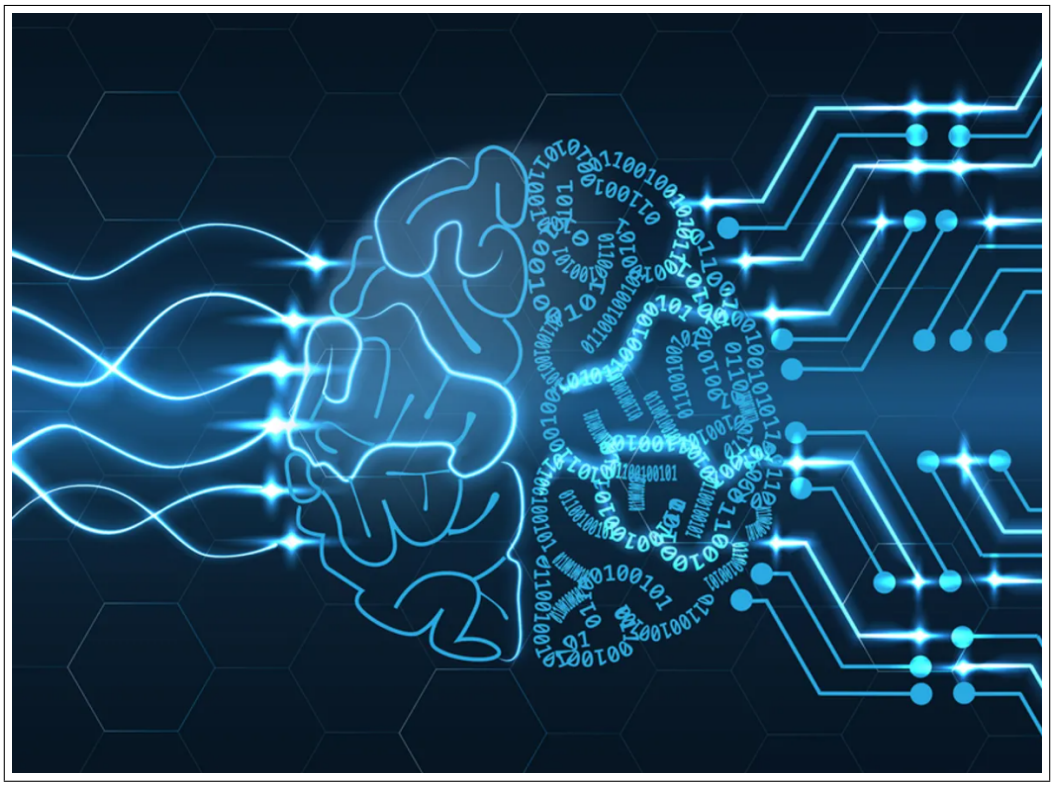
Introduction to Artificial Intelligence - 236501

Homework #1: Search Spaces



Name	ID	Email
Lior Karne	211813878	lior-yehudak@campus.technion.ac.il
Yotam Granov	211870241	yotam.g@campus.technion.ac.il

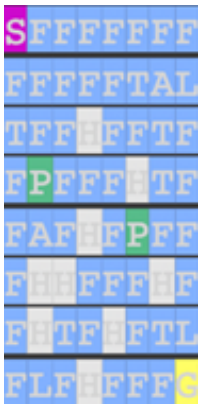
Faculty of Computer Science
Technion - Israel Institute of Technology

Winter 2022/23

Theoretical Questions

Theoretical Question 1 - Introduction

The questions in this section are based on the 8×8 board that appears in the notebook unless written otherwise:



Part 1.1

First we would like to define the search space as we learned in the tutorial. Define (S, O, I, G) for the frozen lake environment. What is the size of the state space S? Explain.

Solution:

In this case we will let $S = \{0\} \cup [63]$, where each state in the state space is defined by an index (in a linear manner) between 0 and 63, and therefore the state space is of size **64**. The initial state is always in slot 0 and the final state is in slot 63, and so we define $I = 0$, and $G = \{63\}$.

Finally, we can define $O = \{DOWN = 0,\ RIGHT = 1,\ UP = 2,\ LEFT = 3\}$. We will denote the portal slots on the board as $p_1(= 25),\ p_2(= 37)$ and define a function $P : S \to S$ in the following way:

$$P(s) = \begin{cases} p_2 & s = p_1 \\ p_1 & s = p_2 \\ s & \text{otherwise} \end{cases}$$

Thus, we can define the operators as follows:

$$DOWN(s) = \begin{cases} P(s+8) & s < 56 \\ \emptyset & s \text{ is a hole} \\ P(s) & \text{otherwise} \end{cases}$$

$$RIGHT(s) = \begin{cases} P(s+1) & s \ mod\ 8 \neq 7 \\ \emptyset & s \text{ is a hole} \\ P(s) & \text{otherwise} \end{cases}$$

$$UP(s) = \begin{cases} P(s-8) & s > 7 \\ \emptyset & s \text{ is a hole} \\ P(s) & \text{otherwise} \end{cases}$$

$$LEFT(s) = \begin{cases} P(s-1) & s \ mod\ 8 \neq 0 \\ \emptyset & s \text{ is a hole} \\ P(s) & \text{otherwise} \end{cases}$$

Part 1.2

What will the *Domain* function return to us on operator (UP) 2?

Solution:

The *Domain* function on operator (UP) 2 will return:

$$Domain(2) = S \setminus \{19, 29, 35, 41, 42, 46, 49, 52, 59, 63\}$$

The operator can be applied to all states except for holes (on which nothing can be applied) and the final state (if we have reached the end of the path).

Part 1.3

What will the *Succ* function return to us on the initial state 0?

Solution:

The *Succ* function on the initial state 0 will return:

$$Succ(0) = \{0, 1, 8\}$$

We are at the top left corner, so if we try to move left or up we will just stay in the same place (state 0), whereas we can indeed move right (to state 1) or down (to state 8).

Part 1.4

Are there cycles in our search space?

Solution:

There indeed may be circles in the search space. If we move from a certain state to the right (assuming we are not moving to a portal or a hole) and then to the left, we will reach the same state again.

Part 1.5

What is the branching factor of this problem?

Solution:

The branching factor of this problem is 4, since there are at most 4 options for movement in a given state: right, left, down, or up.

Part 1.6

In the worst case, how many operations will it take for the random agent (as implemented in the notebook) to reach the final state?

Solution:

In the worst case, the random agent may never reach the final state (i.e. it will conduct infinite actions).

Part 1.7

In the best case, how many operations would it take for the random agent (as implemented in the notebook) to reach the final state?

Solution:

In the best case, it is possible that the random agent will reach the final state after 9 actions, since it takes 4 steps to get from the initial state to the portal's entrance and another 5 steps to get from the portal's exit to the final state (and this is the optimal path in terms of length).

Part 1.8

For a general board, when the initial and final states are at the edges of the board (similar to an 8×8 board), will an agent searching for the shortest path always decide to go through the portal?

Solution:

No, as sometimes the cost of moving through the portal is greater than the cost of moving through several regular slots instead.

Theoretical Question 2 - BFS-G

The questions in this section are based on the 8×8 board that appears in the notebook unless written otherwise.

