EECS 492 - Homework 1 (Winter 2024)

Due: January 31, 2024 at 23:59

Logistics

Submission Guidelines

You must work alone for the conceptual problems, though you may discuss ideas and high-level concepts with classmates. You may optionally work with a partner on the coding sections, though you can work alone if you want.

The due date for this assignment is **January 31, 2024 at 23:59**. The submission is tracked using Gradescope. The written section and programming section are graded and penalized separately. Also note that any change you make to the homework already submitted on Gradescope counts as a resubmission. If you make any changes to the assignment after the due date has passed you will be assigned a late penalty based on the number of days that have passed. For example, if you edit an assignment on February 5 and it was due on February 2 you will be assigned a 30% penalty (10% per day). This is non-negotiable.

You must submit the following files to Gradescope (note that there are two separate Gradescope submissions):

1. A (legible) PDF containing the solutions to the written section. You can write your solutions by hand, use a tablet, or typeset your solutions in LaTeX. We only ask that you make it easy to read and not too verbose so that graders don't struggle to understand your writing. You must submit the written section to the Gradescope individually.
2. The completed `Agent.py` file to the programming section. If you work with a partner, you must put both people's names on the Gradescope submission.

Late Day Tokens

To accommodate for coinciding deadlines you may have from other courses, or personal unforeseen events such as sickness, each person has 3 slip day tokens. Each token provides an automatic extension of 1 calendar day—no questions asked.

1. Each late day token covers the whole homework (both written part and programming part) for extension of 1 calendar day. For example, if you submitted written part 1 day late and programming part 2 days late. Then, if you use 1 late day token, you don't have any penalty on the written part, but will have 10% penalty on the programming part.
2. Each homework has a Gradescope submission for the late day token, in addition to the written part and programming part.

3. Late days are rounded up to the nearest integer. For example, a submission that is 4 hours late will count as one day late.
4. Extreme circumstances, like medical emergencies, etc.: In such cases, additional, no-penalty extensions will be granted. Contact your section faculty instructor with some written documentation (like doctor's note).
5. Emails requesting exceptions for other reasons will be ignored with no reply.
6. Homework turned in after three days will not be accepted.

Grading Policy

You have one week to submit a regrade request once the grades for your homework are out. We will provide solutions, along with the exact grading rubric used.

Collaboration

Collaborating with people is encouraged, but plagiarism is strictly prohibited. As mentioned in **Submission Guidelines**, you must work alone for the conceptual problems, though you may discuss ideas and high-level concepts with classmates. You may optionally work with a partner on the coding sections, though you can work alone if you want.

Generative AI

Learning how to use AI functions such as ChatGPT is important for all of us. Used properly, ChatGPT can enhance our work; used improperly, it can border on plagiarism. You are allowed to use ChatGPT or similar only for conceptual questions, and **it is strictly forbidden for programming**. If you have used ChatGPT on anything you submit for EECS 492, please include an explanation as to:

- State which model you used
- Include the prompt(s) that you gave the model
- Explain your reasoning as to why you agree (or disagree) with the response
- Mention any incorrect responses or erroneous behavior that you may have seen

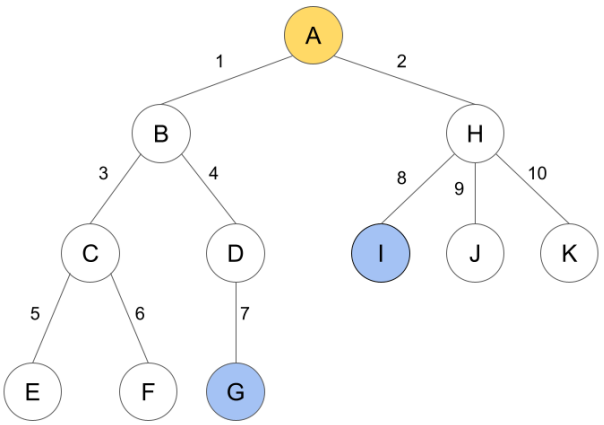
1 Written [50 Points]

1.1 Hands-On Search [18 Points]

Consider the search tree shown in Figure 1a, including nodes A through K. The state of node A is our initial state (shown in yellow). Nodes G and I share the same state and are valid goals (shown in blue). The label by an edge is the cost associated with the path between the edge's vertices.

Assume that **when a node is expanded, its children are added to the frontier in alphabetical order**. For instance, when node A is expanded, node B will be added to the frontier first, and then node H. If the last element is popped, the node popped would be node H.
For frontiers that use a priority queue, nodes with the same priority (path-cost, heuristic value, or the sum of path-cost and heuristic value) will be popped in alphabetical order. For instance,

if node B and node C have the same priority, node B is popped first, followed by node C.



