Prof. Dr. Eirini Ntoutsi
M.Sc. Arjun Roy & M.Sc. Philip Naumann
AG Künstliche Intelligenz und Maschinelles Lernen

17.05.2021

Artificial Intelligence
Project 1

Summer Semester 2021

1 Organization

- In order to pass you need to complete the questions Q1-Q7. Each question has at least one practical implementation part which requires your coding skills.
- Some questions also have additional sub-parts requiring intuitive answers identified by **IQ** and highlighted by color light blue. You need to write these answers in a document file.
- You may participate in this project as a group of 2 persons.
- Submit your solution no later than June 17th, 23:59.
- You can make re-submissions until the deadline, but the latest received submission will be graded only.
- Submit your solution as a zip file with the following name format: **"[YourNames seperated by commas]_AI21Project1"** containing your files: 'search.py', 'searchAgents.py', 'explain.pdf' (containing your explanation regarding the observations) files.

The term "fringe" means the same as frontier, both can be used synonymously. If you encounter any bugs or have any further question, please write to: arjun.roy@fu-berlin.de

Disclaimer: We are reusing a project from UC Berkeley[1]. The whole documentation can be found at https://inst.eecs.berkeley.edu/~cs188/fa20/project1/. The required code can is available at https://inst.eecs.berkeley.edu/~cs188/fa20/assets/files/search.zip

2 Introduction

In this project, your task is to solve various search problems in a given problem environment. The problem environment is a variation of the famous Pacman game[2]. Typically, the game of

[1] http://ai.berkeley.edu
[2] https://en.wikipedia.org/wiki/Pac-Man

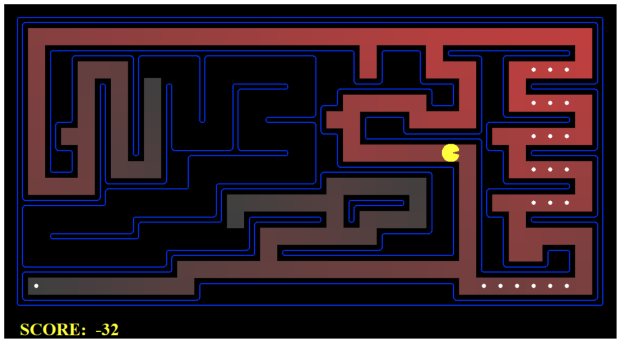

Figure 1: Example: Pacman game environment

Pacman is simple: The entity Pacman must navigate a maze and eat all the (small) food pellets in the maze without being eaten by the malicious patrolling ghosts. However, in this project you will not require to consider the complexity of patrolling ghosts in any of the solutions. Figure 1 demonstrates an example of a Pacman game environment you will require to solve. In each of the problem the Pacman is a rational agent, an entity that has goals or preferences and tries to perform a series of actions that yield the best/optimal expected outcome given these goals. The agent will be given a start state and a goal state to reach. The goal states will be one of the following:

- Eat a given food pallet (pathing): This problem will require you to only find an optimal path to the goal state using the information:
  - States: (x,y) locations
  - Actions: North, South, East, West
  - Successor: Update location only
  - Goal test: Is (x,y)=END?
- Find all corners (corners problem): This search problem finds paths through all four corners. You must maintain an array of booleans corresponding to each corner and whether or not it has been explored in the given state. The information to use:
  - States: (x,y) location, corner booleans
  - Actions: North, South, East, West
  - Successor: Update location and booleans
  - Goal test: Are all corner booleans True?
- Eat all dots (food search problem): Here you must maintain an array of booleans corresponding to each food pellet and whether or not it's been eaten in the given state. The information you will use:
  - States: (x,y) location, dot booleans
  - Actions: North, South, East, West

- Successor: Update location and booleans
- Goal test: Are all dot booleans false?

By considering the start state you will begin your search. Then exploring the state space using the successor function, iteratively compute successors of various states until a goal state is arrived, at which point you will have determined a path from the start state to the goal state (typically called a plan). Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will require to build general search algorithms using the defined problem functions and apply them to Pacman scenarios.

