Middle East Technical University Department of Computer Engineering

CENG 462

Artificial Intelligence

Fall '2021-2022

Homework 2

Due date: 27 November 2021, Saturday, 23:55

1 Objectives

In the first assignment, you'd been asked to implement uninformed search methods (Breadth-First Search (BFS), Depth-Limited Depth-First Search (DLDFS), Uniform-Cost Search (UCS), Iterative-Deepening Depth-First Search (IDDFS)) with the `TREE-SEARCH` algorithm. This assignment aims to broaden the set of search methods (so far) with an informed search method, namely A*, along with the `GRAPH-SEARCH` algorithm.

2 Problem Definition

Informed search methods aim to guide a search operation with some additional information to attain the objective of a problem more quickly than uninformed search methods. In order to retain the optimality of solutions, the information incorporated has to conform to some specific requirements.

In this assignment, you are expected to implement Uniform-Cost Search (UCS) and the A* search with the `GRAPH-SEARCH` algorithm in order to solve 8-puzzle and maze problems.

2.1 8-puzzle Problem

The 8-puzzle problem (Figure 1) consists of eighth tiles that can be slid into a gap location and demands the sequence of tile movements that results in a specific tile ordering by starting from an initial configuration. Basically, when a movement action takes place in the problem, the gap and the picked tile switch their places. Therefore, the movements of tiles are restrained solely by the location of the gap. There are four possible movements (actions) that can be applied to the problem. Figure 2 depicts resulting configurations after applying a particular action on the configuration shown in Figure 1. When the left action (moving the tile on the left of the gap) is applied to the configuration of Figure 1, the tile labeled with '4' is slid rightwards (Figure 2a). Configuration changes for the other actions (up, right, down) are straightforward similarly. When an invalid action is applied to a particular configuration, no configuration change occurs. For instance, if another left-action is applied to the configuration depicted in Figure 2a, since there is no tile on the left, the same configuration is retained as the result of this action (similarly, up-action on Figure 2b, right-action on Figure 2c, down-action on Figure 2d).

Solution of a 8-puzzle problem corresponds to providing an action sequence that alters the initial configuration of the puzzle into a demanded final configuration gradually, when applied on the puzzle.

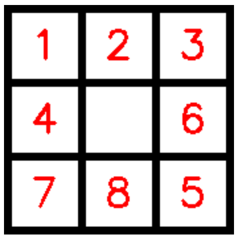
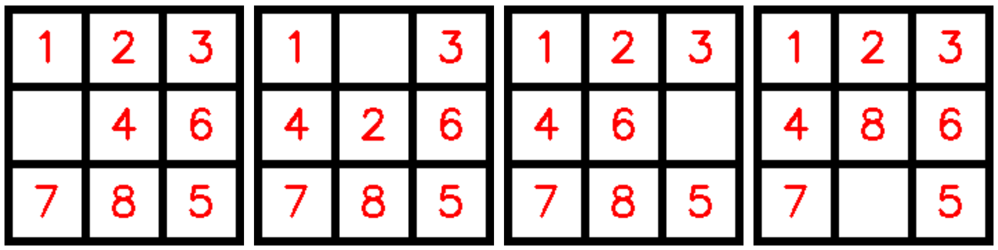


Figure 1: A sample 8-puzzle problem



(a) left-action (b) up-action (c) right-action (d) down-action

Figure 2: Resulting configurations after applying actions on the configuration of Figure. 1

Figure 3 depicts the action sequence to follow to transform a given 8-puzzle configuration into a target configuration step by step.

