University of Michigan
EECS 592 - Fall 2016

Homework 1 - Search
Assigned: Sept 16, 2016, Due: Sept 27, 2016 at 11:59 PM

For this assignment, you will be determining the attack strategy for a swarm of enemies in a tower defense game. In such a game, a swarm of enemies is seeking to destroy the base of the player. The creeps enter the game at a specific location, and are trying to reach the location of the enemy base. The player can defend themselves by building walls, traps, and towers that can block, hinder, and attack the creeps.

Screenshot from game Warzone Tower Defense

In our version, you are playing as a single enemy unit, and, given the player's defenses, must plan a route from the spawn location to the enemy base, while navigating through and around the various obstacles. The world consists of a grid of tiles and movement is divided into discrete time steps. On each time step, the unit can chose to move 1 square north, south, east, or west. There are the following obstacles:

Walls: The unit cannot move into a tile containing a wall.

Swamp: Some tiles are covered with swamp, these require 3 time steps to move through instead of 1. They do not block the unit's path, but do slow it down.

Towers: Towers can shoot at enemies in adjacent tiles, causing damage. Every time step the unit spends in a tile next to a tower (NSEW only, no diagonals) it takes a point of damage. Note that if the unit moves through a swamp adjacent to a tower, it would take 3 points of damage. Also note that towers are also obstacles, so the unit cannot move into a tile that contains a tower.

Programming Requirements

You are allowed to implement your solution to these problems in C++, Java, or Python. You are not allowed to use any external libraries or dependencies, and are limited to those libraries and classes that involve I/O and data-structures:

- C++: The containers in the standard template library
- Java: The collections in java.util
- Python: The collections in the python standard library

This is an individual project, your work must be your own. Do not share code or solutions with other students. You will upload your source code along with your results.

Submission: This homework is due at 11:59 PM on Tuesday, September 27. Submit your work to Canvas (https://umich.instructure.com). In the page for this course there is a tab for Assignments, under Homework 1 there will be a place to upload your source code and to enter your answers for the different questions. Please upload a single zip file containing all source code.

Part 1: Uninformed Search

For the first part of the assignment, you are only concerned with finding a path through the maze of walls to the goal (enemy base), and are not concerned with towers or damage. You will be given a map as a grid of tiles, each tile can either be *start, goal, open, wall, or swamp.* The outer edge of the map will always be a ring of walls.

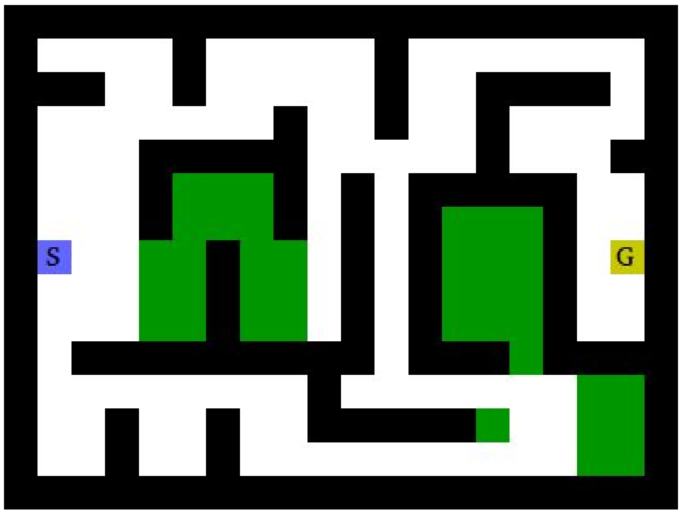
The map will be provided as a text file in the following format:
The first line will have 2 integers, the number of rows (R) and the number of columns (C).
The next R lines will contain C characters each, determining the grid as follows:

`'X'`	- wall
`' '`	- open tile (space character)
`'S'`	- start tile
`'G'`	- goal tile
`'*'`	- swamp
`'O'`	- tower

For example, on the next page is the file '`test_map1.txt`' and a picture of the world (dark green = swamp, light green = start, blue = goal). This is a smaller test map which you can use when debugging your program. All the maps for this assignment are available on the Canvas course website.

`test_map1.txt`

```
15 20
XXXXXXXXXXXXXXXXXXXX
X    X     X       X
XXX  X     X  XXXX X
X       X  X  X    X
X   XXXXX     X   XX
X   X***X X XXXXX  X
X   X***X X X***X  X
XS  **X** X X***X GX
X   **X** X X***X  X
X   **X** X X***X  X
X XXXXXXXXX XXX*XXXX
X        X       **X
X  X  X  XXXXX*  **X
X  X  X          **X
XXXXXXXXXXXXXXXXXXXX
```



Your task is to search for a path from the start to the goal on the map `large_map1.txt`. You will use three different types of uninformed search: Depth-First Search, Breadth-First Search, and Uniform-Cost Search. Your solution should follow the general form of the GRAPH-SEARCH algorithm from the textbook, figure 3.7:

```
**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure
    initialize the frontier using the initial state of *problem*
    ***initialize the explored set to be empty***
    **loop do**
        **if** the frontier is empty **then return** failure
        choose a leaf node and remove it from the frontier
        **if** the node contains a goal state **then return** the corresponding solution
        ***add the node to the explored set***
        expand the chosen node, adding the resulting nodes to the frontier
          ***only if not in the frontier or explored set***
```

Hint: Your implementation should be almost very similar for different searches, the main thing that should change is how the frontier is managed and what data structure is used.

