INTELIGENȚĂ ARTIFICIALĂ - TEMA 1
ALGORITMI DE CĂUTARE

Cătălin-Alexandru Rîpanu, 341C3
Facultatea de Automatică și Calculatoare
Universitatea Națională de Știință și Tehnologie Politehnica București
catalin.ripanu@stud.acs.upb.ro

10 Decembrie 2023

ABSTRACT

Această temă propune implementarea, folosind diverse tehnici, a unor algoritmi (informați și neinformați) de căutare ce au fost prezentați la cursul de **Inteligență Artificială** cu scopul de a rezolva problema **Cubului lui Rubik** de dimensiune **2x2x2** ce posedă un spațiu de căutare cu aproximativ 3.7×10^6 **stări posibile**.

1 Introducere

Lucrarea de față își propune analiza performanțelor algoritmilor de căutare **A***, **Bidirectional BFS** și **Monte Carlo Tree Search** folosind descrieri **succinte**, **concrete** și **tehnice** ce au drept suport **tabele** și **grafice elocvente**. Fiecare secțiune din lucrare va trata diverse comparații între anumiți algoritmi din cei 3 menționați anterior folosind criteriile următoare: **timp de rulare**, **numărul de stări expandate în căutare**, **lungimea soluției găsite** și **memoria utilizată pe parcursul execuției** în cazul celor 4 teste inițializate în fișierul **tests.py**.
Motivația din spatele acestei analize este dorința de a descoperi, experimental, dacă un algoritm informat (A*, MCTS) **"învinge"** sau nu un algoritm neinformat, ce nu folosește o euristică (Bidirectional BFS), în cazul acestei probleme specifice. Desigur, în cazul algoritmilor informați se vor folosi 3 euristici: **distanța Manhattan**, **o funcție care află numărul de rotații necesare pentru a ajunge la starea finală** și **o funcție care folosește tehnica Pattern Database**.

2 Cerința 1: A* și Bidirectional BFS

În această secțiune se vor detalia diferențele de performanță, respectiv de implementare, în cazul celor 2 algoritmi menționați mai sus, astfel încât să se poată formula o concluzie în ceea ce privește alegerea modalității **eficiente** de rezolvare a problemei, în cazul în care dispunem doar de **A*** și **Bidirectional BFS** pe un sistem **clasic** de calcul cu resurse limitate.
Euristica **admisibilă** propusă pentru **A*** poartă denumirea de **rotation_cube()** și aceasta calculează numărul de rotiri **necesare** pentru a putea aduce fiecare mini-față pe fața corectă (dacă mini-fața respectivă este pe fața opusă feței iei, atunci se adaugă valoarea 2 în calculul final, dacă este pe o față oarecare diferită de cea opusă, se adaugă valoarea 1, iar dacă este pe fața corectă, se adaugă 0).

Se va observa, în cele ce urmează, faptul că **Bidirectional BFS** domină algoritmul **A*** în ceea ce privește timpul de execuție pentru cele 2 teste mai **mari**, și anume **case3** și **case4**, însă partea interesantă este că **A*** are un timp mai bun pentru cele 2 teste **mici**, **case1** și **case2**.

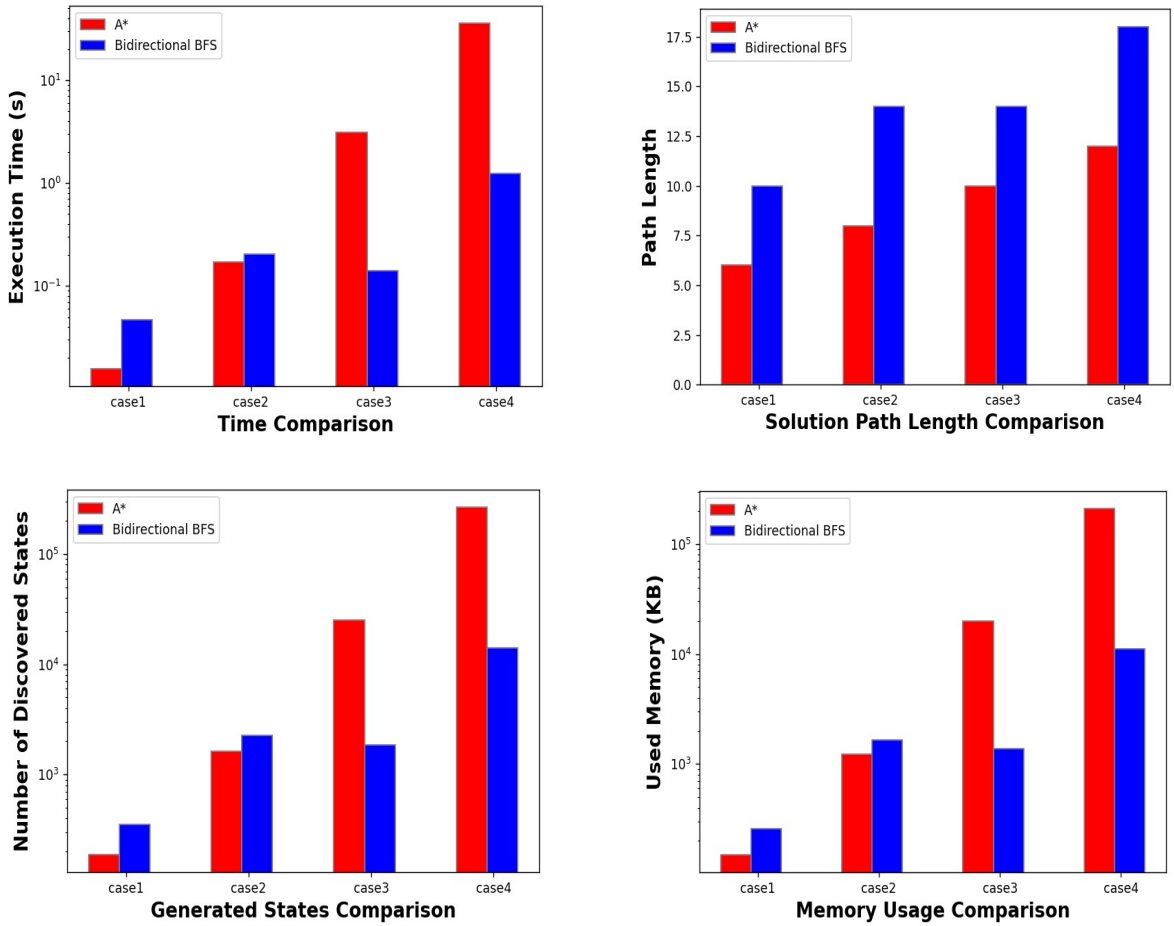

Figure 1: Performanțele algoritmilor A* și Bidirectional BFS

Având la bază cele 4 grafice ce ilustrează eficiența și consumul de resurse în cazul algoritmilor respectivi, se poate **afirma** cu certitudine faptul că **Bidirectional BFS** este **mai potrivit** decât **A*** atunci când vorbim de teste cu **multe mișcări / rotații**, însă **A*** are un avantaj atunci când cubul ce trebuie rezolvat a fost **rotit de mai puține ori** (găsește, de asemenea, calea mai mică spre soluție, comparativ cu **Bid. BFS**, descoperind mai puține stări și consumând mai puțină memorie). Mai mult, se poate constata că **Bidirectional BFS** întoarce, la fiecare test, căi mai mari spre soluție, ceea ce era de așteptat având în vedere că este un **algoritm de căutare neinformat**, spre deosebire de **A***. În această situație particulară nu putem spune că un algoritm este mai bun decât altul dacă nu cunoaștem dimensiunea intrării (adică numărul de rotiri asupra cubului inițial). Se mai poate spune că **Bid. BFS** este prea mult pentru primele teste.

