Artificial Intelligence

Computer Assignment 1

Kian Fotovat

SID : 810003008

In this project, we're going to use informed and uninformed search algorithms to solve a game, and analyze the cons and pros of each algorithm and learn how to implement them.

Here we have a game like sokoban (monster's factory) and we're going to write an AI that solves the game

1) The definition of state in this particular game, could be the map of the warehouse in each moment. But I didn't consider it this way, and that is because of the info we need from that state, what details do I really need? In my case, I only work with the player's location and boxes' locations, because they are really changing in each move and by having them, I can uniquely describe my state. So all that walls, goals, portals' locations, etc would be overkill because they're static and fixed and I improved its simplicity in this way of only storing box locations and player's location.

2) So considering the definition I use for the state, I define my actions as follows:

. UP

. DOWN

. RIGHT

. LEFT

And there would be some rules that must be followed:

➔ The player can move only if the destination cell is empty or contains a box that can be pushed

➔ If the player moves into a box, the box must be able to move in the same direction (without hitting a wall or another box)

➔ The action updates the player's position and (if pushing a box) updates the box's position as well

3) So starting with State Representation :

➔ As I defined, I only store my states as a tuple of player's location, and boxes' locations, so it would add to simplicity and caring about the things that are actually changing, while the other fixed objects' locations are known to me. So they are stored like this : (player's location, boxes' locations)

Next would be my Actions :

➔ My actions would be going up, down, left and right. I do these actions or by the game.py attribute, 'apply_move' , either a bunch of consecutive moves containing a combination of those 4 options to move, and is done by the game.py attribute, 'apply_moves' . and there are some rules in for these actions and they are as followed :

- The **player moves** if the target cell is **empty**.
- The **player pushes a box** if the next cell contains a **box**, and the cell after it is **empty**.
- The action updates the state accordingly.

Formally, the transition functions updates:

- The **player's position** if the move is valid.
- The **box's position** if it is pushed.

Next would be my Initial state and goal state :

➔ My initial state model would be a state, that describes the location of player and the boxes, at the same time, and that is done, again, by a tuple of containing player's position as the first element, and a list of boxes' locations as the second element, and they are achieved in the beginning by using the attributes that game.py gave us

➔ For the goal state, I only care about the boxes' locations and the goals' locations. Here my player's location doesn't matter because we only care about the boxes being at the right place. In the code, I am aware of, if the boxes are at the same locations as the goals, with the attribute "is_game_won()" , so if true, the goal state is met, otherwise, nope !

4) In this game, the search space grows exponentially with the number of boxes and possible moves. To improve efficiency, we need to avoid unnecessary states and use better search techniques.

<u>But do we need to expand all the possible moves?</u>

In bfs, dfs, ids, A* , at each step we consider all possible 4 actions. But expanding all of them causes more memory and time usage, so I was needed to:

➔ Avoid redundant states and revisiting the same state. So I handled this in my code by storing all the states visited and check if the new one is redundant or not, so it caused reduced duplicate computations, making the search faster

➔ Detect deadlock situations. I actually added this to my A* algorithm's heuristic function, so it would return a huge h for the states containing deadlocks. What is deadlock? When the box is stuck in a corner, etc and we're not able to move it. So by just passing these situations, we won't waste time on those dead states and just skip em.

➔ If two states looked the same, except the player's position, treat them redundant and skip it. I actually implemented it in one of my commented algorithm codes, and it is done by looking just at the second parameter of the state tuple we were storing

➔ Prioritizing the promising states. Yes I implemented this one, too, for the A* algorithm, and using a Manhattan distance or the more advanced heuristic calculator I added later, it is achieved to go only for those with lower g+h scores (the promising ones).

So Instead of expanding **all** possible next moves, we should **filter and prioritize only the useful ones (as methods mentioned above)**. This **reduces search space**, improves speed, and makes solving the game **more efficient**.

