Tema1 – Pocket Cube

Avramescu Cosmin-Alexandru – 344C3

Euristica A *:

Pentru h1, parcurg dictionarul de colturi (CORNERS) si verific pe rand culorile de pe fiecare colt ca sunt pozitionate pe fetele pe care ar trebui (indicii colturilor si fetele vizibile de acestea sunt disponibile in CORNERS). Adun 1 de fiecare data cand vad o culoare care nu e la locul ei. De asemenea, daca toate cele 3 culori de pe un colt se afla pe fata buna, inseamna ca acel colt este deja in pozitia corecta (si imi incrementez variabila corners pentru a contoriza colturile corecte).

Am folosit o functie auxiliara count_neigh_colors() care numara cate zone de forma urmatoare exista in cub (conturul cu negru):

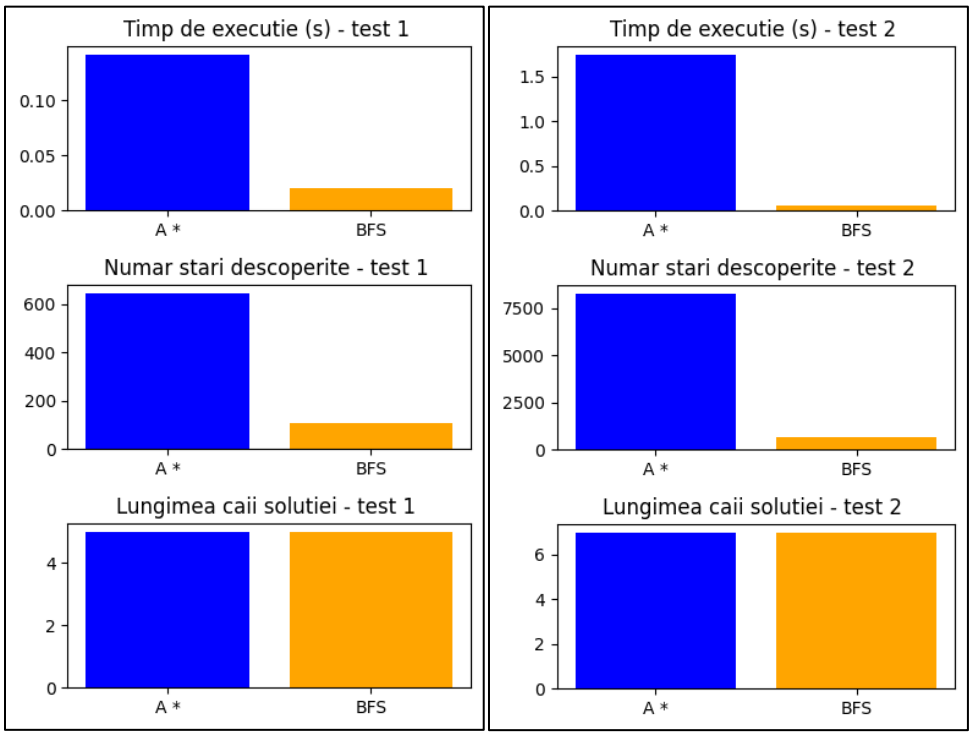

Practic, daca pe o muchie gasim 2 patratele de aceeasi culoare intre ele pe o parte si inca 2 patratele de aceeasi culoare intre ele pe cealalta parte, dar culorile sunt diferite intre ele (alb cu albastru – poza 1 si verde cu rosu - poza 2), atunci incrementez cu 1 variabila count. Ideea este ca aceasta metoda imi sugereaza gradul de amestecare al cubului (deoarece daca avem mai multe astfel de zone, e clar ca acel cub nu a fost amestecat foarte rau). In functia count_neigh_colors() verific fiecare muchie pentru aceasta conditie si verific culorile din state dupa indicii “sticker indices” din fisierul constants.py.

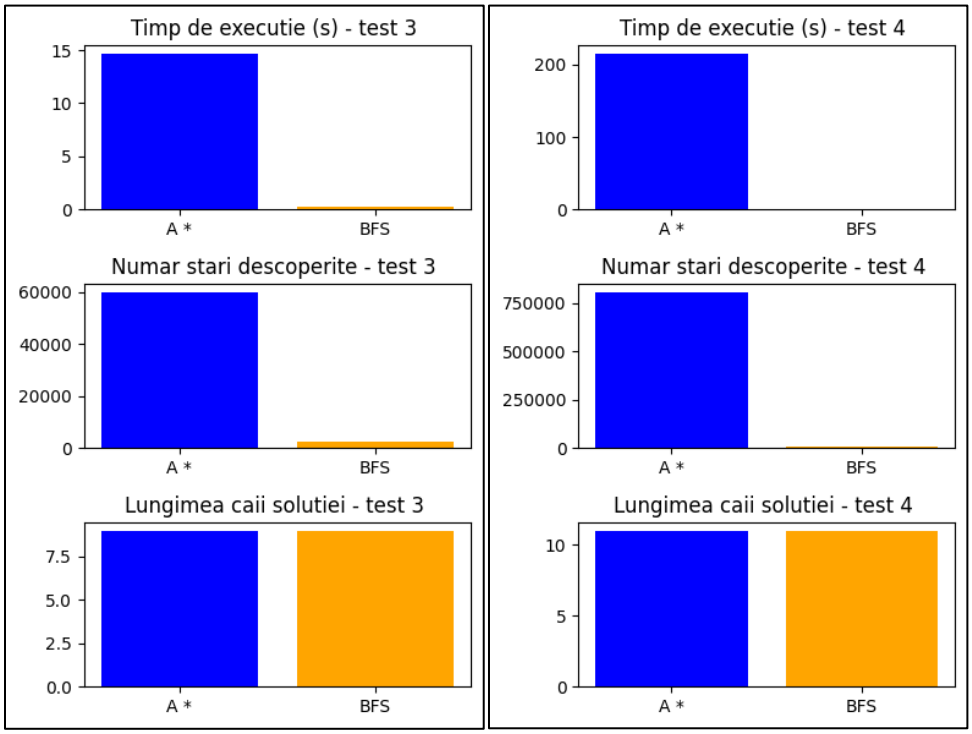
Asadar, daca am cel putin un corner la locul lui, mai putin de 20 de culori pozitionate gresit si cel putin 2 zone colorate cum am descris mai sus, atunci intorc sum // 8 (sum este maxim 24 si impartit la 8 va da maxim 3). Altfel, returnez sum // 6 (adica 24 / 6 = maxim 4 costul estimat). Practic, orice cub cu mutari <=3 (cost 3) va intra pe primul caz datorita conditiei, deci nu va putea intra pe cazul 2 ca sa fie supraestimat. Iar pe cazul 2 vor intra cuburile cu mutari >= 4 (cost 4), ceea ce respecta conditia de admisibilitate de mai mic sau egal. Am printat multe cazuri de

cuburi apeland h1 pentru a verifica admisibilitatea euristicii si faptul ca nu este supraestimat costul.