3 Cerința 2: Monte Carlo Tree Search

Această parte propune analiza calitativă a algoritmului de căutare informată **UCT** (Upper Confidence Bound for Trees) din familia de algortimi **Monte Carlo Tree Search**. Un lucru important de menționat este faptul că de interes a fost o variantă modificată a algoritmului **UCT** ce presupune eliminarea părții în care arborele rezultat era construit dintr-o altă stare (motivația este că nu avem o problemă cu mai mulți jucători în contextul curent). Implementarea acestuia s-a realizat folosind 2 euristici: **rotation_cube()** (admisibilă) și **manhattan_distance()** (neadmisibilă). Merită menționat aspectul conform căruia au existat, în total, **64 de combinații** posibile, fiecare combinație rulându-se de **20 de ori**, timpul totat de execuție pentru cele **1280 de iterații** fiind în jur de 6h și 20 de min (un timp nu prea plăcut în contextul altor teme de implementat în paralel). Mai jos se regăsesc cele 12 grafice aferente comparațiilor, și anume determinarea **celor mai bune** valori pentru **bugetul** și **coeficientul experimental C** împreună cu aflarea **celei mai bune euristici** folosind datele numerice anterioare. Ultimul pas constă în a compara **A***, **Bidirectional BFS** și **MCTS-ul** cu cele mai optime valori pentru a găsi **soluția eficientă** în acest context particular de joc cu un singur jucător.

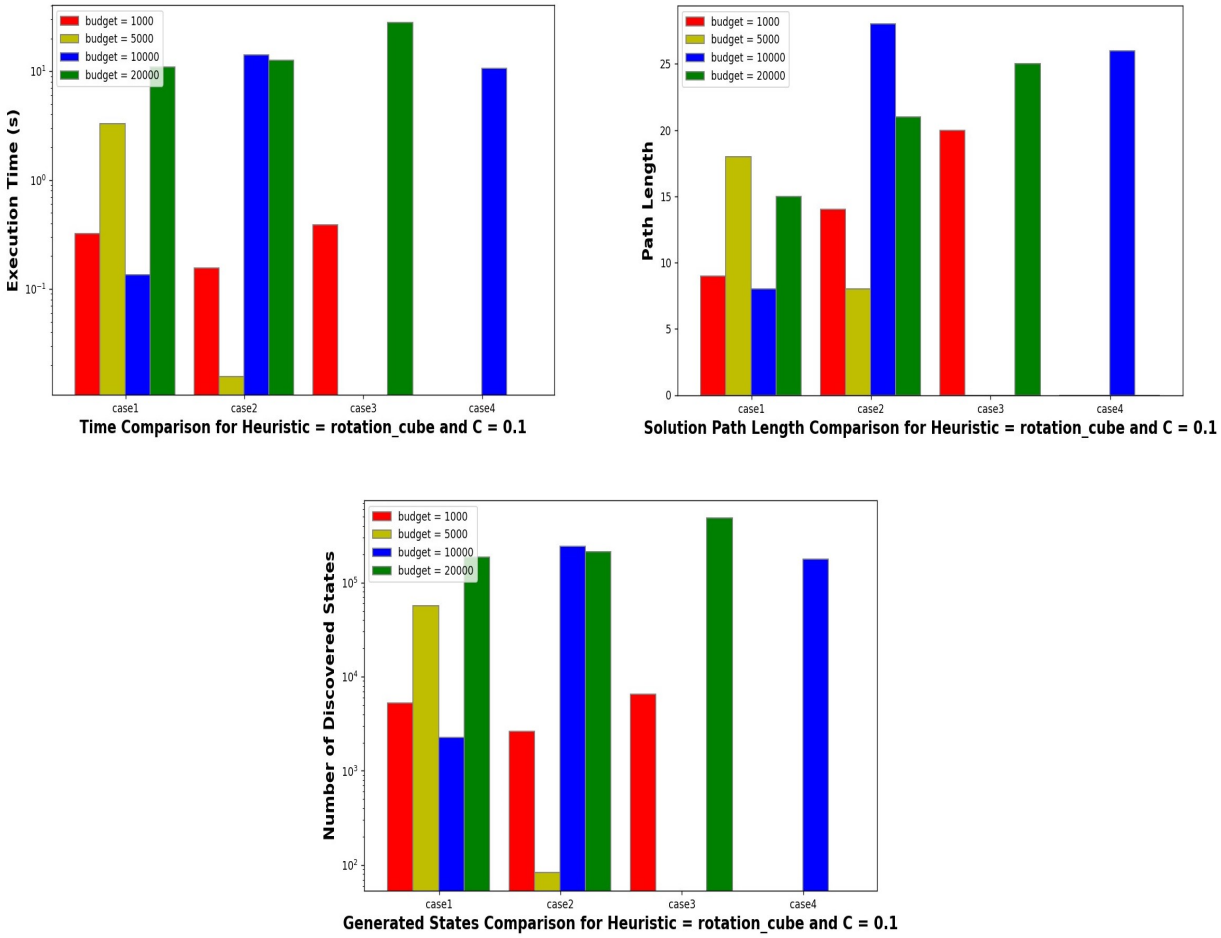

Figure 2: Performanța MCTS-ului cu C=0.1 și euristica rotation_cube

Buget	Caz	Timp	Lungime Soluție	Număr de stări descoperite
1000	1	0.3203125 s	9	5223
1000	2	0.15625 s	14	2648
1000	3	0.390625 s	20	6461
1000	4	0 s	0	0
5000	1	3.30859375 s	18	56865
5000	2	0.015625 s	8	83
5000	3	0 s	0	0
5000	4	0 s	0	0
10000	1	0.135416 s	8	2262
10000	2	14.09375 s	28	240448
10000	3	0 s	0	0
10000	4	10.53645 s	26	178259
20000	1	10.92187 s	15	184853
20000	2	12.51562 s	21	210992
20000	3	27.88281 s	25	482135
20000	4	0 s	0	0

Așa cum este de așteptat, se observă faptul că algoritmul nu găsește mereu soluție pentru anumite cazuri, la unele valori pentru buget, deoarece este bazat pe alegeri de acțiuni aleatoare.

