Introduction to AI - 236501
HW1

Yair Nahum 034462796
and
Hala Awwad 209419134

May 4, 2022

1 Intro

1.1

We use the notation as in the tutorial about the sets as follows:

$$[n] \equiv \{1, 2, ..., n\}$$

Next, we define the $\{S, O, I, G\}$ as follows:
For the states we know we have 25 places for the taxi to be in the grid.
We have 5 states for the passenger to be $\{R, Y, G, B\} \cup \{inthetaxi\}$. and 4 states for the destination $\{R, Y, G, B\}$.
Thus,

$$S = [500] \Rightarrow$$

$$|S| = 25 * 5 * 4 = 500$$

The valid states as defined in the hw notebook are a subset of this states space as the passenger and destination are not at the same location (3 options are left to seleect from for the destination). So,

$$|S| = 25 * 5 * 3 = 375$$

The operations/actions that our taxi agent can perform are:

$$O = \{0 = "South", 1 = "North", 2 = "West", 3 = "East", 4 = "Pickup", 5 = "Dropoff"\} \Rightarrow$$

$$|O| = 6$$

The initial states can be random, but as defined in the hw notebook, we always starts at $s = 328$. Thus,

$$I = \{328\} \Rightarrow |I| = 1$$

The goal state is when the taxi and passenger are at the destination after the passenger was dropped there. So we have 4 different goal states:

$$G = \{s \in S | \{Taxi \wedge Passenger \wedge Destination \in R\} \cup$$
$$\{Taxi \wedge Passenger \wedge Destination \in Y\} \cup$$
$$\{Taxi \wedge Passenger \wedge Destination \in G\} \cup$$
$$\{Taxi \wedge Passenger \wedge Destination \in B\}\} \Rightarrow$$
$$|G| = 4$$

1.2

The Domain of the "North" operation are all the states.
There are states in which the taxi will stay at the same state if it goes north. More formally:

$$Domain(o_1 = "North") \equiv \{s \in S | o_1(s) \neq \phi\} = S$$

One can think that it's illegal to go north on the top row of the grid world or when there are horizontal walls north to a reachable grid cell. But the env allows the taxi to go north with reward -1. More formally, if it was illegal:

$$Domain(o_1 = "North") \equiv \{s \in S | o_1(s) \neq \phi\} =$$
$$\{s \in S | s \in \{\text{All states in which the taxi is not at the north-est grid row}\}\}$$

1.3

The function Succ over the initial state 328 will return all the neighbors of that state, and as we got from running getNeigbours on the initial state

$$South(328) = 428$$

$$North(328) = 228$$

$$East(328) = 348$$

On West/Dropof/Pickup operation we stay at the same location (there is a wall to the west):

$$West(328) = Pickup(328) = Dropoff(328) = 328$$

More formally:

$$Succ(s = 328) \equiv \{s' \in S | \exists o \in O, s.t.[s \in Domain(o) \land o(s) = s'\} =$$

$$\{428, 228, 348, 328\}$$

1.4

for the worst case, we can randomly switch between several cases and never reach a destination. so for the random agent we can get infinite number of actions for the agent.
Since the state space is finite, and there is an equal probability for each action, the agent eventually will find the goal (finite MDP with positive probability and we can compute the expectation for reaching the goal).

1.5

If the agent does the optimal actions by chance, we can see the optimal path is:
1. 4 actions to reach the passenger (North,West,South,South)
2. 1 action to Pickup
3. 4 North actions
4. 1 action to Dropoff
Therefore, we have total of 10 actions at the optimal path.

1.6

As we wrote on the previous section, the optimal path is: (North, North, West, South, South, Pickup, North, North, North, North, Dropoff) The rewards are therefore:
(-1, -1, -1, -1, -1, -1, -1, -1, -1, 20) And the total reward is:

$$\text{total reward} = 9 * (-1) + 20 = 11$$

1.7

Yes. There are circles in our search space (if we don't do some graph search that denotes already visited states).
For example the following path will get us back to our initial path:
(North,East,South,West)
If our algorithms of search will maintain the CLOSE lookup table, we can easily check for such circles.

