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

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:

Breadth-first

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
1	68	7	23	7	17	8
2	31	2	13	10	30	11
3	231	7	81	31	77	32
4	330	8	99	27	60	28
5	182	5	76	44	104	45

Depth-first

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
1	24	6	9	9	21	10

2	24	5	10	10	30	11
3	100	21	37	35	89	36
4	261	28	81	41	90	42
5	164	18	68	52	128	53

Iterative deepening

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
1	196	5	67	7	17	8
2	148	5	65	10	30	11
3	4225	21	1460	35	89	36
4	9192	28	2653	41	90	42
5	4346	17	1861	52	128	53

Uniform-cost

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
1	70	7	25	7	17	8
2	33	2	15	10	30	11
3	233	6	83	31	77	32
4	330	10	100	27	60	28
5	189	7	80	44	104	45

A* Manhattan

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

		once				
1	65	8	23	7	17	8
2	33	2	15	10	30	11
3	233	8	83	31	77	32
4	330	10	100	27	60	28
5	182	7	77	44	104	45

A* Euclidean

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
1	65	8	23	7	17	8
2	33	2	15	10	30	11
3	233	8	83	31	77	32
4	330	10	100	27	60	28
5	182	7	77	44	104	45

Grading: 1 pt per part

2. (3 pts) Which search algorithm
    a. Generated the fewest total nodes?
    Depth-first search

    b. Stored the fewest nodes at one time?
    There was not one algorithm that dominated in all grids, though breadth-first search stores the fewest on average, primarily because it tests on generation rather than expansion.

    c. Found a solution in the fewest number of iterations?
    Depth-first search

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?

The uninformed algorithms were BFS, DFS, IDS, and UCS. The informed algorithms were A* Manhattan and A* euclidean.

In terms of total nodes generated, the uninformed depth-first search beats the informed searches easily.

In terms of number of iterations, DFS and BFS outperform A*, probably because they goal test on generation rather than expansion. However, A* outperforms UCS in some cases, and does just as well in others, in this regard.

In terms of number of nodes stored at once, breadth-first search and UCS outperform A*.

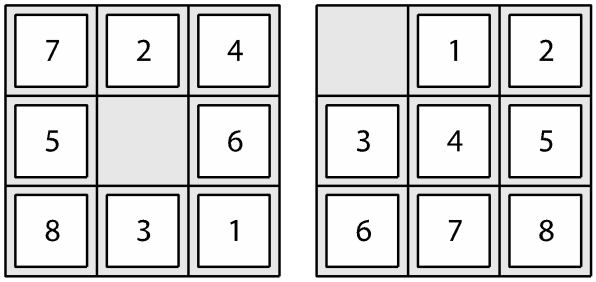
In terms of solution optimality, both UCS and A* are optimal, and the other uninformed searches are not.

Overall, the uninformed searches could be said to outperform the informed searches in this environment, though the margins were slim. This is because in this particular environment, path cost does not have a strong influence on the best path to the solution: a shortest depth solution will have to have some fixed number of vertical and horizontal steps, regardless of the cost of these steps. The order of node expansion also had an effect: BFS naturally expanded nodes in the same order as UCS would. Thus BFS happened to be optimal in this case, in addition to generally using fewer nodes due to goal testing at generation. The heuristic function did cause A* to use fewer iterations in some cases, but it was not useful enough in this environment to make a significant difference.

In other environments, like 8-puzzles, for example, the performance difference between A* search with a good heuristic and other searches is immense.

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.

   ```
   About half of the 25000 random puzzles are not solvable
   even in principle because they are odd permutations, so we
   ignore them. But even considering the expected 12500
   solvable puzzles, fewer than 100 are solvable with
   hill-climbing. This is less than a 1% success rate, so hill
   ```

climbing local search does not do well here.

In the 8-puzzle, only a small number of tiles can be moved at once, and those moves interfere with future moves of other tiles. There is also only one global solution. By contrast, the 8-queens puzzle allows pieces to move largely independently within columns, has far more possible successors per state, and has multiple solutions. This reduces the likelihood that no successors will be improvements and increases the chance that a random starting state will find a solution. So hill-climbing search works better with 8-queens.

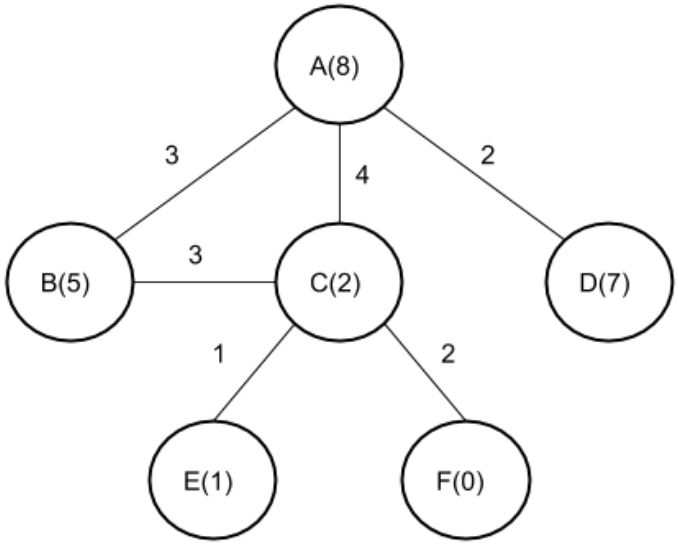
2. (4 pts)What advantages might simulated annealing offer in this case? What about random restart hill climbing?

