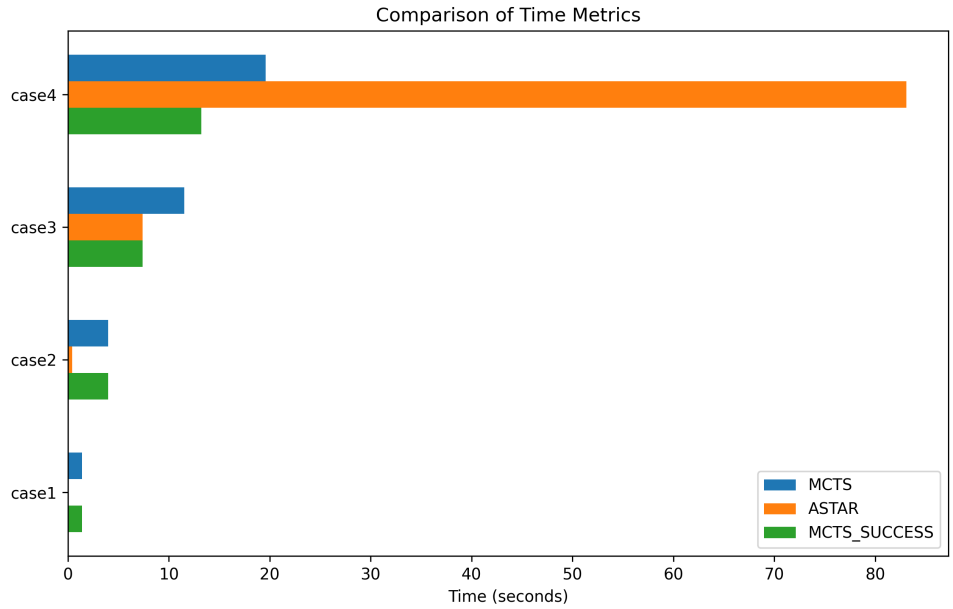

Figura 9: Comparatie timp executie MCTS si A*.

Uitandu-ne la graficele pentru timp, MCTS nu are un inceput bun. Chiar si atunci cand acesta gaseste solutie, nu reuseste sa fie mai rapid decat A* in cazul 4, iar la cazul 3 pare a fi un timp similar. De mentionat este faptul ca, pentru cazurile 3 si 4, sansele de gasire solutie sunt de 0.7 si 0.2, deci acesti timpi chiar daca par impresionanti si ar da impresia ca MCTS scaleaza mai bine, trebuie luat in calcul si faptul ca sansa de a gasi solutie scade semnificativ.

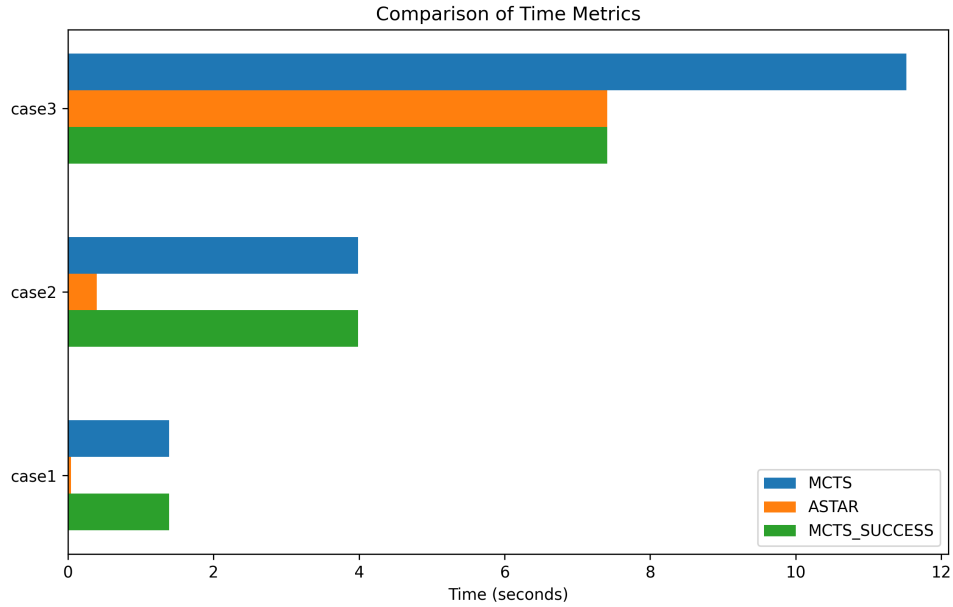

Figura 10: Comparatie timp executie MCTS si A* pe primele 3 cazuri.

Am aratat si acest grafic care are doar primele 3 cazuri, pentru a se vedea mai exact cat de rapid este A* fata de MCTS pentru primele cazuri, unde pe graficul mare nu se puteau observa

asa bine timpi pentru A*.