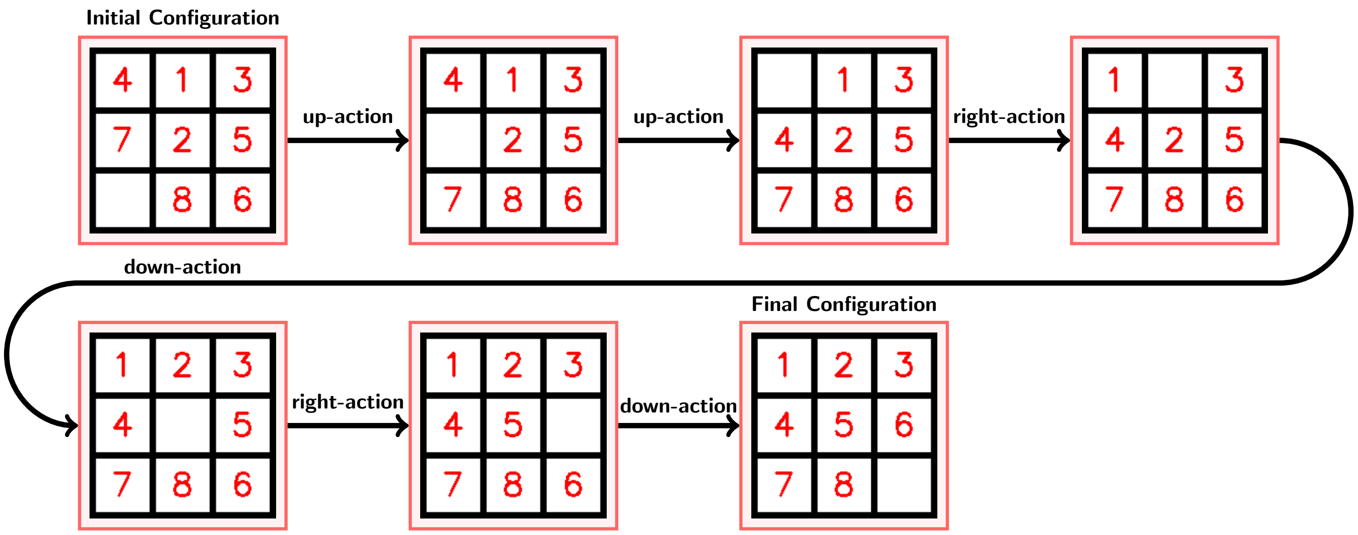


Figure 3: Solution steps of a given initial 8-puzzle configuration to reach a final (target) configuration.

2.2 Maze Problem

The maze problem involves searching for an exit path among obstacles by starting from a particular location, as shown in Figure 4 (the blue and red dots represent the start and target locations, respectively). The solution of the problem is the path with the smallest number of steps (the green-colored path as shown in Figure 5). Obstacles are represented with black squares and available pathway locations are depicted with white squares. The action set of the problem is the same as the 8-puzzle problem. left-action, right-action, up-action, down-action mean to move to the square on the left, right, up, down, respectively.

If an invalid action is taken, such as attempting to run over a block location, no place location change occurs.

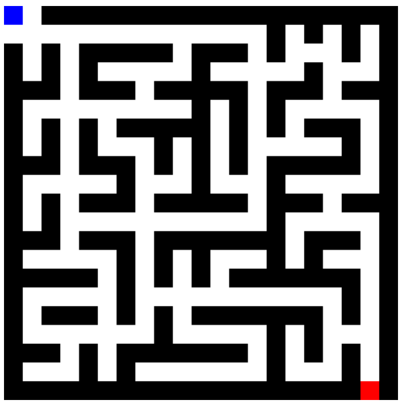


Figure 4: A sample of the maze problem.

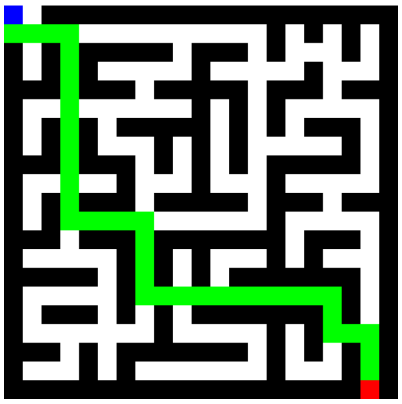


Figure 5: One of the optimal solutions of the maze problem depicted in Figure 4.

3 Specifications

- You are going to implement UCS and A* search in Python 3.
- Each problem is represented with a file and this file will be fed to your implementation along with one of the methods.
- To unify all methods in a single place, you are expected to write the following function:

```python
def InformedSearch(method_name, problem_file_name):
```

  The parameter specifications are as follows:

  - `method_name`: specifies which method to be run for the given problem. It can be assigned to one of the values from `["UCS", "AStar"]`.
  - `problem_file_name`: specifies the path of the problem file to be solved. File names are dependent on the problem specified and take values such as "eightpuzzle1.txt", "eightpuzzle5.txt", "maze1.txt", "maze10.txt".