Pentru această combinație de date, se poate spune că doar la valori mari de buget (10000 și 20000) MCTS-ul are șanse să rezolve ultimele 2 teste ce au un nivel de complexitate mai mare (asta nu înseamnă că un buget mare este garanția conform căreia se va găsi mereu o soluție pentru testele mici, se poate vedea că, de exemplu, pentru un buget de 10000 nu se găsește soluție pentru **case3**, dar se găsește la un buget de 10 ori mai mic, conform celor 3 diagrame).
Alt lucru de menționat este ideea că s-a luat decizia de **a nu mai afișa o bară** specifică unui buget dacă nu s-a găsit o soluție pentru a putea extrage timpul ei de căutare în prima diagramă din stânga. De interes, în acest moment, este cercetarea celorlalte combinații pentru a vedea pentru **ce buget** și pentru **ce constantă C** algoritmul reușește să rezolve toate cazurile de test, într-un **timp optim**.

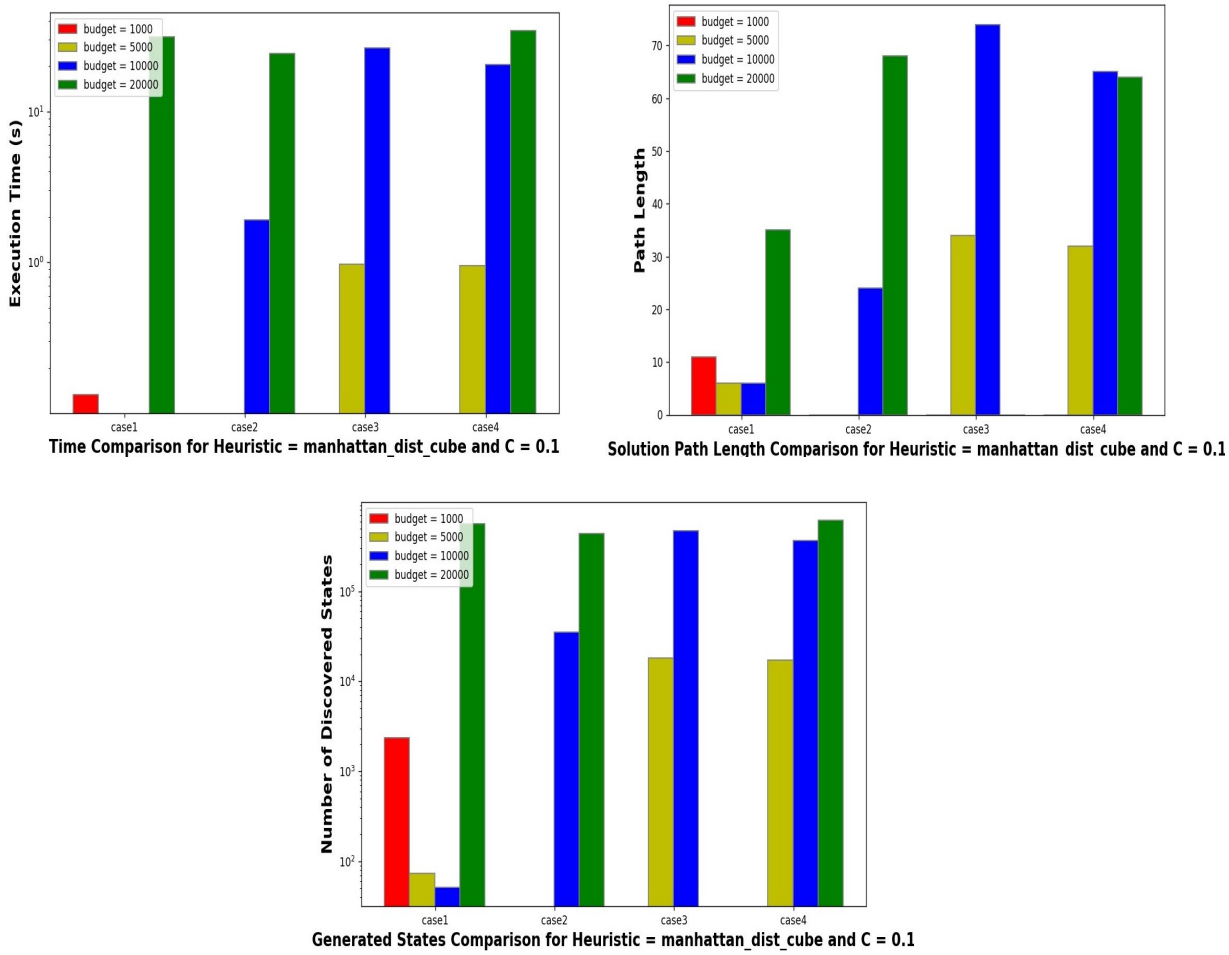

Figure 3: Performanța MCTS-ului cu C=0.1 și euristica manhattan_distance

Buget	Caz	Timp	Lungime Soluție	Număr de stări descoperite
1000	1	0.13281 s	11	2375
1000	2	0 s	0	0
1000	3	0 s	0	0
1000	4	0 s	0	0
5000	1	0.00001 s	6	73
5000	2	0 s	0	0
5000	3	0.968750 s	34	18105
5000	4	0.953125 s	32	17235
10000	1	0.00001 s	6	51
10000	2	1.921875 s	24	35332
10000	3	26.29687 s	74	471211
10000	4	20.45312 s	65	370153
20000	1	31.1250 s	35	559572
20000	2	24.3281 s	68	440104
20000	3	0 s	0	0
20000	4	34.2343 s	64	613882

Uzitând euristica neadmisibilă **manhattan_distance** s-a produs un caz excepțional, și anume că folosind bugetul cu valorile 5000 și 10000 algoritmul s-a terminat aproape instant, funcția built-in **process_time_ns()** din Python nu a

reușit să ofere o valoare diferită de 0.0 s. Revenind la acest caz, se poate constata, conform diagramelor, că bugetul cu valorile 10000 și 20000 pare să domine atunci când ne referim la a găsi o soluție în toate cazurile de test. Deoarece 10000 < 20000, bugetul de interes rămâne, în această situație, 10000, iar valoarea C rămâne, acum, 0.1, cu euristica **manhattan_distance**. Mai avem, acum, analizarea / verificarea celeilalte valori în cazul constantei experimentale C în raport cu cele 2 euristici.

