Homework 1

Administrations

General Guidelines

Please pay attention! If you do not follow procedures without special approval, you will receive a zero in the exercise.

- Due Date: 16.12.21. It can be submitted after the due date if a fine is paid, as per the delay procedure specified in the webcourse.
- The exercise solution should only be submitted in pairs, although you can request permission to submit it individually.
- Exercise solutions must be typed, i.e. not had written.
- Questions about the exercise should be sent to Tal Swisa (talswisa@campus.technion.ac.il) with the title AI_HW1, after first checking the FAQ on the website.
- You can request Tal Swisa to postpone the submission date of the exercise on a reasonable basis.
- During the exercise, there may be updates. Updates are required - please check the FAQ on the course website.
- The exercises should be solved independently, do not copy answers from other students. Any cheating will be punished severely.
- In the dry parts, in addition to the correctness of your answers, you are also tested on presenting your data and results in a readable and orderly manner in the places where you have been asked to do so.
- An automated testing system will check your code thoroughly. The system will compare your results with the results obtained from our implementation. We expect you to receive the same results. We will examine, among other things, the route obtained, the cost, and the amount of node expansions.

Technical Instructions

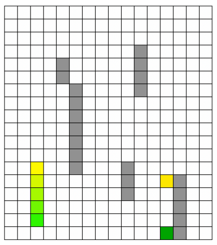
- Python 3.8 is the version you must use. The source files you received are also compatible with this version of Python. After you submit the exercise, you will not be able to correct errors caused by using another Python version.
- Don’t change any code you weren't requested to change.
- Write your code in the marked places only.
- Don’t create new code files.
- Don’t add imports to existing code files.
- The exercise requires you to install matplotlib, pandas, numpy, and heapdict packages (not HeapDict).
- Reports should contain answers to the sections where the ✍️ appears.
- In all the dry questions, you need to explain your answers.
- All changes in the exercise since the first version are marked with an orange marker.

Part A - Introduction

This task deals with the application of search algorithms to large navigation problems.
Before starting the exercise, it is recommended that you review the relevant lecture and recitation slides.

Problem Description

In our exercise, a rectangle-shaped robot needs to find a route from a starting point to a defined destination point in the maze.
The following image illustrates an example problem:



The robot appears at the left bottom corner in light yellow-green colors. The green end of the robot is its head and the yellow end is its tail. The goal is to find a path which will bring the robot tail to the yellow dot on the right (the destination point of the robot tail), and the head to the green point below it (the destination point of the robot head).
The robot can move forward ~~and backwards~~ (only forward) in the direction of its head, and can also turn right or left, with its center of rotation being the axis of rotation. The gray cells on the map represent walls. As the robot moves through the maze, it shouldn't collide with the wall. In the exercise, a problem of this kind is referred to as a "maze problem".

Problem Formulation

State Representation

A state is represented as a

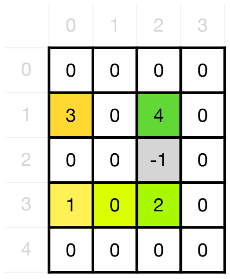
1. Matrix that represents the maze map.
2. The current location of the robot.

In a matrix, a square that is empty will be marked as 0 and a square that is a wall as -1. For example, the following matrix can represent a maze map:

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	-1	0
3	0	0	0	0
4	0	0	0	0

A robot's position is determined by its head and tail positions, which are as two entries in the matrix.
For example, in the maze map in the example, we can set the position of the robot's head to be (3, 2), and the position of the tail to be (3, 0).

Maze problems can be defined by their initial states and by the target locations of the robot's head and tail. For convenience, the maze problem can be pictured with a matrix of values -1, 0, 1, 2, 3, and 4, where 1 represents the position of the robot tail, 2, 3 represents the position of the robot head, and 4 represents the position of the robot tail. By setting the target positions of the tail and head to (1, 1) and (1, 3), we get the following matrix:



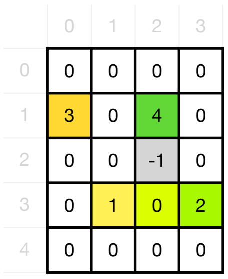
	0	1	2	3
0	0	0	0	0
1	3	0	4	0
2	0	0	-1	0
3	1	0	2	0
4	0	0	0	0

Denote the maze problem represented by this matrix as $M_{example}$.
However, not all maze problems can be described in this way, as sometimes the initial position of the robot head is the same as the target position of the robot tail. Because of this, this representation of the maze problem is not adequate, and it will only be used for explaining the problem (and creation of maze problems from csv files, later on).

