**Introduction to Artificial Intelligence**

**236501**

****

**Submission due date: 24.05.2023**

**Submitters:**

**Ophra Journo : 342791456**

**Ayelete Berkovits:**

**Question 1: Introduction**

1. Read the notebook.
2. We’ll define the Search space (מרחב חיפוש) formally :

(S,O,I,G) such that

S={0,1,2…,63}

O={Up, Right, Down, Left}

I= {0}

G={63}

The size of the state space S is 64. The state is defined by the location of the player on the board so there are 64 states.

1. Domain(UP)={ x | the case x is not a hole}:

Domain(UP) corresponds to all the states in which UP can be applied. In our case UP can be applied from every case that is not a hole (because a hole means the game is terminated). It can even be applied on the top line (it is defined in the Notebook that the player will stay at the same case).

1. Succ(I)={0,1,8}:

Succ(I) corresponds to all the states to where an action from I can lead: If the next action is Up or Left, the player stays on the same case 0, if it’s Right then the next state is case 1 and if the action is Down then the next case is 8:

1. Yes there are cycles.

For example the 4 actions followed from Initial State (in order down right up left) will lead to the same initial state so it will form a cycle.

1. The branching factor is b=4:

It is the maximum among all numbers of successors each state has. So in our case there are 4 actions that can lead to 4 different states for the middle cases without holes around.

1. As we’ve seen in the Notebook it can be infinite.

It can randomly moves and never reach the goal state.

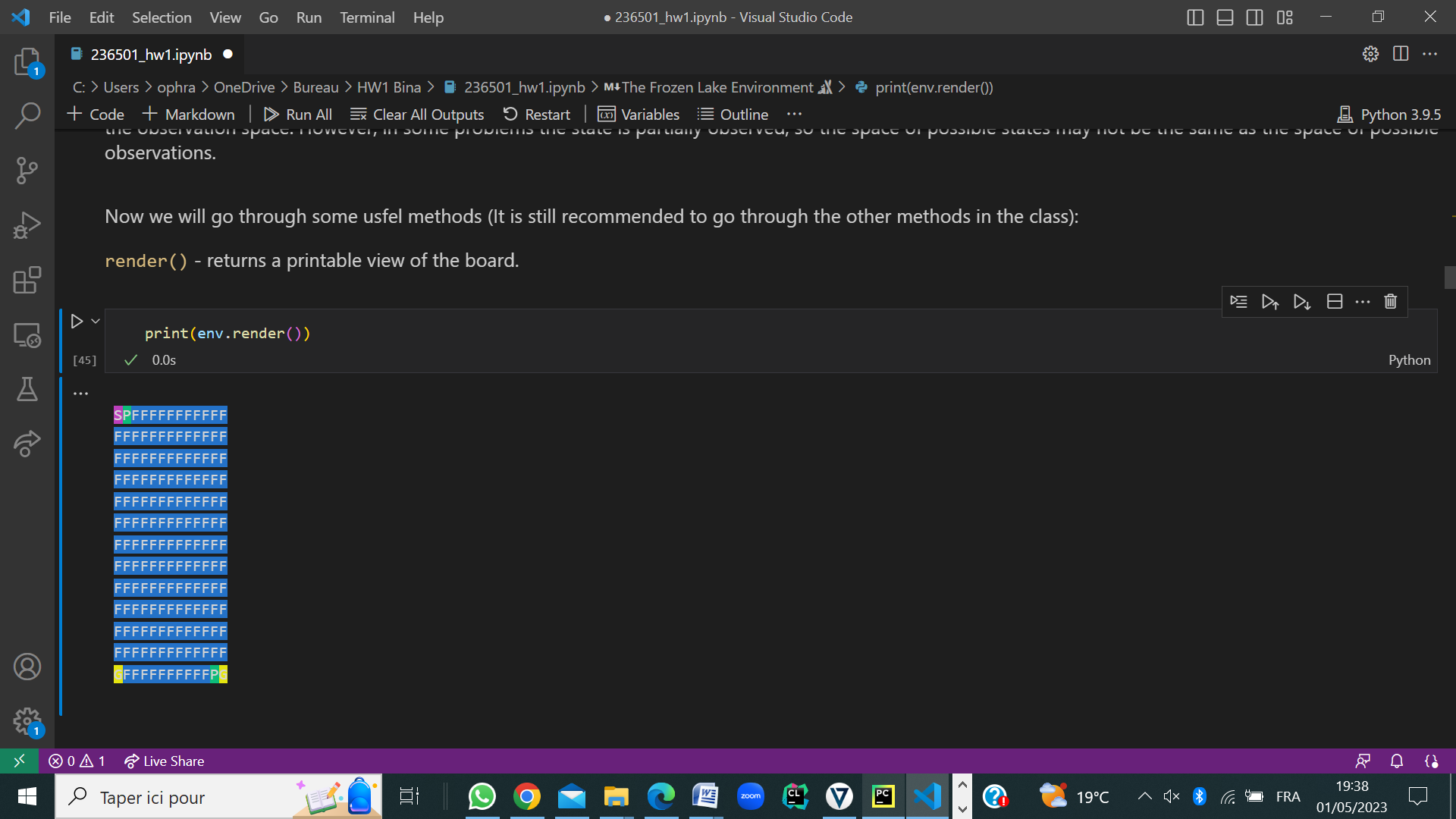
1. In the best case 9 actions are necessary:

Action Right

Action Down

We can count 9 actions.