5) First I used breath-first-search (bfs) . this algorithm is optimal, but also memory heavy. It explores all possible moves level by level and by using a queue with First in First out behavior, it goes through all the cases. Also we're sure that the first answer it would reach, is the shortest one. And it was also simple to implement with respect to others. But as mentioned, it has high memory usage, and is also slow for large maps. So it acts nice in short to medium maps or with maps that are solved with short number of steps but is not that fast in bigger ones.

➔ Next I used depth-first-search (dfs) . it explores one path deeply before backtracking. It uses a stack with Last in Last out behavior. It also doesn't guarantee to find the shortest path, because it goes deep and if that shortest path is placed at layers near to surface, we wouldn't reach it. But it is not all cons, because it has low memory usage, and can find a solution quick (if it's lucky enough). Ans there's obvious cons such that it might stuck in deep, unnecessary paths, and also as mentioned, it doesn't guarantee the shortest path (despite bfs) . it's good to use when we need just one answer, without caring about it being the best. So overally it does not guarantee an optimal solution, it may stuck in loops and deadlocks (need to handle) and there is also the risk of infinite deph.

- ➔ Next we have iterative-deepening-search (ids) . they are the same as DFS but with optimality. It runs dfs, but increases the deph gradually until we reach the answer. Like BFS, it also kinda guarantees the shortest path, but also with less memory usage. Its pros would be that optimality and less usage memory. It is nice for big maps that are memory-heavy but also optimality. It is kind of dfs and bfs combined. So what problems does ids solve? First it solves the high memory usage of bfs, by only storing the current path , like dfs. Next it solve the non-optimality of dfs. Ids ensures to find the shortest path like bfs, by gradually increasing depth limits.
- ➔ And the final would be A* (or weighted A*) . it has a priority queue (I used heap) which prioritizes states based on a second value that is stored beside the path, and that is : f = g + h , where that g is the cost from the start to the state (backward), and h is the heuristic estimate of the cost to the goal (forward). So now we act like we are informed to which state we should explore, and obviously it should find the optimal solution, and also it is faster than bfs (when a good heuristic is used). The problems would be that many thing depends on that heuristic we design, and that should be a well designed one and this algorithm can still cause a lot of memory. It is awesome for large game maps and when we need both optimality and efficiency.

Here I first go for questions number 7 and 8, and then come back to question number 6

6) Skip it for now

7) Here for A* algorithm, first I used the two common ones (Manhattan distance and the Euclidean distance) , then I introduced an extra more advanced one, which is similar to Manhattan distance, but with added

parameters. First **Manhattan dstance** : This heuristic calculates the sum of the Manhattan distances of each box from its current position to its goal position. The Manhattan distance is the sum of the horizontal and vertical distances between two points. This heuristic assumes that each box can be moved directly to its goal without obstruction. **YES IT IS ADMISSIBLE** : The Manhattan distance is a **lower bound** of the true cost because it does not take into account obstacles (walls and other boxes). Therefore, it **underestimates or equals** the true cost, making it admissible. **AND YES IT IS CONSISTENT** : Moving a box closer to its goal (by one step) reduces its Manhattan distance by 1. The actual cost of moving the box (1 move) matches the reduction in the heuristic, satisfying the consistency condition. Next is the **Euclidean distance** : The Euclidean distance heuristic calculates the straight-line distance (the shortest possible path) between each box's current position and its goal position. **YES IT IS ADMISSIBLE** : Like Manhattan distance, Euclidean distance never overestimates the true cost. It is always a lower bound on the actual cost because it assumes a direct, unobstructed path between the box and its goal. Thus, it is admissible. AND **YES IT IS CONSISTENT** : The Euclidean distance is consistent because for any two states moving a box one step closer to its goal reduces the Euclidean distance in a way that is consistent with the cost of the move (1 unit). Because our movement is limited and we don't have diagonal movements, it doesn't have any privilege with respect to Manhattan distance.