6.2 Analiza lungime solutiei

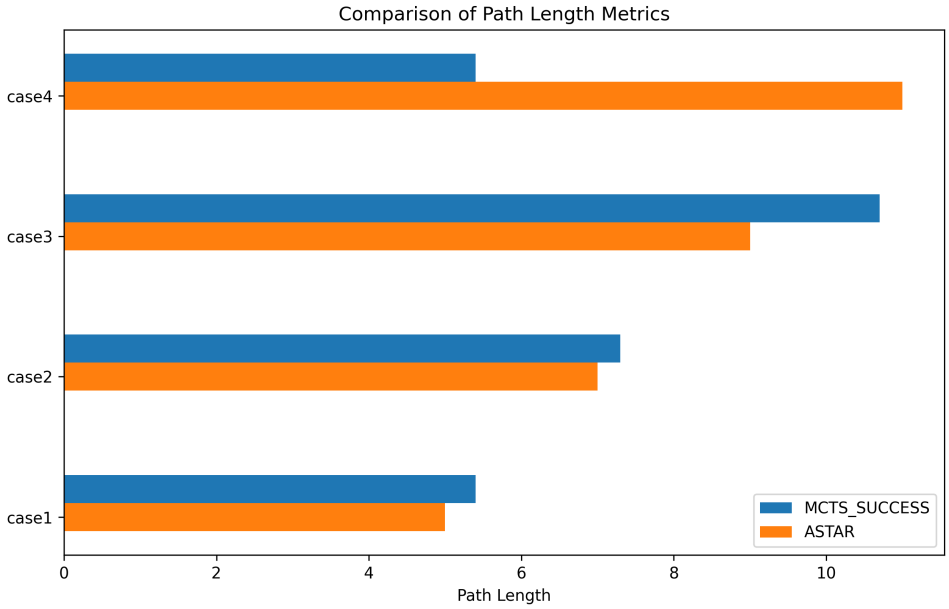

Figura 11: Comparatie lungime solutiei MCTS si A*.

Am inclus doar cazurile cand MCTS gaseste solutie pentru aceasta comparatie, pentru ca altfel datele ar fi parut eronate. Putem observa, pe langa faptul ca nu avem un castig pe partea timpului de executie, MCTS nu gaseste mereu nici macar solutia optima, ceea ce este si mai dezamagitor, mai ales ca nu reuseste acest lucru nici pe cazurile mici.

6.3 Analiza numar stari

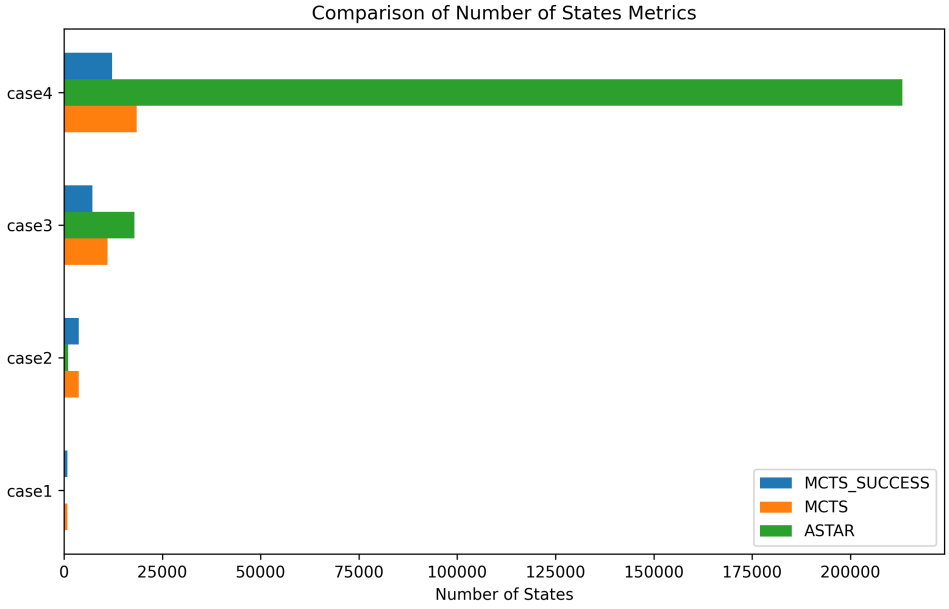

Figura 12: Comparatie numar stari MCTS si A*.

Numarul de stari la A* devine exponential mai mare cu cat distanta fata de solutie este mai mare, lucru care era de asteptat. Pentru cazul 4, A* dureaza foarte mult si evident, avem si un numar foarte mare de stari, iar MCTS nu gaseste mereu solutie, dar atunci cand gaseste (0.2 sansa), exploreaza putine stari. Am putea spune ca avem un castig aici pentru MCTS doar pentru cazul 4, dar din nou, trebuie luata in calcul si sansa mica sa gaseasca acea solutie, chiar daca timpul si numarul de stari sunt mai mic decat la A*.

Sa comparam primele 3 cazuri doar.