1. Wrong. Example:

In a NxN board where N=13 where the Initial case is in case 0 and there are 2 goal states: G1 in case (N-1)\*N (bottom left corner) and G2 in case N2 -1 (bottom right corner). There’s a portal door in case 1 and one in case N2-2 and all the other cases are F cases.

* G1 is closer to the Initial state (12 cases away in Manhattan distance) and the cheapest way to get there is with 12 ‘Down’ actions (white path): 11 into F cases that will cost 110 and one into G case that’ll cost 1, so in total **111**.
* G2 is farther from the initial state (24 cases away in Manhattan distance) and the cheapest way to get there is through the portal that will cost 100 and then one Right action into the G case that will cost 1 (black path), so in total **101**.

**The least cost path** is the one that reaches the goal that is **not the closest** to the Initial State in Manhattan distance so the statement in this question is wrong.

**Question 2: BFS-G**

1. **Wet part –** Implementation of BFS-G in code.
2. A difference between BFS on a graph and BFS on a tree is that when we are doing the algorithm on a graph we have another set where we can keep and remember the nodes that we already opened to avoid revisiting them- so we didn’t open a node few times like it can happen on a tree.

But in both we are walking on the tree we the same logical, we always choose the shallowest node to open.

So the condition on the search graph so that BFS on a graph and BFS on a tree generate and expand identical nodes in the same order is that for each node it exist just one way to join them. With this condition we can be sure that we will not visit node two times in the tree. And we will generate and expand the nodes in the same order like in the graph.

1. State graph of the “4×4” shown in the notebook:

Like we can see in the notebook and like we explained in the first question; there are 16 states, that are the 16 cells where the agent can stand up. We can move from a state to another if the agent is avoid to move between the two cells. There isn’t any portal on this board so the only moves that the agent can do are: right, left, up and down. Obviously, if the agent is in a hole he is stuck so we didn’t allow any move from a hole.

So the state graph is the follow:

Diagram, schematic

Description automatically generated

1. Given an N×N board without portal, we will use the BFS-G algorithm to find an optimal solution (with a minimum cost).

In G, the cost of each transition is based on the mark of the cell we pass to. So, each node on the graph G has a weight.

When I will run the algorithm BFS-G, it will return the shortest path from the initial state to the goal state. So the idea is to transform a node with a cost of a to a nodes separated by a edges with a unified weight of 1.

We can start to build a new graph G’ from G as follow:

For each node in G:

* If the cost of the node is 1 we will stay the node with his incoming and outgoing edges like in the initial graph G.
* If the cost of the node is a> 1, we will duplicate the node to have a nodes separated by edges like a lanyard.
* All the incoming edges to the initial node will be incoming edges to the first node of the lanyard and all the outgoing edges from the initial node are going to come out from the last node of the lanyard.
* The weight of all the edges is unified to 1.

And with this new graph, when we will run BFS-G on the new graph we will find the shortest path to the goal state that is if we join all the lanyards, the path with the minimum cost in the initial graph.

1. Given an N×N board without holes and portal. The board is composed of the cell of the initial state, the state of the goal state and regular cells.

During the BFS-G search, the number of expanded nodes is:and the number of openednodes is:

The BFS-G is a search algorithm that works in layers, he is first visiting all the nodes at a distance of 1 from the initial state before all the nodes at a distance of 2 from the initial state…

About the expanded nodes, the goal state has 2 neighbors that both are at same distance from the initial distance (-1 than goal state). When we will visit the first of the two neighbors, we already expanded all the nodes in the graph except 3 (the goal state and his two neighbors) and so we will expand it and find in his neighbors the goal state. And we will stop the search with no need to expand the second neighbor and the goal state. That’s why we expandedstates.

About the opened nodes, we have to count the number of times that we are doing the action “make node”. At the beginning we are opening the first node by creating it (the initial state) and after for all the cells that we talked about them before (that we are expanding) we are doing the action of “make node” 4 times (for his 4 neighbors). Even if he hasn’t 4 different neighbors (because that he is on the first line for example) he will make 4 nodes because it’s defined that the up neighbor of a state of the first line is itself so he will make again itself. From this, we have to subtract 2 because we will find the goal state by his left neighbor when he is making his second son (the goal state) and so he will not make his two last neighbors.