➔ My improved heuristic : it works like Manhattan distance, but with added complexity. It also handles a case where at least one of the boxes is stuck (deadlock situation) and gives a high score of h to that state, so the state would go right down the bottom of my priority list and is checked at last. ⟶ and yes this one is also both consistent/admissible

8) for this part I tried those three algorithms with some of the test case maps and reported the numbers :

	map 3	map 5	map 7	map 8	map 10
Manhattan Distance	0.0s	0.0s	4.2s	0.0s	28s
Euclidean Distance	0.0s	0.0s	5.2s	0.0s	—
improved Manhattan Distance	0.0s	0.0s	0.6s	0.0s	2.6s

↳ in all cases Weight = 1 (for optimality), Also the results achieved on my Linux virtual machine

Took average ← tried 4 times ←

NOW I tried different Weights (1, 1.2, 1.5, 1.7, 2)

	map 3	map 5	map 7	map 8	map 10
W = 1.2	0	0.0s	1.7s	0.0s	11.6s
W = 1.5	0	0.0s	1s	0.0s	4.4s
W = 1.7	0	0.0s	0.6s	0.0s	2.7s
W = 2	0	0.0s	1.6s	0.0s	1.5s
W = 3	0	0.0s	0.3s	0.0s	3.4s

a good Combination

Greed and optimality

بررسی نتایج ← در improved heuristic نسبت به دو تای دیگر پیشرفت زیادی مشاهده شد و بین manhattan و Euclidean نیز manhattan Distance عملکرد بهتری داشت ← پس از heuristic بهبود یافته خود استفاده خواهم کرد

Test cases for maps 1-4 → all less than 0.1s

Test case for map 5

	States seen	output	runtime
BFS	7936	U,L,D,D,R,D,L,L,L,U,U,U,R,U,L	0.1s
DFS	932	----------	0.1s
IDS	104	------	0.8s
A*	681	----	0.0
Weighted A*	84	----	0.0

W = 1.7

Test case for map 6

	States seen	output	runtime
BFS	20823	U,U,U,U,U,R,R,R,L,L,L,L,L,L,L,R,D,D,D,D,D,D,D	0.7s
DFS	345	------	0.0s
IDS	10806	---	25.5s
A*	—	—	—
Weighted A*	—	—	—

Testcase for map 7

	stats seen	output	runtime
BFS	802152	-------	38s
DFS	—	— — -	—
IDS	—	—	—
A*	12774	- - - -	1.2s
Weighted A*	11386	- - - - -	0.6s

Testcase for map 8

	stats seen	output	runtime
BFS	10756	U,U,R,D,L,D,R,R D,R,U,R,D,R	0.1s
DFS	—	←	—
IDS	4074	- - - - -	1.9s
A*	134	— -	0.0s
Weighted A*	31	— - - -	0.0s