1. A * si BFS bidirectional

Test 1 vs Test2 vs Test3 vs Test 4





Test 1

	Timp executie (s)	Numar stari descoperite	Lungime cale pana la solutie
A *	0.14	646	5
BFS bidirectional	0.02	110	5

Test 2

	Timp executie (s)	Numar stari descoperite	Lungime cale pana la solutie
A *	1.78	8281	7
BFS bidirectional	0.06	657	7

Test 3

	Timp executie (s)	Numar stari descoperite	Lungime cale pana la solutie
A *	14.68	59968	9
BFS bidirectional	0.22	2251	9

Test 4

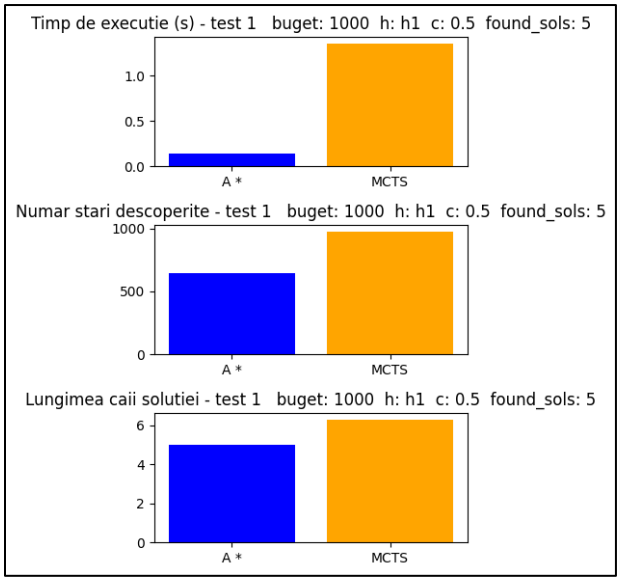
	Timp executie (s)	Numar stari descoperite	Lungime cale pana la solutie
A *	215.52	808230	11
BFS bidirectional	0.88	7488	11

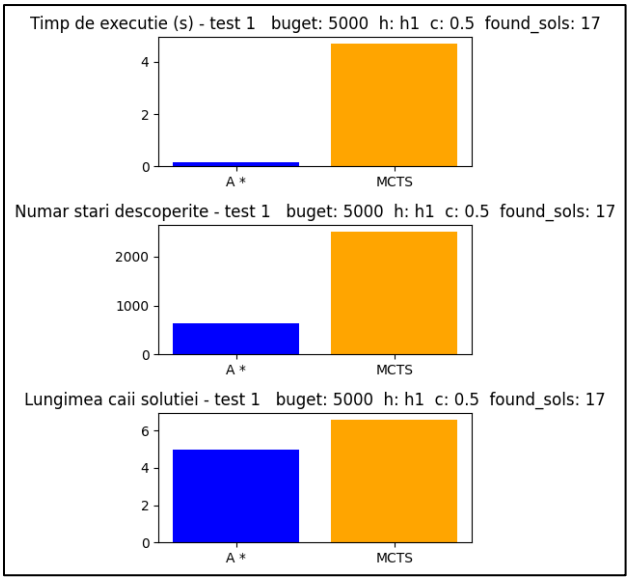
Se poate observa ca pe cel mai simplu caz (test 1), timpul de executie este asemanator, insa pe masura ce crestem gradul de amestecare al cubului rubik, bfs-ul se descurca exponential mai bine deoarece are timpi de executie foarte buni (este rapid) si nici nu expandeaza atat de multe stari precum A *. Mai mult decat atat, bfs-ul reuseste sa gaseasca solutia de lungime minima, la fel ca a star. Cu toate acestea, singurul dezavantaj al BFS este ca garanteaza gasirea solutiei de drum minim. In schimb, A *, datorita euristicii admisibile, gaseste mereu solutia cea mai scurta. Discutam de un trade-off si consider ca viteza mult mai mare a BFS este mai importanta, asa ca il consider pe acesta un algoritm mai eficient per total.

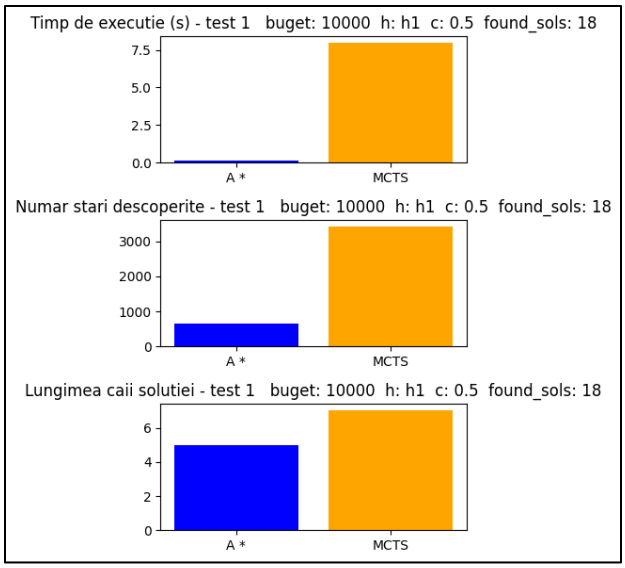
2. Monte Carlo Tree Search

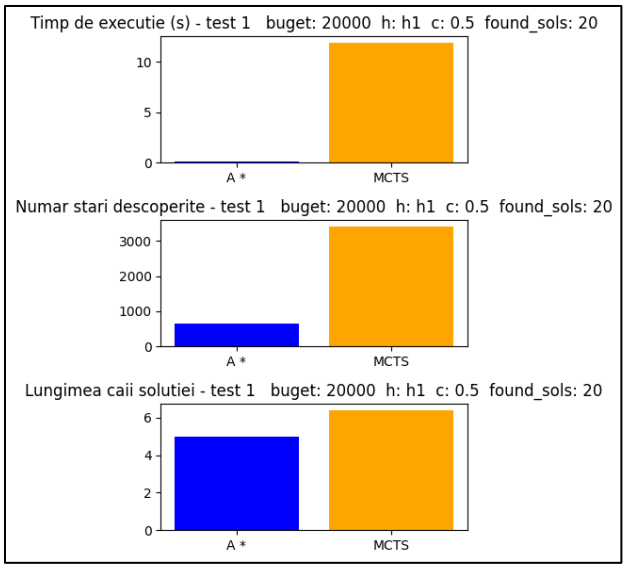
Test 1

	Timp executie (s) (best_avg)	Numar stari descoperite (best_avg)	Lungime cale pana la solutie (best_avg)	Best c & Best h	Solutii gasite (din 20)	Buget
1	1.35	976.5	6.3	0.5 h1	5	1000
2	4.70	2516.3	6.6	0.5 h1	17	5000
3	7.99	3426.7	7.0	0.5 h1	18	10000
4	11.93	3417.6	6.4	0.5 h1	20	20000