The matrix enteries which are in between the robot's head and tail are called the robot body. The robot body can't lay upon a wall.
Throughout the exercise, the robot will be a rectangle with width 1 and odd length, at least 3.

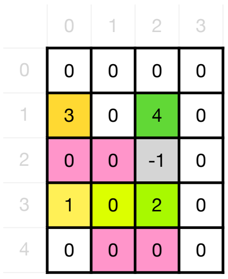
Operators

Forward - The robot's head and tail will advance one square in the direction the robot is looking (on continuation with the line from the tail to the robot's head). This action is possible only if it does not lead the robot out of the maze, and if the robot's head does not reach a slot with a wall.
Applying a forward move on the initial state of $M_{example}$ would result in the following state:



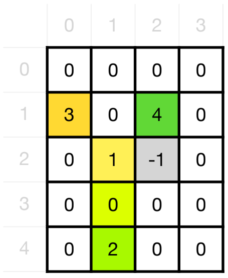
	0	1	2	3
0	0	0	0	0
1	3	0	4	0
2	0	0	-1	0
3	0	1	0	2
4	0	0	0	0

Right rotation - the robot will rotate right, relative to the direction it is looking in, with the rotation axis is the center of the robot. This action can only take place if the robot's head, tail, and body do not collide with the walls, and if it does not exit the maze. In order for the robot to rotate right from the initial state of $M_{example}$, the pink cells must be free.



	0	1	2	3
0	0	0	0	0
1	3	0	4	0
2	0	0	-1	0
3	1	0	2	0
4	0	0	0	0

After applying the rotation operator the we would receive the following state:



	0	1	2	3
0	0	0	0	0
1	3	0	4	0
2	0	1	-1	0
3	0	0	0	0
4	0	2	0	0

Left rotation - the same as right rotation, but to the right. A left rotation couldn't be applied on the initial state of $M_{example}$ because the robot head collides with a wall.

Task 1 (2 points)

✍ Is there exists a solution for $M_{example}$?
If there exists a solution, what is the path that achieves an optimal solution?
It there isn't, explain why.

Task 2 (2 points)

✍ Could there be cycles in the states graph of some maze problem?
If there could be, give an exmaple for such maze problem and a sequence of operators that achieves a cycle. Otherwise, explain why.

Task 3 (4 points)

✍ Is there a sink reachable from any maze problem state. If so, proove it. Otherwise, give an example to a state from which a sink is unreachable, and explain.
Reminder: as sink is a state upon which no operator could by applied (or equivalently the return value of any operator would is $\varnothing$)

Part B - Getting to Know The Code

Understading this part well is imprtant for the rest of the exercies.

Problem Representation in The Code (MazeProblem.py)

The MazeProblem Class

A class representing a maze problem. To produce a maze problem you need a matrix that represents the map (contains zeros and ones), the initial positions of the head and tail, their target locations, and the cost of the operators. The class implements the following functions:
expand_state
A function that accepts a state and returns one after the other the following consecutive states, with the cost of operating the appropriate operators. There is no need to check that the states returned from this function are valid. Note that the number of nodes expanded while running a search algorithm is the number of times the expand_state function is called.
is_goal:
A function that receives a state and returns whether it is a target state.

The Class MazeState

This class represents a state in the search space. To create a state, you need a maze problem (object of type MazeProblem), and the current positions of the robot's head and tail. The class implements the following functions:
robot_direction:
A function that returns an array that represents the direction in which the robot is looking (as defined in the "Forward" operator). For example, in $M_{example}$, the direction of the robot is [0, 1]. The first value is 0 because in the first axis, which corresponds to the matrix rows, the position of the robot's head and tail is the same. The second value is 1 because in the second axis, the robot head is located in a larger index than the robot tail. If the robot were looking up, the direction of the robot would be [-1, 0].

In addition the MazeProblem.py file implements the compute_robot_direction function which calculates the direction in which the robot is looking for any head and tail positions.

Tools for Solving Graph Search Problems (GraphSearch.py)

The Class Node

This class represents a node in the search tree. To generate a node you need the state that the node represents, the parent of the node in the search tree (None if the node is the root) and the g value of the node. The class implements the following functions:
get_path:
A function that returns the route in the search tree to the node.