That’s why we have:

**Question 3: DFS-G**

1. **Wet part** – Implementation in the code Algorithms.py
2. The DFS-G algorithm is complete but is not optimal. In the worst case, the algorithm goes through the whole board so it will find a goal if there is one. It is not optimal because BFS-G is optimal. It can find a goal that is not the closest from the Initial State because it’s going through the board in depth and doesn’t check for closer cases.
3. No it wouldn’t. The DFS algorithm on a tree doesn’t keep in memory the visited states so it will go down to the bottom left corner and will continue to do the action “Down” and revisit the same case again and again.
4. Expanded : 2n-2 Opened: 4n-4

The DFS-G algorithm will traverse the n-1 rows by moving down and expand every time the first neighbor but will open the right neighbor on every row. Then it’ll go through the last row until the last column by expanding the right neighbor every time and opening the neighbor above (not the left one because already expanded).

1. Expanded: 2n-2 Opened: 2n-1

The DFS-G backtracking algorithm will take the exact same path as DFS-G except it will open the node in a lazy way which means it won’t open the right neighbors when going down through the first column and won’t open the neighbors above when going through the last row so it will open 2n-1 nodes (the ones expanded because of the order of the actions) and won’t expand the last one so 2n-2 expanded.

**Question 4.: DFS-L**

We want to find a path in the frozen lake using DFS-L. Given that the path length to the closest goal is d. But Ricky limited the search to a depth of d/2.

1. We will modify the search space (S,O,I,G) to be able to find a path without exceeding the limit imposed by Ricky.

We let the possibility to make leaps of 2. The meaning is that if in the original search space I can go from the state 1 to the state 2 and from the state 2 to the state 3, in my modified search space, I’m giving the possibility to go straight from the state 1 to the state 3. And now, we have the possibility to find the goal with just a depth of d/2.

So the modified search space is the below:

,

O’= { Left-Left, Left-Up, Left-Down, Right-Right, Right-Down, Right-Up, Down-Left, Down-Right, , Down-Up, Up-Left, Up-Right, Up-Up} U O.

I’=I= {0}

G’= G={63}

1. Yes, the branching factor has changed. It was 4 and now he is 13.

Because we have 13 successors states, the 4 initials, the four diagonals, the 4 possibilities do move two times in the same direction and also the state from where we started itself (if I do the move up and after the move down I’m staying at the same place so I’m my own successor.

For example, if I’m on the state 18, the successors states are: 17 (left), 19 (right), 10(up), 24(down), 16 (left-left), 20(right-right), 2 (up-up), 32 (down-down), 9 (left-up), 11 (right-up), 23(left-down), 25(right-down) and 18 (myself).

We can prefer the second formula because she has a meaning. For each one of the b successors I can have b successors (that’s the meaning of the ) and I have to substract the commun successors (there are b like that) and to add the state itself.

1. We will compare the time and space complexity between DFS-L regular with a depth of d and our DFS’ (in the new search space).

|  |  |  |
| --- | --- | --- |
|  | Time Complexity | Space Complexity |
| DFS’ (in the new search space with depth of d/2 |  |  |
| DFS-L (depth of d) |  |  |

The answer will not change if you use DFS-L with backtracking about the time complexity.

But the space complexity may be lower in some cases because we don’t have to save all the visited nodes but just those that I need in the recursion (maximum d/2).

1. Who is better?

For the next examples, we are going to define that the new edges, those that can do leaks of 2 have priority.

Let be state 1 be the initial state, state 3 be the goal state.

The black arrows represent the operations in the initial search space. And the red arrows are the operator that we add in the modified search state.

Example when modified search is better:

In the regular DFS-L, the number of expanded nodes is 2 (state 1 and state 2).

However in the modified algorithm, the red arrow has a priority so we have just one expanded node (the state 1).

Example when regular DFS-L is better:

Here, in the regular DFS-L the number of expanded nodes is 1 (just the state 1).

However, in the algorithm in the modified space search we are going from the state 1 (by expanding him) to the state 2 and here we will expand the state 2 and after come back to the state 1 and go to the state 3 to find the goal. So the number of expanded nodes is 2 (states 1 and 2).

**Question 5: ReverseDFS**

Let we assume that we have some prior knowledge about an upper bound from the depth of a goal state denoted D.

1. The ReverseDFS algorithm is complete.

It’s meaning that is guaranteed to find a solution if one exists. This is because like we proved in class the DFS-L algorithm is complete with a finite distance. Since the distance is finite, the algorithm is guaranteed to terminate and either find a solution or determine that no solution exists.

Since we know from the rules of the frozen lake that there exists a solution in the diameter of search of D and since DFS-L algorithm is complete so the ReverseDFS algorithm is also complete.

