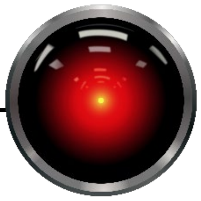

A2

Part I – Do this part on your own. Remember, that as always, you must write answers in your own words. Failure to do so is plagiarism.

Questions are 20 points each

1. You have been asked to develop an agent for a new US-based credit card company, *TarjetasRU$*. Your employer has purchased a large database from Experian and is proposing that you use income, credit history, and zip code for developing a credit approval agent. Do any of these fields pose ethical considerations that you should consider before create the agent?
2. We explained how the 8-queens problem could be solved more quickly by constraining the search space. Provide an example of another problem that can be solved more quickly through incremental search.
3. What is the difference between tree search and graph search? If you were solving an 8-puzzle, which one would you want to use and why?
4. An A* search expands 800 nodes and finds a solution at depth 7. What is the average branching factor? If the actual branching factor is 5, how many nodes would we have expanded in a worst case scenario and what percentage of those the A* search expand?

   If you solve this by hand, show your work. If you use a root finder, show your program. (See slide 52 for hints on setting up the formula. numpy has a polynomial root finder numpy.roots, Matlab has function roots).
5. The number of misplaced tiles heuristic for an 8-puzzle simply counts the number of tiles that are out of place. Suppose that our 8-puzzle has a single goal where the empty tile is in the lower right corner. Prove that the number of misplaced tiles is consistent.

Part II – Searching for love or power pills in all the wrong places… (100 points)

This assignment uses the same Berkely Pacman framework as your last assignment. This time, you will be teaching Pacman how to plan for a goal. While this assignment has Pacman exploring without having to worry about being chased by ghosts, you might start to think about how an agent could begin to execute a plan and change its plans when conditions change, this might make for a fun side project if you want to explore further.

The board configurations used in this game have the word maze in them , such as mediumMaze (file layouts/mediumMaze.lay). You will use the fully implemented searchAgents.SearchAgent to perform your first set of searches, and will implement code in two files:

- search.py – Your graph search code and g/h functions for breadth first search, depth first search, A* search and a Manhattan distance heuristic.
- explored.py – A class for managing an explored set.

The following command line arguments control the game:

--layout mazename (e.g. bigMaze)

--pacman agent (e.g, SearchAgent)

--agentArgs Takes a series of comma separated name=value pairs, you only need worry about the search_fun argument for this assignment. The prob argument can be useful if you want to go beyond the assignment, but the way we have constructed the problem, the code's argument for a heuristic function will be ignored.

search_fn=search_type (e.g., breadthFirstSearch)

prob=problem type (defaults to PositionSearchProblem)

Example invocation for a PositionSearchProblem:

```
python pacman.py --layout bigMaze --pacman SearchAgent \
--agentArgs search_fn=breadthFirstSearch
```

This causes the search agent to invoke the specified search_fn. Once the search_fn is called and a plan is returned, the agent executes the plan. You will see a maze that illustrates through shading which positions were considered. The shading shows the order in which these positions were explored, with the oldest positions being more saturated with color.

In **search.py** you must implement the following three classes:

- **BreadthFirstSearch**
- **DepthFirstSearch**
- **AStarSearch**

All three should consist of the following class methods with the identical signatures for each of three methods. If you have not used Python class methods, they are similar to any other object oriented programming language's class functions in that they are associated with a class, but not any specific instance of the class. In Python, these are indicated with a decorator @classmethod immediately proceeding the define (**def**). It is tradition to use the name **cls** as the first formal parameter instead of **self** for these functions.

- g(cls, node) – Returns the cost to SearchNode node as defined by the search algorithm being implemented.
- h(cls, node, problem) – Returns the cost to the goal as defined by the search algorithm being implemented. Argument problem contains the search problem and may be used to obtain information about the problem such as goal states.
- search(cls, problem) – Executes a graph search using the class methods g and h from the class. Returns a solution path or None if there is no solution to the problem.

As the Berkely code looks for functions rather than classes, three convenience functions have been added that call the class search methods. (e.g. breadthFirstSearch calls BreadthFirstSearch.search). **When starting Pacman, be sure to use the function calls as the search_fun argument, as the code has not been modified to directly find the classes.**

In addition, in search.py you must implement function **graph_search** which takes 5 arguments:

- problem – problem instance. Use problem.getSuccessors(state) to generate legal moves from a specified agent state.
- g – function handle that will be invoked on SearchNode instances. Hint: If you have not used function handles before, pass the function to graph_search without the (). Afterwards, parentheses can be used with the formal parameter name to invoke the function. Example: graph_search(…, g=BreadthFirstSearch.g, …) and within graph_search, use g(some_node).
- h – function handle that will be invoked on a SearchNode and heuristic function h.
- verbose and debug – These are boolean variables that you may wish to use to control output in your debugging. While it is mandatory to accept the variables, you are not required to implement them, but may find it useful to do so in your development.

search.py contains a SearchNode class that will be useful to you.

Implement class Explored in explored.py. A class skeleton is provided for you. Do not think you did something wrong if this class seems pretty trivial, a correct implementation does not require extensive work.

What to turn in:

- Part I: Submit your answers as a Word or PDF document to Canvas A1 Part I. Remember that everyone must work individually and that penalties will be applied as outlined in the syllabus should you plagiarize. If you have any concerns about what plagiarism means, take the SDSU plagiarism tutorial.
- Part II: Submit these to gradescope via the Canvas link:
  - Your solo or pair affidavit, see submitting work for details, you may include it in your source code at the top of the search.py file.
  - search.py
  - explored.py

  Do not submit the rest of the Pacman program. Your file must be interpretable by Python, as it will be uploaded to a virtual machine that will check your assignment for correct output. It is critical that you stick to the provided interfaces as these will be called directly to test your code. If you modify the interface, the tests will break, and you will not receive credit.

If you find yourself wanting more, play around with some of the other problem types defined in searchAgents (e.g. CornersProblem, ClosestDotSearchAgent, FoodSearchProblem, etc.). As this code is fairly generic, you can use it to solve other problems as well. As an example, see eightpuzzle.py. If you invoke it with "python eightpuzzle.py" and your code is implemented properly, it will solve an 8 puzzle using breadth first search, even though you designed your code for the Pacman world. Pretty cool 😊.