Part 2.1

What should be the condition on the search graph (not necessarily in the frozen lake problem) so that BFS on a graph and BFS on a tree produce and expand identical nodes in the same order?

-0.5 no cycles is not enough

Solution:

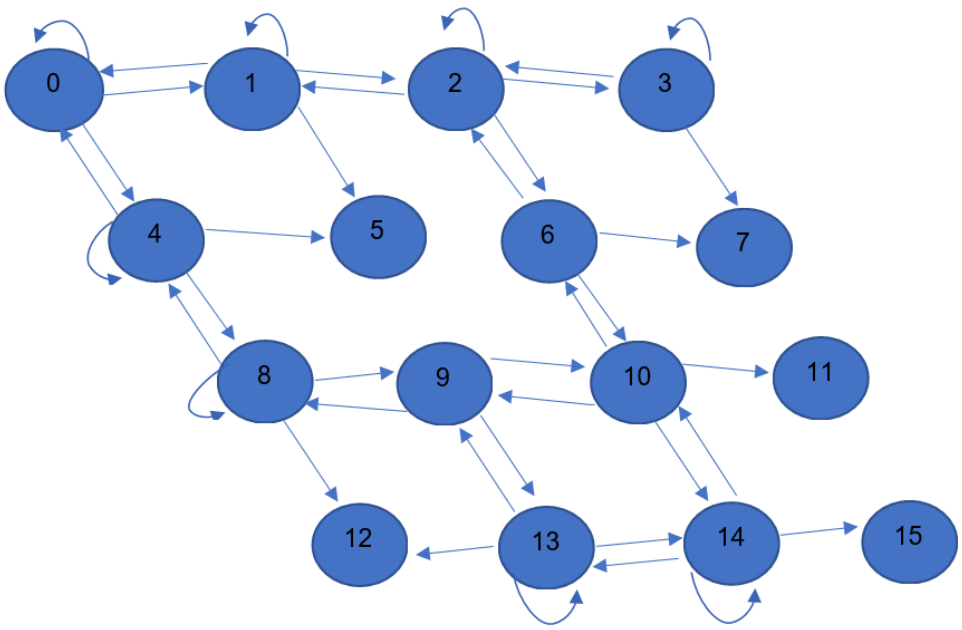
The condition on the search graph such that BFS on a graph and BFS on a tree produce and expand identical nodes in the same order is that there must be **no cycles** in the search graph. If there are cycles then there will be a difference between the two cases, as a search in the tree will expand the nodes in the cycles over and over again whereas a search in the graph will not.

Part 2.2

For the 4×4 board that appears in the notebook, draw the state graph.

Solution:

The state graph for the 4×4 board looks as follows:



Part 2.3

You're given a board of size $N \times N$ that does not contain portals. Suggest a way to use the BFS-G algorithm so that it returns an optimal solution (minimum cost) and explain.

Hint: You must provide a function $T : G \to G'$ that receives the state graph G and creates a new graph G', and with its help find the optimal path in the graph G.

Solution:

We'll define the function $T[G(V, E)] = G'(V', E')$ by first defining the cost of stepping on slot (state) s as a function $cost(s)$. Thus, we can let:

$$V' = \{s_i \mid s \in S, 1 \leq i \leq cost(s)\}$$
$$E' = \{(s_{cost(s)}, s'_1) \mid s, s' \in S,\ (s, s') \in E\}$$

If we run BFS on G' we will get a solution of optimal length, since for each state s whose slot we enter, we will have to go through a number of nodes equal to the cost of that slot, and therefore the number of nodes we will go through will be equal to the cost of the route in the original graph. Thus, the less nodes we go through, the lower the cost will be, and so the optimality of BFS (which guarantees the solution with the minimal number of nodes) will allow us to find the solution with the smallest total cost.

Part 2.4

You're given a board of size $N \times N$, without holes and without portals, containing $N - 2$ normal slots (F, T, A, L) and two slots for a final state and an initial state. How many nodes will be expanded and created during a BFS-G search? Explain.

Solution:

-1 more nodes will be created

We will expand $N^2 - 2$ nodes and create N^2 nodes during a BFS-G search on this problem. The reason for this is that the search is performed laterally: goal node G is the farthest node (distance $N-2$) from start node S, therefore every node will be expanded for every slot that is at a (Manhattan) distance of $0 \leq i < N-3$ from S (that is, every slot except the goal slot, the one above it, and the one to the left, which gives a total of $N^2 - 3$ squares) before the algorithm advances to a greater distance. This means that all the squares with a distance of up to $0 \leq i \leq N - 3$ will be created (the total number of such squares is $N^2 - 1$).

Finally, we will expand the first node that is $N - 3$ away and in it we will create and find the goal node, and therefore a total of $N^2 - 2$ nodes will have been developed and N^2 nodes will have been created.

Note: In this graph there are no holes, therefore any state that is not of the target slot can be developed.

Theoretical Question 3 - DFS-G

Part 3.1

For the frozen lake problem with an $N \times N$ board, is the algorithm complete? Is it optimal?

Solution:

Given that the search space is finite, the DFS algorithm is guaranteed to find a solution, however it is not guaranteed to find the optimal one. Thus, the DFS algorithm is **complete but non-optimal**.

Part 3.2

Would the DFS algorithm (on a tree), for the frozen lake problem on an $N \times N$ board, find any solution? If so, what path will be taken? If not, how would the algorithm work?

Solution:

As we mentioned during the tutorial, the DFS algorithm acting on a search tree is **incomplete** since cycles can exist in our tree and cause infinite loops, and unlike with DFS-G algorithm for graphs we do not keep track of previously visited nodes and thus we don't conduct any cycle-checking during our tree search to prevent such infinite loops.

Part 3.3

You're given a board of size $N \times N$, without holes and without portals, containing $N^2 - 2$ normal slots (F, T, A, L) and two slots for a final state and an initial state. How many nodes will be expanded and created during a DFS-G search? Explain.

Solution:

-1 more nodes will be created

Since we add nodes to the stack in *ascending* index order (i.e. we add the nodes with the smallest state index first), then we will always pop the node with the largest state index from the top of the stack in order to expand it. By doing so, we will start by creating and expanding node 0, and then we will create the nodes to the right and bottom of 0 (i.e. 1 and N respectively). Node N will have been added to the stack last, so it will be first to be expanded, and thus we will create the nodes to the right $(N + 1)$ and bottom $(2N)$ of it. Once again, we choose to expand the node with the larger state index $(2N)$, and we continue this trend until we have expanded all nodes along the left boundary of the board (this amount to N nodes) and created $2N$ nodes total (we created all nodes on the left boundary as well as their siblings, which make up the entire second column from the left).

Once we've reached the bottom left node $(N \cdot (N-1))$, we will expand every node along the bottom boundary until we reach the goal node at the bottom right coroner - we will also create the sibling nodes of all the nodes we expand along the bottom boundary (i.e. the middle $N - 2$ nodes of row $N - 1$). We create the goal node but we do not expand it (whereas for the start node we both created and expanded it). Thus, in total our DFS-G search we will have created $2N + 2(N - 2) + 1 = \mathbf{4N - 5}$ nodes and expanded $N + N - 2 = \mathbf{2N - 2}$ nodes.

