Searching Module

Classical Search

The problem we will be working with is **an agent finding a path from one position to a different position in a grid**. Each position in the grid is represented by a numeric value indicating the cost to move to that position from any neighbor. For example,

```
1 5 2 4 1 5 0 0 4 0
1 5 0 2 4 1 0 2 5 0
1 0 5 5 1 2 1 4 4 3
2 1 2 2 0 0 2 4 4 0
2 0 4 2 0 0 1 5 1 2
0 3 4 1 3 0 4 2 1 3
1 0 3 5 2 5 1 0 4 3
2 4 0 1 0 2 4 2 4 0
0 4 5 5 2 5 3 1 4 3
3 0 3 3 5 5 3 4 1 1
```

A value of 0 indicates that the location cannot be traversed. The agent can move in the four cardinal directions one position at a time. The agent **cannot** move diagonally.

Problem Formulation

The first task is to formulate this problem as a state space problem. You will need to define a state representation, and then describe all components of the problem formulation.

Uninformed Search

Implement the Breadth-First Search and Depth-First Search algorithms to solve this problem. A list of requirements is below.

1. Load the grid from a text file.
2. Allow for different start and goal pairs.
3. After running the algorithm to solve the grid, display the final path on the grid visually and write the new grid to a separate file.
4. Implement **both** Breadth First Search (BFS) and Depth First Search (DFS). **Do not use separate functions for these.**

Record and print to console:

1. The number of expanded nodes when running an algorithm
2. The path cost of the solution

Get code to read a text file into a 2d list, and code to write a path to a file:
https://pastebin.com/S5yzfJPf

You can test your code further by generating random grids and/or visualizing the grid and path. Functions to provide this functionality are at this link https://pastebin.com/tRWgXnAi - copy in the functions genGrid, labelTile, and visualizeGrid and use them as you like.

Informed Search

Implement the A* algorithm to solve this problem. You can use your uninformed search code as a base set of functions to implement A*, and then make whatever changes need to be made.

The requirements for uninformed search apply to the A* code too, except toggling between two algorithms.

Run the same tests as in uninformed search to generate many grids and compute many paths.

Local Search

Implement the Simulated Annealing algorithm to solve the N-queens problem. You may use the nqueens.py script in Canvas files that provides a Board class, number of attacking queens heuristic cost function, and getSuccessorStates function.

Guidelines:

1. Use a linear scheduling function: f(T) = T*decay_rate where decay_rate is a constant in range (0,1). **Note that this function is taking the current temperature as the input, not the time value "t".** At each iteration, pass in T to your scheduling function and return a new T value. Do not use the "t" value shown in the textbook.

2. Set initial T=100.

3. Terminate the algorithm if T is smaller than a threshold value, e.g., 0.00001.
    a. Note that this is not the only termination condition.

4. Try different pairs of decay_rate and the threshold for T to terminate the loop:
    a. decay_rate=0.9, T=0.000001
    b. decay_rate=0.75, T=0.0000001
    c. decay_rate=0.5, T=0.00000001

5. The pseudocode in the book assumes an objective function! We are using a cost function! Therefore, the line *next.VALUE - current.VALUE* should be reversed.

Create a loop to run 10 simulated annealing executions with board sizes 4, 8, and 16. Each run should have a random starting board of the same size. For each run, print the following information to console:

a) Initial state and its h-value
b) Final state and its h-value

For each board size, print out the average h-value of the final solutions over all 10 runs.

Refer to the last section of this document to see example output for your local search code.

Report

A report is required to summarize the work completed. You should include the following:

- Classical search
  - Description of how the problem is formulated as a state space problem
  - Description of uninformed search algorithms
  - Description of how you solved the problem with an uninformed search algorithm
    - Include a brief outline of what your functions do
      - Only include the important ones - applying actions to states, modifying the open list, and implementing the main algorithm loop.
  - Brief discussion of uninformed search results
    - Go to the link https://pastebin.com/tRWgXnAi and copy in all the code into your script. Run the "runTests" function to run a set of randomized tests and see the generated results. Pass in displayGrids=True to see some of the grids used.
  - Description of informed search algorithms
    - How are they different from uninformed search algorithms?
    - How are greedy and A* different?
  - Description of how your uninformed search code changed to implement A*
  - Brief discussion of informed search results
- Local search
  - Description of the problem
  - Description of local search algorithms
    - How are these different from classical search algorithms?
  - Description of how you solved the problem with a local search algorithm.
  - Brief description of the results
    - Include some of the boards
    - Discuss how the average h-value changed with different board sizes, **and why it changed**.

This maximum length is 7 pages single spaced with 12 point font.

Submission

Please submit the following:

1) Your report as a PDF.
2) Your uninformed search Python code as a .py file.
3) Your informed search Python code as a .py file.
4) Your local search Python code as a .py file.

Note that if you worked with a notebook, such as Jynpter, then you need to convert your notebook file to a python file. You can submit all the files individually or as a zip.

Local Search Output

Example final output: (your output should be readable and organized, but doesn't need to be exactly like this)

```
*********************************
Board size: 4
*********************************
######################################
Decay rate 0.9 T Threshold: 1e-06
######################################
Run 0
Initial board:
[1, 0, 0, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]
[0, 0, 0, 1]
h-value: 6
Final board h value: 2
[0, 0, 1, 0]
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
Run 1
Initial board:
[0, 0, 1, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]
[0, 1, 0, 0]
h-value: 6
Final board h value: 2
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]
[0, 1, 0, 0]
Run 2
Initial board:
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 1, 0, 0]
[1, 0, 0, 0]
```

h-value: 6
Final board h value: 0
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]
Run 3
Initial board:
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 0, 0, 1]
[0, 0, 1, 0]
h-value: 4
Final board h value: 0
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]
Run 4
Initial board:
[0, 0, 1, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
h-value: 6
Final board h value: 0
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]
Run 5
Initial board:
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]
[1, 0, 0, 0]
h-value: 4
Final board h value: 2
[0, 0, 0, 1]
[1, 0, 0, 0]

[0, 0, 1, 0]
[0, 1, 0, 0]
Run 6
Initial board:
[0, 1, 0, 0]
[0, 0, 1, 0]
[1, 0, 0, 0]
[1, 0, 0, 0]
h-value: 6
Final board h value: 0
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]
Run 7
Initial board:
[1, 0, 0, 0]
[0, 1, 0, 0]
[0, 1, 0, 0]
[0, 1, 0, 0]
h-value: 8
Final board h value: 2
[1, 0, 0, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]
Run 8
Initial board:
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[1, 0, 0, 0]
h-value: 2
Final board h value: 2
[0, 1, 0, 0]
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 0, 0, 1]
Run 9
Initial board:

[1, 0, 0, 0]
[0, 0, 1, 0]
[0, 1, 0, 0]
[0, 1, 0, 0]
h-value: 4
Final board h value: 2
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]
[0, 0, 1, 0]
Average h-cost of final solutions: 1.2
****
Board size: 8
****
##
Decay rate 0.9 T Threshold: 1e-06
##
…

Continue with the rest of the board sizes...