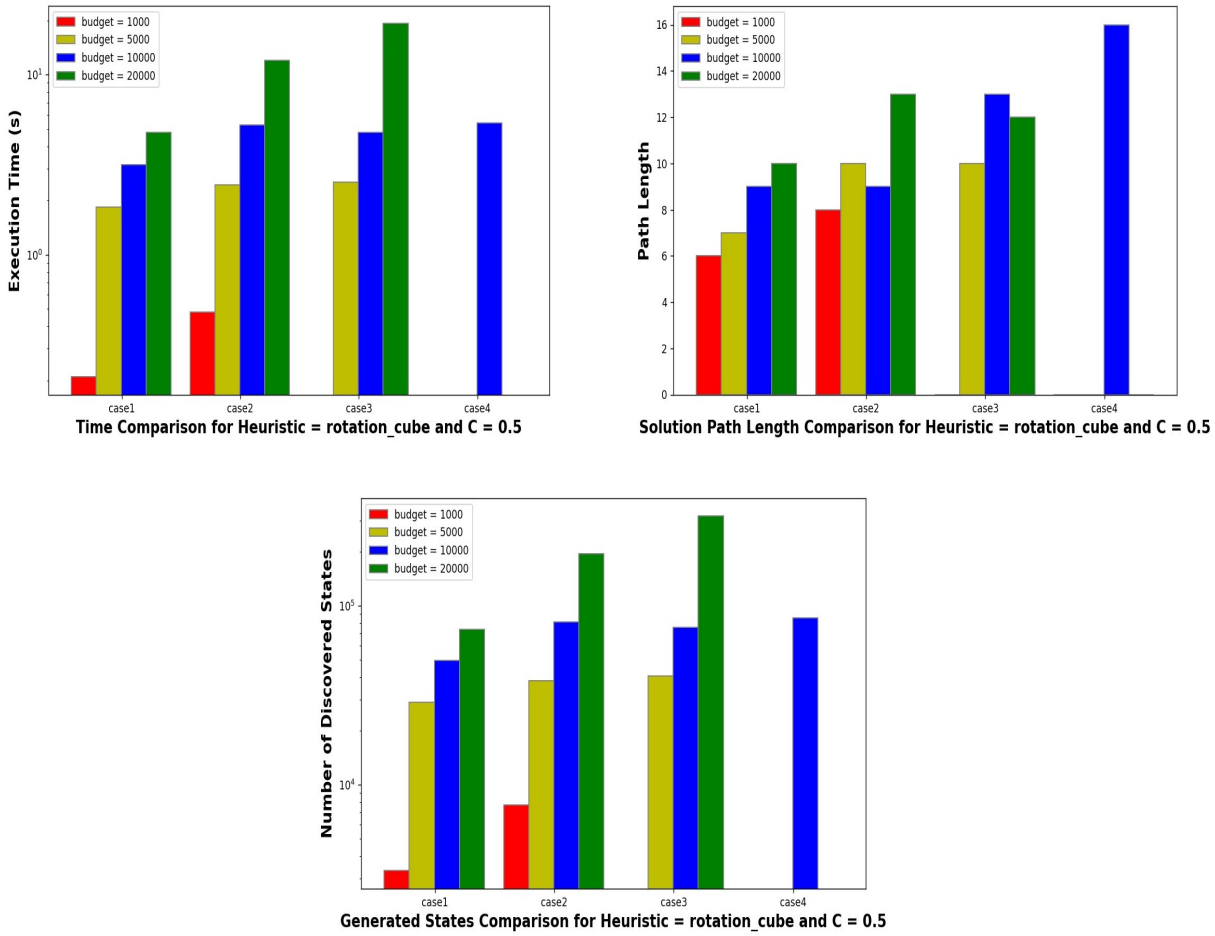

Figure 4: Performanța MCTS-ului cu C=0.5 și euristica rotation_distance

Buget	Caz	Timp	Lungime Soluție	Număr de stări descoperite
1000	1	0.209375 s	6	3316
1000	2	0.479166 s	8	7735
1000	3	0 s	0	0
1000	4	0 s	0	0
5000	1	1.83125 s	7	28901
5000	2	2.43489 s	10	38276
5000	3	2.53125 s	10	40510
5000	4	0 s	0	0
10000	1	3.15885 s	9	49603
10000	2	5.27148 s	9	81580
10000	3	4.79296 s	13	75589
10000	4	5.40625 s	16	85839
20000	1	4.78808 s	10	73605
20000	2	11.9750 s	13	194971
20000	3	19.2343 s	12	317542
20000	4	0 s	0	0

Cu valoarea de 0.5 pentru C se poate observa o îmbunătățire cu privire la rata de succes a algoritmului, mai ales pentru un buget de 10000 (întrucât la un astfel de buget există o soluție pentru orice tip de test). Se poate afirma cu siguranță

faptul că pentru această euristică admisibilă **bugetul optim** este cel de 10000 și valoarea experimentală ce maximizează **potențialul MCTS-ului** este **C=0.5**. În ultima analiză se va stabili și euristica potrivită astfel încât să se poată formula o **concluzie finală** pentru **prima etapă**. Aici se poate vedea cum un buget **mai mare decât celălalt** nu a reușit să găsească o soluție pentru problema dată.