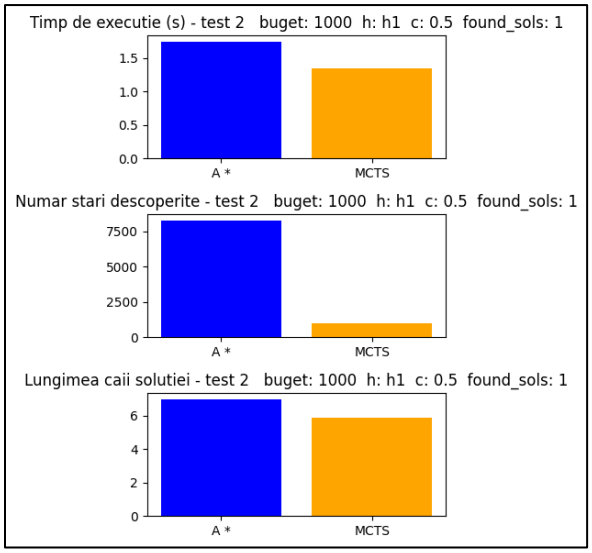


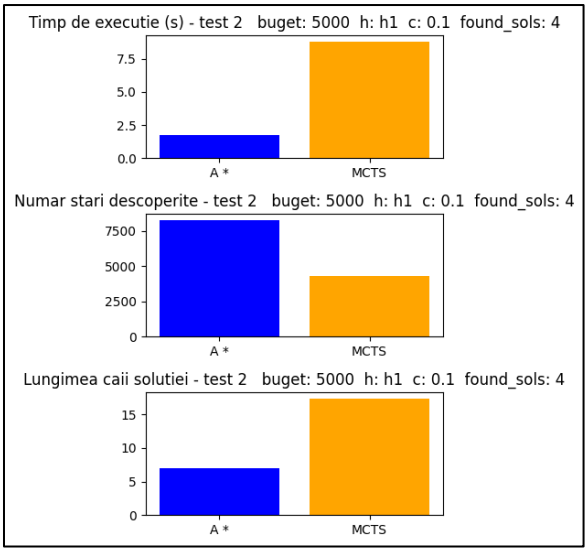


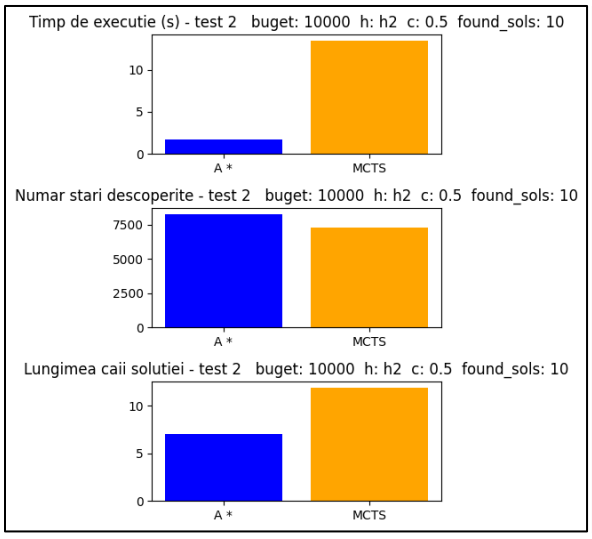


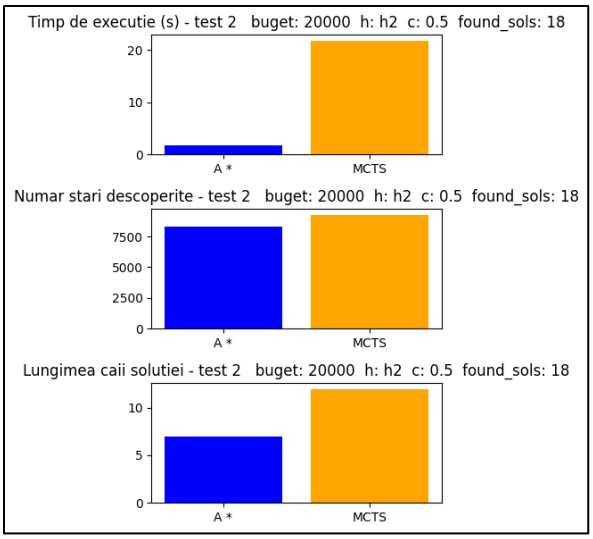
Test 2

	Timp executie (s) (best_avg)	Numar stari descoperite (best_avg)	Lungime cale pana la solutie (best_avg)	Best c & Best h	Solutii gasite (din 20)	Buget
1	1.35	971.8	5.9	0.5 h1	1	1000
2	8.81	4320.3	17.4	0.1 h1	4	5000
3	13.48	7309.5	11.9	0.5 h2	10	10000
4	21.80	9267.6	11.9	0.5 h2	18	20000