(a) The Search Tree, with Path Costs

n	$h(n)$
A	10
B	11
C	14
D	7
E	12
F	9
G	0
H	8
I	0
J	3
K	5

Figure 1: Heuristic Values for the states

An admissible heuristic function is provided to you in Figure 1, which gives the value of the heuristic function for each node's state. For the following search methods, list the order in which nodes were expanded from the frontier (i.e., all nodes that you called `EXPAND(node)` on) from the start till the goal is reached.

(a) Breadth-First Search (3 Points)

(b) Depth-First Search. Goal test after popping from the frontier and before calling Expand(node). (3 Points)

(c) Iterative Deepening. Goal test after popping from the frontier and before calling Expand(node). Also, assume that the depth cutoff check is done immediately after goal testing and before Expand(node) is called. (3 Points)

(d) Uniform Cost Search (3 Points)

(e) Greedy Best-First Search using the heuristic function provided (3 Points)

(f) A-Star search using the heuristic function provided (3 Points)

1.2 Reservoir Search [10 Points]

You have two reservoirs: Reservoir A with a capacity of x units and Reservoir B with a capacity of y units. There is an infinite source of a liquid at your disposal. You may perform the following actions:

1. Completely fill either reservoir with the liquid.
2. Empty any of the reservoirs entirely.
3. Transfer the liquid from one reservoir to the other until the receiving reservoir is full or the original reservoir is empty.

Design a search algorithm (no heuristic should be used) that determines the **minimum number of actions** to use these two reservoirs to precisely measure c units. You can assume

that $c <= x + y$. You don't need to provide detailed pseudocode unless you want to - a general explanation of your solution is sufficient. Please make sure you explicitly state:

1. The pre-existing search algorithm you are using. (1 Point)
2. The data structure you are using. (1 Point)
3. The start and goal nodes. (2 Points)
4. The actions you used for generating successor states. Simply indicating actions in language is not enough, please use x (capacity of A), y (capacity of B) and other symbols as needed. (4 Points)
5. Describe how to save reached nodes help us in some way and justify how that helps, and edge case checking you might have to do. (2 Points)

1.3 Heuristics [10 Points]

For all these questions, assume that the cost of actions LEFT, RIGHT, UP, and DOWN are 1 each.

1. A knight is a chess piece that moves in an 'L' shape. One knight move is two squares vertically and one square horizontally, or two squares horizontally and one square vertically. You are given a chessboard with a start square and an end square, and your task is to find the minimum number of knight moves needed for a knight to get from the start square to the end square. Explain why Manhattan distance in a grid world like the previous case would not be a consistent heuristic for A-Star here, and develop your own consistent heuristic. Please justify why your proposed heuristic is consistent. (5 Points)

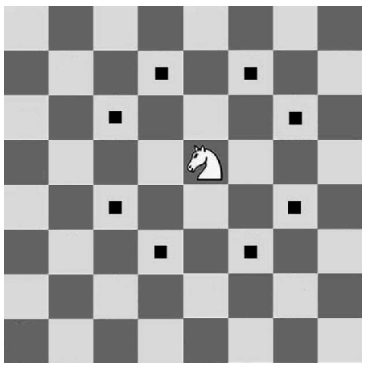


Figure 2: Valid Knight moves, indicated by the small black squares

2. Consider a grid world with a single goal, where the allowed moves are LEFT, RIGHT, UP, and DOWN (you are once again unable to move diagonally). Let $M(s)$ be the Manhattan distance from state s to the goal, and let $E(s)$ be the Euclidean distance from s to the goal. You also have the following function h :

Function *h(s)*:
    $r \leftarrow$ a random number from 1 to 10, inclusive of both;
    if $r \leq 6$ **then**
        return $M(s)$;
    end
    return $E(s)$;

Is h a consistent heuristic? Justify your answer. (5 Points)

1.4 Hill Climbing [12 Points]

Your friend guards each of their files with a 2-digit PIN code, but since they keep forgetting it, they decide to implement an AI agent that uses hill climbing to find it automatically. Each PIN comprises two integers, P_1 and P_2, each of which can take values from 0 to 9. They implement the following agent - the agent automatically submits a candidate PIN, and if it's correct, the file unlocks. If the guessed PIN is incorrect, the agent receives a score that measures how good the guess was. If the two digits entered are x and y, then the score returned is

$$h(x, y) = -(|x - P_1| + |y - P_2|)$$

The agent then uses a hill-climbing algorithm to attempt to find the PIN. The algorithm generates neighbors by creating all guesses that are different by one digit. In other words, if the current state is (x, y), the neighbors generated are $(x-1, y), (x+1, y), (x, y-1),$ and $(x, y+1)$. The algorithm doesn't generate any states outside the valid (0,0) to (9,9) range. In the event of a tie in the h value, the state with the lowest x value is selected.

