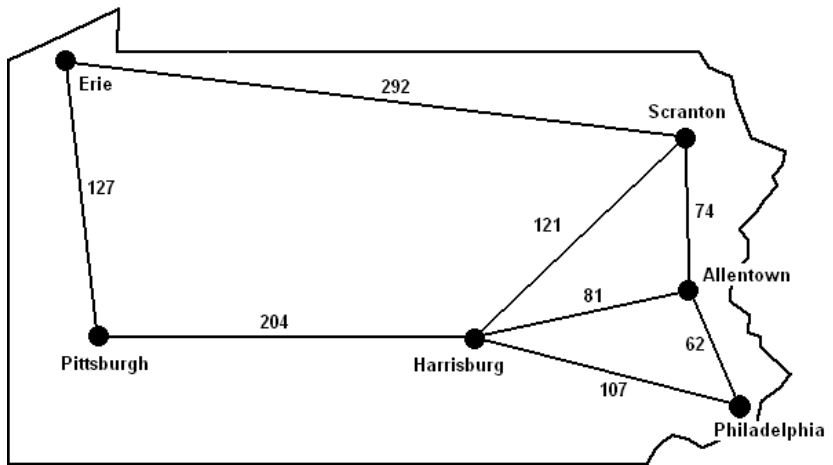


Homework #2: Chapters 3 and 5

The following exercises are due at the beginning of class on **Tuesday, February 21**. This assignment includes an extra-credit assignment worth up to an additional 20 points.

- [30 points] In a world where Google Maps does not exist, you are trying to find an efficient route from Erie to Philadelphia using the following map and distances. Develop one route using graph-based greedy best-first search and another route using graph-based A* search with the straight-line distance to the goal as your heuristic. For each algorithm, show your search tree and label each node with the order in which it is **expanded** (this may be different from the order in which it is **generated**). For the greedy search, show the $h(n)$ value for each node. For A*, show the $f(n)$, $g(n)$, and $h(n)$ values for each node. How do the two routes compare? What benefits and drawbacks of each search algorithm did you observe?

Straight-line Distance		
Erie	Pittsburgh	100
Erie	Harrisburg	225
Erie	Scranton	250
Erie	Allentown	300
Erie	Philadelphia	350
Pittsburgh	Harrisburg	180
Pittsburgh	Scranton	225
Pittsburgh	Allentown	200
Pittsburgh	Philadelphia	225
Harrisburg	Scranton	100
Harrisburg	Allentown	60
Harrisburg	Philadelphia	85
Scranton	Allentown	65
Scranton	Philadelphia	100
Allentown	Philadelphia	55



- [20 points] Use A* to solve the 8-puzzle with the initial and goal states shown below. Assume that your path cost is 1 per move and that your heuristic function is h_2 , the Manhattan distance of all tiles from their correct placement (note, the blank does not count as a tile). Show your search tree, complete with $f(n)$, $g(n)$ and $h(n)$ values for each node and label each node with the order in which it is expanded. Also show the current game board at each node in the tree. Once again, use the graph search version of the algorithm.

Initial State

	2	3
1	4	6
7	5	8

Goal State

1	2	3
4	5	6
7	8	

3. [40 points] This problem looks at playing the game tic-tac-toe. Assume that X is the MAX player. Let the utility of a win for X be 10, a loss for X be -10, and a draw be 0. There are two parts to this question, each using one of the two game boards given below:

Board1	Board2																		
<table border="1" style="border-collapse: collapse; width: 60px; height: 60px;"> <tr><td style="text-align: center;">O</td><td style="text-align: center;">X</td><td style="text-align: center;"> </td></tr> <tr><td style="text-align: center;">X</td><td style="text-align: center;">O</td><td style="text-align: center;">O</td></tr> <tr><td style="text-align: center;"> </td><td style="text-align: center;">X</td><td style="text-align: center;"> </td></tr> </table>	O	X		X	O	O		X		<table border="1" style="border-collapse: collapse; width: 60px; height: 60px;"> <tr><td style="text-align: center;">O</td><td style="text-align: center;"> </td><td style="text-align: center;"> </td></tr> <tr><td style="text-align: center;"> </td><td style="text-align: center;">X</td><td style="text-align: center;">O</td></tr> <tr><td style="text-align: center;"> </td><td style="text-align: center;"> </td><td style="text-align: center;">X</td></tr> </table>	O				X	O			X
O	X																		
X	O	O																	
	X																		
O																			
	X	O																	
		X																	