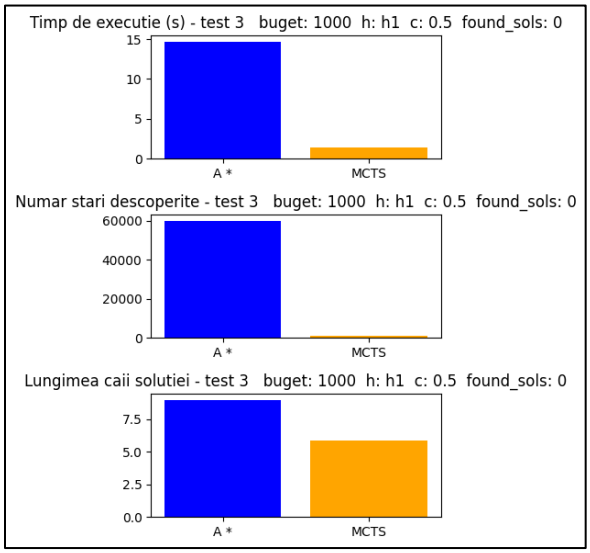


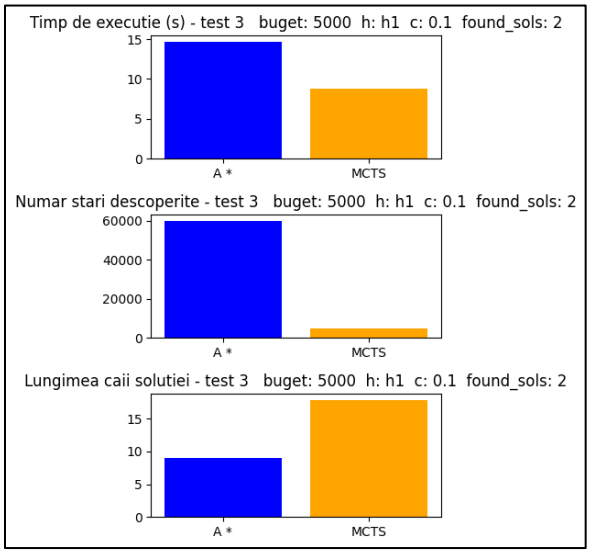


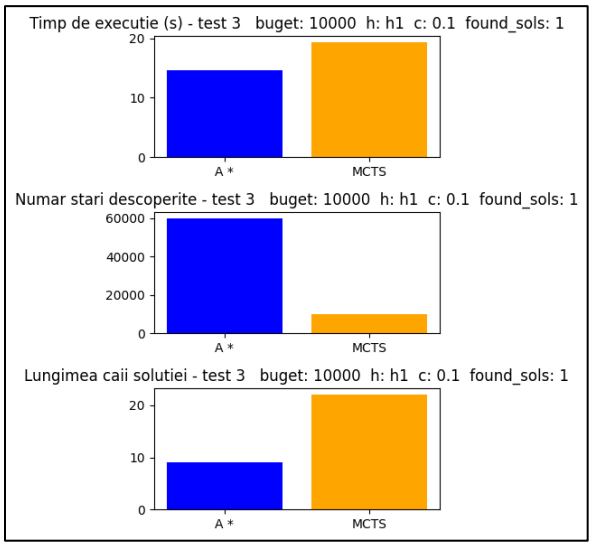


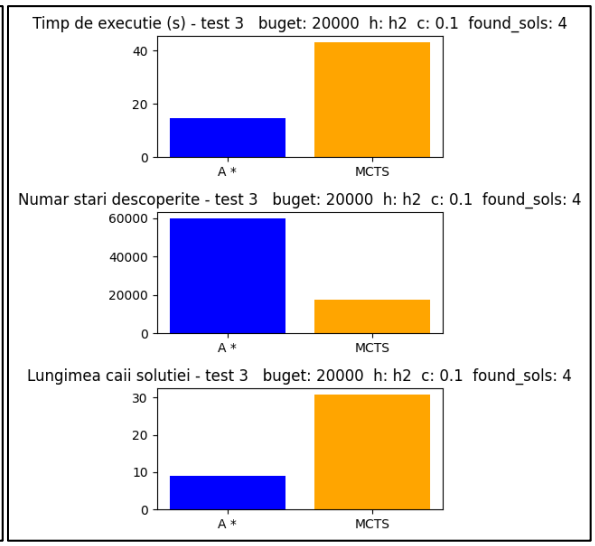
Test 3

	Timp executie (s) (best_avg)	Numar stari descoperite (best_avg)	Lungime cale pana la solutie (best_avg)	Best c & Best h	Solutii gasite (din 20)	Buget
1	1.37	1000	5.85	0.5 h1	0	1000
2	8.76	4682.4	17.85	0.1 h1	2	5000
3	19.37	9799.6	22.1	0.1 h1	1	10000
4	43.21	17356.7	30.9	0.1 h2	4	20000