  During the evaluation process, this function will be called and returned information will be inspected. The returned information is dependent on the method fed and as follows:

  - `method_name=="UCS"`: `UnInformedSearch` returns the optimal solution sequence, the list of processed nodes/states during search, the depth value of the found target node and the optimal solution cost. If no solution is found then it returns a single `None`.
  - `method_name=="AStar"`: The same output structure as UCS should be generated.

  NOTE: For inspecting the algorithms' behavior, the order of nodes/states/locations in the processed nodes/states/locations list is important so a generic node expansion strategy should be adapted. This strategy is as follows: for both types of problems while expanding a node, (left-action, up-action, right-action, down-action) order should be followed. That is to say, first the child node generated as the result of applying the left-action should be processed, then the up-action's, and so forth.

  As in the first assignment, the depth value for the starting node is 0.

  Each action (for both problem types) is considered to incur a cost value of 1.

- 8-puzzle problems are described in ".txt" files and have the following structure:

```
a b c
d e f
g h i

j k l
m n o
p q r
```

  The first 3 lines and the last 3 lines specify the initial configuration and the demanded final configuration of the puzzle, respectively (There is a blank line between the configurations). Each character represents a value from [0, 1, 2, 3, 4, 5, 6, 7, 8] (The number 0 specifies the gap).

- Maze problems are described in ".txt" files too and have the following structure:

```
starting coordinate
target coordinate
XXXXXXXXXXXXX......
XXXXXXXXXXXXX.....
XXXXXXXXXXXXX.....
.
.
.
.
```

The first two lines specify the starting and target coordinates of the given maze problem. The coordinates are represented as "(a,b)" where a and b are values of the x-axis and y-axis, respectively. By starting from the third line, the maze is described. It is divided in to cells (represented via Xs) and each cell is either `' '` or `'#'`, where `' '` represents an available pathway and `'#'` is for the obstacles. The origin (of the coordinate system, (0, 0)) is located at the top leftmost cell. x and y coordinate values increase rightwards and downwards, respectively as shown in Figure 6.

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)

Figure 6: Coordinate values on a 8x5 maze problem. Each square represents a single cell.

Maze problems can be of any rectangular size (10x10, 20x20, 30x20, etc.), where the first and second numerical values on the size tuple are the width and height of the maze, respectively.

- No `import` statement is allowed and all methods should be implemented in a single solution file. You are expected to implement all the necessary operations by yourself without utilizing any external library.
- Commenting is crucial for understanding your implementation and decisions made during the process. Your implementation can be manually inspected at random or when it does not satisfy the expected outputs.

NOTE: Unlike the first one, in this assignment you are expected to utilize the `GRAPH-SEARCH` algorithm for both search methods. With the A* search, you are expected to make use of any heuristic function that does not violate the optimality of the method for each type of problem. Please specify which heuristic function you have utilized for each problem at the very beginning of your implementation as a comment.

4 Sample I/O

eightpuzzle1.txt:

```
6 1 2
4 0 3
7 8 5

1 2 3
```

```
4 5 6
7 8 0
```

Expected outputs of the InformedSearch function with different methods on applied on eightpuzzle1.txt:

```
>>> import hw2solutioneXXXXXXX as hw2
>>> print(hw2.InformedSearch("AStar", "eightpuzzle1.txt"))
(['DOWN', 'LEFT', 'UP', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'DOWN',
    'RIGHT', 'RIGHT', 'UP', 'LEFT', 'DOWN', 'RIGHT'], ['6124 3785', '6
   2413785', '61243 785', ...], 16, 16)
>>> print(hw2.InformedSearch("UCS", "eightpuzzle1.txt"))
(['DOWN', 'LEFT', 'UP', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'DOWN',
    'RIGHT', 'RIGHT', 'UP', 'LEFT', 'DOWN', 'RIGHT'], ['6124 3785', '6
   2413785', '61243 785', ...], 16, 16)
```

