EECS 492: Introduction to AI
Homework 2 (100 pts)

General Information

Due: 11:59 PM, October 3rd, 2016
Notes:

- For the written questions (Problem 1 Parts 2 and 4, Problems 2 and 3), put your answers in a pdf (with your name in the pdf) and submit to the corresponding assignment on Gradescope.
- For the coding questions (Problem 1 Parts 1 and 3), submit your code files firstname_lastname_search.py and firstname_lastname_8puzzle.py to the corresponding assignment on canvas, and also **append the code** to the end of your pdf.
- Late homework will be penalized 10% per day (each day starts at 11:59 PM on the due day).
- Homework turned in after three days **will not be accepted**.

Reminder: All solutions must be your own - no looking online or copying other students' solutions. For any solutions that require programming, please comment and format your code to make it readable. You should code your solutions in **Python 2**, and your code should be able to run on CAEN.

Problem 1: Implementing Search Algorithms (60 points)

You will implement a subset of the search algorithms covered in this class. The algorithms you will implement are the *graph* versions of: breadth-first search, depth-first search, iterative deepening search, uniform-cost search, A* search. You will also implement hill climbing search.

Problem Setup

There is a robot in a room. There is also a battery. The robot would like to move from its initial position (marked by a '2') to its battery (marked with a '3') in an optimal manner. The robot will use a search algorithm to get there.

The text grid that you process looks like this:

Figure 1: Example environment grid

```
2 0 1 1
1 1 1 1
0 0 0 1
```

```
1 1 1 1
3 1 1 0
```

Walls are marked with ‘0’ and free squares are marked with ‘1’. The robot’s initial position is given by the number ‘2’ and batteries are marked by the number ‘3’ (there is only one battery per environment).

The robot can move one grid square at a time. It can move north, east, south, or west. It cannot move outside of the grid, nor can it move into interior walls. **The path cost of moving north is 1, east is 2, south is 3, and west is 4.**

Part 1: Implementing the Algorithms (30 pts)

Fill in the following search functions in the provided file “search.py”: `breadth_first_search`, `depth_first_search`, `iterative_deepening_search`, `uniform_cost_search`, `a_star_search`. Use **graph search**, not tree search - so you must maintain a list of visited nodes. Do not change the arguments or the return values on these functions. To aid in your search, we define a priority queue (min heap) data structure, a node class, and an outline of the search structure. You do not need to follow our outline, or use our data structures - these are just there to help. As long as you implement the five search functions listed above, and the `extract_path`, `manhattan`, and `euclidean` functions, you are fine - just make sure to document your code structure well if you deviate from ours. You may create additional classes and helper functions, as long as they are within the file search.py.

Your code will be run with the following commands:

```
python simulator.py <search_module_name> <gridfile> <search
type> [heuristic]
```

“Search type” is one of ‘dfs’, ‘bfs’, ‘ucs’, ‘ids’, ‘astar’. So, for example, you could run

```
python simulator.py search grid1.txt dfs
```

If you run A* search, include the heuristic (either ‘manhattan’ or ‘euclidean’) as a command line argument:

```
python simulator.py search grid1.txt astar manhattan
```

Data Structures

For different algorithms, different data structures are needed. The following data structures, which are included in the provided code, may prove helpful:

- deque from the Python collections library. This can be used as a stack or a queue.
- Set from the Python sets library. Good for maintaining a set of visited objects.
- MinHeap: a class we define for you, built on top of the Python heapq library functionality. Implements a data structure that always maintains the minimum element of a collection of items. (You may ignore, modify, or extend our provided class).

Node ordering and tracking

Your code should track the total number of nodes generated during the search, as well as the maximum number of nodes stored in the frontier data structure at one time. You should also track the number of iterations run by the algorithm, and the depth and total cost of the goal node.

- Since the exact number of nodes and iterations depend on where you place the updates for these values in your loop, we will tolerate answers that differ from ours by a few iterations or nodes.
- Always insert successor nodes into data structures in the following order: north, east, south, west
- Don't forget the path costs of different successors: 1 for north, 2 for east, 3 for south, 4 for west.