- a) Given the game board **Board1** above where it is X's turn to play next, show the entire game tree. Mark the utilities of each terminal state and use the minimax algorithm to calculate the optimal move.
- b) Given the game board **Board2** where it is X's turn to play next, show the game tree with a cut-off depth of two ply (i.e., stop after each player makes one move). Use the following evaluation function on all leaf nodes:

$$\text{Eval}(s) = 10X_3(s) + 3X_2(s) + X_1(s) - (10O_3(s) + 3O_2(s) + O_1(s))$$

where we define $X_n(s)$ as the number of rows, columns, and diagonals in state s with exactly n X's and no O's, and similarly define $O_n(s)$ as the number of rows, columns, and diagonals in state s with exactly n O's and no X's. Use the minimax algorithm to determine X's best move.

4. [10 points] Consider the following variant of the Hearts card game. Each player is dealt 5 cards, and both players can see each other's cards. The first player must play a card to lead the first "trick." If the other player has a card in the same suit they must play one of those cards. Otherwise, they can play any card. The player with the highest card in the suit that was led takes the trick and then leads the next trick. After all cards have been played, utilities are assigned as follows: -1 for each heart that you have taken in a trick, and an additional -13 if you have taken the queen of spades.
- a) In what ways is this game different from the two-player zero-sum games we have discussed in class?
- b) For a given deal, what is the maximum number of games (i.e., distinct sequence of moves) that can be played? Be sure to explain how you derived your answer.

Extra Credit (+20 points):

This optional exercise requires you to do some Java programming in order to conduct an experiment that compares uniform-cost, greedy best-first, and A* search. In order to do this, download and read the code I have made available from the course web site (under Additional Class Materials). This code implements all three search algorithms and provides two *abstract* classes: `SearchProblem` and `State`. In order to solve a particular problem, you only need to extend these two classes with details specific to your problem.

Your task is to extend the code to solve the following three configurations of the 8-puzzle, where the goal state for all three configurations is as specified to the far right.

Initial State #1	Initial State #2	Initial State #3	Goal State																																				
<table border="1"><tr><td>7</td><td>1</td><td>4</td></tr><tr><td>6</td><td>3</td><td>2</td></tr><tr><td></td><td>8</td><td>5</td></tr></table>	7	1	4	6	3	2		8	5	<table border="1"><tr><td>4</td><td>8</td><td>2</td></tr><tr><td>6</td><td>3</td><td>5</td></tr><tr><td>1</td><td></td><td>7</td></tr></table>	4	8	2	6	3	5	1		7	<table border="1"><tr><td>7</td><td>5</td><td>3</td></tr><tr><td>6</td><td></td><td>4</td></tr><tr><td>8</td><td>1</td><td>2</td></tr></table>	7	5	3	6		4	8	1	2	<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>8</td><td></td><td>4</td></tr><tr><td>7</td><td>6</td><td>5</td></tr></table>	1	2	3	8		4	7	6	5
7	1	4																																					
6	3	2																																					
	8	5																																					
4	8	2																																					
6	3	5																																					
1		7																																					
7	5	3																																					
6		4																																					
8	1	2																																					
1	2	3																																					
8		4																																					
7	6	5																																					

You will then run each of the three search algorithms on each puzzle, recording the path cost of the solution (if found), number of nodes expanded, number of nodes generated, and time to perform the search. To do this, you'll need to write a class that extends `State` and can record the state of the game (i.e., the current position of each tile). It is important that this class implements an `equals()` method that can be used to compare the current state to another state. You'll need to write a second class that extends `SearchProblem` and implements the four methods: `getInitialState()`, `goalTest()`, `getSuccessors()`, and `getHeuristicValue()`. You may want to include a constructor that allows you to initialize the class with different initial states. Note, `getSuccessors()` returns a list of `Successor` elements, where each `Successor` records the `State`, a string describing the action to reach it, and the step cost of executing that action. By including the step cost in the successor information, we can avoid providing a separate path cost function. For the heuristic, use the sum of the Manhattan distances of all tiles from their goal positions. Finally, you'll need to write a `main()` method that runs the tests.

After you collect your data, write an analysis of it. What appears to be the strengths and weaknesses of each algorithm based on your experiment? Did the experimental results agree with the theoretical properties of the algorithms discussed in class and in the book? What, if anything, surprised you?

Attach a hardcopy of your code, the output of your experiment, and your analysis to your homework submission. Submit your source (`.java`) and compiled (`.class`) files via e-mail to hflin@cse.lehigh.edu with subject line: "CSE 327: HW #2 Extra Credit".