Simulated annealing allows some transitions to states that are not improvements, which can move the search out of local minima. With the right temperature function, the solution will be eventually found with probability 1.

Random restart hill climbing does not offer any advantages, because it makes no sense to do here. You start out with a fixed initial state, so there can be no random perturbations.

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 `ABC|DACAB`
    b. Depth-first *tree* search `ABABABAB`
    c. Iterative Deepening *tree* Search `AABC|DABA`
    d. Uniform Cost (graph) Search `ADBCEF`
    e. Greedy best-first (graph) search `AC|FEBD`
    f. A* (graph) search `ACEF|BD`

    `Note: the vertical lines are where your search would stop if you assume that F is a goal state.`

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.
We represent each state as a tuple of integers <a,b,c> designating a) the number of missionaries on the left side of the river, b) the number of cannibals on the left side of the river, and c) where the boat is (0 for left side and 1 for right side)

b. (1 pt) How many states are representable in your notation? (Show your calculation).
4*4*2 = 32 possible states that are representable.

c. (1 pt) What is the initial state in your representation?
<3,3,0>

d. (1 pt) What is the goal state in your representation?
<0,0,1>

e. (2 pts) Define a transition model, i.e., describe the potential successors of an arbitrary state.
Given <a, b, c>, the potential successors are <a-1, b-1, 1>, <a-2, b, 1>, <a, b-2, 1>, <a-1, b, 1>, or <a, b-1, 1> if c is 0. If c is 1, then the possible successors are <a+1, b+1, 0>, <a+1, b, 0>, <a, b+1, 0>, <a+2, b, 0>, <a, b+2, 0>. Some of these successors are not possible from some states due to the quantities and constraints of the problem.

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, 3, 0>, <2, 2, 1>, <3, 2, 0>, <3, 0, 1>, <3, 1, 0>, <1, 1, 1>, <2, 2, 0>, <0, 2, 1>, <0, 3, 0>, <0, 1, 1>, <0, 2, 0>, <0, 0, 1>

3. (4 pts) Prove that every consistent heuristic is admissible. (Hint: use mathematical induction).
Let h be our heuristic function. Let C(n, n') denote the (minimum) cost of a direct edge between nodes n and n'. Let d(n,n') be the number of edges on a path of minimum cost from n to n'. Let P(n,n') be the minimum path cost between

n and n'.

(Base case, 0 distance away from a goal node) If g is any goal node, then h(g) = 0 by definition of a heuristic function. Since P(g, g) = 0, h is admissible at any goal node.

(Induction step: k distance to k+1 distance away) Next, suppose that h is admissible for all nodes n such that d(n, g) = k, for k >= 0 and the nearest goal node g to n. Consider a node n' such that d(n', g) = k+1 for the nearest goal g to n'. On its minimum path to the goal, it must be connected to some node n such that d(n, g) = k. By hypothesis, h(n') <= C(n', n) + h(n). But h(n) <= P(n, g) by hypothesis. So h(n') <= C(n', n)+P(n,g) = P(n', g). Thus h(n') is admissible.

By induction, h is admissible for all nodes connected to a goal by some path. For nodes that have no path to a goal, the true path cost is infinite, so any heuristic value is admissible for them.

Wrong solutions to watch out for: failing to realize that we are inducting over paths of *minimum cost*, NOT paths of minimum number of edges between nodes and the goal. You could be one edge away from the goal, but have a lower cost path that takes two edges.

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.
      Advantage 1: Local search can take up much less space than other algorithms - often O(1) space.
      Advantage 2: Local search can solve problems in very high dimensional spaces that can't feasibly be solved with other algorithms.
      Disadvantage 1: Local search is not an optimal search, and may get stuck in local optima
      Disadvantage 2: If neighboring solutions are very similar, local search may take a long time to move to an acceptable solution

   b. (2 pts) Explain the difference between hill-climbing search and genetic search.

Hill climbing search maintains one potential solution and perturbs it so that its utility increases to a local maximum. Genetic search maintains multiple potential solutions, and can combine them to generate new solutions, as well as randomly perturb them.

c. (2 pts) Explain the difference between simulated annealing search and random restart search.
Simulated annealing picks a random successor and transitions to the successor with some probability dependent on the gain or loss of utility, even if it is a worse solution. Random restart search performs hill climbing from a number of randomly chosen starting points, and maintains the best solution found so far.