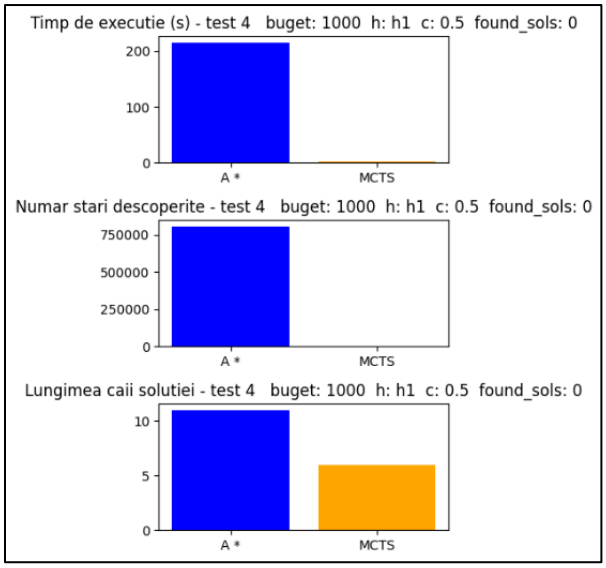


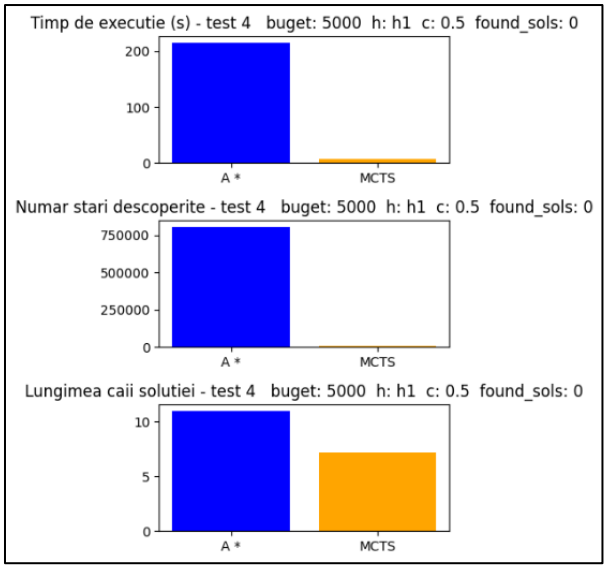


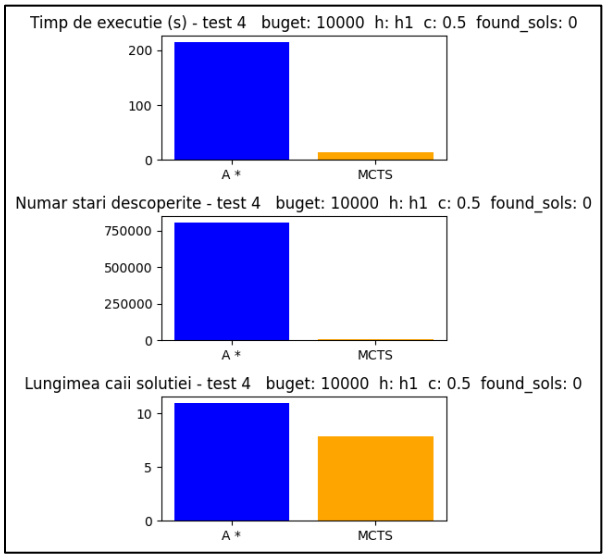


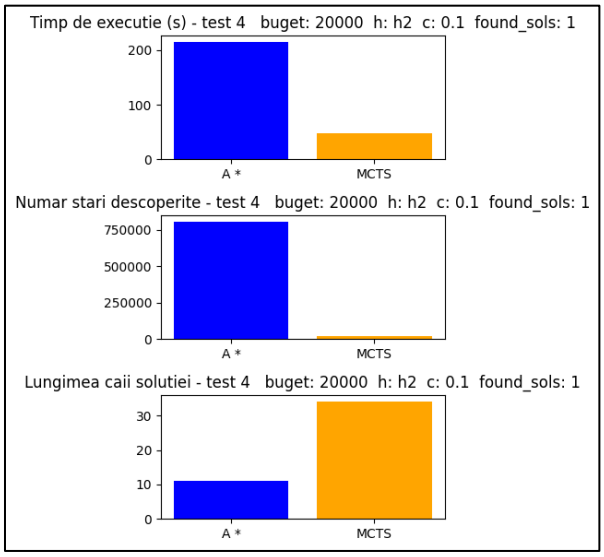
Test 4

	Timp executie (s) (best_avg)	Numar stari descoperite (best_avg)	Lungime cale pana la solutie (best_avg)	Best c & Best h	Solutii gasite (din 20)	Buget
1	1.35	1000	6.0	0.5 h1	0	1000
2	7.17	5000	7.2	0.5 h1	0	5000
3	14.61	10000	7.8	0.5 h1	0	10000
4	47.69	19048	34.15	0.1 h2	1	20000









Algoritmul Monte Carlo Tree Search are cele mai slabe rezultate din cauza alegerilor aleatorii pe care le face. Este nevoie de bugete foarte mari pentru a reusi sa se gaseasca solutie si chiar si asa cu bugete mari, nu se gasesc des solutii pentru cuburile amestecate mai puternic. Fata de A *, algoritmul MCTS expandeaza mult mai multe stari pe masura ce creste bugetul. Mai mult decat atat, timpul de executie este mai mare si cu toate acestea, solutiile gasite (daca se gasesc) nu sunt de lungime minima (sunt solutii chiar si de lungime dubla). Singurele momente in care lungimea solutiei este mai mica la MCTS decat la A* pe grafice sunt doar atunci cand nu se gasesc solutii (found_sol = 0).

3. Pattern database

A * (h1 vs h3) – comparatie doua cate doua

	Test	Timp executie (s)	Numar stari descoperite	Lungime cale pana la solutie
A * (h3)	1	0.04	103	5
A * (h1)	1	0.14	646	5
A * (h3)	2	6.83	29301	7
A * (h1)	2	1.74	8281	7
A * (h3)	3	171.35	624878	9
A * (h1)	3	14.68	59968	9
A * (h3)	4	1463.80	3300649	11
A * (h1)	4	215.52	808230	11

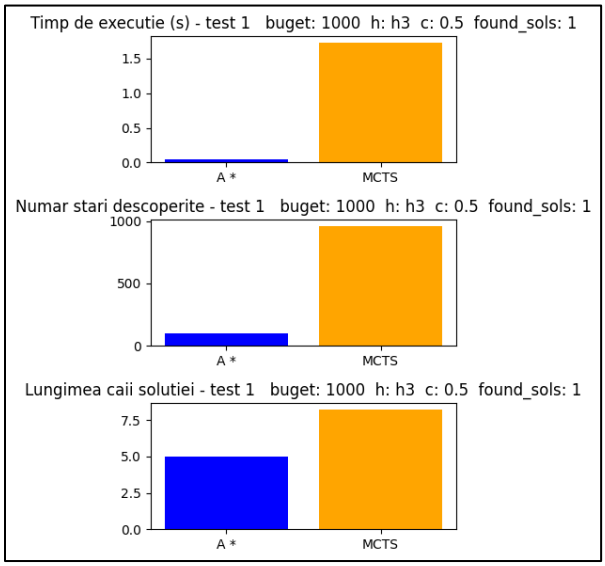
Se poate observa ca rezultatele A* cu h3 sunt mult mai slabe decat cele de la A* cu h1 (in special la cazurile de cuburi mai amestecate). Insa acest lucru nu se intampla deoarece euristica h3 este slaba, ci se intampla deoarece euristica h1 nu este la fel de performanta (sau macar pe aproape de h3). h1 intoarce un cost de maxim 4, pentru a avea grija sa se pastreze admisibilitate. In aceste conditii, avand in vedere ca A * este implementat ca un min heap, vor avea prioritate starile cu un cost mai mic (adica cele din h1).

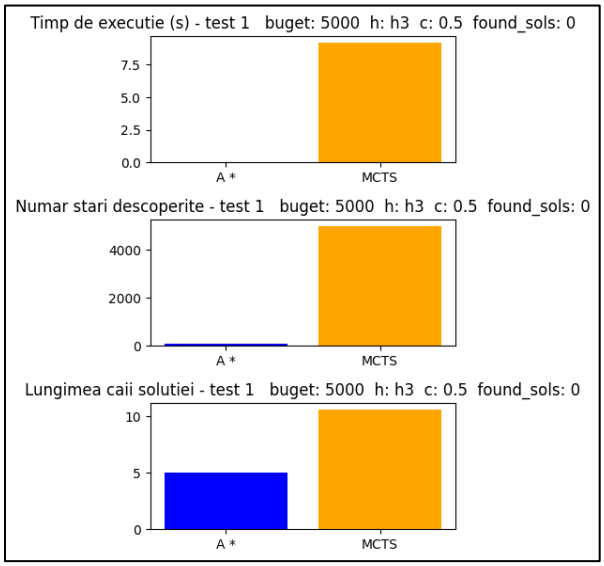
Avand in vedere ca s-au retinut in catalog starile de adancime 7 fata de solutie, este interesant cum acest fenomen apare pentru cuburile amestecate cu mai mult de 7 miscari, deoarece pana sa se ajunga la o stare din catalog, se vor introduce in coada de prioritati stari cu cost estimate de h1 (cost mult mai mic decat cele de la h3), iar acestea vor iesi primele din coada si din acest motiv dureaza mult mai mult sa se ajunga la solutie.