2 BFS-G

2.1

We've implemented the BFS-G in code (not tree search).
That is, Besides the frontier (the OPEN queue implemented with python list), we maintain a set of states that were already reached (the CLOSE set as described in the book).
We've optimized the number of nodes created when expanding some node in the frontier by checking if its neighbor state was already reached or in frontier.
A helper set of frontier (OPEN) states was added, although we could check existence of the node in nodes queue (for simplicity of checking vs reached/frontier states' sets) We've added also some counters for the created and expanded nodes through the search.

2.2

The amount of states expanded were 31.
BTW, the amount of created nodes was 36 (about 5 nodes remained in the

frontier)

2.3

This is the same as previous section as we've optimized to create only none reached (not in CLOSE set) and none frontier (OPEN) states.

2.4

The main advantage of the BFS/BFS-G compared to DFS (tree search) is that it's a complete (guaranteed to find a path to the goal) and optimal (that path is the optimal path to the goal state), while DFS (tree search) isn't as it may stuck in infinite loops repeating the same states.

In our taxi env, the state space is finite, so if we use DFS-G with graph search (next section), we will find a solution but it may not be optimal.

The disadvantage of BFS is its memory consumption which is $O(b^d)$ while the DFS-G $O(bD)$ (with backtracking $O(D)$) When D is the max depth of the taxi env when we start from some state (bounded by the state space size). DFS with tree search may not end and have an infinite memory consumption as we can repeat the same state in the tree (circles). One can run DFS-G or add some circles detection mechanisms (going back several parent nodes for example) as described in the course book.

2.5

Yes. In the taxi environment it will always find the optimal solution, as we saw in the lecture,

BFS finds the shortest path (number of edges) to any state in the env. In our case, the taxi env, the edges (actions) doesn't have the same cost but we will still find the optimal solution as the shortest path in edges is also the shortest path in cost.

Proof

Completeness: Assuming a solution of cost (depth) d exists. BFS will return the solution before exposing the d+1 layer, We know it will reach the d+1 layer as it works in layers and we can by induction prove that it finds all the nodes in the next layer.
Optimal: Assume that a solution of cost (depth) d exists and the BFS returns a none optimal solution of cost $d' > d$.
It will return the optimal solution before exposing the $d + 1$ layer so we get a contradiction. ■

3 DFS

3.1

We've implemented the DFS-G similar to BFS-G but the strategy to pop from the frontier (OPEN) was based on LIFO (stack) and not FIFO(queue) as in BFS.
Initially, we've implemented a tree search in which the algo got into infinite loops since we didn't check vs reached states (CLOSE).
We've also played with the tie breaking between neighbor nodes (by random shuffle) or by just reversing the order of addition to frontier and got different paths.
This is not surprising, as DFS-G may return some solution which is not optimal. Again, DFS-G ***on finite state space*** , as in the taxi world, is a complete search algorithm (DFS tree search is not since we can get stuck in a loop) but not optimal.

3.2

The number of states expanded when we didn't find the optimal path were 96.
BTW, we experimented with reversing the list of neighbors of the node expanded and got much better results. so we can see DFS is not proven to get the optimal results.

3.3

Since we expand only different states as we run DFS-G, the number of states expanded is the same.

4 ID-DFS

4.1

Implemented the ID-DFS by calling DFS-L with increasing depth starting from 0 depth.

4.2

We got that the total expanded nodes (sum of all nodes expanded in all depths runs) is 186.
This is the summation of the following list which stands for number of expanded nodes in each ID-DFS iteration (increasing depth):
expanded_nodes = [1, 4, 8, 13, 19, 24, 27, 28, 30, 32]

4.3

As we repeat the same expanded nodes when we increase the depth (and add more new states we didn't reach at the max depth), the amount of different states developed is actually the last iteration expanded nodes calculation. In our environment and start state, as we saw on previous section, these are 32 different states.

