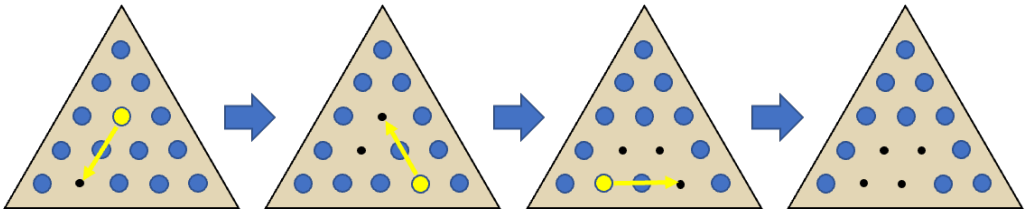
CS 380: Artificial Intelligence

Assignment 1

IQ Puzzle (20 pts)

The IQ Puzzle is a single-player game with pegs arranged in a triangular pattern:

At the start of the game, every space is filled with a peg except one. Then, at each turn, the player executes a "jump" where a peg jumps over another peg into an empty slot, thus removing the jumped-over peg from the board (as in checkers). For example, the figure below shows a sequence of three jumps, with each jump highlighted in yellow. The game eventually reaches a point where no more jumps are possible, and at that point, the player counts how many pegs are left on the board. The ultimate goal of the game is to have only one peg left on the board.



In this assignment, we give you the state representation for the IQ puzzle, and you take the next step and implement search algorithms to find a solution.

Implementation Setup

As part of the assignment, you have been given a file **iq.py** that provides an implementation of the state representation for the game, including computation of possible legal actions. We use a simple string as a base representation for a state:

```
Default      "O|OO|O-O|OOOO|OOOOO"
```

This string shows 5 rows of the triangle, where each row contains a combination of pegs (represented by 'O') and empty spaces (represented by '–'). The rows are separated by the vertical-line character '|'.

The code in **iq.py** parses this string for you and creates an internal representation with a 2-dimensional array, with helper functions to get and put characters in the cell locations. (Note that locations are represented as (x, y) coordinates, and the triangular game board is represented as a grid where we only use half the grid on one side of the diagonal.) The code also includes methods **is_goal()** to determine whether the current state is a solution state, **actions()** to return the possible legal actions from the current state, and **execute(action)** to execute an action within the current state. In addition, the code allows for command-line arguments, with some extra utility functions to print color states to the

terminal. You can try these commands as examples (all of which use the default state string above):

```
> python3 iq.py print

     O
    O O
   O - O
  O O O O
 O O O O O

> python3 iq.py goal
False
> python3 iq.py actions
jump(1,4,1,2)
jump(3,4,1,2)
```

Implementing an Agent

For this assignment, you will be coding an agent in a file **agent.py** that interacts with and solves a given game using various algorithms. This assignment, and all assignments for this course, will use **python3**. Please note that **you may only use built-in standard libraries (e.g., math, random, etc.); you may not use any external libraries or packages.** (If you have any questions at all about what is allowable, please email the instructor.)

Node and Agent Classes

As discussed in lecture, when searching through a state space, we typically use a node structure where each node includes both a state and some extra information used by the search—namely, a pointer to its parent node (to maintain path information for how we arrived at this state), and sometimes a state value (for informed search algorithms like A* where we assign a numeric value to a state). Your first task is to implement a class **Node** that captures this information. This code should be put in **agent.py**.

Next, also in **agent.py**, implement a new class **Agent** which will serve as the foundation for building the search algorithms below. Specifically, each of the search algorithms will be a method within the Agent class. Note that this separation between Agent and State classes allows us to use the same agent for multiple problem domains, and also allows us to use different agents on the same problem domain; we will not make much use of this flexibility in this assignment, but future assignments will exploit this architecture to a fuller extent.

Random Walk

Write a method for your Agent class, **random_walk(state, n)**, that does a random walk through the state space. Specifically, given a positive integer *N*, the random walk should generate all the possible next states from the current state, select one at random, then repeat until *N* states have been visited. Your code should build nodes (using your Node class) and, when the walk is completed, use the final node to generate and return the list of *N* states visited on the random walk.