The Class GraphSearchSolution

This class represents a solution to a graph search problem. To produce a solution you need the final node, the time it took to find the solution, and the number of nodes expanded during the search. If no solution is found, an object from the class must be created with final_node = None, and with the reason that no solution is found (time is running out, or there is no solution).

The Class NodeCollection

This class implements a collection of states associated with nodes in the search tree. We will use this class as a close set in the search algorithms.
The class implements the following functions:
add:
Adding a node to the collection. The class assumes that the state represented by the node is not in the collection (as is the case with search algorithms when adding a node to close). If it is found you will get an error.
remove_node:
Deleting a node from the collection.

get_node:
Returns the appropriate node to the state.
In addition you can check if a situation is in the collection by the in (s in close) operator.

The Class NodesPriorityQueue

This class implements a priority queue of nodes. We will use this class as an open set in search algorithms.
add:
Adding a node to a queue, with some priority. The class assumes that the state represented by the node is not in the collection (as is the case with search algorithms when adding a node to open). If it is found you will get an error.
pop:
Removes and returns the node with the lowest priority. If the queue is empty, None will be returned.
get_node:
Returns the appropriate node to the state.
remove_node:
Removes the node from the queue.

The Class Queue

The class implements a queue.
add:
Adding an item to a queue.
pop:
Returns and removes an item from the queue in FIFO order. If the queue is empty, None will be returned.

In addition you can check if a state is in a queue using the in operator (s in q), and check the number of nodes in the queue using the len function, (len (q)).

Search Algorithms (Robot.py)

The Class Robot

This class is an abstract class for a search algorithm in a states graph. Classes that implement this class should implements the following functions:
solve:
A function that accepts a problem of type MazeProblem and returns a solution of type GraphSearchSolution

The Class BreadthFirstSearchRobot

A class that inherits from Robot and implements the breadth first search.

The Class BestFirstSearchRobot

This class is an abstract class that inherits from the Robot class, and implements the best first search.
In addition to the solve function, departments that implement this class need to implement the function:
calc_node_priority_:
A function that calculates the priority (f-value) for a node in the search tree.

The Class WAStartRobot and UniformCostSearch:

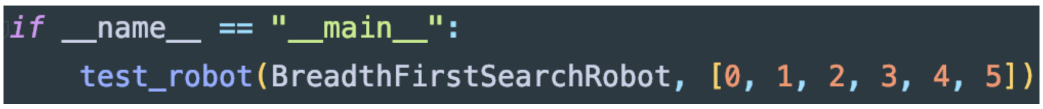
Those classes inherint from BestFirstSearchRobot and implements $wA*$ and UCS respectively.

Part C - Uninformed Search

In this section we will start writing code. The file main.py will not be tested, so you can feel free to change the main as you wish in order to perform the exercise tasks.

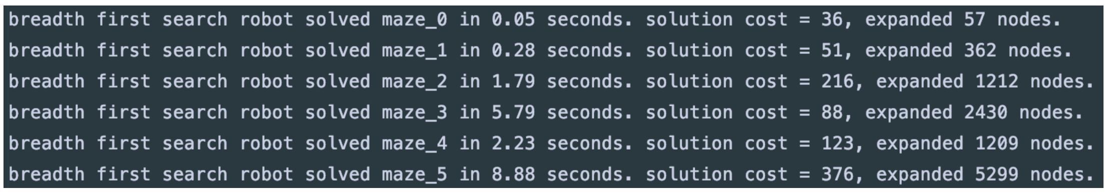
Task 4 (6 points)

1. Complete the implemetation of BreadthFirstSearchRobot in Robot.py.
   Although the breadth first search algorithm does not refer to the prices of the operators, the returned solution must contain the cost of the route (and not the length of the route in terms of the number of operators in it).
2. In the Utilities.py file, the test_robot function is implemented, which receives a non-abstract type that inherits from the Robot class, indexes of mazes, and parameters for initializing the robot if necessary. The function loads the mazes from csv files, runs the robot on the mazes and prints the results. Add a call to test_robot with a BreadthFirstSearchRobot object and 0 to 5 mazes:



```
if __name__ == "__main__":
    test_robot(BreadthFirstSearchRobot, [0, 1, 2, 3, 4, 5])
```