(a) Assume that the initial guess for the PIN is (5,5) and that the actual PIN is (2,8). List the steps taken by the hill-climbing algorithm till it terminates. If it does not, explain why. (4 Points)

(b) Is this algorithm guaranteed to find the correct PIN code? Feel free to use a graphing tool like GeoGebra to help you solve this. (2 Points)

(c) Now assume that the algorithm is hard-coded to terminate after n steps. Does your answer to part (b) change if
   (i) n = 10 (2 Points)
   (ii) n = 20 (2 Points)

(d) It turns out that the code had a typo. The parenthesis around the first $-$ sign was missing. The heuristic is now

$$h(x, y) = -|x - P_1| + |y - P_2|$$

Is this version of his hill-climbing search guaranteed to find a solution? Explain your answer. (2 Points)

2 Programming [50 Points]

2.1 Setup

All programming assignments for this course use Python. If you're new to Python or not very experienced with it, please come to office hours; we're happy to help! We recommend also checking out the Python Cheatsheet. The topics you'd need from here are: Basics, Built-in Functions, Control Flow, Functions, Lists, Tuples, Dictionaries, Sets, OOP, and Virtual Environments. We have prepared a document Debugging Tips to help you debug your code. There's also a pinned Piazza post with more Python resources.
This assignment was designed for **Python 3.10** (though the newest version, 3.11 should also work out of the box).
We highly recommend creating a new virtual environment for the assignment. You can then simply install the requirements by running the command
`python -m pip install -r requirements.txt`

2.2 Problem Statement

In this programming section, you'll be coding an agent that searches a maze. More specifically, you'll be implementing the following search algorithms -

- BFS
- DFS
- UCS
- A-Star

Unlike lectures and previous examples, these mazes may (or may not) have multiple goals. As such, we'll be modifying our heuristic to be the Euclidean distance to the closest goal (See Q 1.4.1 in the Written Section)
Complete the `Agent.py` file and submit it to Gradescope. You will be evaluated using the Autograder on 10 test mazes on every algorithm. In `Agent.py`, you must complete the following functions

i. `heuristic_function` : returning the minimum Euclidean (straight line) distance between the AStar Node's coordinate and the end coordinates
ii. the three comparator functions for A-star nodes `__eq__`, `__lt__`, `__gt__`
iii. `goal_test`
iv. `expand_node`
v. `bfs`
vi. `dfs`
vii. `ucs`
viii. `astar`

For simplicity, please add nodes to the frontier directly instead of updating an existing node.
Please go through `maze.py` and `agent.py` before you begin! The files are annotated with TODO in every location where you need to add code and have multiple comments, tips, and recommendations for how to proceed. In addition, you will never need to modify `maze.py`.

Maze structure

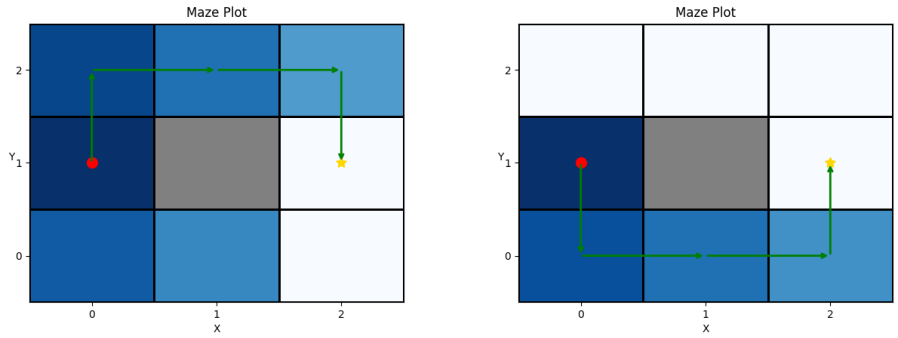
Each test case is a text file (saved as a `.test` file, but you can open it in any text editor). Here's what Test case 1 looks like:

```
3 3
1 1 1 1
...
A*B
...
```

The first line of input lists the width (w), and the height (h) of the maze.
The second input line lists the cost of going left, right, up, and down correspondingly. The next three lines delineate the elements in the grid. '`A`' denotes the starting point of the agent. '`B`' denotes a goal state. '`.`' denotes a spot where the agent can freely land, while '`*`' denotes a wall.

2.3 Testing locally

We've provided four test cases that you can use to test your implementation. See `LocalTest.py` for instructions on how to run it. That being said, we have not provided solutions for them. The LocalTest.py file will show you what the maze looks like and the path your agent found. You can solve the maze and determine if your algorithm works correctly.
Test cases 1,2 and 3 are identical on Gradescope. There are no hidden test cases - your score on Gradescope is the score you can expect to receive for this section. An example of what you should see after running `LocalTest.py` for Test1 on BFS and DFS can be seen in Figure 3. The red dot is the start position, the yellow star is the goal, and the green arrows indicate the path found by the algorithm. The grey cells indicate walls, while the cells are shaded based on how early on into the algorithm's execution they are expanded. The darker the shade of blue, the earlier it was expanded. White cells have not been expanded.



(a) BFS (b) DFS

Figure 3: Expected paths found by BFS and DFS on test case 1