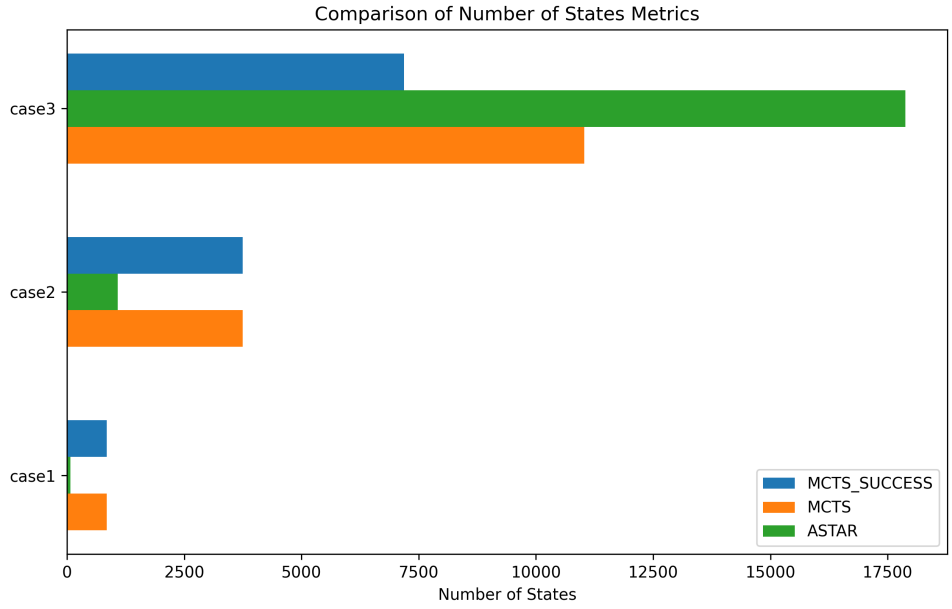

Figura 13: Comparatie numar stari MCTS si A*.

Pentru primele 2 cazuri, MCTS nu ofera un rezultat satisfacator, explorand mult mai multe stari decat A*, iar pentru cazul 3, desi numarul de stari este mai mic, acest lucru nu il face neaparat mai bun, deoarece are minusurile mentionate mai devreme(timp mai lung de expansionare stari, sansa sa nu gaseasca solutia, sansa ca solutia sa nu fie optima).

6.4 Concluzii analiza MCTS si A*

Personal as putea spune ca datorita faptului ca MCTS are la baza un factor de alegeri aleatorii, acesta nu este un algoritm tocmai optim pentru rezolvarea cubului rubic 2x2x2. Cu toate acestea, cu o euristica mai buna, tind sa cred ca ar putea scala mult mai bine si sa dea rezultate mult mai bune decat ceea ce am obtinut, mai ales pe un cub mai mare (3x3x3, etc).

7 Analiza Monte Carlo Tree Search si BFS Bidirectional

7.1 Analiza timp executie

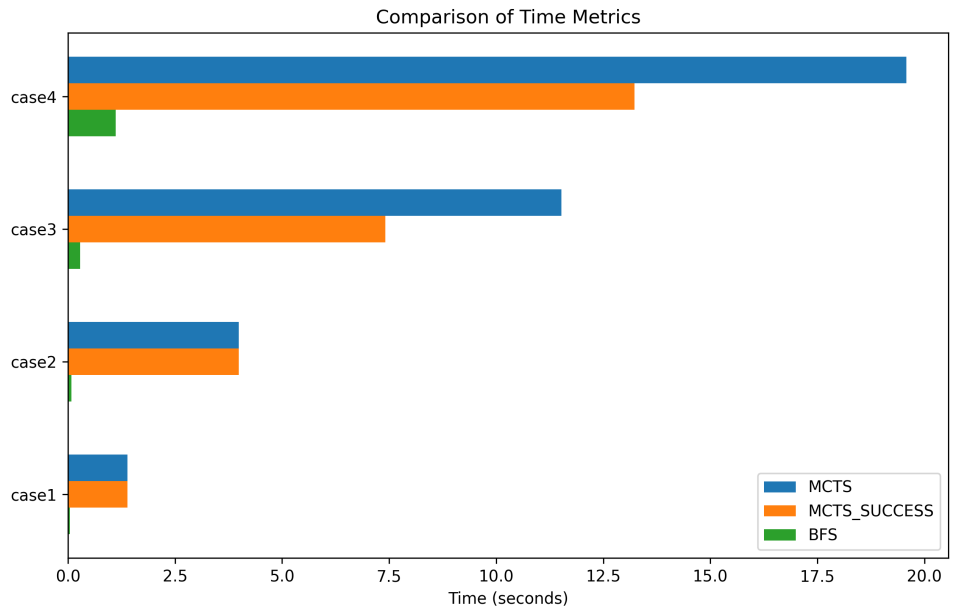

Figura 14: Comparatie timp executie MCTS si BFS.

Timpul pentru BFS cazul 1 nu se vede, acesta este acolo, dar este mult mai mic decat cel pt MCTS, 0.063s vs 1.38s, din aceasta cauza pare ca nu este pe grafic. Diferenta intre A* si BFS era una mare, dar acum este si mai mare. BFS bidirectional este incomparabil de rapid pentru pocket cube fata de MCTS.