The optimal action sequence is represented as an array each element of which defines which action to apply by starting from the initial puzzle configuration (LEFT is for left-action, RIGHT is for right-action, and so forth), similar to what is depicted in Figure 3. The processed search tree nodes have been shown partially (due to their number). Each puzzle configuration has been represented as a string which consists of 9 characters whose first three characters (indices 0:3) specify the upper row of the puzzle, similarly, the next three characters (indices 3:6) and the last three characters (indices 6:9) represent the middle row and bottom row of the puzzle, respectively. The space character (' ') is for the gap. The last two numerical values are the depth and cost values of the solution, respectively.

maze1.txt:

```
(0,0)
(19,20)
  ###################
              # # # #
# # ##### ### #   # #
# # #     #     #   #
### ### ##### ### ###
#         # # #     #
# # # ##### # # ### #
# # #   # # #     # #
### ### # # # ##### #
#     #   #   #     #
# # ### ##### ### ###
# #           #     #
### ### ####### ### #
#     # # #   # #   #
##### # # # ####### #
#     #           # #
# ### # # ####### # #
#       #     # #   #
### # ####### # # # #
#   # #       #   # #
################### #
```

Expected outputs of InformedSearch function with different methods applied on maze1.txt:

```
>>> import hw2solutioneXXXXXXX as hw2
>>> print(hw2.InformedSearch("AStar", "maze1.txt"))
([(0, 0), (1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4,
   5), (5, 5), (5, 6), (5, 7), (6, 7), (7, 7), (7, 8), (7, 9), (7, 10), (7, 11)
   , (7, 12), (7, 13), (7, 14), (7, 15), (8, 15), (9, 15), (10, 15), (11, 15),
   (12, 15), (13, 15), (14, 15), (15, 15), (16, 15), (17, 15), (17, 16), (17,
   17), (18, 17), (19, 17), (19, 18), (19, 19), (19, 20)], [(0, 0), (1, 0), (0,
    1), ...], 39, 39)
>>> print(hw2.InformedSearch("UCS", "maze1.txt"))
([(0, 0), (1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4,
   5), (5, 5), (5, 6), (5, 7), (6, 7), (7, 7), (7, 8), (7, 9), (7, 10), (7, 11)
   , (7, 12), (7, 13), (7, 14), (7, 15), (8, 15), (9, 15), (10, 15), (11, 15),
   (12, 15), (13, 15), (14, 15), (15, 15), (16, 15), (17, 15), (17, 16), (17,
   17), (18, 17), (19, 17), (19, 18), (19, 19), (19, 20)], [(0, 0), (1, 0), (0,
    1), (1, 1), ...], 39, 39)
```

The optimal action sequence is represented as an array each element of which defines the next location coordinate to visit by starting from the initial maze location (both start and target coordinates should be included.). The processed search tree nodes are the location coordinates traversed during the search and shown partially (due to their number). The last two numerical values are the depth and cost values of the solution, respectively.

5 Regulations

1. **Programming Language:** You must code your program in Python 3. Since there are multiple versions and each new version adds a new feature to the language, in order to concur on a specific version, please make sure that your implementation runs on İnek machines.
2. **Implementation:** You have to code your program by only using the functions in the standard module of python. Namely, you **cannot** import any module in your program.
3. **Late Submission:** No late submission is allowed. Since we have a strict policy on submissions of homework in order to be able to attend the final exam, please pay close attention to the deadlines.
4. **Cheating: We have zero-tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from the internet included) will be punished according to the university regulations.
5. **Discussion:** You must follow ODTUClass for discussions and possible updates on a daily basis. If think that your question concerns everyone, please ask them on ODTUClass.
6. **Evaluation:** Your program will be evaluated automatically using the "black-box" technique so make sure to obey the specifications. A reasonable timeout will be applied according to the complexity of test cases. This is not about the code efficiency, its only purpose is avoiding infinite loops due to an erroneous code. In case your implementation does not conform to expected outputs for solutions, it will be inspected manually so commenting will be crucial for grading.

6 Submission

Submission will be done via OdtuClass system. You should upload a **single** python file named in the format **hw2_e<your-student-id>.py** (i.e. hw2_e1234567.py).

7 References

- For the `GRAPH-SEARCH` algorithm, you can refer to the lecture's textbook.
- Announcements Page
- Discussions Page