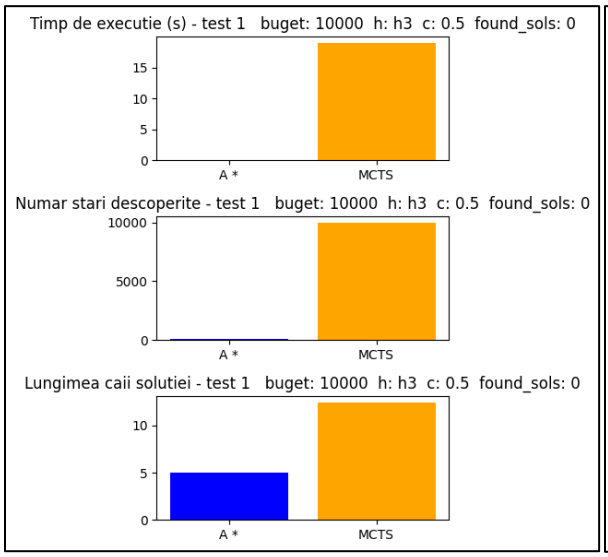
MCTS (h3) vs A * (h3)

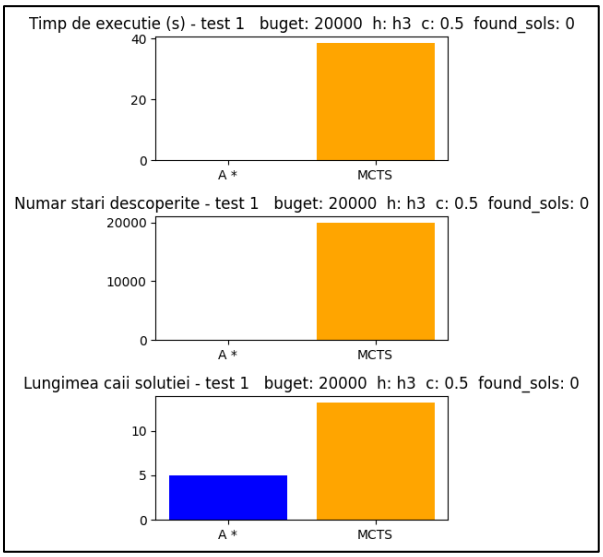
Test 1

	Timp executie (s) (best_avg)	Numar stari descoperite (best_avg)	Lungime cale pana la solutie (best_avg)	Best c & Best h	Solutii gasite (din 20)	Buget
1	1.73	961	8.25	0.5 h3	1	1000
2	9.21	5000	10.65	0.5 h3	0	5000
3	19.03	10000	12.45	0.5 h3	0	10000
4	38.65	20000	13.2	0.5 h3	0	20000