Other Implementation tips

- If you need to make deep copies of objects, import the copy module and use `copy.deepcopy`. Otherwise, Python may make shallow copies, which can produce strange errors. You should be able to write your code without needing to make deep copies, however.
- If you write your own Python class, and want to be able to put it into a Set or MinHeap, you will need to overload built-in operators. For example, if you want an object to be comparable to others using the less-than (<) operator, implement the class function `def __lt__(self, other)`. If you want an object to be hashable, implement `def __hash__(self)`. If you want to be able to judge equality, implement `def __eq__(self, other)`. And so on. Search online if there is something else you want to overload for a class.
- A neat search implementation will have a node class and separate functions for check_goal and get_successors. Don't bunch everything together in one function - that's hard to read.
- Comment your code well and make it neat.

Grading

The code itself is worth 5 points per search function. The last 5 points come from your code running smoothly on our held-out test cases.

Part 2: Questions about Informed and Uninformed Search (12 pts)

1. (4 pts) For each search algorithm in Part 1, and each provided environment, fill in a table with the following columns:

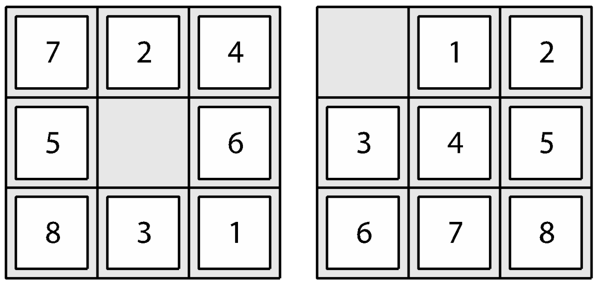
Grid #	**Total Nodes generated**	**Max nodes stored at once**	**Number of iterations**	**Depth of Goal**	**Cost of path to goal**	**Length of path to goal**

2. (3 pts) Which search algorithm
    a. Generated the fewest total nodes?
    b. Stored the fewest nodes at one time?
    c. Found a solution in the fewest number of iterations?

3. (5 pts) Compare the performance of the uninformed search algorithms vs the informed search algorithms. Which type did better in this environment? Why do you think that is?

Part 3: Implementing Local Search for the 8-Puzzle (10 pts)

In class we talked about constructing searches for the 8-puzzle. The 8-puzzle is a tile sliding game. There is an initial state in which the tiles are in a random configuration and a goal state in which the tiles are in a pre-specified configuration.



(Left) Initial state (Right) Goal state

You have decided that it will be interesting to investigate the 8-puzzle problem in the context of local search. What we are going to try to figure out in this problem is how well local search algorithms work in this domain. Use the sum of the Manhattan distances between current location and goal location for each tile as the heuristic value of a state.

Create a file firstname_lastname_8puzzle.py. When the command

```
python  firstname_lastname_8puzzle.py
```

is run, the code should generate 25000 random 8-puzzles, do hill-climbing search on each, and print out the *initial* states of the puzzles that were **solved** using hill-climbing search. There should be about 25-40 of them on average. Print out the puzzles in the format of the following example:

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

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

where '0' represents the empty slot, and the numbers are separated by spaces and newlines, with an empty line between separate puzzles. This question is worth 6 points for your code and 4 points for your code running smoothly.

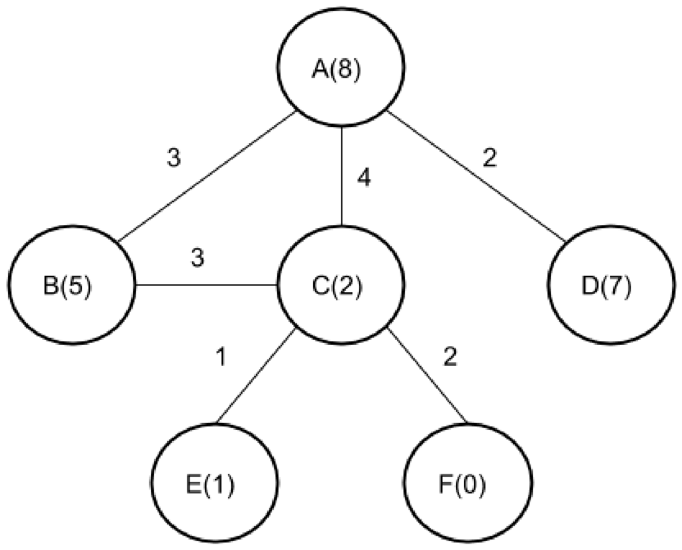
Part 4: Questions about Local Search for the 8-Puzzle (8 pts)

1. (4 pts) Does local search do well? Which aspects of the 8-puzzle make it difficult for local search to solve? Contrast this to a problem like 8-queens for which local search is better suited.