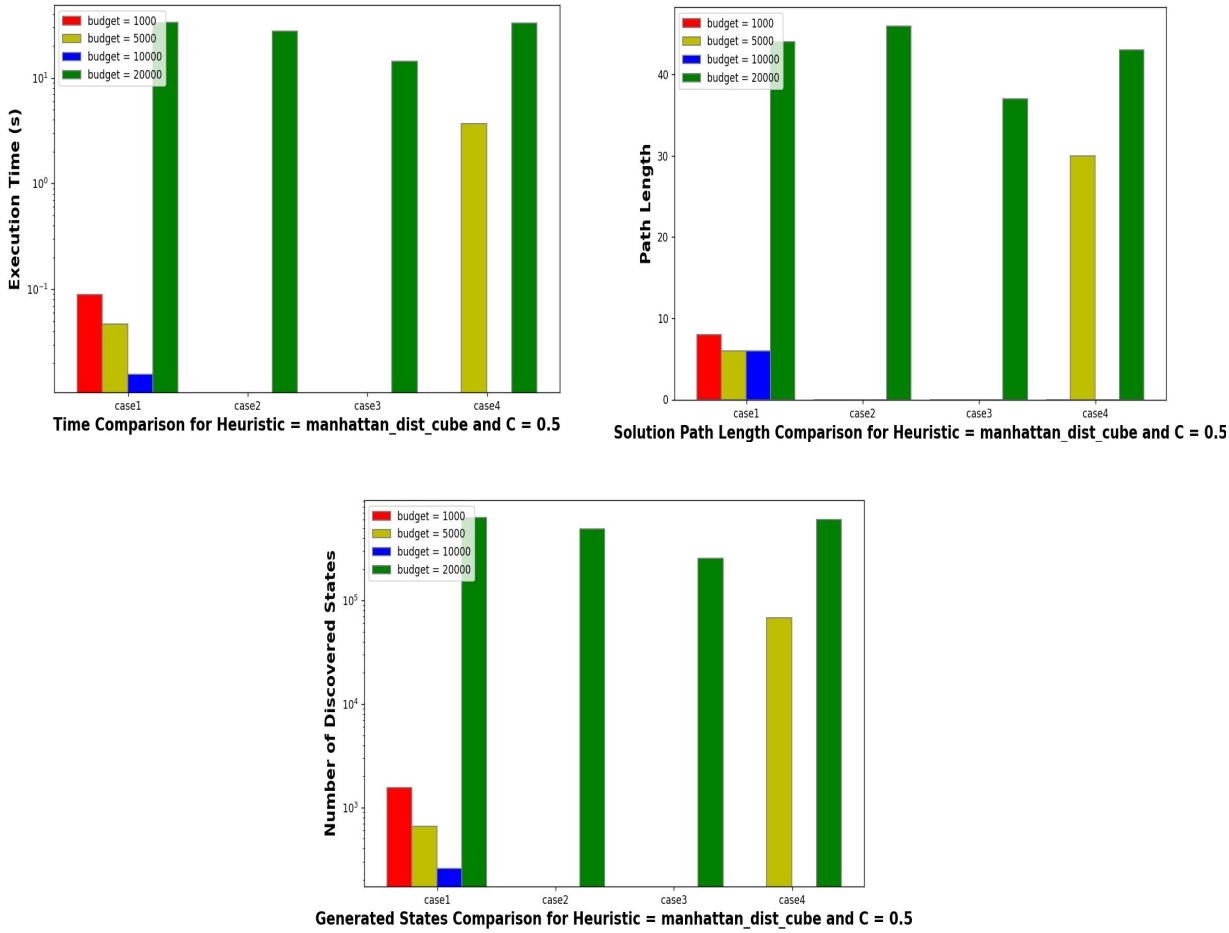

Figure 5: Performanța MCTS-ului cu C=0.5 și euristica manhattan_cube

Buget	Caz	Timp	Lungime Soluție	Număr de stări descoperite
1000	1	0.088541 s	8	1565
1000	2	0 s	0	0
1000	3	0 s	0	0
1000	4	0 s	0	0
5000	1	0.046875 s	6	659
5000	2	0 s	0	0
5000	3	0 s	0	0
5000	4	3.68750 s	30	68062
10000	1	0.015625 s	6	257
10000	2	0 s	0	0
10000	3	0 s	0	0
10000	4	0 s	0	0
20000	1	33.28125 s	44	628002
20000	2	27.60937 s	46	490726
20000	3	14.32031 s	37	253682
20000	4	32.82812 s	43	600657

Având diagramele de mai de sus la dispoziție, se poate extrage informația conform căreia bugetul cu valoarea de 20000, pentru un C=0.5, permite **cea mai mare rată de succes cu un timp decent**, de ordinul zecilor de secunde. Se vede cu

simplitate faptul că celelalte valori de buget nu prezintă un interes de o mare intensitate având în vedere că nu excelează în contextul rezolvării problemei date.

În concluzie, având în vedere toate aspectele expuse mai sus, folosind **lungimea căii soluției** drept criteriu / factor decisiv și important, putem spune că **cel mai potrivit buget** pentru acest MCTS adaptat acestei probleme este **10000**, **cea mai potrivită valoare** pentru constanta experimentală este **0.5**, iar **cea mai potrivită euristică**, care este și **admisibilă**, este **rotation_cube**, aceasta, fiind, încheierea primei etape. Intuitiv, era de așteptat ca valoarea de buget să fie una mare deoarece MCTS este un algoritm cu **alegeri aleatoare**, iar problema Cubului lui Rubik de dimensiune 2x2x2 are un număr mare de stări posibile în **spațiul de căutare**.

În etapa a doua se vor analiza aspectele legate de **eficiență** în rezolvare / consumarea resurselor pentru algoritmii **A***, **Bidirectional BFS** și **MCTS (cu valorile găsite)**.

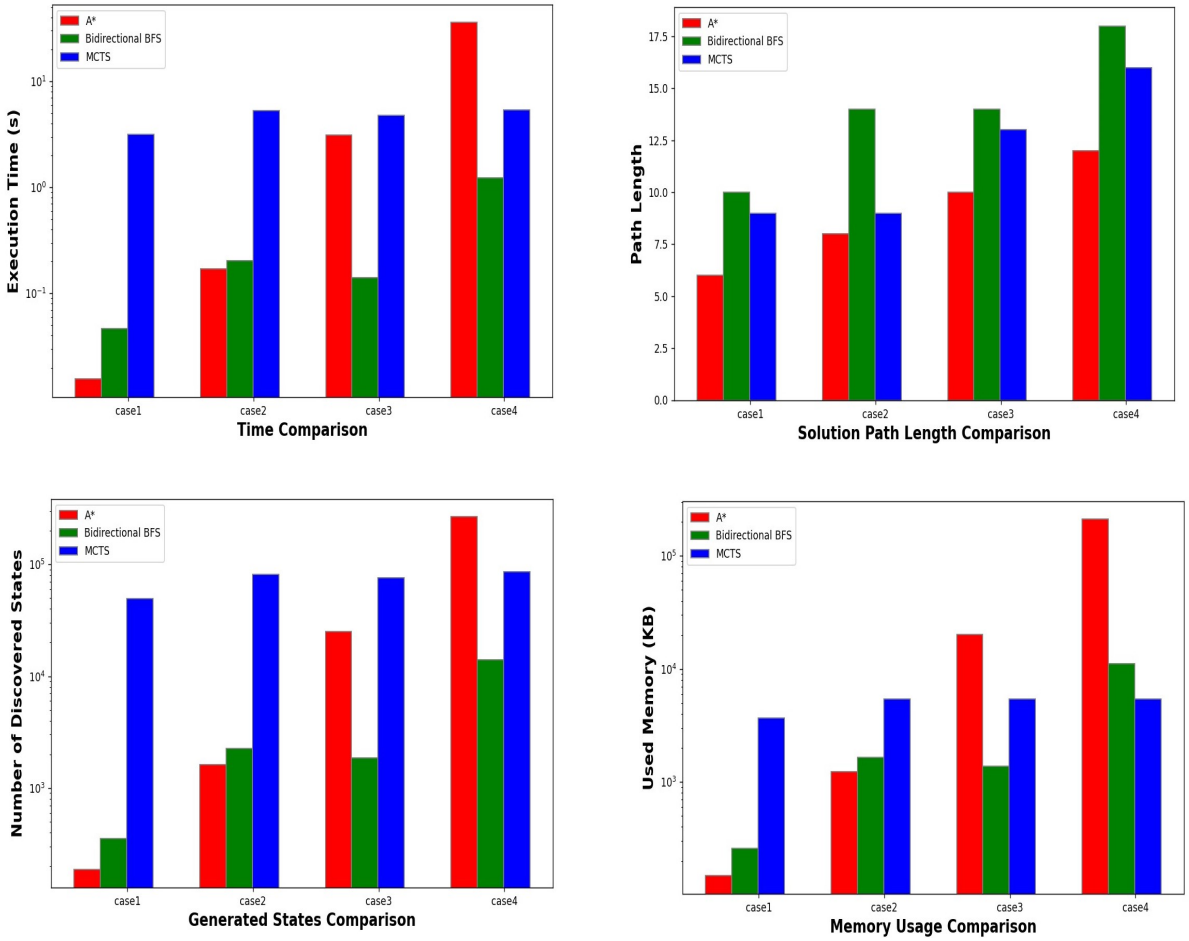

Figure 6: Performanțele algoritmilor A*, Bidirectional BFS și MCTS cu C=0.5 și euristica manhattan_cube