4.4

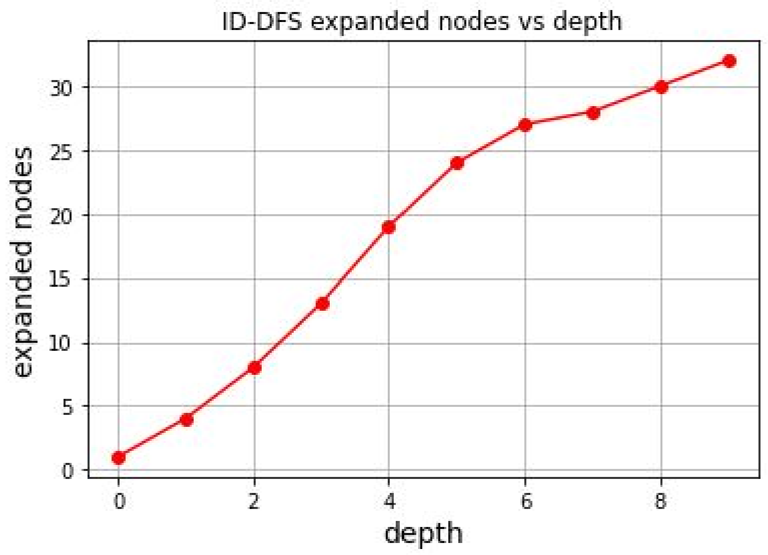
The ID-DFS advantage compared to DFS-G is that it's complete and admissible, while DFS-G on finite state space is only complete.
The disadvantage of ID-DFS is that it expands the same nodes multiple time and not just once as in DFS-G.

4.5

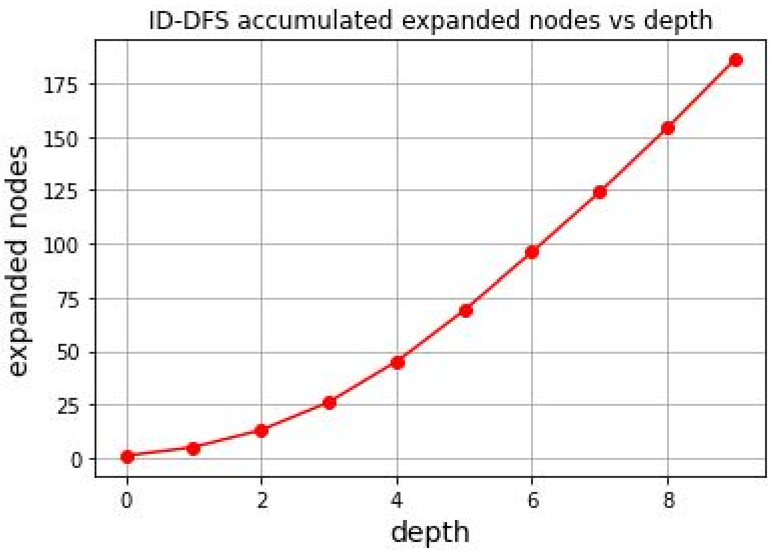
The advantage of ID-DFS compared to BFS is its memory complexity. In ID-DFS we have $O(bD)$ (b - branching factor, D - is maximal depth from start state as we run DFS-G on finite state space), while in BFS it's $O(b^d)$ (d - optimal path depth).
The disadvantage of ID-DFS compared to BFS-G is that it expands the same nodes several times while BFS-G expands each node only once.

4.6

The following list stands for number of expanded nodes in each ID-DFS iteration (increasing depth):
expanded_nodes = [1, 4, 8, 13, 19, 24, 27, 28, 30, 32]



As we go deeper at each iteration of ID-DFS, we expand states that we haven't visited before due to limited depth.



Above, we see the accumulation of expanded nodes vs max depth.

5 WA^*

5.1

see 4.6 section

5.2

Implemented the WA^* in code.
When w=0.5, we get the A^* search.

5.3

Yes. we can expand a reached state (after it's in the CLOSE set) and re-add it to OPEN and expand again:

```
if s not in OPEN and not in CLOSE then
...
else if s in OPEN then
...
```

```
**else**                                                    ▷ s in CLOSED
    **if** $new\_g < n\_curr.g()$ **then**
        $n\_curr \leftarrow update\_node(s, n, new\_g, new\_g + h(s))$
        $OPEN.insert(n\_curr)$
        $CLOSED.remove(n\_curr)$
    **end if**
**end if**
```

As shown in pseudo code above, when the next state (neighbor of the now expanded node) is already in CLOSE set we ***update*** it's f value and re-add it to the OPEN set. BTW, In case it's in OPEN we also update it.

5.4

When called with the null heuristic we get the UCS solution with 34 states expanded with the optimal cost (10 actions as before, with total reward of 11). Note that since the A^* is implemented for admissible heuristic, the expanded nodes are more than in BFS, as we finish only when we expand the goal state (not when encountered as in BFS with 31 nodes expanded).