Part 3.4

-1 more nodes will be created

You're given a board of size $N \times N$, without holes and without portals, containing $N^2 - 2$ normal slots (F, T, A, L) and two slots for a final state and an initial state. How many nodes will be expanded and created during a Backtracking-G search? Explain.

Solution:

Since we add nodes to the stack in *ascending* index order (i.e. we add the nodes with the smallest state index first), then we will always pop the node with the largest state index from the top of the stack in order to expand it. By doing so, we will start by creating and expanding node 0, and then we will only create the node to the bottom of 0 (i.e. N), since the Backtracking-G algorithm only creates nodes that are about to be expanded. We then expand node N and only create the node below it $(2N)$, and we will expand that one again and so on until we reach the bottom boundary. Until that point, we'll have created *and* expanded N nodes.

Once we reached the bottom left corner (i.e. node $N \cdot (N - 1)$), we'll create the node directly the right of it and expand that one, and we'll continue with this process until we reach the goal node at the bottom right corner of the board. Along the bottom boundary, we'll have created $N - 1$ nodes (including the goal node) but expanded only $N - 2$ nodes (since we don't count the goal node), and thus in total our Backtracking-G search will have created $N + N - 1 = \mathbf{2N - 1}$ nodes and expanded $N + N - 2 = \mathbf{2N - 2}$ nodes.

Theoretical Question 4 - ID-DFS-G

The questions in this section are based on the 8×8 board that appears in the notebook unless written otherwise.

Part 4.1

Is the ID-DFS-G algorithm complete?

Solution:

The ID-DFS-G algorithm is indeed **complete**. For every solution of finite length there is an iteration that will reach all the states of that length, therefore if there is a solution it will be found.

Part 4.2

Supposing that the cost of every action is 1, the ID-DFS-G algorithm is then non-optimal. Explain why. In addition, suggest a way to update the algorithm in order to be optimal for this case.

Solution:

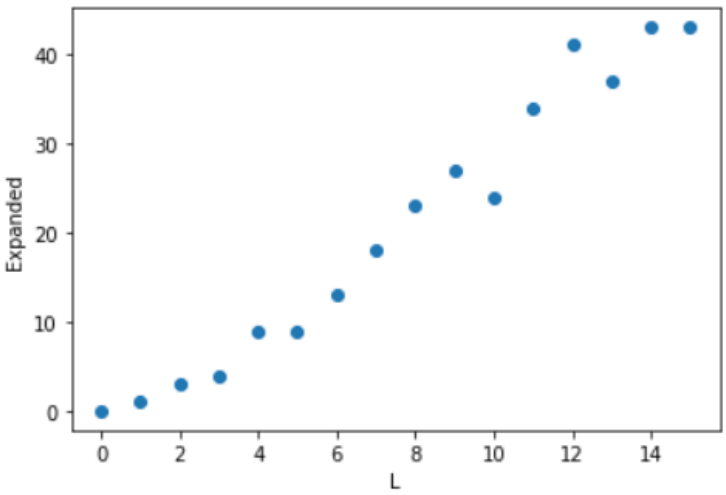
Let's assume that the optimal solution is of a certain length L. It is possible that in the DFS-L run, in the iteration where we will try to find the solution of length L, we will pass through one of the nodes that are on the route that provides the optimal solution and withdraw from it (that is, we have not yet passed through the optimal route, but have reached the node as part of another route). The next times we reach this node we will not be able to expand it because it has been entered into the closed list. In such a situation, we will not be able to find the optimal solution (because we will not be able to go through it further in the current iteration) and therefore the optimality of the algorithm is not guaranteed.

In order to avoid such a situation, we will run DFS-L on a tree instead of a graph in each iteration (that is, not DFS-L-G), and then we can expand the same node several times and go through all possible routes such that optimality will be guaranteed.

Part 4.3

Present a graph showing the effect of L (at least 5 different values) on the number of nodes that are expanded at each depth. Briefly explain the graph.

Solution:



It can be seen that there is an increase in the total number of nodes expanded as the depth of the search increases, since when we increase the depth of the search there are new possibilities for us to

reach new routes. However, there is no monotonous growth in relation to each point and the one that follows it, because, as we mentioned in the previous section, there may be certain situations where the expansion of a certain node just once will result in the prevention of the possibility of reaching new nodes and a consistent increase in the number of nodes expanded at each depth.

Theoretical Question 5 - UCS

The questions in this section are based on the 8×8 board that appears in the notebook unless written otherwise.

Part 5.1

For which search problems will the UCS algorithm and the BFS algorithm perform in the same way? Explain.

-0.5 it doesnt have to be 1, it can be any constant number

Solution:

The UCS and BFS algorithms are equivalent if we set the cost of all edges to be 1, i.e. the lowest cost path (in the case of UCS) is the one with the least number of edges. In such a case, the priority queue for the UCS algorithm will develop identically to the FIFO queue of the BFS algorithm, and so the nodes of the search tree will be expanded in an identical order for each algorithm.

Part 5.2

In our search problem, for an $N \times N$ board, is the algorithm complete? Is it optimal?

Solution:

Given that our search space is finite and that the cumulative cost of the optimal path is finite, then the UCS algorithm will always return the optimal solution, and thus is **both complete and optimal**.

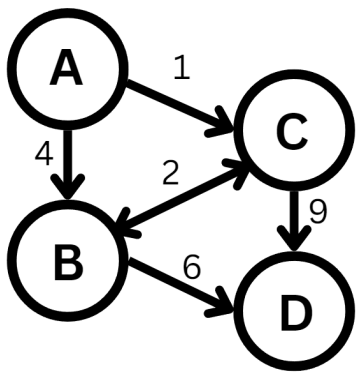
Part 5.3

Dan made a mistake in the implementation of the UCS algorithm and mistakenly checked when creating a node whether it was a target node instead of when the node was expanded. Give an example of a search graph for which Dan will still return the easiest route and an example of a search graph for which Dan will not return the easiest route. For each example explain what route and cost the incorrect UCS returned, and what route and cost the correct algorithm would have returned.

We emphasize that the search graph does not necessarily have to represent the frozen lake problem. You can give an example of a graph that represents another search problem. The graph should contain directed arcs and the cost of each arc.

Solution:

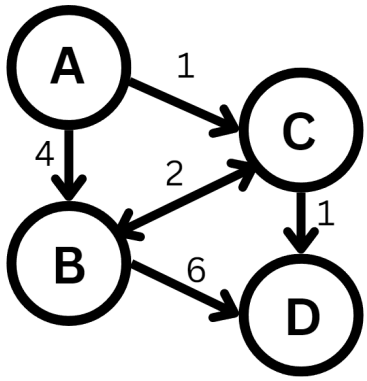
Let's take a look at a graph on which Dan's implementation of UCS would surely fail:



Here node A is the initial state and D is the goal state. In this case, the correct shortest route would be $[A, C, B, D]$ with a total cost of 9, however Dan's algorithm would return $[A, C, D]$ with a total cost of 10 instead. This is because Dan's algorithm would begin by expanding $[A]$ and thus creating $[A, C]$ and $[A, B]$, with costs 1 and 4 respectively (neither of which are the goal state). Since $[A, C]$ has a lower cost than $[A, B]$, Dan's algorithm would then expand $[A, C]$ and create $[A, C, B]$ and $[A, C, D]$, with costs 3 and 10 respectively. Dan's algorithm would see that $[A, C, D]$ is a solution upon creating the node, and thus would immediately return it as the final result (even though it is not optimal).