Testcase for map 9

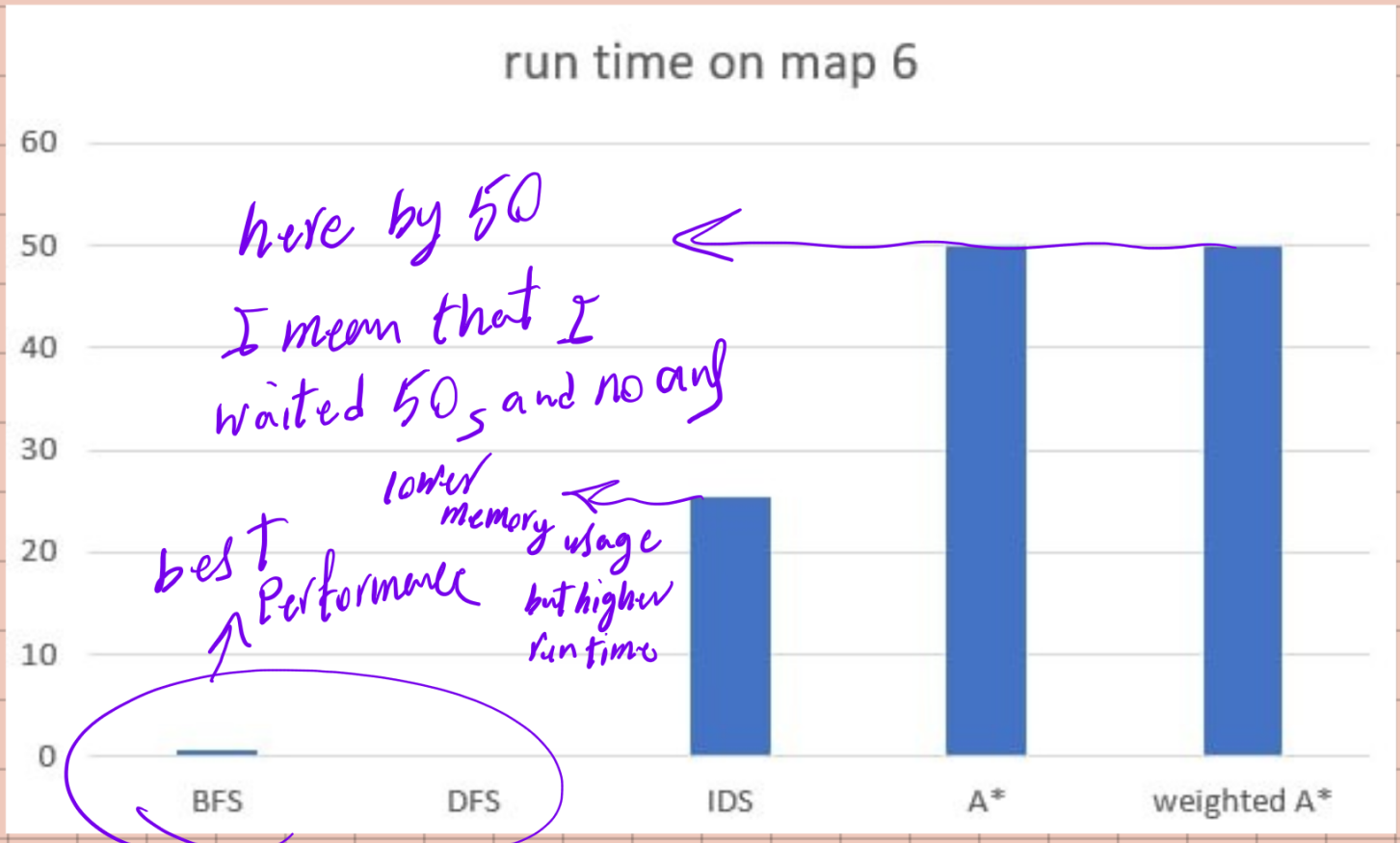
	stats seen	output	runtime
BFS	—	←	— → Took several minutes
DFS	—	—	—
IDS	—	—	—
A*	—	—	—
Weighted A*	—	—	—