For each search you must report the following:

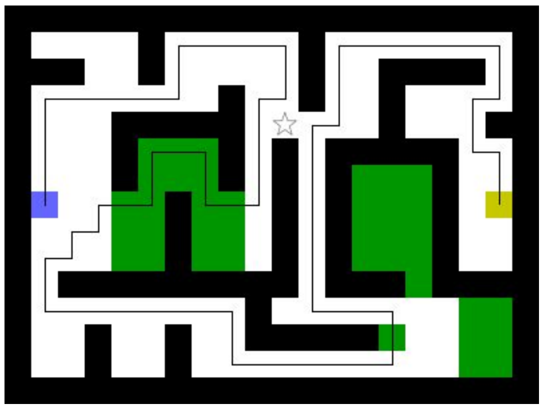
1. The sequence of actions that led to the goal. This should be as a string of letters NSEW (for example, NNEESENNNWWNEEE).
2. The length of the path in number of moves made.
3. The total cost of the path, which includes the cost of going through swamps (each move has a cost of 1 or 3, depending on whether it is moving out of a swamp).
4. The number of nodes removed from the frontier and examined during search.

IMPORTANT: In order to standardize results and ensure that your results match ours, it is crucial that you break ties in a specific order. If there are two nodes in the frontier with equal preference, choose the one with the lower row number first, and if they are in the same row choose the one with the lower column number first. This should result in the ordering North, West, East, South (the tile with row 0, column 0 is at the North-West corner).

Also make sure that you do not add a node to the frontier twice. Note the last 2 lines in the above algorithm. You should know which nodes have been added to the frontier at some point so that you do not add a duplicate node. Two nodes are equivalent if they contain the same state, for this problem they are equivalent if they represent the same tile.

1A: Depth First Search (8 points)

Implement depth-first search on the 1st large map and report the results. Remember that you want the successors of a node to be removed from the frontier in the order NWES. And make sure that you keep track of which tiles have been visited so that you don't add the same node more than once to the frontier.

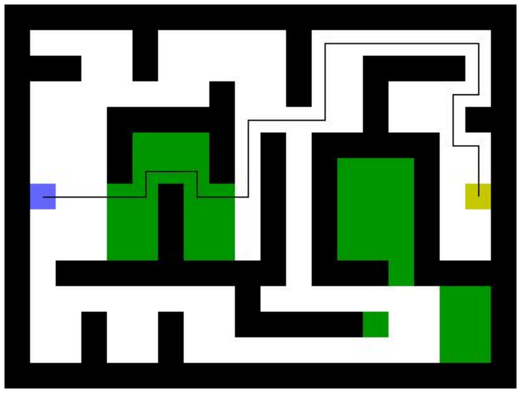


```
DFS Solution for test_map1.txt:

Path:
NNNNEEEEEENNEEEEESSWSSSSWWNNWWSSWWSWSW
SSEEEEEEEESSEEEEEENNWWWNNNNNNNENNNEEE
EEESSWSSESS

Path Length: 83
Path Cost: 103
# Nodes Examined: 97
```

Note: You may wonder why it doesn't go through the gap with the star. When it reaches the tile directly north of the star, it adds the tiles to the west and south to the frontier. Due to tie-breaking rules, it chooses the tile to the west. Later, it avoids the tile with the star because that node is already in the frontier.

1B: Breadth First Search (8 points)

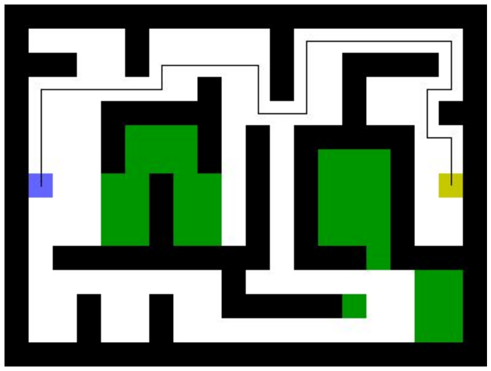
Implement breadth-first search on the 1st large map and report the results. Again, you want the successors of a node to be removed from the frontier in the order NWES.



```
BFS Solution for test_map1.txt:

Path:
EEEENEESEENNNEEENNNEEEEEEESSWSSESS

Path Length: 33
Path Cost: 47
# Nodes Examined: 160
```

1C: Uniform Cost Search (8 points)

Both the breadth-first and depth-first searches did not consider the actual costs involved when finding a path. Uniform cost search will always find the lowest cost path. Implement UCS to find the best path, taking into account the path costs. Remember that the cost of moving from one tile to another depends on the first tile's type, 1 if it is clear, and 3 if it is a swamp.