7.2 Analiza lungime solutie

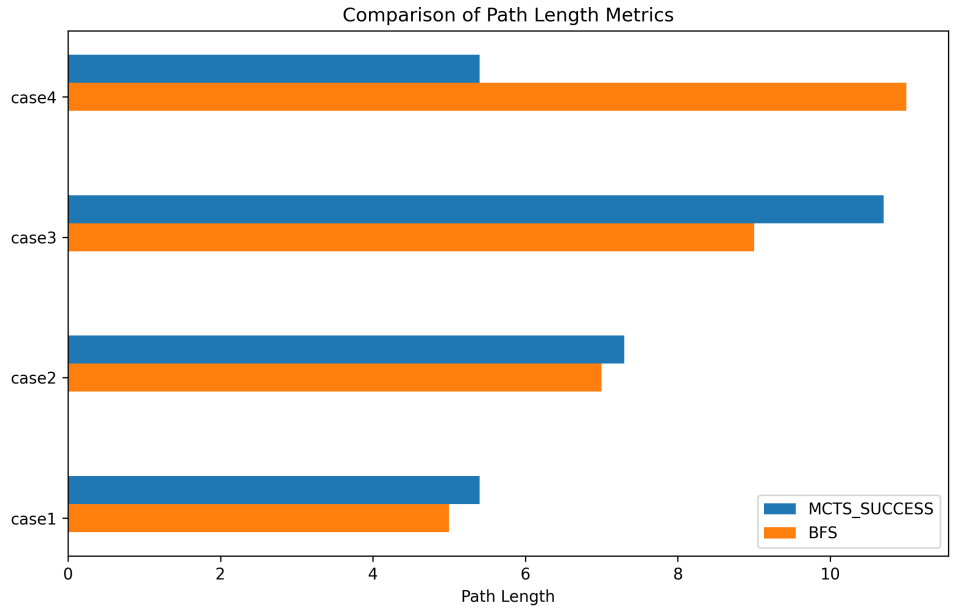

Figura 15: Comparatie lungime solutiei MCTS si BFS.

Din moment ce A* si BFS au solutii de lungimi egale, aceste grafice arata identic. Aceleasi concluzii ca la A*, dezamagitor ca nu gaseste solutia optima pe primele 3 cazuri.

7.3 Analiza numar stari

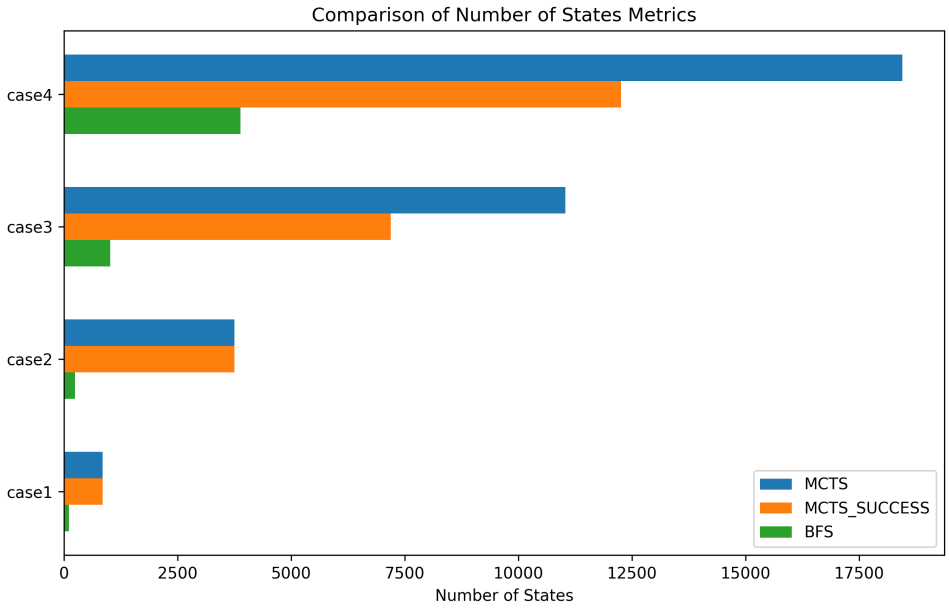

Figura 16: Comparatie numar stari MCTS si BFS.

BFS-ul bidirectional exploreaza foarte putine stari, are un cost redus de explorare al unei stari si gaseste solutii relativ optime, acest lucru putand fi observat si din numarul de stari reduse. Timp mic de explorare, numar mic de stari, impreuna duc la un algoritm rapid.

7.4 Concluzii analiza MCTS si BFS Bidirectional

Daca ar fi sa aleg un algoritm pe care sa-l implementez intr-un robot pentru a rezolva un pocket cube, cel mai probabil ar fi BFS. Algoritmul este mai rapid, complet consistent si consuma putina memorie. Dar totusi, BFS Bidirectional are o limitare, nu poate fi imbunatatit mai mult de atat. Sa vedem cum vor arata rezultatele cand avem euristici si mai bune.

8 Pattern database

8.1 Creare euristici

Pentru a crea Pattern Database-ul, am folosit o metoda foarte rudimentara, construim toate cuburile care sunt la maxim 7 mutari distanta dupa care

- Daca starea nu se afla in database, o adaugam cu distanta egala cu numarul de mutari facute pe cub.
- Altfel, ignoram starea.

Din acest Pattern Database am creat doua euristici:

1. Euristica h3, care intoarce fie valoarea din Patter Database daca se afla starea cubului in ea, altfel intoarce valoarea euristicii H1.
2. Euristica h4, care intoarce fie valoarea din Patter Database daca se afla starea cubului in ea, altfel intoarce valoarea 14.