The correct implementation of UCS would've just added $[A, C, D]$ to the open list without immediately declaring it to be the solution. It would've deleted $[A, B]$ from the open list (since it found a shorter path to node B, as $cost([A, C, B]) = 3 < cost([A, B]) = 4$) and expanded $[A, C, B]$ next since it is now first on the priority queue - this would create $[A, C, B, C]$ (which would immediately be rejected from the open list since its cost of 5 is greater than the current shortest cost from A to C, which is 1) and $[A, C, B, D]$ with cost 9. The open list at this stage would only contain $[A, C, B, D]$ (with cost 9) and $[A, C, D]$ (with cost 10), and so upon expanding $[A, C, B, D]$ first (since it has a lower cost) the UCS algorithm would recognize that it's trying to expand a goal node, and thus it would choose to return $[A, C, B, D]$ as the (optimal) solution.

Dan's implementation could still return an optimal solution for some graphs though, for example we could considering the following variant of the graph we were just looking at:



In this case, the correct shortest route would be $[A, C, D]$ with a total cost of 2. Dan's algorithm would begin by expanding $[A]$ and thus creating $[A, C]$ and $[A, B]$, with costs 1 and 4 respectively (neither of which are the goal state). Since $[A, C]$ has a lower cost than $[A, B]$, Dan's algorithm would then expand $[A, C]$ and create $[A, C, B]$ and $[A, C, D]$, with costs 3 and 2 respectively. Dan's algorithm would see that $[A, C, D]$ is a solution upon creating the node, and thus would immediately (and correctly) return it as the final result. Likewise, the proper implementation of UCS would recognize that $[A, C, D]$ is now first on the priority queue (it has the smallest cost of all nodes on the open list), and return it as well. Thus, this is a case where Dan's mistake doesn't prevent his algorithm from returning the correct solution.

Theoretical Question 6 - Heuristics

Let's define define a new heuristic:

$$h_{SAP}(s) = min\{h_{Manhattan}(s, g),\ Cost(p)\},\ g \in G$$

where the first expression is the Manhattan distance from the current state to the final state and the second expression is the cost of an arc resulting in a "launch" slot.

Part 6.1

Is the heuristic admissible on all boards? If yes explain, if not give a counterexample.

Solution:

The heuristic is indeed **admissible**. For every state s, we see that:

- If we can't reach a goal state g from s, then $h^*(s) = \infty$ and thus it's clear that $h(s) < h^*(s)$. Since $Cost(p) = 100 > 0$ and the Manhattan distance between two points is always non-negative ($h_{Manhattan}(s, g) \geq 0$), it must be that $h_{SAP}(s) \geq 0$, and so $0 \leq h(s) \leq h^*(s)$ must indeed hold.
- If g is indeed reachable from s, then there are two cases of interest for us to consider:
  1. If the lowest cost route from s to g contains no portals, it will be built from squares whose weight is at least 1. A path on the board is essentially in the form of a grid path, so it's cost will necessarily be at least the length of its Manhattan distance (i.e. the cost from s to g will always be at least the Manhattan distance between them), and therefore it will hold that $h^*(s) \geq h_{Manhattan}(s, g) \geq h_{SAP}(s) \geq 0$ (i.e. the Manhattan distance underestimates the cost to goal and is at least equal to h_{SAP}, which is greater than zero by definition since the Manhattan distance is non-negative).
  2. If the lowest cost route from s to g contains a portal, then only the step through the portal will have weight $Cost(p)$, and the rest of the weights in the route will be positive. Suppose that the sum of the weights of the rest of the route is $C > 0$, then it's clear to see that $h^*(s) = C + Cost(p) > Cost(p) \geq h_{SAP}(s) \geq 0$ (the last inequality is due to the fact that the Manhattan distance between two points is always non-negative and $Cost(p) = 100 > 0$).

Thus, we see that $h^*(s) \geq h_{SAP}(s) \geq 0$ is always true, and therefore the heuristic is admissible.

Part 6.2

Is the heuristic consistent on all boards? If yes explain, if not give a counterexample.

Solution:

No, this heuristic is **not always consistent**. For example, let's consider the following 3×3 board:

S	H	F
F	F	F
F	F	G

In this case, the cost of moving from state 0 to 1 is zero, i.e. $cost(0, 1) = 0$ (since state 1 is a hole), but $h_{Manhattan}(0, g) = 4$ and $h_{Manhattan}(1, g) = 3$, and since both of these are smaller than $Cost(p) = 100$ we see that $h_{SAP}(0) = 4$ and $h_{SAP}(1) = 3$. In this case, it's clear that:

$$h_{SAP}(0) - h_{SAP}(1) = 4 - 3 = 1 > 0 = cost(0, 1)$$

and this is a contradiction to the definition of consistency for a heuristic.

Theoretical Question 7 - Greedy Best First Search

The questions in this section are based on the 8×8 board that appears in the notebook unless written otherwise.

Part 7.1

Is the algorithm complete? Is it optimal?

Solution:

Since our search space is finite, the Greedy Best First Search algorithm is guaranteed to obtain a solution but it can't guarantee that the solution it obtained is optimal. Thus, it is **complete but not optimal**.

Part 7.2

Give an advantage and disadvantage of the Greedy Best First Search algorithm compared to UCS.

Solution:

The Greedy Best First Search algorithm is clearly disadvantageous when it comes to guaranteeing optimality - for a finite search space, UCS is optimal whereas Greedy Best First Search is not. Despite this, the Greedy Best First Search algorithm is advantageous in that its informed nature will often allow it to reach a solution much faster than UCS can (though this is not always the case).

Theoretical Question 8 - W-A*

The questions in this section are based on the 8×8 board that appears in the notebook unless written otherwise.

Part 8.1

Here are a number of heuristics, prove or disprove (with the help of a counterexample) the admissibility of each heuristic:

(a) $GreedyHeuristic(s) = 0$ if s is goal, else 1
(b) $h_{MD}(s) = h_{Manhattan}(s, g),\ g \in G$
(c) $NearestPortalOrGoalHeuristic(s) = min\{h_{Manhattan}(s, p_1), h_{Manhattan}(s, p_2), h_{Manhattan}(s, g)\}$, $g \in G$, where p_1, p_2 are launch slots ($p_1 \neq p_2$) and g is a goal node
(d) $NearestPortalToFinalHeuristic(s) = NearestPortalOrGoalHeuristic(s) + h_{Manhattan}(p_1, p_2)$

Solution:

(a) This heuristic is **admissible**. If s is the goal state then $h^*(s) = h(s) = 0$, otherwise there will inevitably be a path to the goal state (all non-goal states in this path will fulfill $h(s) = 1 \leq h^*(s)$) and the last step in it (to slot g) is weighted 1, and thus $h^*(s) \geq h(s)$. Therefore, $h^*(s) \geq h(s) \geq 0$ for all s.