2. (4 pts)What advantages might simulated annealing offer in this case? What about random restart hill climbing?

Problem 2: General Questions about Search (30 pts)

Figure 2: A Search Graph



1. (6 pts) For each of the following search algorithms, list the nodes from Figure 2 in the order that they are examined by the algorithm. No need to show work. Start at node A and **examine** nodes in alphabetical order if there is any tie or ambiguity (so BFS would examine A, B, C, D, and DFS would examine A, B, etc. Technically this means that you push nodes into a queue in alphabetical order, and onto a stack in reverse alphabetical order.). Stop either when the algorithm finishes, or when 8 nodes have been examined. The numbers in parentheses by the letters are heuristic values, and the numbers by the edges are edge costs. *Note that the graph is undirected*.
   a. Breadth-first *tree* search
   b. Depth-first *tree* search
   c. Iterative Deepening *tree* Search
   d. Uniform Cost (graph) Search
   e. Greedy best-first (graph) search
   f. A* (graph) search

2. (8 pts) Consider the classic problem of Missionaries and Cannibals. There are 3 missionaries and 3 cannibals on one side of a river. To cross the river, they must use a single boat that can carry at most 2 people at a time (at least one person must be in the boat whenever it crosses the river). At no point during the crossing should

cannibals outnumber missionaries on one of the river banks - otherwise the cannibals will eat the missionaries. How can everyone get across the river safely?

   a. (2 pts) Devise a simple representation of the states of this problem and describe it. Full points will only be awarded for solutions that have at most 32 *representable* states.
   b. (1 pt) How many states are representable in your notation? (Show your calculation).
   c. (1 pt) What is the initial state in your representation?
   d. (1 pt) What is the goal state in your representation?
   e. (2 pts) Define a transition model, i.e., describe the potential successors of an arbitrary state.
   f. (1 pt) Solve the problem, i.e., provide list of states going from the initial state to the goal state, with valid transitions between adjacent states.

3. (4 pts) Prove that every consistent heuristic is admissible. (Hint: use mathematical induction).

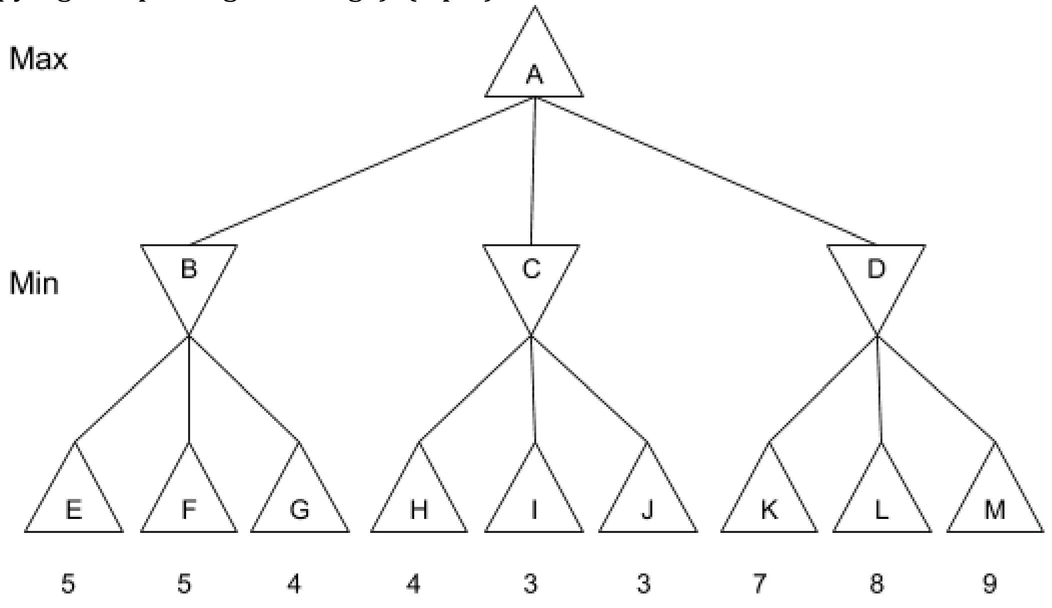
4. (8 pts) Local search in general
   a. (4 pts) Describe two possible advantages of local search over graph/tree search, and two possible disadvantages.
   b. (2 pts) Explain the difference between hill-climbing search and genetic search.
   c. (2 pts) Explain the difference between simulated annealing search and random restart search.