Algoritm	Caz	Timp	Lungime Soluție	Număr de stări descoperite	Memory
A*	1	0.015625 s	6	186	148.0546875 KB
A*	2	0.171875 s	8	1618	1228.875976 KB
A*	3	3.125 s	10	25118	20088.35937 KB
A*	4	35.890625 s	12	267400	211272.81640 KB
Bidirectional BFS	1	0.046875 s	10	351	259.5605468 KB
Bidirectional BFS	2	0.203125 s	14	2246	1651.62304 KB
Bidirectional BFS	3	0.140625 s	14	1863	1370.069335 KB
Bidirectional BFS	4	1.234375 s	18	14018	11128.212890 KB
MCTS	1	3.158854 s	9	49603	3698.286132 KB
MCTS	2	5.271484 s	9	81580	5412.291992 KB
MCTS	3	4.792968 s	13	75589	5420.51074 KB
MCTS	4	5.40625 s	16	85839	5407.819335 KB

Când vine vorba de timpul de execuție, se pare că **Bidirectional BFS** rămâne la statutul de algoritm rapid pentru testele mari, în schimb MCTS-ul consumă mai puțină memorie la testul **case4**, ceea ce generează o veche problemă în lumea ingineriei calculatoarelor, și anume: **timp mare de execuție** sau **multă memorie consumată**?

A* nu ar fi o alegere înțeleaptă pentru cazuri complexe, așa cum s-a mai menționat mai sus, din simplul fapt că acesta are un **timp mai mare de execuție** și, în același timp, **consumă mai multă memorie**.

În consecință, folosirea unui algoritm depinde, mult, de **limitările mașinii clasice** de calcul ce se dorește a fi folosită pentru execuția acestuia.

4 Cerința 3: Pattern Database

Această ultimă secțiune va detalia performanța adusă algoritmilor A* și MCTS în cazul în care există o structură de date (un catalog) ce reține stările precalculate anterior în **11.0156 s** și care ocupă o memorie de **29098.10 KB**. Rămâne, exact ca mai sus, găsirea valorilor **optime** pentru **buget** și **constanta experimentală**, doar că euristica de interes este **cunoscută**, și anume **h3=multi_heuristic**, care se folosește atât de implementarea **catalogului**, cât și de **euristica admisibilă** de la prima secțiune, mai exact **rotation_cube**. Evident, pentru ambele cazuri de valori în ceea ce privește constanta C se vor **efectua 20 de rulări** (**640 de iterații**, având în vedere că sunt **4 valori de buget**).

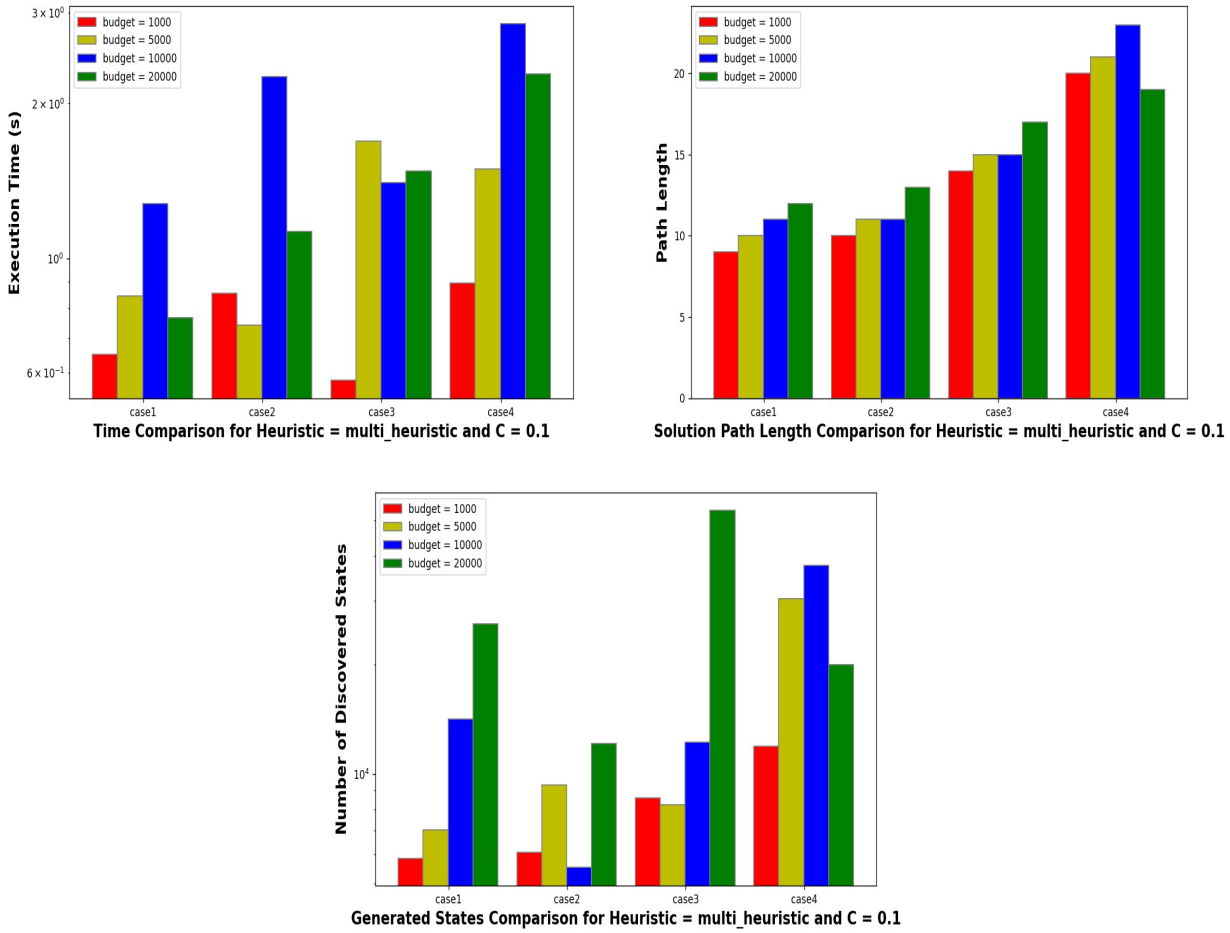

Figure 7: Performanța MCTS-ului cu C=0.1 și euristica multi_heuristic

Buget	Caz	Timp	Lungime Soluție	Număr de stări descoperite	Memorie
1000	1	0.65049 s	9	5855	166.658 KB
1000	2	0.85456 s	10	6081	157.623 KB
1000	3	0.57942 s	14	8597	271.117 KB
1000	4	0.89620 s	10	11931	98.0608 KB
5000	1	0.84457 s	10	7017	326.745 KB
5000	2	0.74218 s	11	9340	619.649 KB
5000	3	1.6875 s	15	8233	474.543 KB
5000	4	1.4929 s	21	30453	520.289 KB
10000	1	1.2765 s	11	14206	633.5401 KB
10000	2	2.2515 s	11	5530	271.581 KB
10000	3	1.4031 s	15	12229	460.801 KB
10000	4	2.8554 s	23	37683	594.108 KB
20000	1	0.7671 s	12	25992	173.797 KB
20000	2	1.1289 s	13	12160	241.262 KB
20000	3	1.47734 s	17	53315	277.336 KB
20000	4	2.28437 s	19	20045	771.643 KB