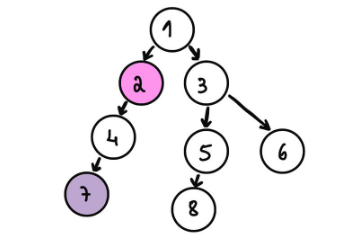
1. But the ReverseDFS algorithm is also optimal. The algorithm pass at all the possibly depths.

Because we are passing at all the depths from the D that is an upper bound from from the depth of a goal state to zero (decreasing the depth at each iteration by 1). And for each depth we are running the algorithm DFS-L. At each iteration (=each different depth) if we find a solution we improve the final solution if the solution that we find is better.

And because we are decreasing until 0, it’s sure that if exists we will find the optimal solution.

1. Comparisons between ReverseDFS and ID-DFS.

The main difference between both algorithms is that in ID-DFS we are starting from the beginning, from a depth of zero and increasing it at each iteration. And in ReverseDFS we are starting from the maximal depth and decreasing at each iteration.



(The draw is for both two examples)

Example when ReverseDFS is better than ID-DFS:

In this example, the only goal state is 7 (in purple on the draw), at a depth of 3 (the maximal depth). With ReverseDFS, the search will start at the depth 3 and will find the goal. After that we are decreasing to the depth 2 by trying to improve this goal. But in the next iteration (depth 2) we will not find the goal and so we will return failure for this iteration. That’s why we will return the first solution that we found. However, the ID DFS will start with depth 0 and after increasing it at each iteration. And it will take more time until we achieved the depth 3 and find the goal state.

Example when ID-DFS is better than ReverseDFS:

No, it’s the contrary. The goal state is 2 (in pink on the draw), at a depth of 1. The algorithm will find it at the depth 1 from the beginning and will return it.

However in Reverse DFS we are starting with the biggest depth and at each iteration trying to improve the solution until we will return failure (didn’t find any goal) at some iteration. So, we will run the DFS-L with a depth of 3, of 2, of 1 and for all we will find a goal that is at the depth 1. Until we will run it with a depth of 0 and return failure.

1. How to make the algorithm more efficient?

It could be to use adaptive depth limits, where we adjust the depth limit L in theReverseDFS algorithm dynamically based on the results of previous iterations.

For example if we find a goal state at certain depth level, we could increase the depth limit L for the next iteration to allow for deeper iteration. However if we not find a goal state at a certain depth level, we could decrease the depth limit L for the next iteration to focus on shallower exploration.

By using those adaptive updates, we can improve the efficiency of the ReverseDFS algorithm and make it more effective at finding optimal solutions in a better time.

**Question 6 : UCS**

1. **Wet part –** Implementation of UCS algorithm in code.
2. The purpose of the BFS algorithm in a search problem is to find the shortest path from the initial state to the goal state accorded to the distance between the two states (in terms of number of edges between the two states).

On the second side, the purpose of the UCS algorithm, is to find the cheapest path from the initial state to the goal state in terms of cost of the nodes that we visit in our search path.

BFS and UCS algorithms behave the same in search problems where all the nodes of the same level (= at the same distance in terms of number of edges from the initial state) have the same cost.

In this search problem, at which time UCS will choose an edge in the same level that is (until now) the shortest path until now. It’s the same way BFS works.

1. For an N×N board the UCS algorithm is also complete and also optimal.

We learned in the class that the algorithm is complete if the cost functionis blocked from below or if the number of states is finite. Here, the number of states is finite (, we didn’t pass on the same state more than one time so, in the worst case we will pass on all the states and at the end we will find a solution.

The UCS algorithm is optimal because he is based on Dijekstra algorithm. When we take out a node from the OPEN queue, the cost of the path until this node is the cheapest (because of the mode of operation of the algorithm). That’s why when we take out the goal state the path to the goal state is the cheapest.

1. Dan made a mistake in the implementation of the UCS algorithm. He checked if the new node is a goal node when we are opening it instead of when we are expanding it. It can work and return the optimal path but it can also doesn’t work and not returning the optimal path.

An example when it’s work:

Diagram

Description automatically generated

Here, the path that the algorithm, will choose is the first edge with weight 0 and after again the second edge with weight and will come to the goal with the good path.

An example when it doesn’t work:

Diagram

Description automatically generated

Here, the path that the algorithm, will choose is the first edge with weight 0 and after again the edge from c to the d with the weight of 11.

We find a path to the goal state d but it’s not the good one.

The problem is because we are checking if a state is a goal state when he is opened and not when he is expanded. So we are maybe pass over other better options in terms of cost. In the example that doesn’t works, if I were check if the node d is the goal state in his expansion (like in the real correct UCS) and not in his opening (like in the error of Dan) it would only be after we went over the second track and updated the good price to the goal to be 10.

**Question 7 : Heuristics**

1. Yes: h1 and h2 are admissible heuristics so for all states s:

and (when h\* is the perfect heuristic).