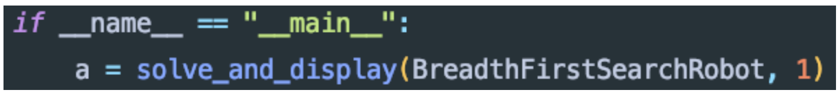
   You should see the following print:



```
breadth first search robot solved maze_0 in 0.05 seconds. solution cost = 36, expanded 57 nodes.
breadth first search robot solved maze_1 in 0.28 seconds. solution cost = 51, expanded 362 nodes.
breadth first search robot solved maze_2 in 1.79 seconds. solution cost = 216, expanded 1212 nodes.
breadth first search robot solved maze_3 in 5.79 seconds. solution cost = 88, expanded 2430 nodes.
breadth first search robot solved maze_4 in 2.23 seconds. solution cost = 123, expanded 1209 nodes.
breadth first search robot solved maze_5 in 8.88 seconds. solution cost = 376, expanded 5299 nodes.
```

   Of course the runtime can vary, but the price of the solution and the number of nodes developed should be the same. If you get other results, you need to correct your implementation.

3. You can view the mazes and solutions found by the robots you implemented by calling the solve_and_display function. The function receives a non-abstract type inherited from the Robot class, an index of a maze, and also parameters for initializing the robot if necessary, and presents the solution of the maze problem found by the robot in animation. Try it now with BreadthFirstSearchRobot and Dungeons 0 through 5

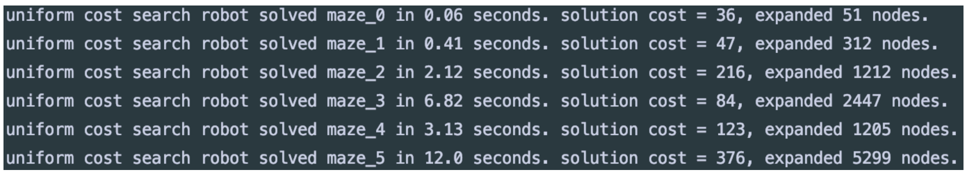


```
if __name__ == "__main__":
    a = solve_and_display(BreadthFirstSearchRobot, 1)
```

   You may have trouble running the animation. A helper document for running the animation appears on the course website (where the exercise is). If you have tried everything that is said in the document and still have not been able to run the animation, please contact Tal so that we can address the problem and add the solution to the document. The animation is not needed for solving the exercise, but it might help. The helper document is written in Hebrew, if you troubles understanding it, please contact Tal.

Task 5 (8 points)

1. Complete the implementation of BestFirstSearchRobot in Robot.py.
   As shown in the recitation, algorithm $A*$ is a particular case of best first search for which $f = g + h$. Its pseudo-code is the same as $A*$'s, except that the priority function is general (determined by the implementation of _calc_node_priority).
2. Complete the implementation of UniformCostSearchRobot by implementing its function _calc_node_priority.
3. call test_robot with UniformCostSearchRobot and mazes 0 to 5. You should see the following print:



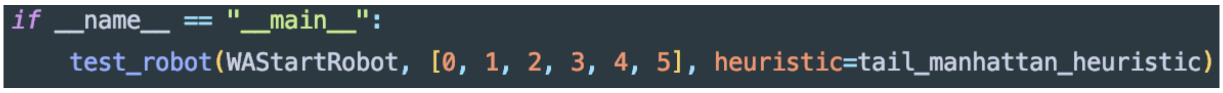
```
uniform cost search robot solved maze_0 in 0.06 seconds. solution cost = 36, expanded 51 nodes.
uniform cost search robot solved maze_1 in 0.41 seconds. solution cost = 47, expanded 312 nodes.
uniform cost search robot solved maze_2 in 2.12 seconds. solution cost = 216, expanded 1212 nodes.
uniform cost search robot solved maze_3 in 6.82 seconds. solution cost = 84, expanded 2447 nodes.
uniform cost search robot solved maze_4 in 3.13 seconds. solution cost = 123, expanded 1205 nodes.
uniform cost search robot solved maze_5 in 12.0 seconds. solution cost = 376, expanded 5299 nodes.
```

Task 6 (5 points)

1. ✍️ The price of the solution obtained from running UniformCostSearchRobot on mazes 1 and 3 is smaller than that obtained with BreadthFirstSearchRobot. Explain why this happens.
2. ✍️ Write as simple a condition as possible about the solutions of any general search problem, so that the price of the solutions returned from each run of the breadth first search algorithm will be the same as the price of the solutions returned from each run of.