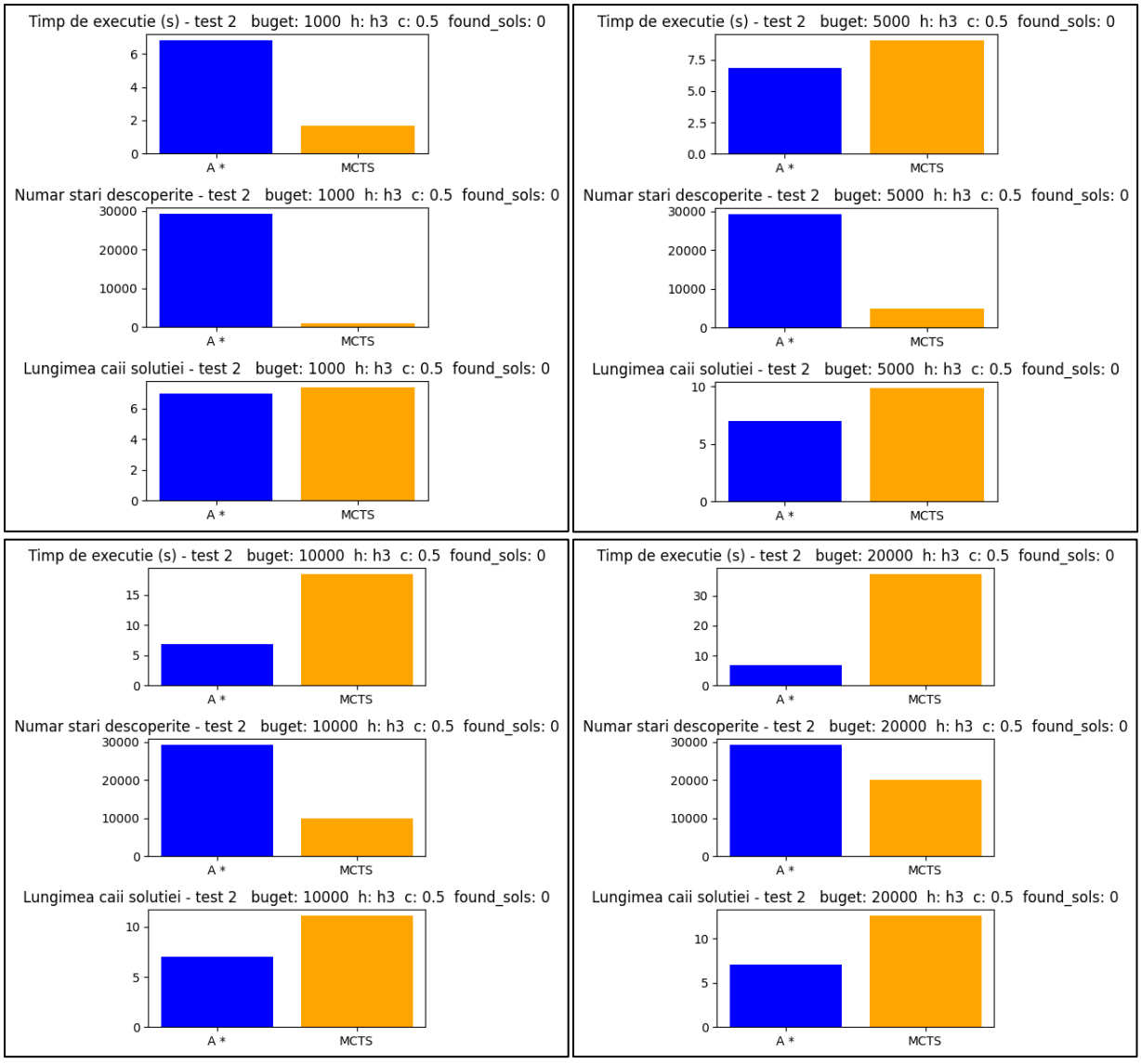






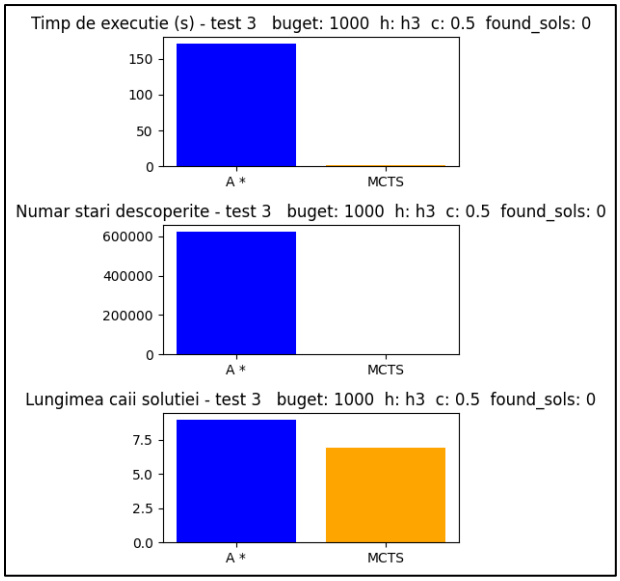
Test 2

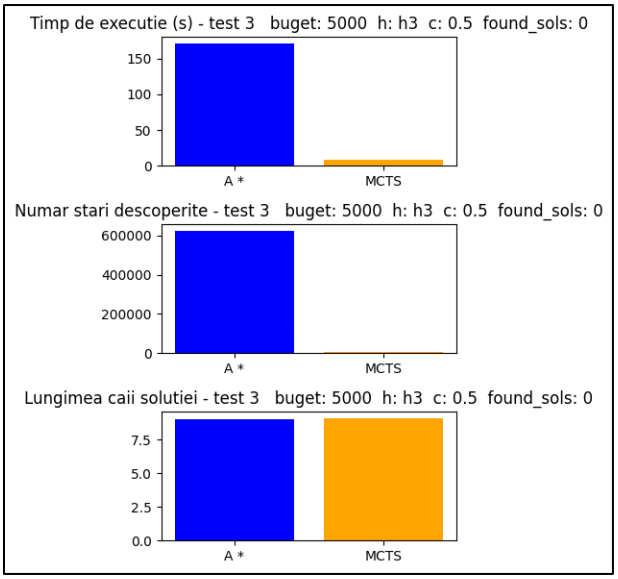
	Timp executie (s) (best_avg)	Numar stari descoperite (best_avg)	Lungime cale pana la solutie (best_avg)	Best c & Best h	Solutii gasite (din 20)	Buget
1	1.66	1000	7.4	0.5 h3	0	1000
2	9.06	5000	9.9	0.5 h3	0	5000
3	18.48	10000	11.15	0.5 h3	0	10000
4	37.35	20000	12.6	0.5 h3	0	20000

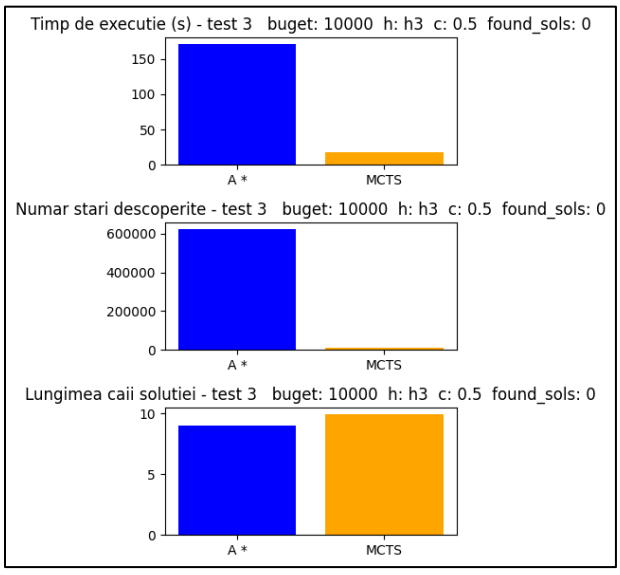


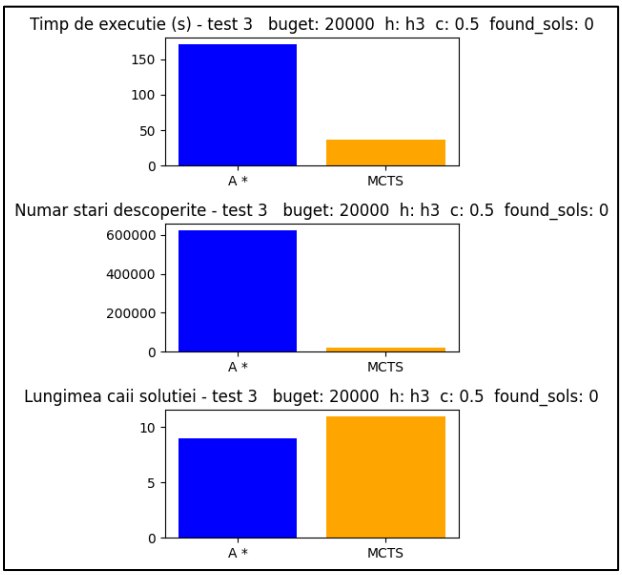
Test 3

	Timp executie (s) (best_avg)	Numar stari descoperite (best_avg)	Lungime cale pana la solutie (best_avg)	Best c & Best h	Solutii gasite (din 20)	Buget
1	1.65	1000	6.9	0.5 h3	0	1000
2	8.70	5000	9.1	0.5 h3	0	5000
3	17.89	10000	9.9	0.5 h3	0	10000
4	35.55	20000	11.0	0.5 h3	0	20000