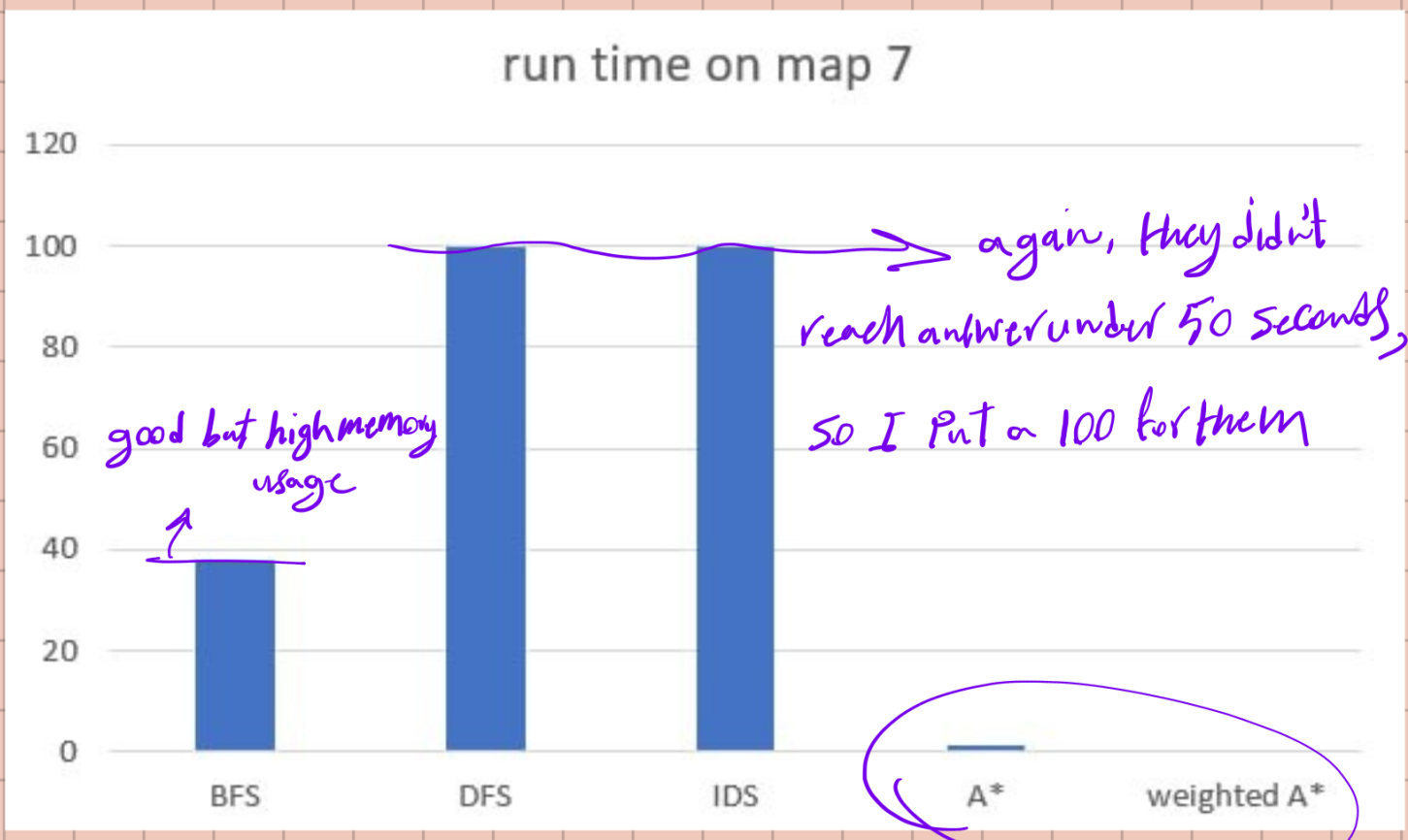
Test case for map 10

	state seen	output	run time
BFS	254735	-----	13.4s
DFS	——	——	——
IDS	—	—	—
A*	37568	----	22.4s
weighted A*	16826	-----	2.6s

→ that missed

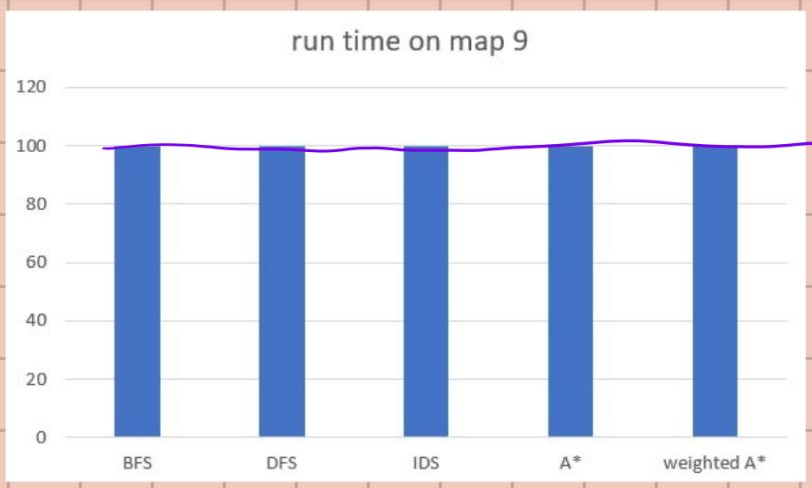
6) between BFS and DFS, in these maps BFS finds the answer faster (DFS don't have luck here) but BFS visits more states meaning more memory usage. IDS combines these two and it visits noticeably smaller number of states and the speed is also something BFS & DFS.