8.2 Analiza Monte Carlo Tree Search folosind h3 si h4

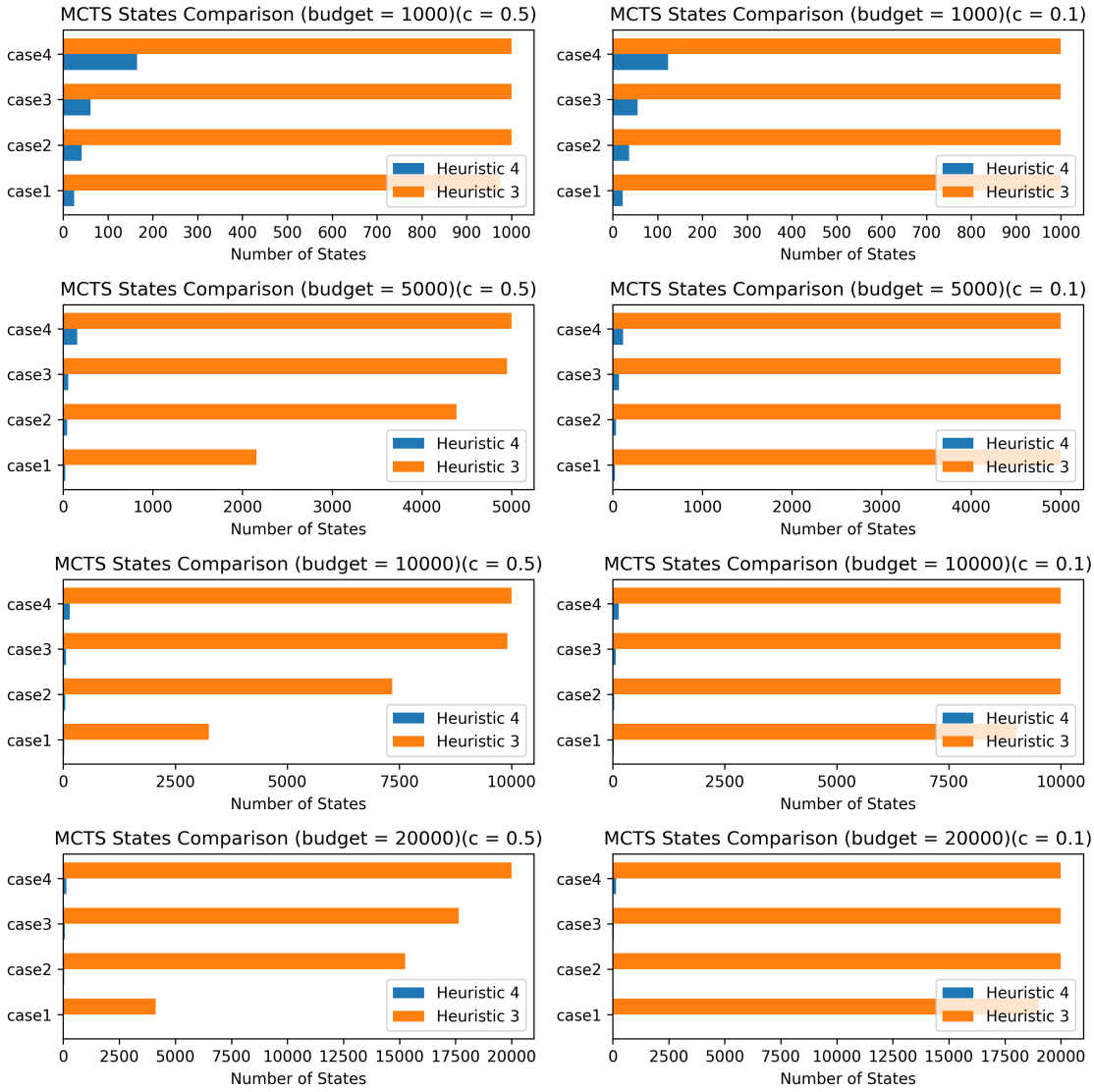

Figura 17: Comparatie numar stari euristici h3 si h3 MCTS.

Am sa includ graficul doar pentru numarul de stari, deoarece toate informatiile necesare se pot deduce de aici. Este clar ca h4 este mult mai buna decat h3. De asemenea, din acest grafic, se

poate observa ca pentru eurisitca h3, constanta c = 0.5 este mai buna, iar pentru euristica h4, constanta c = 0.1 are rezultate mai bune, deci voi continua comparatia folosind aceste constante. Bugetul pentru h3 va fi de 20000, iar pentru h4 este suficient sa luam 1000.

8.3 Analiza Monte Carlo Tree Search si A* folosind h3

8.3.1 Rezultate obtinute

Pentru euristica h3, rezultatele obtinute sunt foarte nesatisfactoare. Monte Carlo Tree Search nu a reusit sa rezolve nici macar o data cazul 4 de test, deci voi include compartia doar pentru primele 3 cazuri pentru h3.

Acestea sunt rezultatele pentru MCTS:

cazul 1:

- Sansa gasire solutie: 0.95
- Numar stari : 4857.05
- Numar stari (solutie): 4060
- Lungime medie solutie: 6.05
- Timp rulare: 5.09s
- Timp rulare (solutie):: 4.24s

cazul 2:

- Sansa gasire solutie: 0.7
- Numar stari : 12115.85
- Numar stari (solutie): 8736.92
- Lungime medie solutie: 7.71
- Timp rulare: 12.66s
- Timp rulare (solutie): 9.05s

cazul 3:

- Sansa gasire solutie: 0.2
- Numar stari : 19042.25
- Numar stari (solutie): 15211.25
- Lungime medie solutie: 9.0
- Timp rulare: 19.66s
- Timp rulare (solutie):: 15.67s

cazul 4:

- Sansa gasire solutie: 0.0
- Numar stari : 20000.0
- Numar stari (solutie): 0
- Lungime medie solutie: 0
- Timp rulare: 21.029s
- Timp rulare (solutie): 0s

Practic, aceasta euristica are rezultate mai proaste decat h1. Acest lucru poate fi explicat prin faptul ca euristica h1 nu intoarce niciodata valori mai mari decat 7, si atunci pattern database-ul ajunge sa nu fie utilizat la potential maxim.