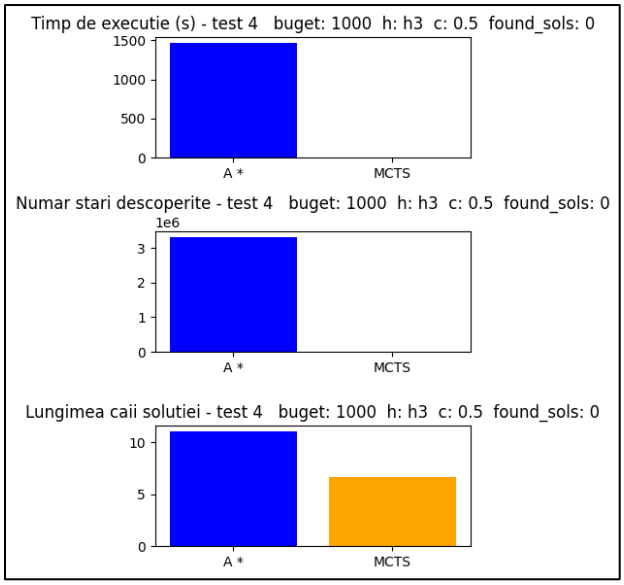


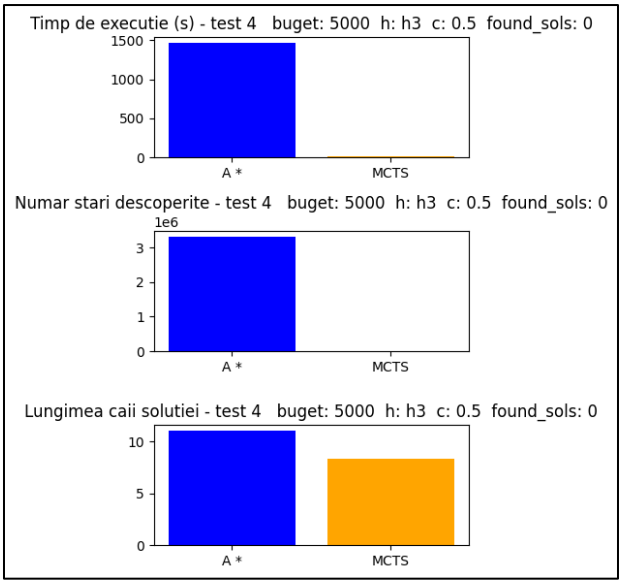


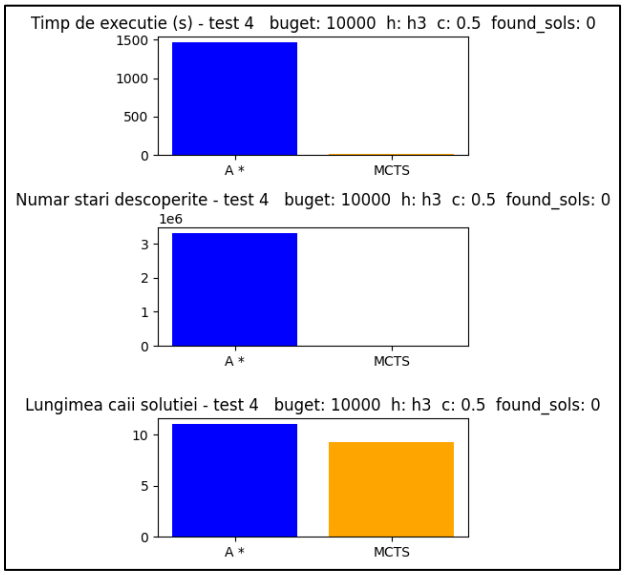


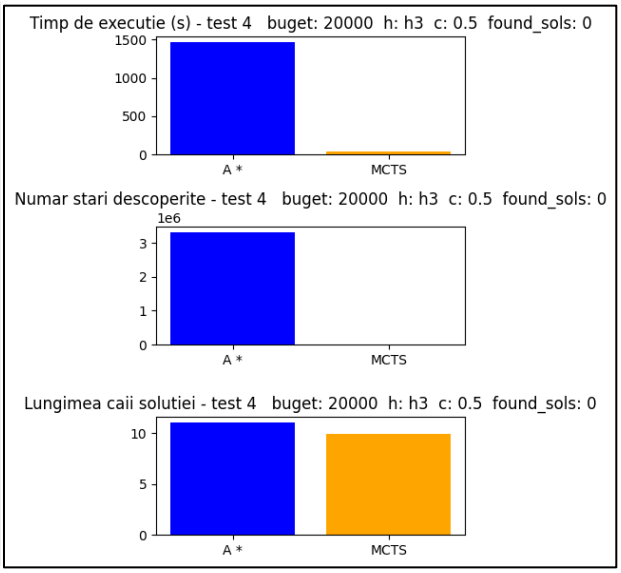
Test 4

	Timp executie (s) (best_avg)	Numar stari descoperite (best_avg)	Lungime cale pana la solutie (best_avg)	Best c & Best h	Solutii gasite (din 20)	Buget
1	1.67	1000	6.6	0.5 h3	0	1000
2	8.56	5000	8.3	0.5 h3	0	5000
3	17.56	10000	9.3	0.5 h3	0	10000
4	35.66	20000	9.9	0.5 h3	0	20000









Se poate observa ca se obtin cele mai slabe rezultate la MCTS cu h3. Nu se gaseste solutie deloc, cu exceptia testului 1 cu buget 1000 cand se gaseste fix o solutie. Problema este legata de alegerile aleatoare facute de algoritmul MCTS si astfel se expandeaza foarte mult spatiul de stari fara sa se ajunga la o solutie. In plus, la asta contribuie si diferenta mare de estimare dintre h3 si h1. Pe masura ce se expandeaza numarul de stari, se extrage costul maxim (r) si reward-ul va deveni reward = 100 – r. Sa luam, de exemplu, testul 1 care s-ar rezolva in 5 mutari. Ei bine, reward va fi 100 – 5 = 95 pentru prima stare. Insa pe masura ce se expandeaza spatiul de stari, se va trece de limita de 7 nivele fata de solutie si va interveni h1 (care are costuri de 1, 2, 3 sau 4). Aceste noduri vor avea reward-uri de 99, 98, 97, 96 (mai mari decat starea initiala, desi sunt mai departe de solutie), iar pe masura ce aceste costuri vor ajunge la backpropagation, se va altera calitatea nodurilor si din aceasta cauza nu se mai gaseste solutia. Acelasi motiv este si la cuburile amestecate mai puternic.