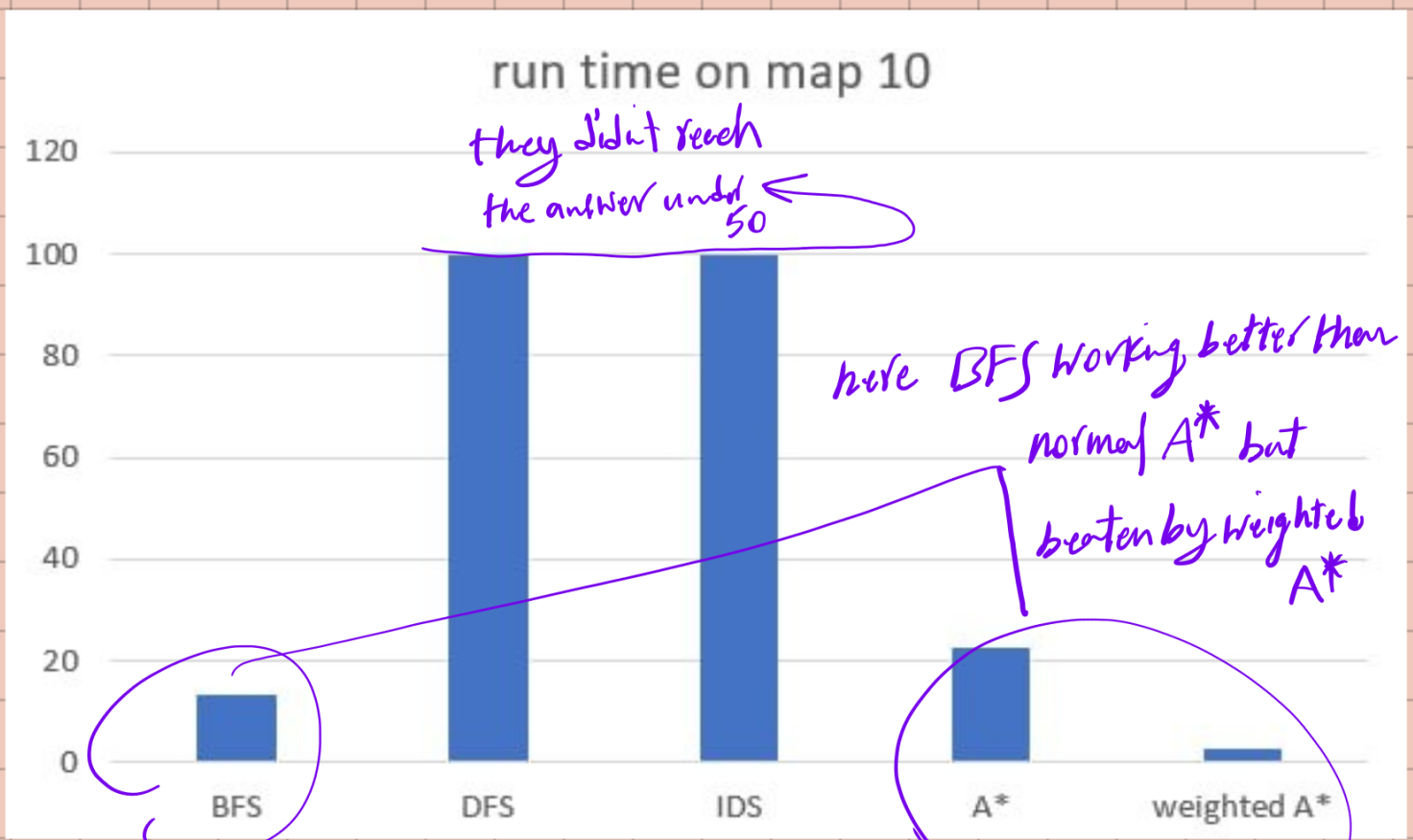


best performance

they all did awesome exect this IDS



no one reached to answer in reasonable time



they didn't reach the answer under 50

here BFS working better than normal A* but beaten by weighted A*