8.3.2 Analiza timp executie

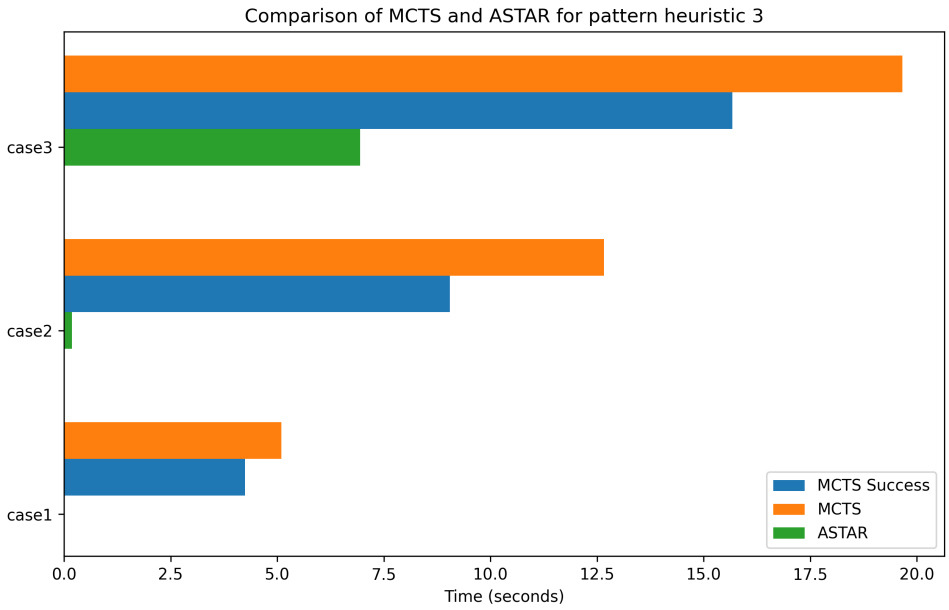

Figura 18: Comparatie timp executie MCTS si A* folosind h3.

Un prim grafic ne arata ca, pe aceasta euristica, A* inca are un avantaj semnificativ in ceea ce priveste timpul de executie. Acest lucru cred ca se datoreaza faptului ca A* pune din start toate miscarile posibile in coada de prioritati, si asa sunt sanse mari sa gasim miscari care sunt in pattern database si care au valori mai mici de 3.

8.3.3 Analiza lungime solutie

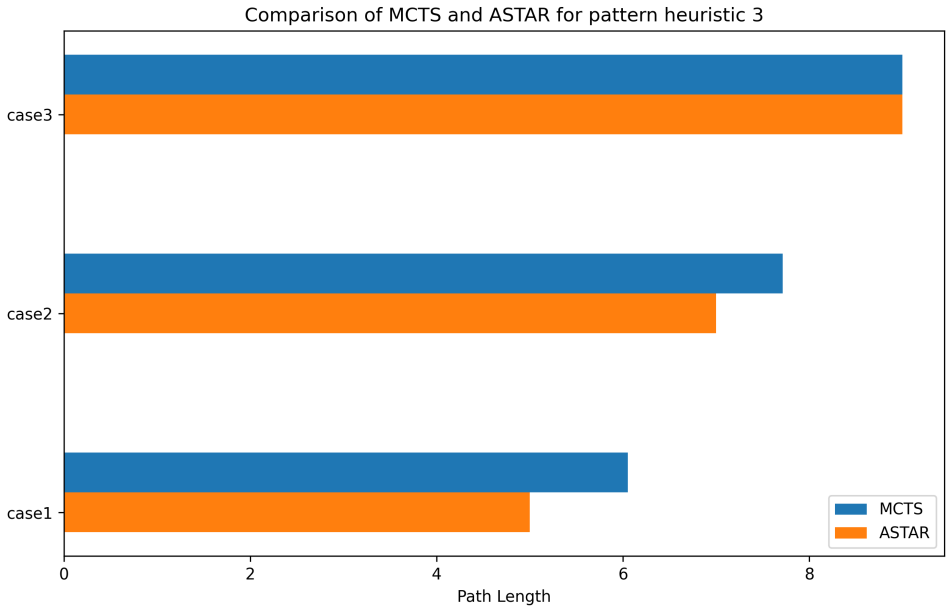

Figura 19: Comparatie lungime solutie MCTS si A* folosind h3.

Acest grafic ne arata, ca din nou, MCTS nu gaseste mereu solutia optima, singura exceptie fiind pentru cazul 3. Nu ma asteptam la imbunatari, tinand cont de statisticele mentionate mai

sus.