<table><tr><td></td><td>

```
UCS Solution for test_map1.txt:

Path:
NNNNEEEEENEEEESSEENNNEEEEEESSWSSESS

Path Length: 35
Path Cost: 35
# Nodes Examined: 156
```

</td></tr></table>

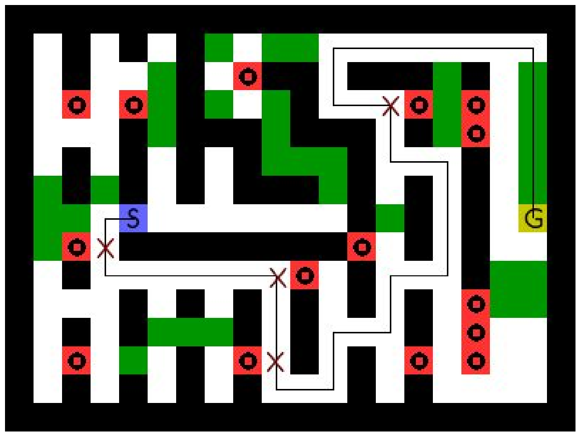
Part 2: Heuristic Search

Good News! You now know the location of the enemy base! You can use this knowledge when deciding which node to choose next. However, the player has added towers that will attack any adjacent enemy units. You must find the fastest route to the goal without being killed, the amount of health remaining doesn't matter. When the unit is on a tile, it will take damage from **each** adjacent tower (only the 4 adjacent tiles, not diagonal). It will take 1 point of damage per tower or 3 points if the tile is also a swamp. Towers are designated in the map file as an 'O' character.

The unit starts with **5 health**, if it receives 5 or more damage it dies. Your task is to find the shortest path from the spawn point to the goal while keeping the unit alive. Below is an example map (towers red squares with circles):
`test_map2.txt`

<table><tr><td>

```
15 20
XXXXXXXXXXXXXXXXXXXX
X X X X* **        X
X X  *X OXX XXX*X *X
X O O*X* *X   O*O *X
X   X*XXX*XXX X*O *X
X X X X X***X   X *X
X*X*X X XXX*X X X *X
X** S       X*X X GX
X*O XXXXXXXXO X X  X
X X       O X   X**X
X   X X X X X X O**X
X X X***X X   X O  X
X O * X O X X O O  X
X X X X X   X X    X
XXXXXXXXXXXXXXXXXXXX
```

</td><td>

(Note the unit takes 4 points of damage)</td></tr></table>

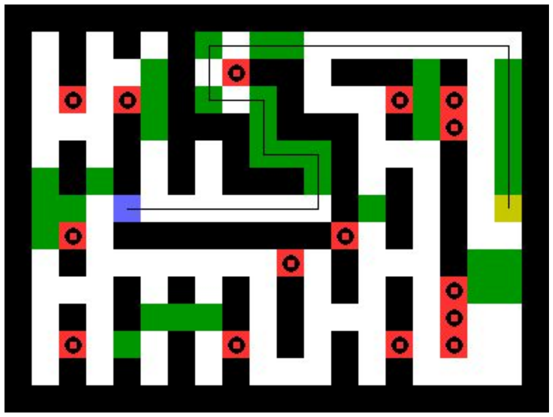
2A: Greedy Best-First Search (8 points)

Expand your state representation to include the health of the unit in addition to its location. Visiting a tile with 1 health vs 5 health is a very different situation. You will also have to modify the successor function so it includes damage taken and doesn't produce nodes where the unit has 0 health.

Then implement greedy best-first search to find a path to the goal. Remember that greedy best-first search chooses the node in the frontier that has the lowest heuristic cost, and ignores the path cost so far. Use the Manhattan distance as the heuristic (# rows + # columns to the goal). As before, it is vital you do not expand the same node twice, and that you break ties using first row, then column (NWES), **then health** (for two nodes on the same tile, choose the one with the higher health).