5.5

The optimal cost defined (as was defined in BFS) is the smallest amount of actions to take from start state to goal state. We define the cost of each action as 1.
The first 2 heuristics are admissible while the last 2 are not.

1. ***Greedy***:

Since our closest state before the goal is the state just before we drop the passenger at its destination (and thus with the smallest cost to go as we defined it).
We get

$$\forall s \in S, s \neq goal\_state \text{ we have } 1 = h(s) \leq h^*(s) = cost\_to\_go(s)$$

and

$$s = goal\_state, 0 = h(s) = h^*(s) = 0$$

Thus, the Greedy heuristic is admissible. ■

2. ***Manhatan Sum***:

We can relax the problem (apply less constraints) and look at it as if there are no walls between different locations on the map, providing we can still move only North/South/West/East.
Thus, this is a tighter lower bound relative to the Greedy heuristic and therefore more informed.
The Manhatan Sum is admissible as at the goal state the location of the passenger is the same as the location of the taxi $MD(s, loc_{passenger}) = 0$ and the same as the location of the drop-off $MD(loc_{passenger}, loc_{drop-off}) = 0$.
Thus, for the goal state we have

$$s = goal\_state, 0 = h(s) = h^*(s) = 0$$

The MD by definition is zero or positive, so:

$$\forall s \in S \text{ we have } 0 \leq h(s)$$

As we wrote above, the actual cost to go can only be greater once we add walls to the map (As in our original problem), so in total:

$$\forall s \in S \text{ we have } 0 \leq h(s) \leq h^*(s) = cost\_to\_go(s)$$

Moreover, the MD doesn't take into consideration the actions of pickup and drop-off so it's even strictly less then the cost to go for all states that are different from the goal (we can even create a better informed heuristic). ■

3. ***Pickup Sum***:

The Pickup Sum heuristic is not admissible as the cost to go on goal state might be different than 0 when the pickup location is different from the drop of location.
The cost to go at goal state is by definition 0, and

$$MD(loc_{pick-up}, loc_{drop-off}) > 0$$

In our main test case in the wet code:

$$MD(loc_{pick-up}, loc_{drop-off}) = 4$$

So to conclude, it's not admissible as

$$h(goal\_state) > h*(goal_state) = cost\_to\_go(goal\_state) = 0$$

■

4. ***Pickup Mult***:
The Pickup Mult heuristic is not admissible as the cost to go on goal state might be different than 0 when the pickup location is different from the drop of location.
The cost to go at goal state is by definition 0, and

$$MD(loc_{pick-up}, loc_{drop-off}) > 0$$

And also

$$MD(goal\_state, loc_{pick-up}) > 0$$

Thus the multiplication is also positive.
In our main test case in the wet code:

$$MD(loc_{pick-up}, loc_{drop-off}) = 4, MD(goal\_state, loc_{pick-up}) = 4$$

(when distance of goal state to pickup is actually the also the drop-off location). So to conclude, it's not admissible as

$$h(goal\_state) > h*(goal\_state) = cost\_to\_go(goal\_state) = 0$$

■

Between the 2 admissible heuristics (***Manhatan Sum*** and ***Greedy***) we cannot say that one is more admissible than the other as on the state right before the drop and reaching the goal state we have $0 = MD(s) < Gready(s) = 1$, while on every other state $MD(s) \geq Gready(s)$ (the definition for more informed stands that it's $MD(s) \geq Gready(s)$ for all states).

Moreover, the distance doesn't take into consideration the actions of pickup and drop-off so it's even strictly less then the cost to go for all states that are different from the goal (we can even create a better informed heuristic by adding 1 or 2 to none goal state).

If we fix a bit the MD distance and add 1 or 2 (depends if passenger was picked up or not yet), the most informed heuristic between the above is the ***Manhatan Sum*** as it's the closest to the real cost to go from any state.

5.6

Implemented in the notebook the ***Manhatan Sum*** as the chosen_h

5.7

With the ***Manhatan Sum*** heuristic we've chosen (and with $W = 0.5$ for pure A*) we've expanded only 13 nodes before expanding the goal state and returning the optimal path.
In the blind search, with the null heuristic ($W = 0$), we've expanded 34 nodes expanded (BFS-G got it with 31 nodes expanded but UCS as implemented for A* for admissible heuristics must end only after expanding the goal state. if it was consistent it was less as in BFS-G).