8.3.4 Analiza numar stari

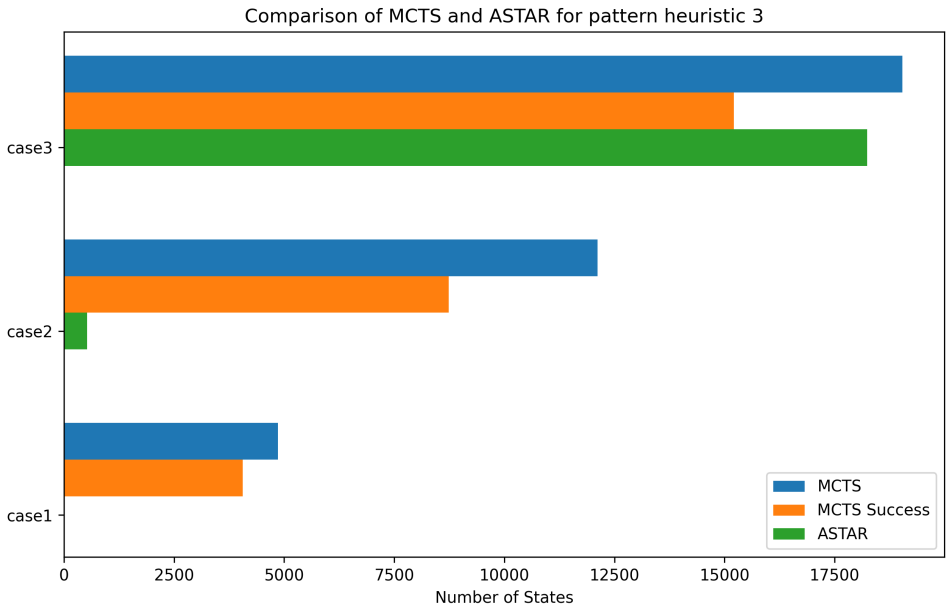

Figura 20: Comparatie numar stari MCTS si A* folosind h3.

Pe langa faptul ca A* exploreaza starile mai repede, pentru aceste cazuri de test exploreaza si mai putine stari. Din nou, acest lucru era de asteptat considerand rezultatele anterioare.

8.4 Analiza Monte Carlo Tree Search si A* folosind h4

Dupa cum s-au vazut in primele grafice din aceasta sectiune, h4 este o euristica foarte buna si interesanta, chiar daca nu este admisibila. Este cu siguranta cea mai buna euristica testata, si acest lucru se datoreaza faptului ca incurajeaza sa exploram stari pentru care stim exact la cate mutari distanta de solutie suntem.

Voi analiza aceasta euristica deoarece are cele mai satisfactoare rezultate.

8.4.1 Analiza timp executie

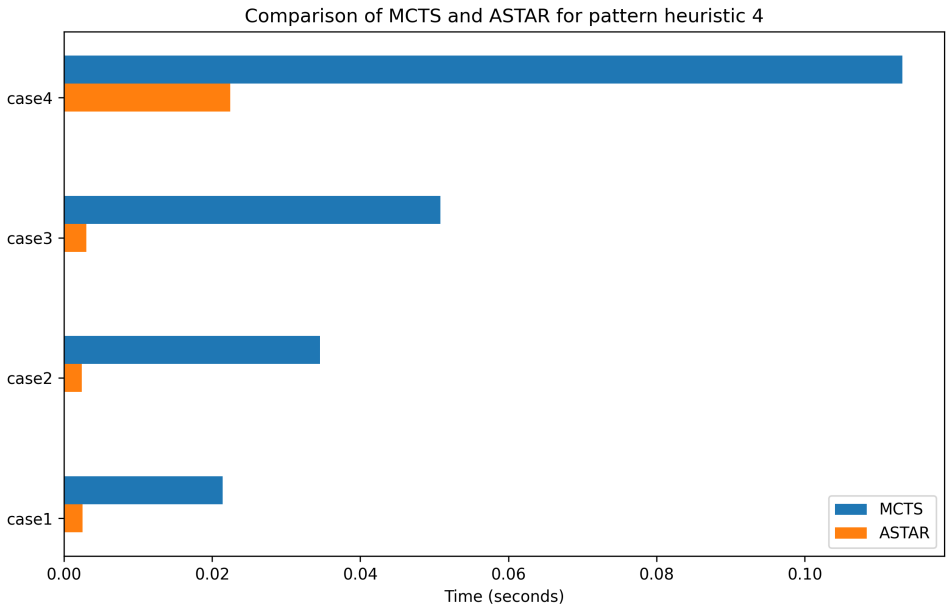

Figura 21: Comparatie timp executie MCTS si A* folosind h4.

MCTS are un speed-up considerabil, si cel mai important, o probabilitate de gasire a solutiei de 1.0 indiferent de caz, prima data cand se intampla asta. Cu toate aceste imbunatatiri, pentru aceasta euristica, A* tot este mai rapid. Totusi, MCTS nu mai pare un algoritm complet inferior in acest caz, ambii algoritmi avand timpi mai mici de jumatate de secunda.