5. (4 pts) Real-world search-based agents need strategies for dealing with nondeterministic actions and partial observability (sections 4.3 and 4.4 in the book, respectively).
   a. (2 pts) Suppose a robot is trying to vertically balance a pole. The pole can be in one of three states: L (leaning left), R (leaning right), or V (vertical). The robot has two actions: NudgeLeft and NudgeRight, which nudge the pole left and right respectively. Whenever a pole is nudged, the result is nondeterministic: the pole may either stand up vertically or lean the way it was nudged (if it was already leaning in the way it was nudged, nothing happens). Write pseudocode for a conditional plan that solves this problem.

   b. (2 pts) Now suppose the robot cannot know the direction that the pole is initially leaning. All the robot can sense is whether or not the pole is vertical. Write pseudocode for an algorithm that will solve this partially observable problem (actions are still nondeterministic).

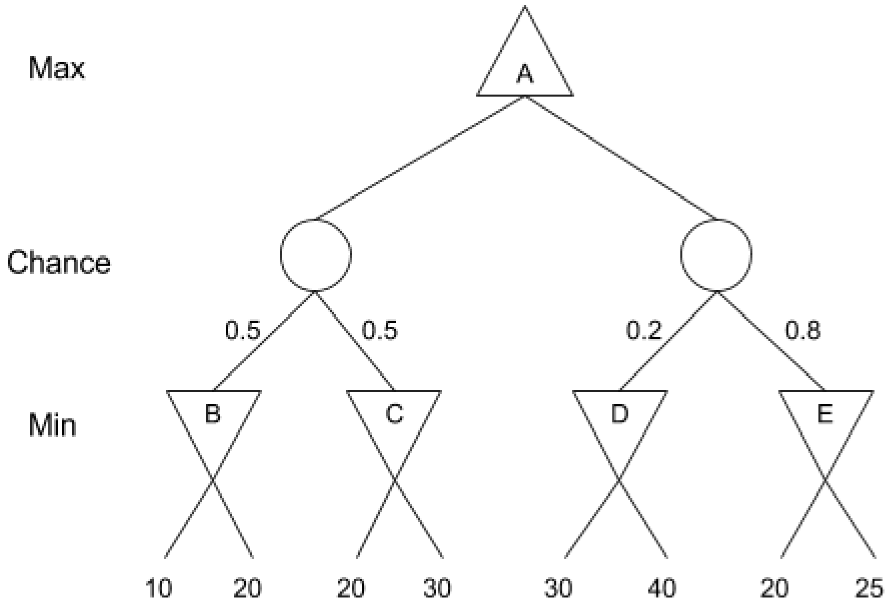
Problem 3: Adversarial Search (10 pts)

Apply minimax to the tree below (you may list the names and values of the nodes, rather than copying and pasting the image). (4 pts)



If we are using alpha-beta pruning, which node(s) can we skip (assuming we fill the tree in left to right)? (2 pts)

Apply minimax to the tree below with chance nodes. (4 pts)



Rules for Programming and Submission

- After you fill out search.py, **rename it** to firstname_lastname_search.py (of course, "firstname" and "lastname" should be replaced with your actual first and last name). Your local search file should be called firstname_lastname_8puzzle.py. Submit both files to Canvas under assignment 2.
- Submit a neat pdf of answers to the questions (*with your code pasted at the end*) to Gradescope. Be sure to identify the correct pages with the correct problems. **Put your name in the header on each page of the pdf.**
- Include a comment at the top of each of your code files with your name
- Your program must run from a command line on CAEN computers.
- Use good style and layout. Comment your code well.

Answer Format and Grading

- Partial or (possibly) full credit will be given to answers that are well-explained, or have work shown, even if they are different from our answers. (Or code that is well-written, even if it has errors).
- Points are broken down as follows:
    - Problem 1: 60 pts
        - Part 1: 30 pts
        - Part 2: 12 pts
        - Part 3: 10 pts
        - Part 4: 8 pts
    - Problem 2: 30 pts
        - Part 1: 6 pts
        - Part 2: 8 pts
        - Part 3: 4 pts
        - Part 4: 8 pts
        - Part 5: 4 pts
    - Problem 3: 10 pts