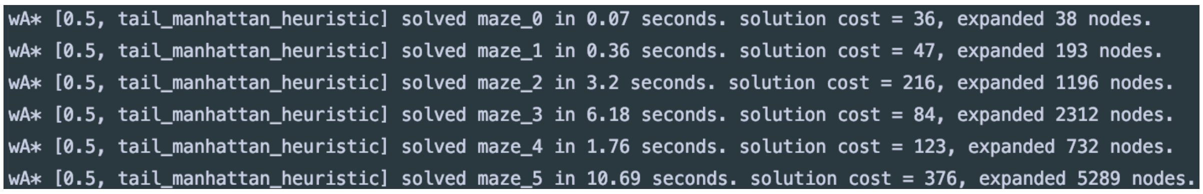
Part D - Informed Search

1. Complete the implementation of the WAStartRobot class by implementing the _calc_node_priority function. Please note, we define $f = (1 - w) \cdot g + w \cdot h$.
2. In order to use WAStartRobot, we need to define a heuristics function. We will define the Manhattan heuristics to be the Manhattan distance between the robot tail and the target position of the tail multiplied by the price of the forward operator. Implement this heuristic in the tail_manhattan_heuristic function in the Heuristics.py file.
3. Call the test_robot function with WAStartRobot initialized with tail_manhattan_heuristic, and with maps 0 to 5.



```
if __name__ == "__main__":
    test_robot(WAStartRobot, [0, 1, 2, 3, 4, 5], heuristic=tail_manhattan_heuristic)
```

You should see the following print:



```
wA* [0.5, tail_manhattan_heuristic] solved maze_0 in 0.07 seconds. solution cost = 36, expanded 38 nodes.
wA* [0.5, tail_manhattan_heuristic] solved maze_1 in 0.36 seconds. solution cost = 47, expanded 193 nodes.
wA* [0.5, tail_manhattan_heuristic] solved maze_2 in 3.2 seconds. solution cost = 216, expanded 1196 nodes.
wA* [0.5, tail_manhattan_heuristic] solved maze_3 in 6.18 seconds. solution cost = 84, expanded 2312 nodes.
wA* [0.5, tail_manhattan_heuristic] solved maze_4 in 1.76 seconds. solution cost = 123, expanded 732 nodes.
wA* [0.5, tail_manhattan_heuristic] solved maze_5 in 10.69 seconds. solution cost = 376, expanded 5289 nodes.
```

Task 8 (12 points)

tail_manhattan_heuristic is not admissible.

1. ✍️ Explain why, and define a necessary and sufficiant condition on the cost of the turn operators, such that the tail_manhattan_heuristic would be admissible.
2. Create a csv file named maze_99 in the Mazes folder that represents a maze problem for which the solution returned by WAStartRobot using this heuristic with w = 0.5 is not optimal. Empty squares in the maze should contain the value 0, wall squares the value -1, in the initial positions of the tail and the robot head the values 1 and 2 should appear respectively, and in the target positions of the tail and the robot head the values 3 and 4 should appear respectively. Note that the maze needs to be successfully loaded by calling the test_robot function.

Task 9 (5 points)

A simple way to make the heuristic admissible is calculating the Manhattan distance between the current robot <u>center</u> point and the robot <u>center</u> point at its target location.

1. ✍️ Prove that this heuristic is acceptable.
2. Implement this heuristic in the center_manhattan_heuristic function.

Task 10 (5 points)

The w parameter of the wA * algorithm affects the greediness level of the search algorithm. We would like to examine the effect of w on the runtime and solution quality of the wA * algorithm on mazes 0 to 5 used with center_manhattan_heuristic heuristics.