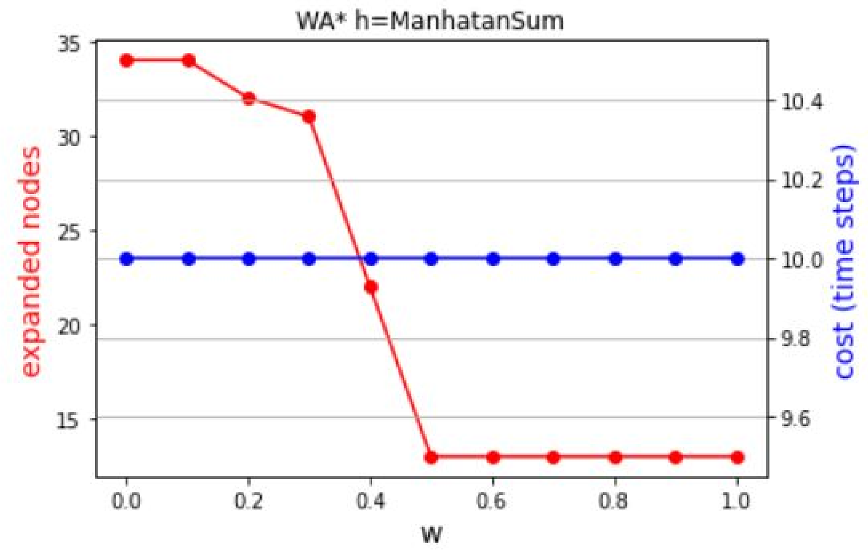
In total we've saved $34 - 13 = 21$ expansions saved.

5.8

Implemented in the notebook. We've added the parameter to our common code for all A* version ($W - A*$ and $A * -\epsilon$)

5.9

When we run W-A* with different W values $W \in [0, 1]$, we get the following graph:



We have printed the cost in time steps. The reward is the same for all as in all we've found the optimal path (10 time steps, reward=11)

5.10

When we run WA* with $W = 0$ we should get UCS and get the optimal cost and path. We do get it but the number of expanded nodes is 34 vs 31 in BFS-G.

The reason for that is that in A* we don't finish our search when we encounter the goal state as part of the neighbors of an expanded node but rather finish only when we expand the goal node itself. This is why we have 3 extra expanded nodes in WA* and USC cost as implemented this way is bit more costly than BFS-G.

It is better to work with $W \in [0.5, 1]$ as A* ($W = 0.5$) uses the heuristic w/o lose of optimal cost (still complete and admissible search) when the heuristic is admissible. We can also view the improvement in speed as we increase W $W \in [0, 0.5]$.

In the region $W \in [0.5, 1]$ in general $W - A*$, we have a trade-off between speed (approaching Greedy search) and A*.

That is, we risk having none optimal solution but with faster (less expansions) reach to goal when we increase W above 0.5.

However, if we take $w_1 < w_2$ in our graph, we can see that the solution we get with both is the same (for example when $(w_1, w_2) = (0.4, 0.5)$ or $(w_1, w_2) = (0.5, 0.6)$), we can also see that for such pairs the number of expanded nodes doesn't necessarily decrease as we increase W (for example when $(w_1, w_2) = (0.5, 0.6)$ or $(w_1, w_2) = (0.6, 0.7)$).

In our diagram, we don't see that we lose/gain from changing W in $(0.5, 1]$. It returns the optimal path and it does it with the same amount of expanded nodes as A*.

6 $A^*\epsilon$

6.1

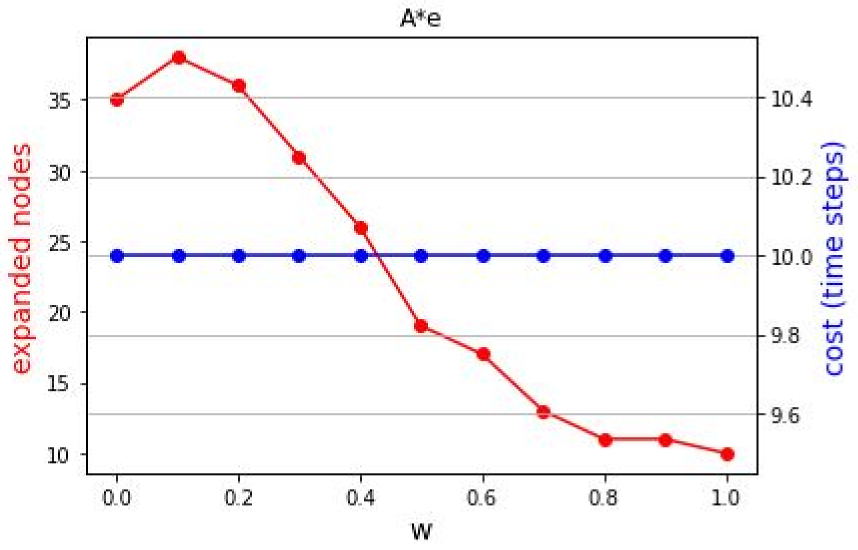
Implemented in the notebook. We've added the parameter to our common code for all A* version (WA^* and $A^*\epsilon$)