Create a new file **search.py** and add a "**random**" command-line command to print the resulting sequence of states returned by **agent.random_walk()**. It should work on a state given on the command line, or, if there is no state argument, on the default state (see above). Please assume that *N*=8 here (this can be hard-coded into **search.py**). The code you've been given in **util.py** includes a helper function that will pretty-print a list of states: **util.pprint(states)**. Here are a couple examples:

```
> p search.py random

    o         o         o         o         o         o         o         o
   o o       o o       o o       o o       - o       - o       - o       - o
  o - o     o o o     o o o     o - o     - - o     o - o     o - o     o - o
 o o o o   o o - o   - - o o   - - - o   o - - o   - - - o   - - - o   - - - o
o o o o o o o o - o o o o - o o o o o o o o o o o - o o o o o - - o o o - o - -

> p search.py random '0|00|000|0000|00-00'

    o         o         o         o         o         o         o         o
   o o       o o       o o       o o       - o       o o       o o       o o
  o o o     o o -     o o o     o o o     o - o     - - o     - o o     o - -
 o o o o   o o - o   o o - -   o o - -   o o o -   - o o -   - - o -   - - o -
o o - o o o o o o o o o o - -o o o - - o o o - - o o o - - o o - - - o o - - - o
```

Of course, because this is a random walk, the code will do different things on each run, so your runs will likely not match those above.

Breadth-First Search, Depth-First Search, and A* Search

Write methods for your Agent class **bfs(state)**, **dfs(state)**, and **a_star(state, heuristic)** that implement the respective search algorithms.

First, note that all of these algorithms have a common base algorithm, described at a high level in the "Graph-Search" algorithm in Lecture 4. In essence, all of these algorithms:

- Maintain a list of open nodes, namely the nodes that have yet to be considered in the search process. The initial list of open nodes contains only the given start state.
- Maintain a list of closed nodes, namely the nodes that have already been considered. The initial list of closed nodes is empty, but as a node is considered, it is added onto this list.
- The algorithms differ with respect to how they select a node from the open-node list to consider: breadth-first search and A* pop off the first node in the list, whereas depth-first search pops off the last node in the list. In addition, A* gives each node a value (i.e., $f(n) = g(n) + h(n)$), and the list of open nodes should be kept sorted by value at each iteration.
- When a new node is considered, if that node is the goal, we are done and can return the solution node. Otherwise, we expand the node by adding its next states to the open-node list, and continue searching.

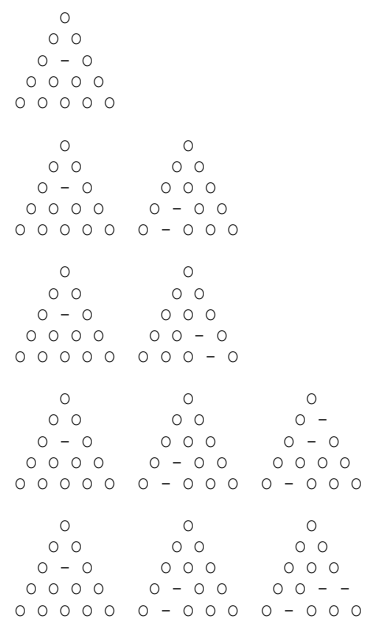
Write a method for your Agent class **_search()** with whatever arguments are needed to implement this base algorithm. Then, your implementations of **bfs()**, **dfs()**, and **astar()** should call this base method with appropriate arguments to differentiate how they work. (In other words, the real work of the search is being done in the base **_search()** method; the other methods are simply telling this method how to search.)

For A* search, note that there is one additional argument, the heuristic function, passed into the agent method **a_star(state, heuristic)**. The heuristic function $h(n)$ provides a value for a given state. For the game, you will need to choose and implement an *admissible* heuristic function $h(n)$, such that A* can reasonably estimate the minimum cost from a given state to the goal state. This heuristic function should be put into **search.py** so that it can be passed to the agent when running the A* algorithm.