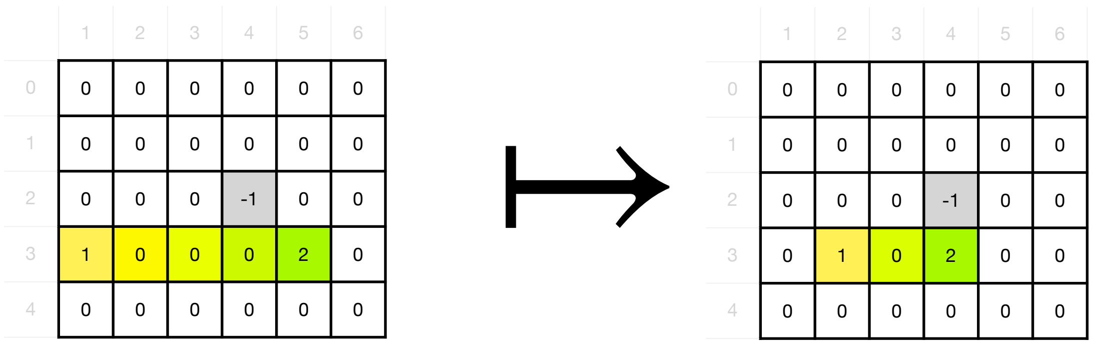
1. Complete the implementation of the w_experiment function that gets an index of maze and saves in the plots folder two graphs - one graph of the runtime of wA * as a function of w, and a second graph of the price of the solution found wA * as a function of w.
2. ✍ Run the function with maps 0 to 2, attach the graphs to the report and explain the results.
   ** It turns out that it is not possible to create files with an asterisk (*) in their name in Windows, so whoever uses Windows should remove the asterisk from the names of the graphs stored in the function, i.e. change wA * to wA in calls to the plt.savefig function.*

Part E - Sophisticated Heuristic

Notations for this part:

- For states s, s', we donote $d(s, s')$ the price of the cheapest path between them. if s' is unreachable forn s, we define $d(s, s') = \infty$.
- For a maze problem, we denote by s^i its initial state, and s^* its goal state.

One of the things that makes it difficult for a robot to move in a maze is its length - shorter robots can make turns in more places and thus reach the destination location in cheaper ways. We can take advantage of this feature of the problem to define the heuristics h_k: $h_k(s)$ it is the price of the shortest path from the state s for a robot shorter by k units from the original robot (for something fixed k, even, at most the length of the robot minus 3). Given a state s, we will define s_k to be the same state except that the robot has been shortened by k units. More precisely, in s_k the positions of the head and tail of the robot are close to its center in $k/2$ squares. For example if the left state in the following example is s, then s_k is the right state for $k = 2$.



	1	2	3	4	5	6
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	-1	0	0
3	1	0	0	0	2	0
4	0	0	0	0	0	0

$\mapsto$

	1	2	3	4	5	6
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	-1	0	0
3	0	1	0	2	0	0
4	0	0	0	0	0	0

Similarly, for s^* the goal state, we define s_k^* to be as s^*, except the head and tail location are closer, in the same manner as before. Now we can write $h_k(s) = d(s_k, s_k^*)$.

Notices that in a naive implementation, using h_k takes a lot of time - for each created state s, we would need to calculate $d(s_k, s_k^*)$ by solving tha appropriate maze problem. Instead, we would perform a preliminary calculation (before starting the search) according to which we can induce $h_k(s)$ for each state s.

For a state s we define $\bar{s}$ an identical state except the head and tail positions have been switched.

Task 11 (5 points)

✍️ Explain we for any two states s, s' it holds $d(s, s') = d(\overline{s'}, \bar{s})$.

According to task 11, it holds $h_k(s) = d(s_k, s_k^*) = d(\overline{s_k^*}, \overline{s_k})$. We will efficiently calculate the $d(\overline{s_k^*}, \overline{s_k})$ for each state s, and thus save precious time.

As you learned (or will learn) in the Algorithms 1 course, you can efficiently calculate the price of the easiest path from a particular node to all the other nodes in the graph using the Dijkstra algorithm. In fact, you already implemented this algorithm in the code when you implemented UniformCostSearchRobot. Calling the solve function with compute_all_dists = True will prevent the search from ending until open is empty, and will return close, which will contain all states reachable from the initial node, and their g values, which are guaranteed to be the shortest path costs from the initial state.

Task 12 (8 points)

For the sake of this task solely, assume that in the preliminary calculation, instead of calculating the cheapest path from $\overline{s_k^*}$ to all reachable states, we calculate the cheapest path from $\overline{s_k^*}$ to $\overline{s_k^i}$ solely, for s^i the initial state of the original maze problem, using A^* and some heuristic h_{pre}.
When solving the original problem, we would use the heuristic $h(s) = g_{\overline{s_k^*}}(\overline{s_k})$ where $g_{\overline{s_k^*}}(\overline{s_k})$ is the minimal g value found for a state $\overline{s_k}$ which was inserted to close during the preliminary calculation. If $\overline{s_k}$ was not inserted to close, we define $g_{\overline{s_k^*}}(\overline{s_k}) = 0$.