În primul rând, se observă faptul că MCTS-ul, cu această euristică, reușește să găsească **o soluție pentru orice buget**, la C=0.1, deci, într-adevăr, tehnica **Pattern Database** reprezintă o mărire a capabilității în ceea ce privește **rata de**

succes a algoritmului. Vizualizând, cu atenție, diagramele de mai sus, se constată că bugetul **cu valoarea 1000** implică **cel mai mic timp de execuție** pentru toate cazurile de test și că acesta reușește să găsească **cele mai mici căi** până la soluție (cu excepția cazului 4). Până aici, putem alege această valoare de buget (care este, de asemenea, **cea mai mică**) pentru C=0.1.

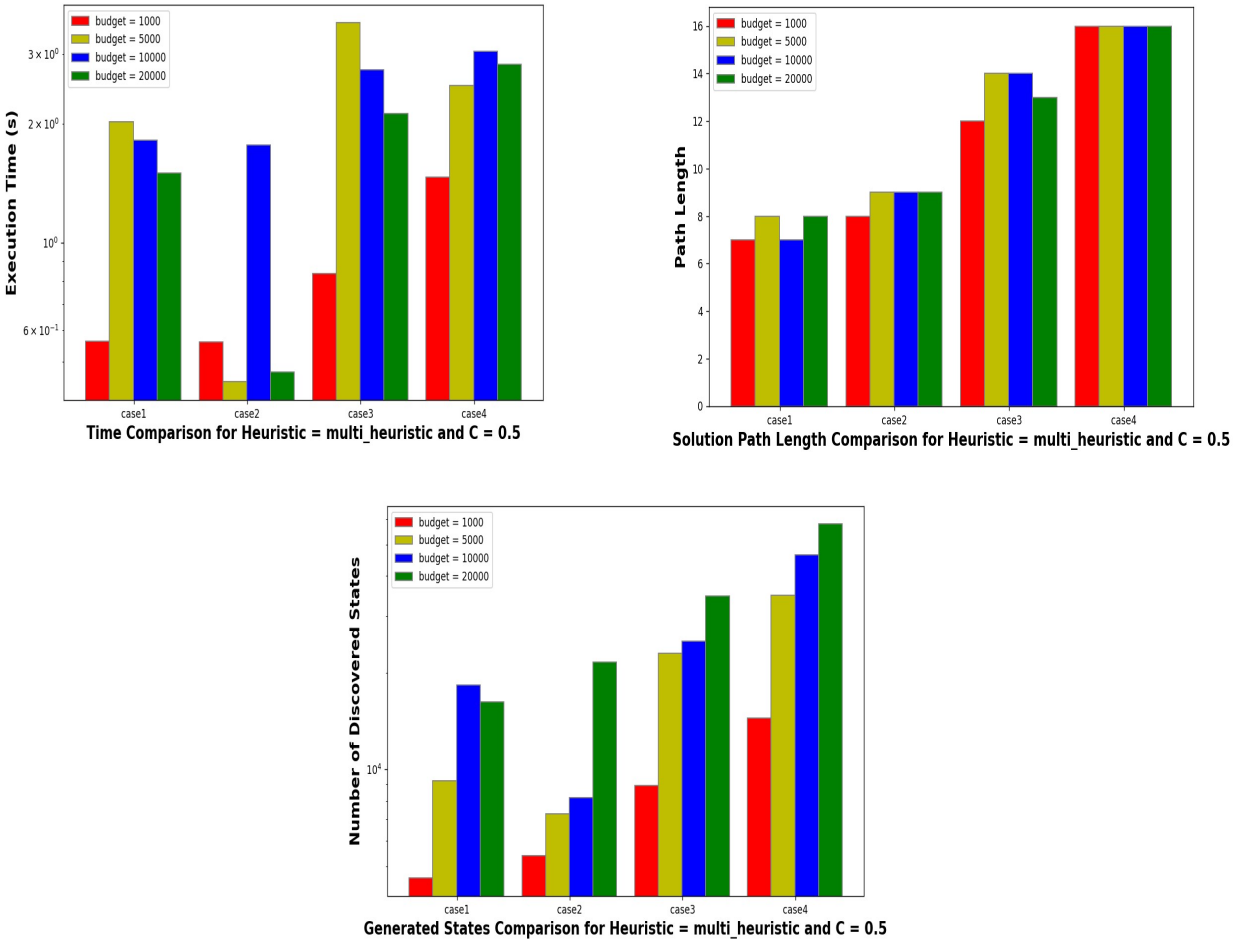

Figure 8: Performanța MCTS-ului cu C=0.5 și euristica multi_heuristic

Buget	Caz	Timp	Lungime Soluție	Număr de stări descoperite	Memorie
1000	1	0.56347 s	7	4602	98.573 KB
1000	2	0.56085 s	8	5415	167.716 KB
1000	3	0.83645 s	12	8914	188.404 KB
1000	4	1.46614 s	16	14487	339.684 KB
5000	1	2.02713 s	8	9236	345.948 KB
5000	2	0.44531 s	9	7280	189.035 KB
5000	3	3.60243 s	14	23017	554.751 KB
5000	4	2.49843 s	16	34712	760.004 KB
10000	1	1.8210 s	7	18352	436.701 KB
10000	2	1.7664 s	9	8184	189.318 KB
10000	3	2.7414 s	14	25005	1342.332 KB
10000	4	3.0531 s	16	46440	1205.426 KB
20000	1	1.5015 s	8	16260	585.774 KB
20000	2	0.4718 s	9	21573	160.2101 KB
20000	3	2.1218 s	13	34546	698.808 KB
20000	4	2.8281 s	16	57951	775.934 KB

Și în această situație algoritmul MCTS oferă câte o soluție pentru fiecare caz de test, la fiecare buget, însă, din punct de vedere al eficienței, la fel ca mai înainte, bugetul cu valoarea 1000 termină cel mai repede, conform primei diagrame, de asemenea, consumul de memorie este cel mai mic, raportat la celelalte valori de buget din cadrul tabelului de mai sus.