(b) This heuristic is **admissible**. For each non-g slot, if there is a portal slot on the route to g, then the cost of the route is at least 100 and this is automatically greater than the Manhattan distance of all slots from the goal slot (since the board is 8×8, the maximal Manhattan distance from any s to g is 14). If there are no portals on the route from s to g, then the route passes through a number of slots equal to the length of the Manhattan route, and at least one slot has a cost greater than 1 (no slot has a route in which all the slots are L) and therefore the cost of the route will be greater than the Manhattan distance, i.e. $h(s) < h^*(s)$. The Manhattan distance is always non-negative, and so $0 \leq h(s) \leq h^*(s)$. For the goal node g, the Manhattan distance is 0 and $h^*(g) = 0$, and so $h^*(g) = h(g) = 0$. Therefore, by definition this heuristic is admissible.

(c) This heuristic is **admissible**. Since

$$\begin{aligned} NearestPortalOrGoalHeuristic(s) &= min\{h_{Manhattan}(s, p_1), h_{Manhattan}(s, p_2), h_{Manhattan}(s, g)\} \\ &\leq h_{Manhattan}(s, g) \end{aligned}$$

and we showed in the previous section that $h_{Manhattan}(s, g) \leq h^*(s)$ (and that the Manhattan distance between any two points is always non-negative), then it must hold that:

$$0 \leq NearestPortalOrGoalHeuristic(s) \leq h_{Manhattan}(s, g) \leq h^*(s)$$

For g, we see that $min\{h_{Manhattan}(g,p_1), h_{Manhattan}(g,p_2), h_{Manhattan}(g,g)\} = h_{Manhattan}(g,g) = 0$, and so $h^*(g) = NearestPortalOrGoalHeuristic(g) = 0$. Therefore, by definition this heuristic is admissible.

(d) This heuristic is **not admissible**. For the goal state g it holds that $h^*(g) = 0$, but according to the heuristic definition we get that:

$$NearestPortalToFinalHeuristic(g) = \cancelto{0}{h_{Manhattan}(g,g)} + h_{Manhattan}(p_1,p_2) = 5 \neq 0$$

and therefore $NearestPortalToFinalHeuristic(g) > h^*(g) = 0$ and the heuristic is automatically inadmissible.

Part 8.2

Which heuristic (among the admissible ones) is the most informed (including the h_{SAP} heuristic)?

Solution:

-0.5 also h_sap on 8x8 board

The most informed heuristic is h_{MD}. First, it's clear to see that:

$$0 \leq h_{SAP}(s) = min\{h_{Manhattan}(s,g),\ Cost(p)\} \leq h_{Manhattan}(s,g) \leq h^*(s)$$

Next, we can reiterate the fact that:

$$0 \leq NearestPortalOrGoalHeuristic(s) \leq h_{Manhattan}(s,g) \leq h^*(s)$$

which we saw in the last section. We also know that the Manhattan distance is greater than or equal to 1 for any state s that is not a goal state, and so:

$$0 \leq GreedyHeuristic(s) = 1 \leq h_{Manhattan}(s,g) \leq h^*(s)$$

for all $s \neq g$. For the goal, we see that $GreedyHeuristic(g) = h_{Manhattan}(g,g) = h^*(g) \equiv 0$. Thus, we conclude that h_{MD} is the most informed one of the set of admissible heuristics we're considering, since it is the closest in value to the perfect heuristic $h^*(s)$ for all states s.

Part 8.3

Run W-A* with different W values and display two graphs:

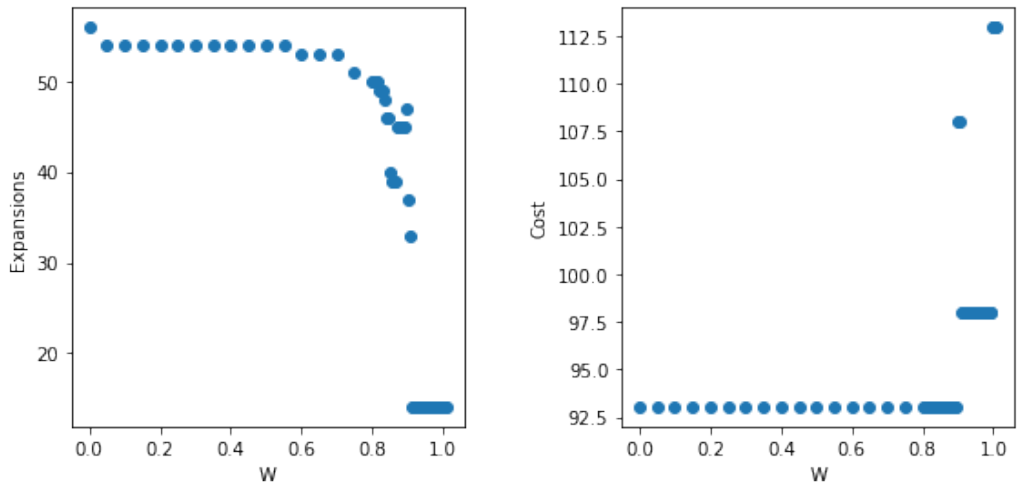
(a) Graph 1: The number of expansions as a function of W

(b) Graph 2: The cost of the solution found as a function of W

The graphs can be drawn by hand or on the computer. Explain each graph separately as well as the relationship between them.

Solution:

The two requested graphs can be seen in the following figure:



We can see in the graphs that increasing the value of w will cause our search to become more greedy, expanding less nodes (and thus running faster) while paying the price in terms of a greater cost incurred by the solution. On the other hand, we see that decreasing the value of w will cause our search to become more uniform (like UCS), finding solutions with lower costs while paying the price in terms of longer runtimes. In this way, the two graphs illustrate the relationship between the quality of the solution and the "price we will have to pay" at runtime to find it.

Theoretical Question 9 - A*-epsilon

Part 9.1

Give an advantage and disadvantage of A*-epsilon as compared to A*.

Solution:

A*-epsilon is disadvantageous when it comes to guaranteeing optimality - for a finite search space, A* is optimal whereas A*-epsilon is not (it is only bounded-suboptimal). Despite this, the A*-epsilon algorithm is advantageous in that its approximate nature will often allow it to reach a solution much faster than A* can.

Part 9.2

Suggest a heuristic to select the next node to expand from FOCAL. Describe the heuristic and present a comparison between the use of this heuristic versus the use of $g(v)$, in terms of the number of expansions, the selected path, and the cost of the selected path.

Note: In the code you submit, the agent must use $g(v)$ and not the heuristic you defined in this section.

Solution:

Let's investigate the use of $2 \cdot g(v) + h_{SAP}(v)$ as our heuristic for selecting the next node to expand from FOCAL, rather than $g(v)$ - this will allow the influence of information about the goal's location to contribute more to our choice of node expansions, while giving more weight still to the cumulative cost of the nodes in the path. When we use just $g(v)$ with $\varepsilon = 0.5$, we get the following:

- Total Cost: 93
- Expanded: 56
- Actions: [1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]

When we use $2 \cdot g(v) + h(v)$ with $\varepsilon = 0.5$, we get the following:

- Total Cost: 93
- Expanded: 54
- Actions: [1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]

We can see that our heuristic returned the same solution with the same cost, but did so by expanding 2 less nodes.