Let’s define h= min{h1,h2} then for all states s, as so for all states s: then h is an admissible heuristic.

1. Yes: h1 and h2 are admissible heuristics so for all states s:

and (when h\* is the perfect heuristic).

Let’s define h= max{h1,h2} so for all states so as for all states s then h is an admissible heurisitic.

1. Yes: h1 and h2 are consistent heuristics so for all states s,

for all s’=Succ(s): h1(s)-h1(s’) cost(s,s’) and h2(s)-h2(s’) cost(s,s’) and for every state g that is a goal h1(g)=h2(g)=0.

Let’s define h=min {h1, h2}:

For every state g that is a goal h(g)= min {h1(g)=0, h2(g)=0} = 0.

For all states s and all his successors s’, let’s split in cases:

* If h1(s)<h2(s) and h1(s’)<h2(s’) then h(s)=h1(s) and h(s’)=h1(s’) so h(s)-h(s’) cost(s,s’).
* If h2(s)<h1(s) and h2(s’)<h1(s’) then h(s)=h2(s) and h(s’)=h2(s’) so h(s)-h(s’) cost(s,s’).
* Without limitation of generalityif h1(s)<h2(s) and h2(s’)<h1(s’) then h(s)=h1(s) and h(s’)=h2(s’) and we know that h2(s) cost(s,s’)+h2(s’) because h2 is consistent so we obtain that

h1(s)<h2(s) cost(s,s’) +h2(s’) so h1(s) – h2(s’) cost(s,s’) and so h(s)-h(s’) cost(s,s’)

In every case for all s and s’, h fulfills the condition so h is consistent.

1. Yes: h1 and h2 are consistent heuristics so for all states s,

for all s’=Succ(s): h1(s)-h1(s’) cost(s,s’) and h2(s)-h2(s’) cost(s,s’) and for every state g that is a goal h1(g)=h2(g)=0.

Let’s define h=max {h1, h2}:

For every state g that is a goal h(g)= max {h1(g)=0, h2(g)=0} = 0.

For all states s and all his successors s’ let’s split in cases:

* If h1(s)<h2(s) and h1(s’)<h2(s’) then h(s)=h2(s) and h(s’)=h2(s’) so h(s)-h(s’) cost(s,s’).
* If h2(s)<h1(s) and h2(s’)<h1(s’) then h(s)=h1(s) and h(s’)=h1(s’) so h(s)-h(s’) cost(s,s’).
* Without limitation of generalityif h1(s)<h2(s) and h2(s’)<h1(s’) then h(s)=h2(s) and h(s’)=h1(s’) and we know that h2(s)-cost(s,s’) h2(s’) because h2 is consistent so h2(s)-cost(s,s’) h2(s’) < h1(s’) so we obtain that h2(s) - h1(s’) cost(s,s’) so h(s)-h(s’) cost(s,s’).

In every case for all s and s’, h fulfills the condition so h is consistent.

1. Yes the heuristic is admissible for all boards.

Let’s assume a board of size NxN and the current state is s. Let’s prove that , when is the optimal cost of the path between s and the goal state g. In the optimal path P from s to g:

* If there is no portal, the cost of P is at least equal to (the total number of edges between s and g \* the minimum cost for an edge which is 1) so in total the cost of P will be at least the number of edges between s and g. Because the actions are vertical or horizontal, the Manhattan distance from s to g will be smaller or equal to this optimal cost so: .
* If there is a portal in P then the cost of P is 100 so will be equal to 100 (minimum between cost of path including the portal or the edge leading to the portal) and in this case too .

In every case, so the heuristic is admissible.

1. Yes, is consistent:
2. For the goal g, h(g)=0 because (g,g)=0 and it is the minimum.
3. Let’s prove that for all states s and its successor s’

For all state s and successor s’:

* If there is no portal going from s to s’ , h(s)-h(s’)=1 because the actions are only Down, Right, Left, and Up and those actions create a difference of Manhattan Distance of 1 exactly and we know that so
* If there is a portal to go from s to s’: because h is the minimum between 100 and the Manhattan distance from the goal. So and so

This is true for all state s and all its successors s’ so the heuristic h is consistent.

1. Yes: Let’s assume a board of size NxN and the current state is s.

Let’s prove that , when is the optimal cost of the path P among all paths between s and all goals g in G. In the optimal path P from s to g:

* If there are no portals there the Manhattan distance of (s,g) for every goals is smaller than the cost of the path P from s to g that is at least going through every edges in path P with cost at least 1 so in this case .
* If there is a portal then =100 as explained in sub question 5 and is smaller than the cost of P which is at least 100. So in this case too .
* In every cases, so the heuristic is admissible.

1. Yes, is consistent.
2. For all g in G: h(g)=0 because (g,g)=0 and it is the minimum.
3. Let’s prove that for all states s and its successor s’

* If there are no portals from s to s’: Let’s prove that .