Ei bine, luând în considerare analiza anterioară, cel mai optim buget rămâne cel cu valoare 1000, dar, cea mai optimă constantă experimentală este 0.1 întrucât memoria folosită este mai mică față de memoria uzitată în configurația cu C=0.5 pentru testul 4. De asemenea, diagramele generate pe baza valorii C=0.1 au afișat informația conform căreia s-au găsit soluții în cel mult 14 pași la prima valoare de buget (la C=0.5 lungimea soluției pentru cazul 4 este de 16 rotații, de asemenea diferențele de timp sunt prea mici pentru a avea un impact major, de interes).

Sumarizând, se poate beneficia de o performanță mai bună a algoritmului MCTS ce folosește un catalog de stări precalculate dacă bugetul este de 1000, iar constanta experimentală are valoarea 0.1.

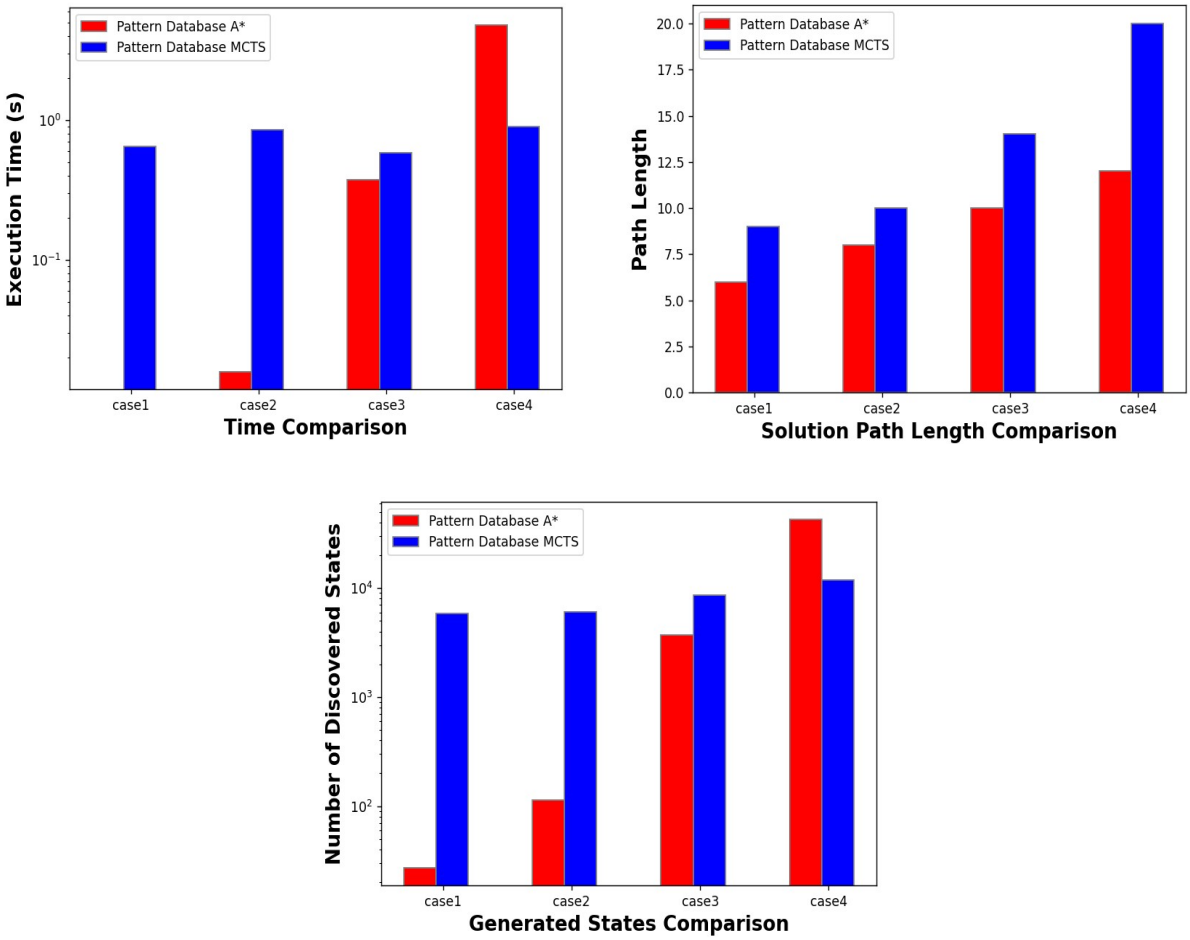

Figure 9: Performanțele algoritmilor A* și MCTS cu C=0.5 și euristica multi_heuristic

Algoritm	Caz	Timp	Lungime Soluție	Număr de stări descoperite	Memorie
A*	1	0.000001 s	6	27	23.075195 KB
A*	2	0.015625 s	8	113	76.024414 KB
A*	3	0.375 s	10	3745	2734.930664 KB
A*	4	4.796875 s	12	42805	33354.33984 KB
MCTS	1	0.65049 s	9	5855	166.658 KB KB
MCTS	2	0.85456 s	10	6081	157.623 KB
MCTS	3	0.57942 s	14	8597	271.117 KB
MCTS	4	0.89620 s	10	11931	98.0608 KB

Această ultimă comparație ilustrează, negru pe alb, **îmbunătățirea algoritmului A***, folosind tehnica **Pattern Database**, în ceea ce privește **timpul de execuție** pentru oricare **intrare** din suita de teste. Însă, comparând cu algoritmul MCTS, A* are **un timp mult mai mare pe ultimul test** și **consumă mai multă memorie**, deci pentru cazuri complexe MCTS este o variantă mai **potrivită**. Evident, pentru testul 1 A* nu are un timp egal cu 0 s (acest 0 este unul **numeric**, nu **matematic**), ci se termină foarte repede (aproape instantaneu) deoarece găsește toate stările procesate în **catalogul construit înainte**.

Dacă nu dorim construirea unui astfel de catalog, având în vedere că spațiul de căutare nu este atât de mare ca în cazul altor probleme de acest gen, mai potrivit decât MCTS este **Bidirectional BFS** care are atât un timp decent spre bun, cât și un consum mic de memorie pentru testele mari. Așa cum s-a mai menționat înainte, un aspect critic este dat

și de arhitectura sistemului de calcul folosit în rezolvare (catalogul respectiv poate să fie foarte mare, generând astfel un dezavantaj cu privire la spațiu).

Concluzia finală ce stă la **baza acestui document** este că nu putem spune dacă un algoritm (fie el informat sau nu) de căutare este mai bun sau nu decât un alt algoritm dacă nu definim, exact, **problema**, **complexitatea dimensiunii intrării** în cazul acelei probleme și **limitele resurselor folosite în cadrul rulării**.

5 Referințe

1. *Cursul 4 de IA - Strategii în jocuri*
   https://curs.upb.ro/2023/mod/resource/view.php?id=53410
2. *Idee de euristică pentru Pocket Cube*
   https://stackoverflow.com/questions/36490073/heuristic-for-rubiks-cube
3. *O lucrare interesantă care implică MCTS într-un context cu rețele neurale*
   https://web.stanford.edu/class/aa228/reports/2018/final28.pdf