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.

```
while pole is not vertical:
          if pole is leaning left:
               NudgeRight
          if pole is leaning right:
               NudgeLeft
```

   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).

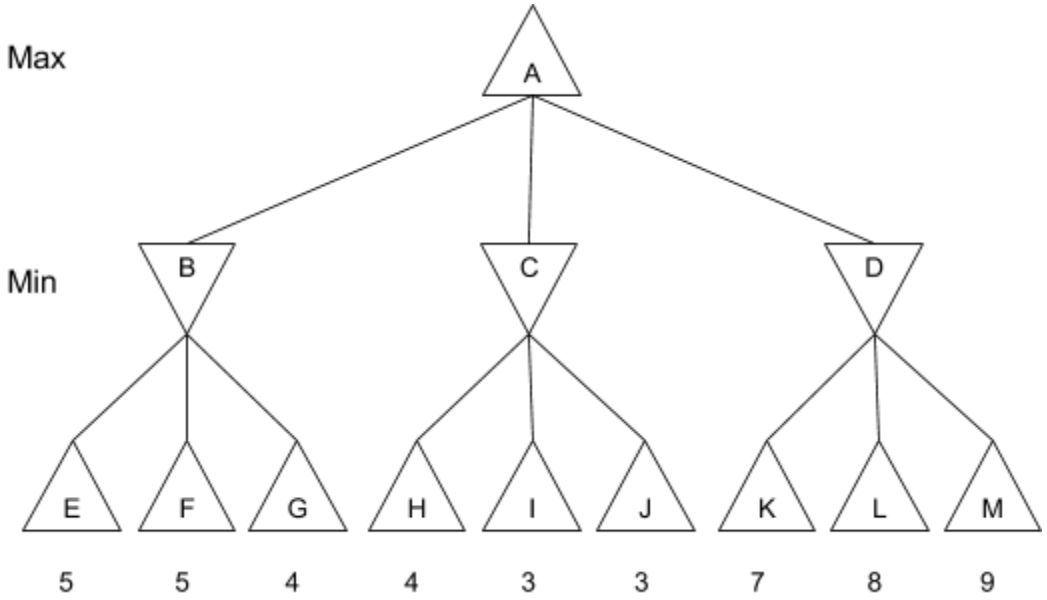
```
If pole is vertical:
     done
NudgeRight
While pole is not vertical:
     NudgeLeft
     If pole is vertical:
```

```
            break
        NudgeRight
```

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)

Answer: B: 4, C: 3, D:7, A: 7

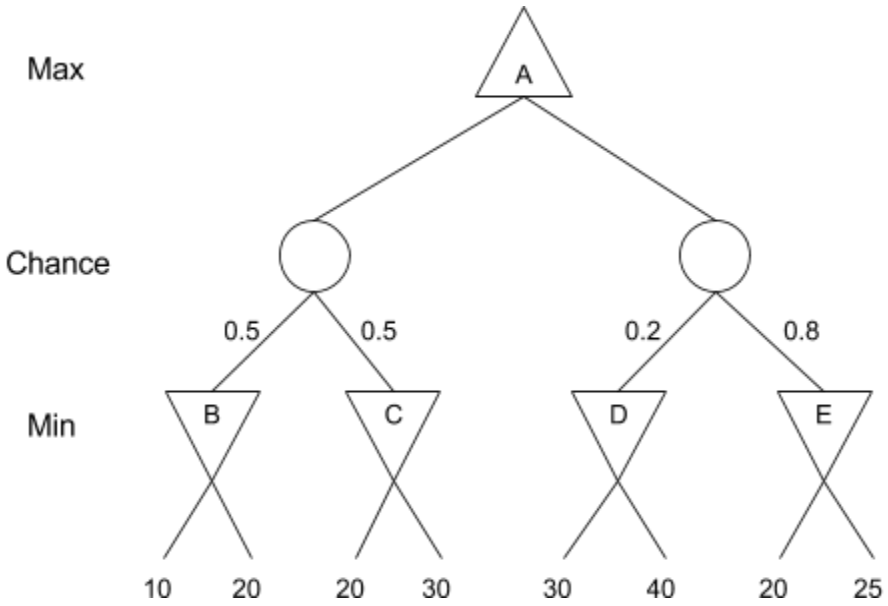


If we are using alpha-beta pruning, which node(s) can we skip (assuming we fill the tree in left to right)? (2 pts)

Answer: I and J will be skipped (since upon seeing H=4, the value of C can never exceed 4, which is no better than the value of B. Thus there is no need to traverse I and J, since they cannot improve the score of C).

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

Answer: B: 10, C: 20, D: 30, E: 20. Chance node over B and C: 0.5*10+0.5*20 = 15. Chance node over D and E: 0.2*30 + 0.8*20 = 22. Thus we have A:22.


Max
A
Chance
0.5
0.5
0.2
0.8
Min
B
C
D
E
10
20
20
30
30
40
20
25