1. ✍️ Is $h(s) = g_{\overline{s_k^*}}(\overline{s_k})$ is admissible? If it is, prove this. Otherwise, write down a condition on h_{pre} that would insure $h(s) = g_{\overline{s_k^*}}(\overline{s_k})$ is admissible.
2. ✍️ In general, is the use of A^* with non-admissible heuristics guaranteed to lead to a non-optimal solution?

* This task have change since the homework was published. If you already solve the previous version of this task, you can submit it instead of the new version.

Task 13 (9 points)

We will implement h_k at the class ShorterRobotHeuristic. Notice that in unlike task 12, in the preliminary calculation we will calculate the shoretest path from $\overline{s_k^*}$ to all the reachable states, as described after task 11. The calculation of the $d(\overline{s_k^*}, \overline{s_k})$ values will be performed at the initialization of the class, by creating the appropriate maze problem and calculating all the distances from its initial state by running the UniformCostSearch algorithm.

1. Implement _compute_shorter_head_and_tails tht receive the head and tail location of a robot and returns the location of the head and tail after shortening the robot by k units (note that k is a field in the class).
2. Complete the implementation of the __init__ function of the ShorterRobotHeuristic according to the following steps:
   1. Create the maze problem with which we will calculate the $d(\overline{s_k^*}, \overline{s_k})$ values. Notice that since we want to calculate the distances from its initial state to <u>all</u> other states, and hence neglect the goal state, the position of the robot's head and tail in the goal state is not important.
   2. Save the close set returned from UCS (don't foget to set compute_all_dists=True).
3. Implement the __call__ function of ShorterRobotHeuristic, which is called when computing the heuristic value of a given state.

Task 15 (8 points)

Although we have saved a lot of time with the efficient implementation of heuristics, it still takes a lot of time due to the preliminary calculation. We are interested in analyzing the new heuristic and understanding when it should be applied. Assume there is a single path $\overline{s_k^*}$ and $\overline{s_k^i}$.

1. ✍️ Prove that if there is a solution to the original problem, then it is unique.
2. ✍️ Prove or refute: $h_k = h^*$.
3. ✍️ Assuming that there is a solution to the original problem, prove that while solving the original problem with h_k, A^* only expands nodes on the optimal solution path.

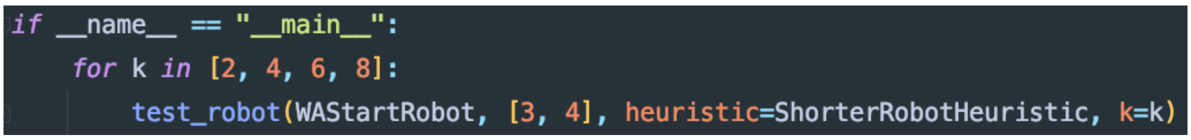
For some given maze problem, denote by $n_{manhattan}$ the number of nodes expanded by runing A^* with the center_manhattan_heuristic, m_k the number of nodes expanded by UCS in the preliminary calculation of the h_k heuristic, and n_k the number of nodes expanded while running A^* using h_k, after the preliminary calculation.
Assume that the running time of the preliminary calculation is $c \cdot m_k$, the running time of A^* using h_k is $c \cdot n_k$, and the running time of A^* using center_manhattan_heuristic is $c \cdot n_{manhattan}$.
Thus, we should use the new heuristic if it holds $n_k + m_k < n_{manhattan}$.

4. ✍️ Denote by a the number of states reachable from $\overline{s_k^*}$, and assume there exsits a single path from $\overline{s_k^*}$ to $\overline{s_k^i}$, and that there exsits a solution for the original problem, that uses r operators. Give a necessary and sufficiant condition for A^* using h_k being faster than A^* with center_manhattan_heuristic, where we consider the preliminary calculation in the running time of A^* with h_k. Use r, a and $n_{manhattan}$ in your solution.
5. ✍️ A bonus question! (5 points for the homework grade, and lots of appreciation). Describe a general maze map for which the running time of A^* using the center_manhattan_heuristic is greater than that of A^* using h_k (including the preliminary calculation) to any extent we desire.