8.4.2 Analiza lungime solutiei

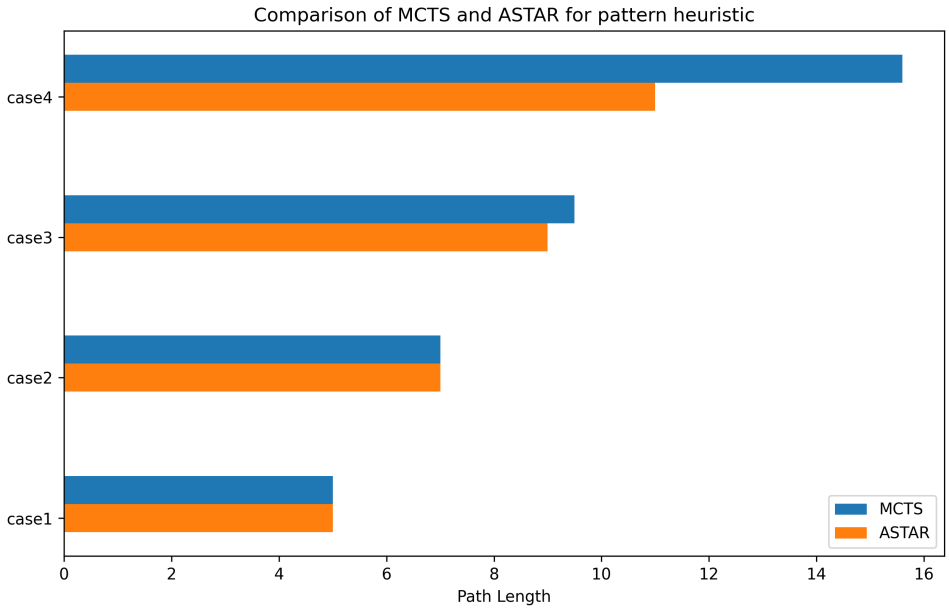

Figura 22: Comparatie lungime solutie MCTS si A* folosind h4.

Nu pot sa-mi explic cum, dar MCTS nu gaseste mereu o solutie optima nici macar cu aceasta euristica. Probabil asta se datoreaza factorului de alegeri aleatorii. Totusi, este relativ dezamagitor.

A* din nou pare a fi algoritmul superior pentru aceasta problema.

8.4.3 Analiza numar stari

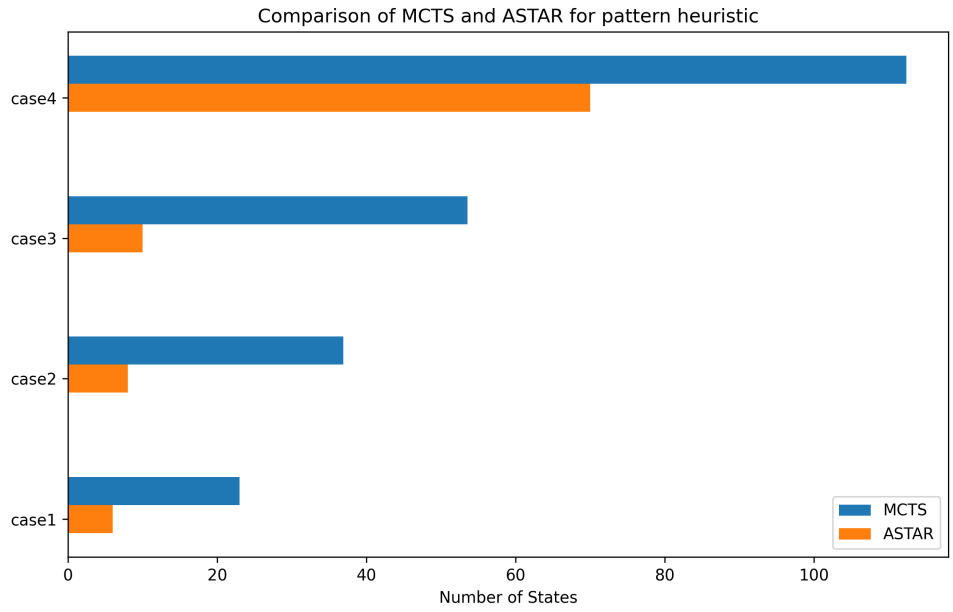

Figura 23: Comparatie numar stari MCTS si A* folosind h4.

Speram ca macar aici sa vad un avantaj pentru MCTS. Din pacate, chiar si in ceea ce priveste numarul de stari, A* reuseste sa fie mai eficient. Partea de alegeri aleatorii este prea defectuoasa pentru algoritmul MCTS pentru aceasta problema.

9 Concluzii

Algoritmul A* poate fi foarte rapid atunci cand euristica este una buna, fiind cel mai performant din toate punctele de vedere cand euristica este cat mai aproape de realitate. Totusi, pentru rezultatele obtinute pentru euristica h4, este nevoie de o euristica de tipul Pattern Database, care ocupa destul de mult memorie.

Luand asta in calcul, BFS-ul bidirectional este un algoritm cu rezultate excelente, atat din punct de vedere al timpului de executie cat si al memoriei utilizate, si care gaseste si solutia optima. Singurul dezavantaj la BFS la care ma pot gandi este modul cum scaleaza in momentul in care problema are mult mai multe stari, de exemplu un cub de 3x3x3, si tind sa cred ca in acest nici BFS nici A* nu scaleaza la fel de bine, ci MCTS are cel mai mare potential de a scala si a gasi solutia.

Bibliografie

Bibliografie

[1] Laboratoarele si cursurile de Inteligenta Artificiala, CTI, ACS, UPB.