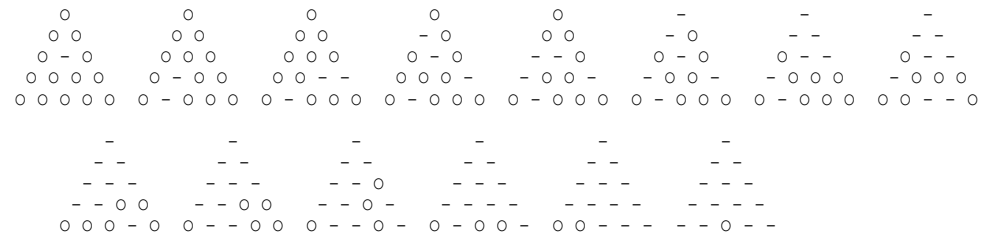
In addition, at each iteration of the search, print the path to the node being considered—the "path" here being the list of states from the initial state to the current state. You should again use the provided utility function **util.pprint(states)** to print the list of states.

Finally, add command-line commands "**bfs**", "**dfs**", and "**a_star**" to **search.py** such that the commands call their respective search algorithms. Here are some examples:

```
> python3 search.py bfs
```



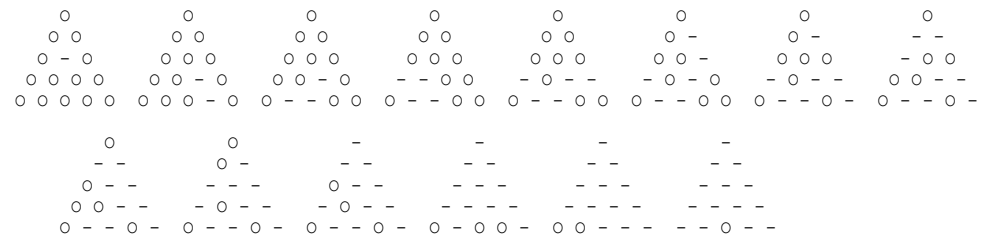
... and the rest of the paths searched,
ending with a solution and the number of paths explored ...



```
1651

> python3 search.py dfs
```

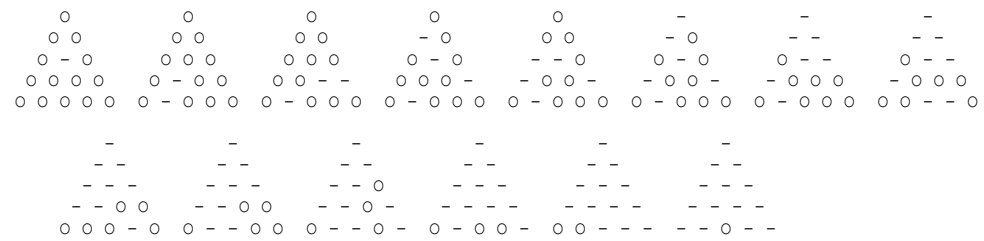
... ending with a solution and the number of paths explored ...



```
425

> python3 search.py a_star
```

... ending with a solution and the number of paths explored ...



```
1565
```

In this case, BFS took 1651 iterations before finding a solution. Because the given code in **iq.py** sorts actions into a particular order, your BFS results should be the same. Your DFS may be faster or slower. (Although DFS's solution may not be optimal in general, for this game, all paths to a solution are the same length.) Your A* result will depend on your own heuristic, but it should be better than BFS and still find the optimal solution path.

You will want to test your code on a variety of game configurations beyond the default initial state. If you're having trouble getting the search to work, you might try board configurations with only 2-3 pegs to check whether the algorithms perform the proper jumps for these states. As you're more confident with your coded algorithms, you could try a variety of initial states; note that any state with only one empty slot could be a valid initial state, so you could try different initial states to ensure that your code performs in a reasonable way.

Submission

For this assignment, you should submit the following Python files:

- **iq.py** – the original game code given to you for this assignment (you should not modify this file in any way!)
- **util.py** – the original utilities code given to you for this assignment (you should not modify this file in any way!)
- **agent.py** – your code for the search agent
- **search.py** – your code for running everything with command-line arguments

Please compress your files into a single ZIP file and submit the ZIP file electronically on Blackboard—please do not email your assignment to a TA or instructor.

Academic Honesty

Please remember that all material submitted for this assignment (and all assignments for this course) must be created by you, on your own, without help from anyone except the course TA(s) or instructor(s). Any material taken from outside sources must be appropriately cited.