Exam-Style Questions

Exam Question 1 - Variations on A*

Part 1.1

Let n' be the parent node of node n in the graph. We will assume that h is an admissible heuristic that is not the zero heuristic and that there exists in the space a goal state that is reachable from the initial state. For each of the algorithms, explain whether it is complete and whether it is optimal:

(a) A* as we learned in the lecture
(b) A* which ignores the h value
(c) A* which ignores the g value
(d) A* such that $f(n) = g(n) + h(n')$
(e) A* such that $f(n) = g(n') + h(n)$

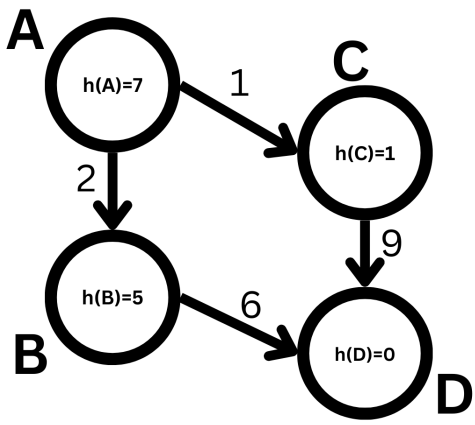
Solution:

(a) A* is known to be **both complete and optimal** given that h is an admissible heuristic and that the cost of every edge in the graph is non-negative.

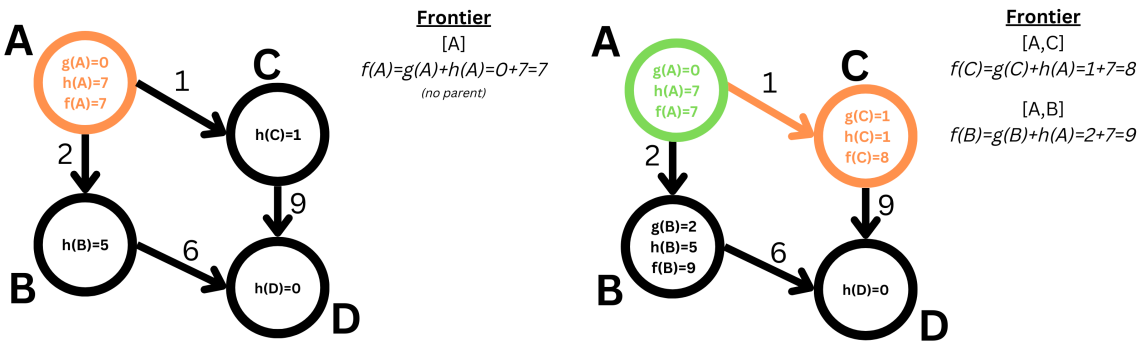
(b) A* which ignores the h value essentially relies only on $f = g + 0 = g$ (the cumulative cost function) to determine the node order in the fringe, and this reduces down to being the same algorithm as UCS. As we discussed earlier, UCS is **both complete and optimal** for a finite search space, and thus the same applies for this variant of A*.

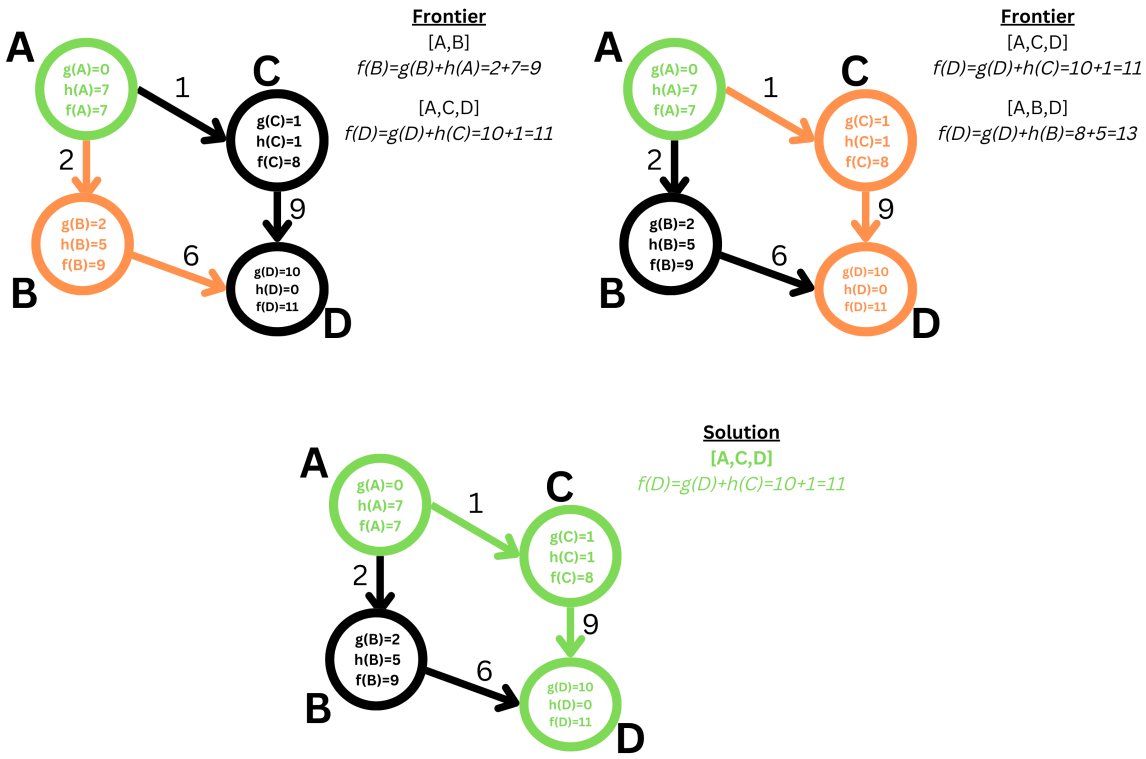
(c) A* which ignores the g value essentially relies only on $f = 0 + h = h$ (the admissible heuristic) to determine the node order in the fringe, and this reduces down to being the same algorithm as Greedy Best First Search. As we discussed earlier, Greedy Best First Search is **complete but not optimal** for a finite search space, and thus the same applies for this variant of A*.