6.2

We've used the ***Greedy*** heuristic from previous section as our admissible heuristic.
For the none admissible heuristic, we've took improved (more informed) ***Manhatan Sum*** by +1 or +2 (depends on state) since originally we didn't take the drop/pick operations distance.
Then, we multiplied it by 1.1 and it has become none admissible any more as we've got $h(s) > h^*(s) = g(s)$.

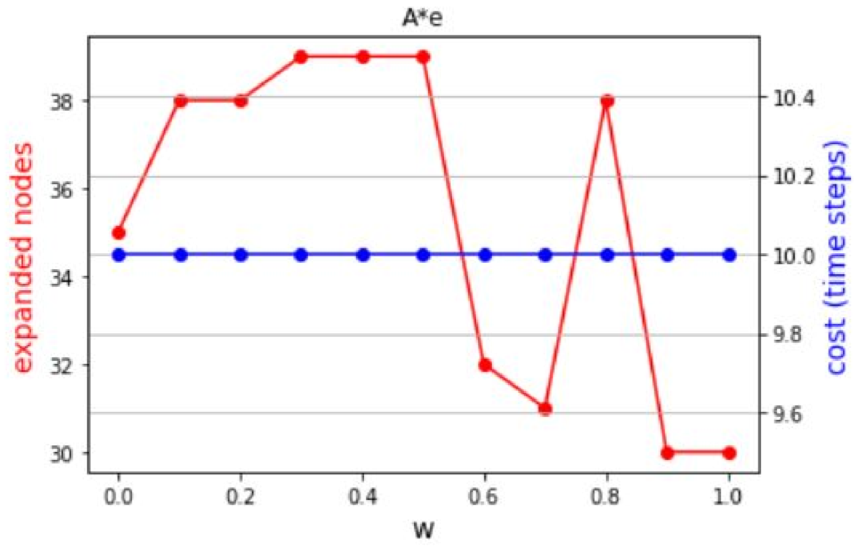
6.3

When we run $A^*\epsilon$ with different W values $W \in [0, 1]$ and $\epsilon = 0.3$, we get the following graph over the none admissible heuristic:



We can see that when $W = 1$ the amount of expanded nodes (for some random selection from focal set) was reduced from 13 to 10 (the minimum as possible, as if we selected the nodes on the optimal path only).

Moreover, when running $A^*\epsilon$ with the admissible heuristic that we chose (***Greedy***) with the same $W = 1$ and $\epsilon = 0.3$ we got 30 nodes were expanded vs the none admissible with only 10 (expanding only the nodes on the optimal path).



When we change the $A*$ to $A*\epsilon$ we sacrifice possible loss of optimal cost finding (not admissible) in order to add greediness for faster move towards the goal (with the guarantee of having an upper bounded degradation in the optimal cost found up to $C \leq C^*(1+\epsilon)$).

7 Test like question

7.1 Should-Return in A^*-Parallel

1.

Complete - as we find the goal state and return it (the first time we encounter it)

Not admissible - as we return at the first time we find the goal state. but there might be a better path to the Goal state which we haven't expanded yet. Meaning, we've added to open the goal state in a parallel worker process which has a lower f value and due to race we've returned the goal state with higher f value (which is the real cost g value as $h(goal\_state) = 0$)

2.

Not Complete - as there might be a busy worker in parallel and we return False. Thus, we might miss the only path that reaches the goal state and never return it.

Not admissible - as it's not complete

3.