Task 16 (6 points)

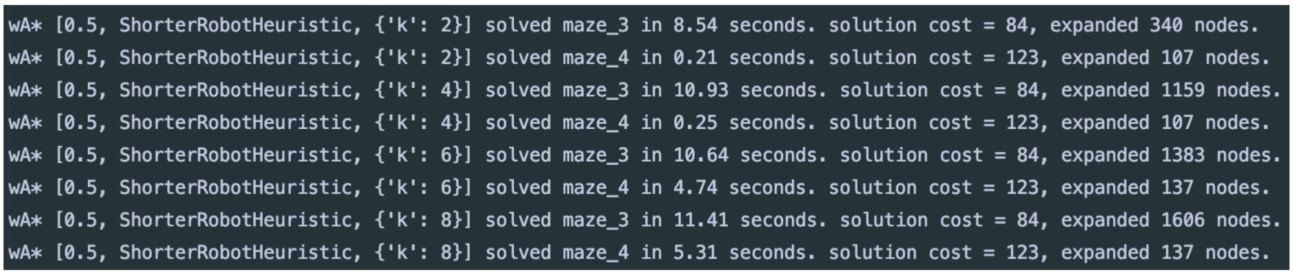
To test whether the new heuristic is useful, we would like to try it out.

1. ✍️ Call test_robot with WAStartRobot with w=0.5 and ShorterRobotHeuristic with various k values and various maps. Does the new heursitic improve performance?
For the following calls to test_robot:



```
if __name__ == "__main__":
    for k in [2, 4, 6, 8]:
        test_robot(WAStartRobot, [3, 4], heuristic=ShorterRobotHeuristic, k=k)
```

You should get the following print:



```
wA* [0.5, ShorterRobotHeuristic, {'k': 2}] solved maze_3 in 8.54 seconds. solution cost = 84, expanded 340 nodes.
wA* [0.5, ShorterRobotHeuristic, {'k': 2}] solved maze_4 in 0.21 seconds. solution cost = 123, expanded 107 nodes.
wA* [0.5, ShorterRobotHeuristic, {'k': 4}] solved maze_3 in 10.93 seconds. solution cost = 84, expanded 1159 nodes.
wA* [0.5, ShorterRobotHeuristic, {'k': 4}] solved maze_4 in 0.25 seconds. solution cost = 123, expanded 107 nodes.
wA* [0.5, ShorterRobotHeuristic, {'k': 6}] solved maze_3 in 10.64 seconds. solution cost = 84, expanded 1383 nodes.
wA* [0.5, ShorterRobotHeuristic, {'k': 6}] solved maze_4 in 4.74 seconds. solution cost = 123, expanded 137 nodes.
wA* [0.5, ShorterRobotHeuristic, {'k': 8}] solved maze_3 in 11.41 seconds. solution cost = 84, expanded 1606 nodes.
wA* [0.5, ShorterRobotHeuristic, {'k': 8}] solved maze_4 in 5.31 seconds. solution cost = 123, expanded 137 nodes.
```

2. ✍️ Describe in general terms how k affects runtime.
3. We would like to test the effect of k on the running time on various maps. Complete the implementation of shorter_robot_heuristic_experiment, which solves the maze problem it receives by running $wA*$ with ShorterRobotHeuristic and all possible k values, and produces a graph of the preliminary calculation running times, and the total running time for each value of k.
4. ✍️ Run the function with mazes 2 to 5, add the graphs to the report, and explain the resuts. (The run might take a long time for some of the mazes).

Part F - Two questions about $IDA*$

<u>Task 17 (10 points)</u>

Assume $IDA*$ is run with admissable heuristics only.

1. ✍️ During a search performed by running $IDA*$, at the end of some iteration (I.e. after returning from some call to $DFS - f$), it was deiceded to increase the size of the robot by 2 units. Is it guaranteed that we will get an optimal solution for the bigger robot if we use the new-limit value obtained by the last iteration? Explain your answer.
2. ✍️ We are interested in solving some maze problems in which the maze map is identical, and the starting and ending locations of the robots are the same, except that the robots' lengths are different (like the $s_k$definition). We decided to use the $IDA*$algorithm. With a smart initialization of the f-limit values, how can we reduce the total running time?

Submission Instructions

- The ID numbers of the two submitters must be written in the report.
- The report should be typed and in PDF format. Reports in word format will be fined.
- The report should be legible and easy to understand.
- Answers in the report should follow the order of the questions.
- You should subnit a zip file with the name AI_HW1_123456789_987654321 (with your ID numbers instead of the numbers).
- The zip should contain:
  - The report is in PDF format.
  - All the code files and folders you received, with all the files that were in them, and the maze_99 csv file (not excel) in the Mazes folder.

Good Luck!