As with part 1, record the following 4 results for running greedy search on the map `large_map2.txt:`

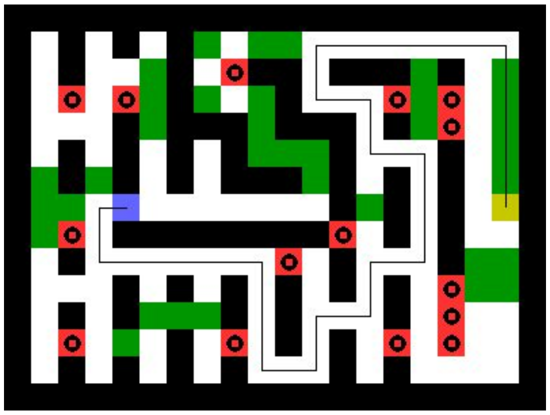
1. The sequence of actions that led to the goal as a string of letters NSEW.
2. The length of the path in number of moves made.
3. The total cost of the path.
4. The number of nodes removed from the frontier and examined during search.



```
Greedy Solution for test_map2.txt:

Path:
EEEEEEENNWWNNWWNNEEEEEEEEEEEESSSSSS

Path Length: 34
Path Cost: 64
# Nodes Examined: 140
```

2B: A* Search (8 points)

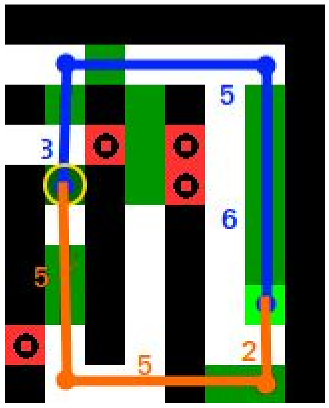
Now implement A* search using the Manhattan Distance heuristic as before. A* chooses the node in the frontier with the lowest total cost = path cost + heuristic cost. Note that it is now possible to generate a successor node that is already in the frontier but that has a lower total cost. So you will have to have a way to change that node's priority in the frontier. Record the same 4 results as in part 2A on the `large_map2.txt` map.

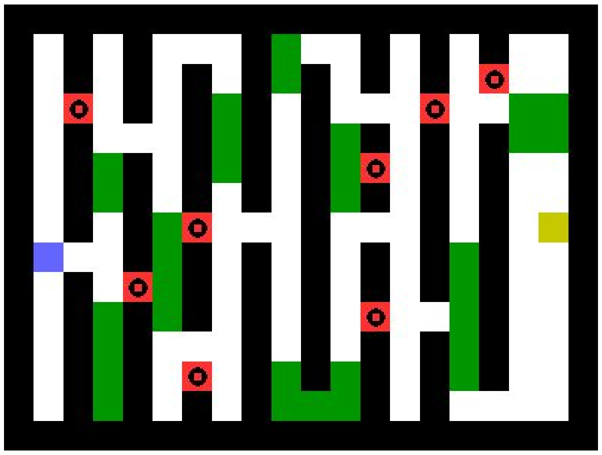
	`A* Solution for test_map2.txt with Manhattan Distance heuristic:`
	
	`Path:`
	`WSSEEEEEEESSSSEENNEENNEENNNNWWNNWWNNEEEEEEESSSSSS`
	`Path Length: 48`
	`Path Cost: 58`
	`# Nodes Examined: 384`

2C: Different Heuristics (10 points)

Here we consider how different heuristics can impact the number of nodes examined during the search. You are to implement these 4 heuristics and use them with A* search on the `large_map2.txt` map. For each heuristic, record the number of number of nodes examined.

1. Zero Heuristic: h(state, goal) = 0
   Note that this makes the search equivalent to a uniform cost search.
2. Manhattan Distance: h(state, goal) = | goal.row - state.row | + | goal.col - state.col |
   Same as in parts 2A and 2B. This heuristic just looks at what the lowest cost path would be if there were no obstacles.
3. Move Horizontally: ¼ * | goal.row - state.row | + | goal.col - state.col |
   Note that the goal is always in the eastern-most column, and the walls are arranged vertically. This modifies the manhattan distance so that horizontal distance is more important than vertical distance.
4. Wall: min(north path length, south path length)
   Determine two paths and return the length of the shorter one. These paths will try to find a way through the column to the east of the tile. Find the nearest open space in the column to the east by going north, then add the Manhattan distance from there to the goal. Do the same going south. Only use the path lengths (number of steps), not actual path costs or tower damage. If a path going north or south is blocked, do not consider it. If the space directly to the east is open, both paths are blocked, or you are in the easternmost column, just return the Manhattan distance.

	Example of the wall heuristic. The northern path has a opening 3 tiles north, then it is 11 tiles from there to the goal (total = 14) The southern path has an opening 5 tiles south, then it is 7 tiles from there to the goal (total = 12) The heuristic would return 12

	Heuristic solutions for test_map3.txt: # Nodes Examined: 1 Zero: 426 2 Manh: 412 3 East: 419 4 Wall: 404 Correct Wall Heuristic Values (#4): (row, col) -> value (10, 15) -> 12 (8, 9) -> 20 (11, 7) -> 15 (4, 4) -> 17 (12, 17) -> 6 (12, 18) -> 5