We know that |G|>=1 so there is at least a goal g1 such thath(s) = hManhattan(s,g1)=x.

* If there’s no other goal g2 such that that hManhattan(s’,g2) < x+1 then h(s’)=x or x+1 and so

.

* If there is another goal g2 such that hManhattan(s’,g2) < x-1 then hManhattan(s,g2)<x-1+1 = x and that’s a contradiction to the definition of h being the minimum so it’s not possible.

We proved that and we know that so we can conclude that

* If there is a portal to go from s to s’: because h is the minimum between 100 and the Manhattan distance from a goal in G. So and so.

This is true for all state s and all its successors s’ so the heuristic h is consistent.

**Question 8: Greedy**

Based on 8x8 grid:

1. Wet part – Implementation in Algorithms.py
2. The algorithm is complete but not optimal:

Complete because in the worst case it will go over all the states in the board and it’s finite so the algorithm will find a goal if there is one.

Not optimal because it does not consider the full path cost to the goal, but only the heuristic function value at each state. Therefore, it may get stuck in a local minimum or overshoot the goal state, leading to a suboptimal solution.

1. GBFS against UCS:

Advantage:GBFS can be more computationally efficient than UCS because it only considers the heuristic function to choose the next node to expand, which can greatly reduce the search space and the number of nodes expanded.

Disadvantage: GBFS is not guaranteed to find the optimal solution because it does not take into account the actual cost of the path from the start node to the current node.

1. GBFS against Beam Search

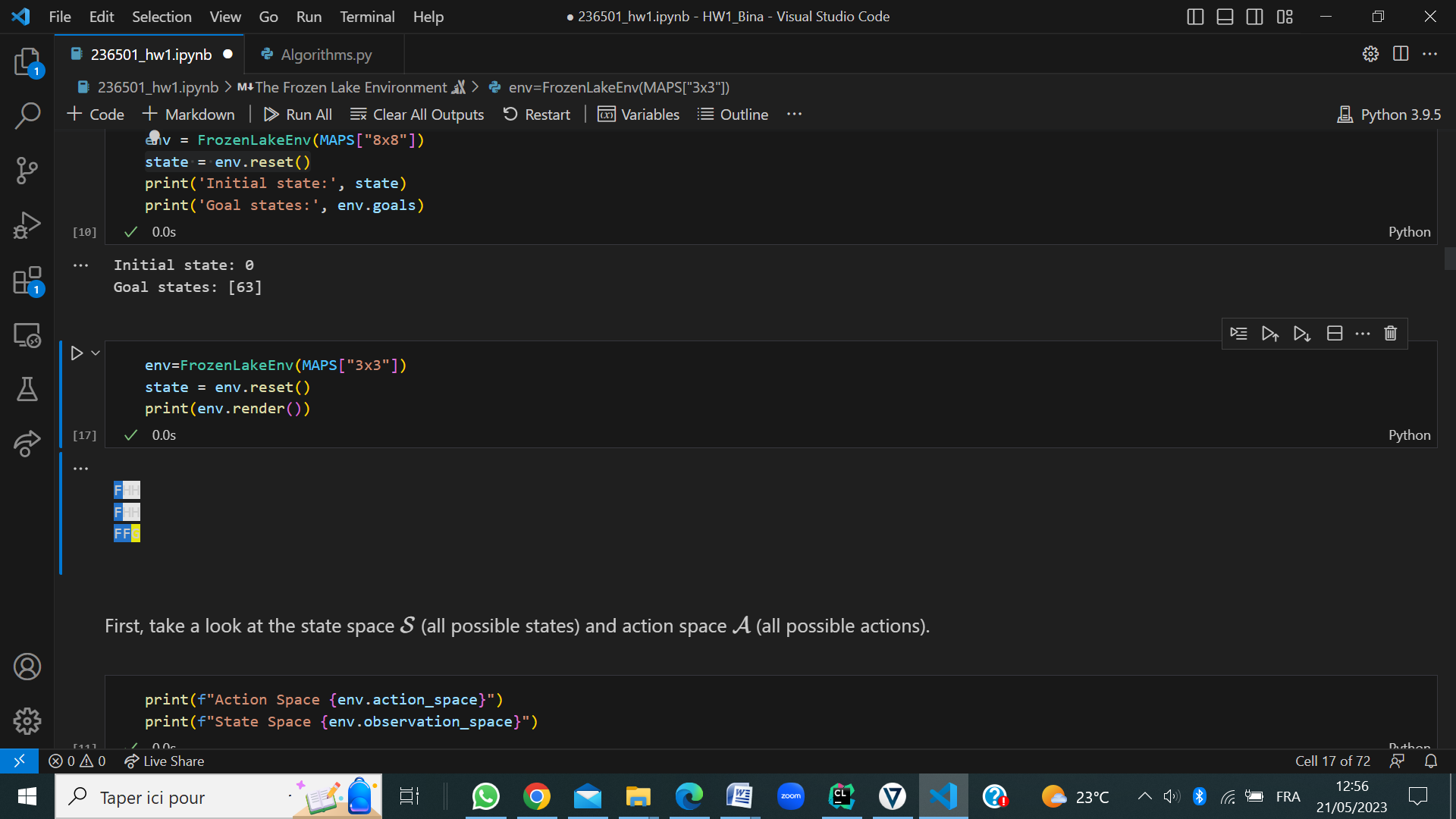
Advantage: GBFS is more memory-efficient than Beam Search, as it only expands the most promising node based on the heuristic function, while Beam Search expands k nodes at each level. This makes GBFS particularly useful in situations where memory is a limiting factor.