(d) A* such that $f(n) = g(n) + h(n')$ is still complete, since the cost is lower bounded by zero, but it is no longer optimal. While $h(n)$ is still admissible, our "new" heuristic $h(n')$ isn't exactly inadmissible, since it can overestimate the distance to the goal (for example, $h(g') > 0 = h^*(g)$ when the parent nodes of the goal node have positive heuristic evaluations). We can see this using the following example:



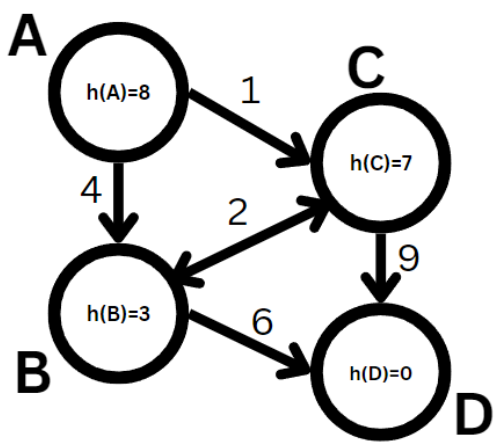
Here we have a search graph, which we would like to traverse from initial node A to goal node D with the help of an admissible heuristic h. If we use the function $f(n) = g(n) + h(n')$ to order our frontier for node expansion, then a run of A* will look as follows (nodes that are about to be expanded are denoted in orange, expanded nodes that will be placed in the closed list are green):



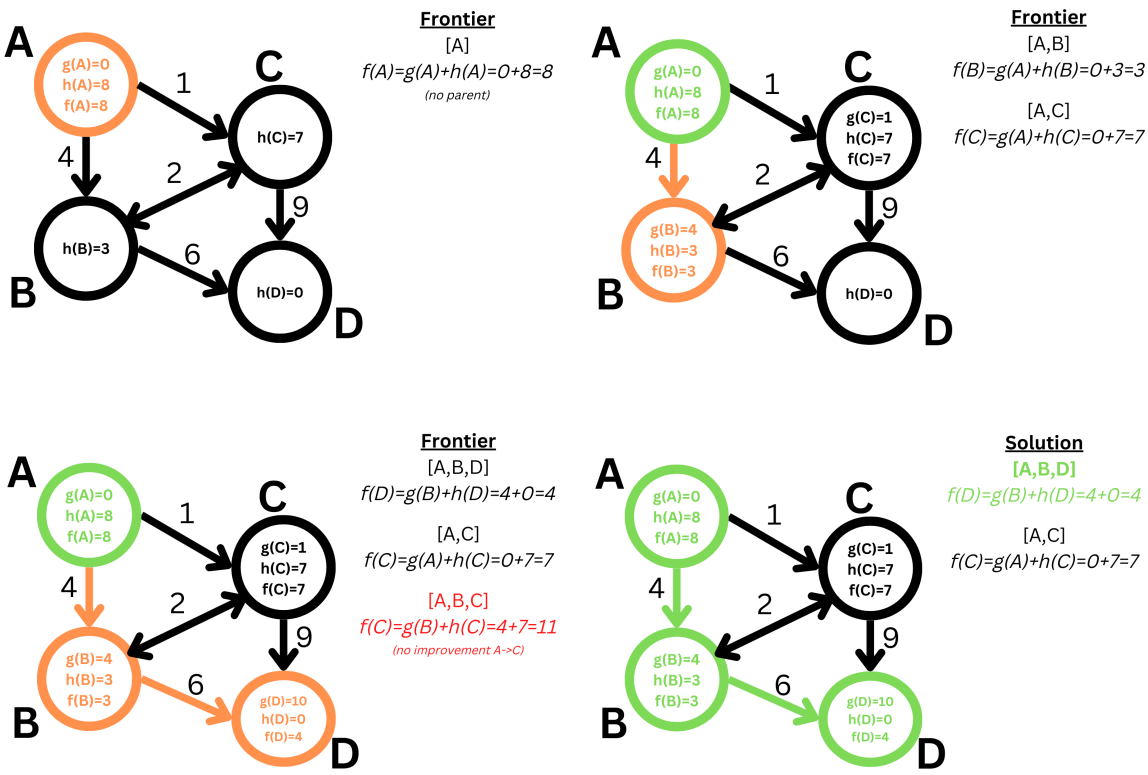


We see that the algorithm decided that the path $[A, C, D]$ is optimal since it was the first goal node to be expanded, however we see that it's total cost would be 10 while the optimal path ($[A, B, D]$) has a cost of only 8. Thus, we can see that this variant of A* is **complete but not optimal**.

(e) A* such that $f(n) = g(n') + h(n)$ is still complete, since the cost is lower bounded by zero and the heuristic remains admissible. It will no longer be optimal, though, and we can see this using the following example:



Here we have a search graph, which we would like to traverse from initial node A to goal node D with the help of an admissible heuristic h. If we use the function $f(n) = g(n') + h(n)$ to order our frontier for node expansion, then a run of A* will look as follows (nodes that are about to be expanded are denoted in orange, expanded nodes are green):



We see that the algorithm decided that the path $[A, B, D]$ is optimal since it was the first goal node to be expanded, however we see that it's total cost would be 10 while the optimal path ($[A, C, B, D]$) has a cost of only 9. Thus, we can see that this variant of A* is **complete but not optimal**.

Part 1.2

Now it's given that $0.5 \leq w_1 < w_2 \leq 1$. Let's denote P_1, P_2 to be the paths returned from running W-A* with the weights w_1, w_2 respectively. Remember to calculate $f(n) = w \cdot h(n) + (1 - w) \cdot g(n)$. Prove/Disprove: $cost(P_1) \leq cost(P_2)$.

Solution:

This claim is **False**, and we can use the 8×8 board that we've been working with this whole assignment in order to disprove it. Running our W-A* algorithm with $w_1 = 0.90$, we obtain the following results:

- Total Cost: 108
- Expanded: 37
- Actions: [0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0]

Now if we run our W-A* algorithm with $w_2 = 0.95 > w_1$, we obtain the following results:

- Total Cost: 98
- Expanded: 14
- Actions: [1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0]

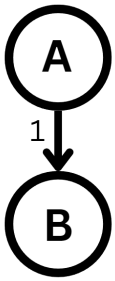
Thus, we see that even though $0.5 \leq w_1 = 0.90 < w_2 = 0.95 \leq 1$, we obtained $cost(P_1) = 108 > 98 = cost(P_2)$, and thus the original claim is disproved by counterexample.

Part 1.3

Prove/Disprove: For *any* search problem, the path returned by the UCS algorithm can change if we add a constant value $C > 0$ to each arc cost during the search.

Solution:

This claim is **False**, and the proof is fairly straightforward. We can consider the following search problem as a counterexample:



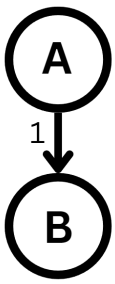
We would like to find a path from the initial node A to the goal node B, and regular UCS will easily return $[A, B]$ with cost 1 as the solution. No matter what constant value $C > 0$ we add to the cost of the arc between A and B, the UCS algorithm will always return $[A, B]$ as its solution. Thus, we have found a search problem for which the path returned by the UCS algorithm will *not* change if we add a constant value $C > 0$ to each arc cost during the search.

Part 1.4

Prove/Disprove: For *any* search problem, the path returned by the UCS algorithm can change if we multiply each arc cost by a constant value $C > 0$ during the search.

Solution:

This claim is **False**, and the proof is again fairly straightforward. We can consider the following search problem from before as a counterexample:



We would like to find a path from the initial node A to the goal node B, and regular UCS will easily return $[A, B]$ with cost 1 as the solution. No matter with what constant value $C > 0$ we multiply the cost of the arc between A and B, the UCS algorithm will always return $[A, B]$ as its solution. Thus, we have found a search problem for which the path returned by the UCS algorithm will *not* change if we multiply each arc cost by a constant value $C > 0$ during the search.