Complete - as opposed to prev solution (2), we push the solution to open before we check if all workers are idle, this way we avoid the chance that we won't get back to the goal state due to some parallel worker still working to expand some other nodes. This way, even if we return False now, eventually some worker will be the last one to expand the goal state while all other workers are idle and we will return a path to the goal.

Not admissible - the solution found might not be the optimal one as there might be some worker that reached the goal last and pushed to open but the f=g value we currently have is not the optimal. Meaning, another worker pushed to open the goal state from a different path with lower f=g value and we missed it as all workers were idle when we checked.

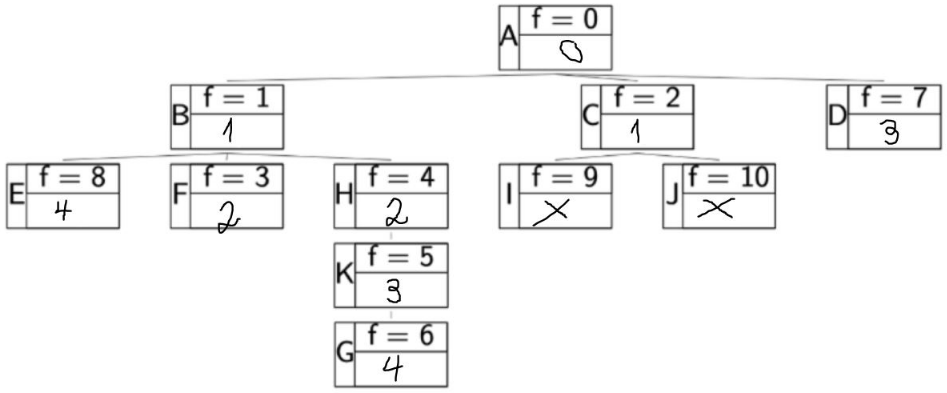
4.

Complete - once we find a goal state as in (3) we push it to the open again to guarantee that it will be poped from heap eventually as the last time might be when the goal state is the only one in open and thus the last condition when we check the front f value vs the current goal state must be True and we return this path to goal.

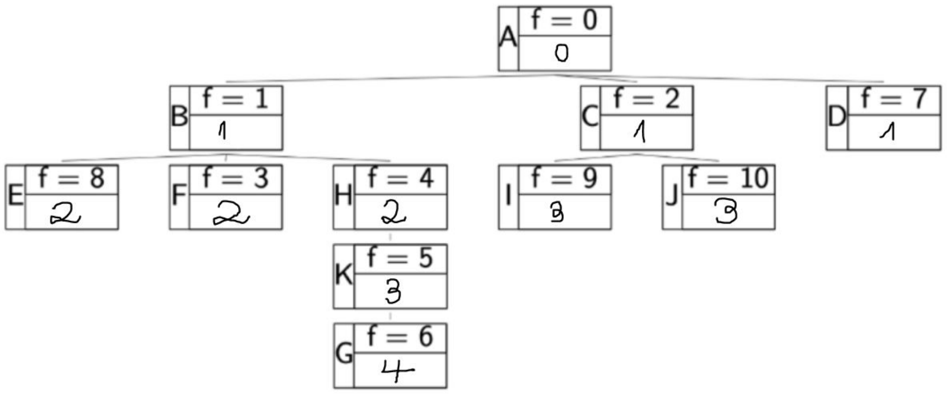
Admissible - the solution returned is also optimal as if we pushed the to open after all workers are idle (no more nodes to add to open that might contain another path to goal state) and thus all possible paths to goal with different f values are in open, what's left is to add current goal node again to open (in order to compare with other goal paths f values) and if it has the lowest f value (Get-Front(open) of the heap returns it) we have found the goal node that has the optimal parent and return it.
if it's not the minimum f value, that is we have another goal node with better path to it, we will not return it and wait for the master process to pop next (other none goal node) the best goal state with the minimum f value and return it.

7.2 Order of expansion

1. השלם את זמני הרחבת הצומת במקרה של **שני עובדים** ומלא 'X' עבור כל צומת שאינו פותח. (2 נק)



2. השלם את זמני הרחבת הצומת במקרה של **שלושה עובדים** ומלא 'X' עבור כל צומת שאינו פותח. (2 נק)



A f = 0
0
B f = 1
1
C f = 2
1
D f = 7
1
E f = 8
2
F f = 3
2
H f = 4
2
I f = 9
3
J f = 10
3
K f = 5
3
G f = 6
4