Disadvantage:GBFS may get stuck in local optima and fail to find the global optimum, especially when the heuristic function is not accurate or the search space is complex. In contrast, Beam Search explores a wider range of nodes, making it more likely to find the global optimum.

**Question 9 : WA\***

1. **Wet part** – Implementation in Algorithms.py

**a**. This is false: Let’s provide a counterexample:



Let’s define h such that h(0)=1, h(3)=1, h(6)=1, h(7)=1, h(G=8)=0, h(s)=0 for all other s.

h is an admissible heuristic because h(s)<=h\*(s) for all node s. Let’s choose w1=0.5 and w2=0.8

Both paths p1 and p2 from the initial state to the goal state, returned by W-A\* under the are the same 0 🡪 3 🡪 6 🡪 7 🡪 G and so cost(p1)=cost(p2)=2 and so it’s not true that .

**b**. This is not true: The counterexample for the question “a” is still ok here.

**Question 10 : IDA\***

1. **Wet part –** Implementation of IDA\* algorithm in the code.
2. In this question we will compare the algorithms IDA\* and A\*. We can first remember that the algorithm IDA\* is derived from the algorithm A\*.

One advantage of IDA\* compared to A\* it’s memory. Because IDA\* is doing a depth-limited search and is only storing a limited number of states in memory at each time. So with IDA\* we doesn’t need to use large amount of memory.

One disadvantage of IDA\* compared to A\* is that it can be slower. Because he has an iterative deepening approach, he can explore redundant states multiple times. And that’s why his runtime will be greater than the runtime of A\* that use a consistent heuristic function.

The final choice between IDA\* and A\* will depends of the specific problem, the search space, the runtime…. Like we explained below, we will prefer IDA\* when the memory to run the algorithm is limited because IDA\* unless A\* only stores a limited number of states at each time. So it can be better to use IDA\* when the problem has a large state spaces because I don’t have to store all the states in the memory. On the other hand we will prefer to use A\* when we don’t have constraint of memory or when the heuristic function that A\* use is consistent because when the heuristic function is consistent A\* guarantees finding the optimal solution because he will find the optimal solution faster than IDA\*

**Question 11 : A\* epsilon**

1. Advantage: A\*-epsilon provides the advantage of customizing the search behavior by adjusting the weight or epsilon parameter.

By increasing or decreasing the weight, A\*-epsilon can prioritize either the heuristic or the actual cost component of the evaluation function. It provides Flexibility.

Disadvantage:A\*-epsilon can potentially return suboptimal solutions, particularly when the weight parameter is set to a value greater than 1.

Increasing the weight gives higher priority to the heuristic estimate, which may result in paths that are not the shortest. So A\* has a higher level of optimality than A\*-epsilon.

1. We can choose the Manhattan Distance heuristic or Euclidean distance heuristic that we’ve both seen in class (just calculate the distance between the actual state and the goal according to the way of calculating)

Comparing using the heuristic h(v) against g(v) when choosing the next node from the FOCAL group (that allows choosing a node within an epsilon distance from the node with the min f(v))

* Nodes expandedchoosing the nodes according to the g(v) value tends to expand fewer nodes compared to selecting the minimum heuristic value within the FOCAL group because it prioritized nodes with lower actual costs from start node. But the difference seems very small because we’re choosing within epsilon nodes that have the minimum f(v) values and f and our heuristic is accurate.
* Returned path: g(v) ensuresto return an optimal path but it takes more time (because more accurate). However, the heuristic may find suboptimal paths very close to the optimal path since the node is always chosen within the nodes with f(v) close from the minimum and the heuristic we chose is accurate.
* Path cost: g(v) will return the optimal cost since the algorithm is designed to find the optimal path based on the given cost function. The heuristic choice may result in a suboptimal path with a higher cost but will still be very close to the optimal path cost.

Since the heuristic we chose is very accurate in a Frozen lake environment, it is likely that both ways will converge to the optimal path with a small difference for the heuristic that doesn’t take into account the portals and the actual cost of the edges.