Exam Question 2 - Rick and Morty

Rick and Morty got lost on our $N \times N$ frozen lake. They are interested in meeting each other, no matter at which slot, as long as the cost of the path to their meeting point is the cheapest possible. Each turn, each of them takes one of the following steps: {right, left, down, up, stay in place}. Assume that the board now has two initial states s_1, s_2.

Part 2.1

Define (S, O, I, G).

Solution:

We will define (S, O, I, G) as follows:

$$S = \{(s, s') \mid s, s' \in \{0\} \cup [63]\}$$
$$O = \{(d, d') \mid d, d' \in \{RIGHT,\ UP,\ DOWN,\ LEFT,\ STAY\}\}$$
$$I = \{(s_1, s_2)\}$$
$$G = \{(s, s) \mid s \in \{0\} \cup [63]\}$$

Each state is actually a pair of squares on which each of the agents (Rick and Morty) stands. By definition, they can move in any direction or stay in place as described earlier in this assignment. The initial state is given and the final state is a state in which both agents are on the same square.

Part 2.2

Define the *Domain* function on an operator of your choice.

Solution:

We'll choose to examine the operator $(Right, Right)$, and so applying the *Domain* function on it we'll obtain:

$$Domain(Right, Right) = \{(s, s') \mid s, s' \in \{0\} \cup [63] \setminus H\}$$

where H is the set of holes. As we already know, the $RIGHT$ operator can be applied to all states that are not holes.

Part 2.3

Define the *Succ* function for the initial state.

Solution:

We define $S^* = \{0\} \cup [63]$. In addition, if there are 2 portal slots on the board, then we will call their indexes (positions) p_1, p_2 and define a function $P : S^* \to S^*$ in the following way:

$$P(s) = \begin{cases} p_2 & s = p_1 \\ p_1 & s = p_2 \\ s & \text{otherwise} \end{cases}$$

This function is required so that we can describe how stepping on a portal slot automatically launches the agent to the state of the other portal (in case there are no portals, this is just an identity function). Next, we'll define another function $L(s) : S^* \to 2^{S^*}$ such that:

$$L(s) = \begin{cases} \{P(s), P(s+1), P(s+N)\} & s = 0 \text{ (No slots to the left or up)} \\ \{P(s), P(s-1), P(s+N)\} & s = N-1 \text{ (No slots to the right or up)} \\ \{P(s), P(s+1), P(s-N)\} & s = N \cdot (N-1) \text{ (No slots to the left or down)} \\ \{P(s), P(s-1), P(s-N)\} & s = N^2 - 1 \text{ (No slots to the right or down)} \\ \{P(s), P(s+1), P(s+N), P(s-1)\} & 0 < s < N-1 \text{ (No slots above)} \\ \{P(s), P(s+1), P(s-N), P(s-1)\} & N \cdot (N-1) < s < N^2 - 1 \text{ (No slots below)} \\ \{P(s), P(s+1), P(s-N), P(s+N)\} & s \ mod\ N = 0 \text{ (No slots to the left)} \\ \{P(s), P(s-1), P(s-N), P(s+N)\} & s \ mod\ N = N-1 \text{ (No slots to the right)} \\ \{P(s), P(s-1), P(s-N), P(s+N), P(s+1)\} & \text{otherwise (there is a slot in every direction)} \end{cases}$$

Thus, with this function we can properly define $Succ$ as follows:

$$Succ(s_1, s_2) = \{(s, s') \mid s \in L(s_1),\ s' \in L(s_2)\}$$

Part 2.4

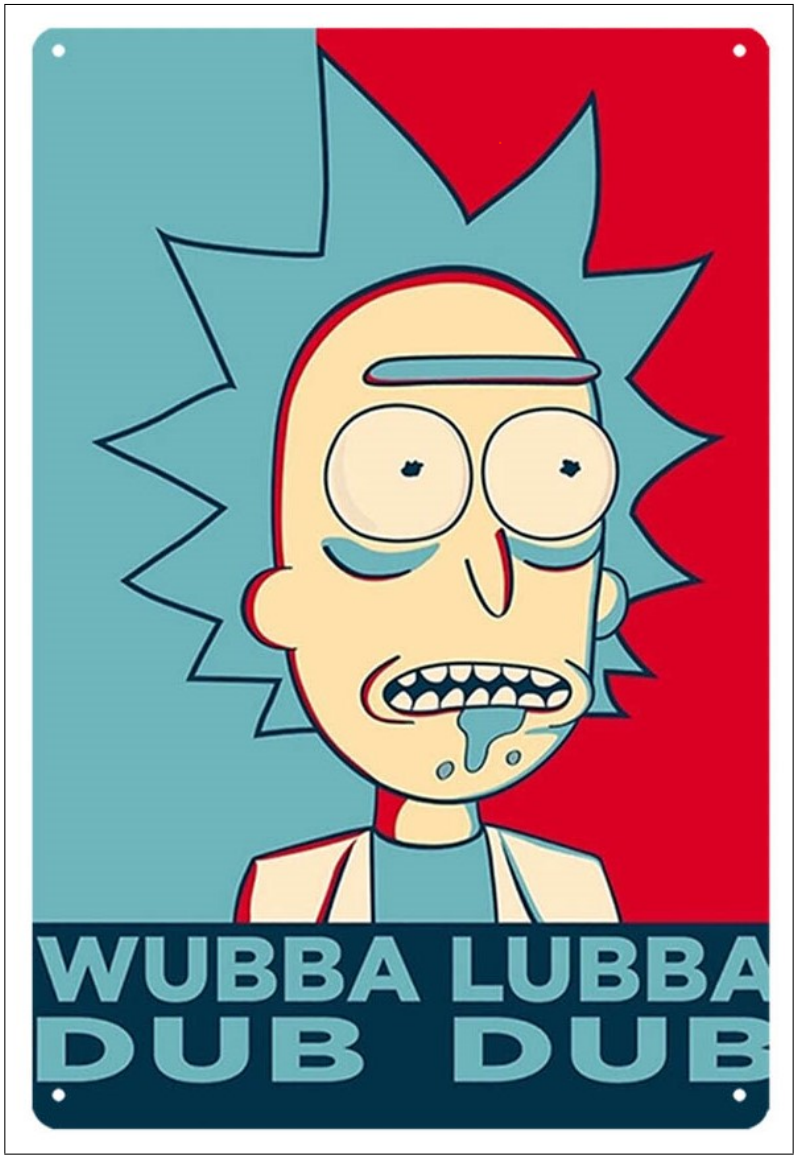
Propose an admissible heuristic (that is not the 0 heuristic).

Solution:

Let's define the following heuristic:

$$h(s, s') = \begin{cases} 0 & h_{Manhattan}(s, s') < 3 \\ 1 & \text{otherwise} \end{cases}$$

This indeed an admissible heuristic, since if the squares of s, s' are more than 2 squares away from each other, then even if each takes a step towards the other there will still be a space left between them. If both agents step on a hole slot (i.e. the cost of both steps is zero) then the value of the optimal heuristic will be infinity since they will get stuck. Otherwise if at least one of the agents did not step on a hole, then this means that the cost is greater than 0, and so $h(s, s') \leq h^*(s, s') = 1$. In the latter case $0 = h(s, s') \leq h^*(s, s')$ will always hold, and therefore it's still an admissible heuristic.



Thank You!