**Question 12 : Benchmarking**

1. We ran the benchmarking in the notebook.
2. In terms of cost, we can see that the optimal paths were found by the UCS algorithm, and the WA\* algorithm. That is not surprising since they are heuristic algorithms based on the actual cost g(v) which means they have information on the actual cost. The UCS algorithm has the better relation ‘cost/expanded nodes’ which is what we expected since it is only guided by the actual cost of the edges/actions. We can see that the weight give the results for the cost of the path (except 0.9 that is slightly higher) but the number of expanded nodes are different: As the weight is bigger the number of expanded nodes is also bigger since we give more and more weight to the h(v) and less weight to g(v) so it is less efficient.

The greedy algorithm is also a heuristic algorithm but it has information only via the heuristic function we defined and not on the actual path cost so it doesn’t return the optimal path so we could maybe expect better with a better heuristic function. The greedy algorithm still has the lowest number of expanded nodes because the heuristic function is based on the Manhattan distance to the goals that doesn’t take into account the cost of the edges but instead the fastest way to get to the goal in terms of ‘number of actions’.

The BFS and DFS algorithm aren’t heuristic so they don’t find the optimal path but they have other qualities:

We can note that BFS doesn’t return the optimal path in terms of cost because BGS is only based on the number of edges/actions so if the objective is to get to the goal with the least number of actions no matter the cost of the path, BFS is the best. It expands a lot of nodes so we can assume it takes a long time to find the path.

The DFS algorithm doesn’t expand a lot of nodes because it tries to look for the goal in depth and that’s where the goal is: at the bottom right corner of the frozen lake. It is fast to find the goal but doesn’t return the optimal path neither in terms of number of actions nor in terms of cost of the path. It is fine if the objective is to find the goal fast no matter what the path is.

In resume, it depends what the objective of the search is. UCS is the best in terms of optimal cost. BFS is the best in terms of number of actions in the path returned. Greedy and WA\* algorithms depend on the heuristic function and allow to create balance between the cost of the path and on the heuristic (in our case the manhattan distance). The DFS will find the goal fast even if it is far without expanding too many nodes.

**Question 13: Local Search**

1. Given the state space of the question , where 𝑎 is the initial state and a utility function

𝑈: 𝑆 → ℝ+ where the 𝑈 value of each state is indicated in each state.

We will use the Stochastic Hill Climbing algorithm who in each state select a state to continue from the successors states that improve the U function with a probability proportional to the improvement.

The U value of the initial state is 0, so the state b improve the U of 3, the state c improve of 2 and the state d improve of 2.

1. The maximum number of steps the algorithm can do is 4 and it will be the following steps:

From the initial state a that has initially a U value of 0, the algorithm can and will choose the state c (with a U value of 2) to attempt the maximum number of steps so that improve it by 2.

From here he can and will choose the state b (that has a U value of 3) and so that improve it by 1.

From the state b, the algorithm can choose the state f (that has a U value of 4) and so that improve it by 1.

After that, from the state f, the algorithm can choose the state g that has a U value greater than 4 and so will improve it by a value greater than zero.

So the paths relative to the maximal number of states is: a 🡪 c 🡪 b 🡪 f 🡪 g.

1. Given that the first transition is from initial state a to state b.

The algorithm will of course reach an optimal (global) solutionthat is the state that improve his U value the more than possible that is the state g. Because from the state b, the algorithm has two possibilities to advance in the search space. He can choose the state f or the state g. If he choosed the state g we finish. And if he choosed the state f, from here he has just one possibility that is to advance to the goal state so he will choose this possibility. He can’t choose e because the U value of b is 3 and the U value of e is 3 so go to the state e does not improve the U value. That’s why the algorithm, will not choose it. And he reached the optimal solution in a,l the cases.



1. Like we saw below, if the algorithm reach the state b it’s sure that he will reach the optimal state.

Otherwise, the algorithm has two possibilities:

* He can reach the state d and from here I will not can advance more.
* Or he can also reach the state c. From the state c, he has also two possibilities: reach state b and from the state b he will reach the goal state like we explained below. Or, he can reach the state h (that will improve the U value of the state c). And, from the state h, the algorithm can’t advance more. The only transition that exists from the state h is to the state c. But the U value of c is not greater than the U value of h so the algorithm will not advance from h and will stay stuck in the state h.

That’s why:

1. The range of values of β such that the probability of reaching an optimal (global) solution in exactly 3 steps is bigger than 1/5 is:

We have two possibilities of path from the initial state a to the goal state g. They are the paths:

* a 🡪 c 🡪 b 🡪 g (1)
* a 🡪 b 🡪 f 🡪 g (2)

We have to calculate the probabilities of those two paths, add them and verify that the result is bigger than 1/5.

So to come back to the question, we want:

So for the range the probability of reaching an optimal (global) solution in exactly 3 steps is bigger than 1/5.