3 Project Structure & Autograding

You will require **Python 3.6** environment. If you have difficulties regarding the technical requirements, please refer to https://inst.eecs.berkeley.edu/~cs188/fa20/project0/.
To check your progress (only for self-assessment) for each question you may run the command "python autograder.py -q " followed by the question number. E.g to check your progress for Question 1 you need to run:

```
python autograder.py -q q1
```

Files you will edit:

- `search.py` | Where all of your search algorithms will reside.
- `searchAgents.py` | Where all of your search-based agents will reside.

Files you might want to look at:

- `pacman.py` | The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this project.
- `game.py` | The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
- `util.py` | Useful data structures for implementing search algorithms.

Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your project. If necessary, we will review individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. We trust you all to submit your own work only.

4 Welcome to Pacman

After downloading the code (search.zip), unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.
The simplest agent in `searchAgents.py` is called the `GoWestAgent`, which always goes West (a trivial reflex agent). This agent can occasionally win:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.
Soon, your agent will solve not only `tinyMaze`, but any maze you want.
Note that `pacman.py` supports a number of options that can each be expressed in a long way (e.g., `--layout`) or a short way (e.g., `-l`). You can see the list of all options and their default values via:

```
python pacman.py -h
```

Also, all of the commands that appear in this project also appear in `commands.txt`, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with `bash commands.txt`.

5 Question 1: Finding a Fixed Food Dot using Depth First Search

In `searchAgents.py`, you'll find a fully implemented `SearchAgent`, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented – that's your job. As you work through the following questions, you might find it useful to refer to the object glossary (the second to last tab in the navigation bar above).
First, test that the `SearchAgent` is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command above tells the `SearchAgent` to use `tinyMazeSearch` as its search algorithm, which is implemented in `search.py`. Pacman should navigate the maze successfully.
Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides and course book. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note: All of your search functions need to return a list of actions that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).
Important note: Make sure to use the `Stack`, `Queue` and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder.
Hint: Algorithms are similar: DFS, BFS, UCS, and Greedy Best First Search (this search function is not present in the `search.py` file and you need to create a function `GreedyBestFirstSearch` from scratch in the `search.py` file) and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need not be of this form to receive full credit).
Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. To make your algorithm complete, write the *graph search version* of DFS, which avoids expanding any already visited states.
Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent
```

```
python pacman.py -l mediumMaze -p SearchAgent
```

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration).

IQ: Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?
Hint: If you use a `Stack` as your data structure, the solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

6 Question 2: Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Does BFS find a least cost solution? If not, check your implementation.
Hint: If Pacman moves too slowly for you, try the option `--frameTime 0`.
Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

```
python eightpuzzle.py
```

7 Question 3: Varying the Cost Function

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider `mediumDottedMaze` and `mediumScaryMaze`.
By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.
Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`. We encourage you to look through util.py for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

```
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
```

```
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

After you have successfully completed the above section, we encourage you to write a search function `GreedyBestFirstSearch` in the `search.py` file which must follow the greedy best first search algorithm. Use the search function in all of the above three layouts and observe the difference.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=GreedyBestFirstSearch
```

```
python pacman.py -l mediumDottedMaze -p SearchAgent -a fn=GreedyBestFirstSearch
```

```
python pacman.py -l mediumScaryMaze -p SearchAgent -a fn=GreedyBestFirstSearch
```

IQ: What difference you observe from the other applied search strategies??explain your observation.
Note: You should get very low and very high path costs for the `StayEastSearchAgent` and `StayWestSearchAgent` respectively, due to their exponential cost functions (see `searchAgents.py` for details). The `autograder.py` file will not be able to assess your `GreedyBestFirstSearch` function.

8 Question 4: A* search

Implement A* graph search in the empty function `aStarSearch` in `search.py`. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The `nullHeuristic` heuristic function in `search.py` is a trivial example.
You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as `manhattanHeuristic` in `searchAgents.py`).

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly).

IQ: What happens on `openMaze` for the various search strategies? Replace `manhattanHeuristic` with `euclideanHeuristic`, what difference you notice? which heuristic is better here and why?

9 Question 5: Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.
In corner mazes, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first! Hint: the shortest path through `tinyCorners` takes 28 steps.
Note: Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.
Implement the `CornersProblem` search problem in `searchAgents.py`. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

```
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

To receive full credit, you need to define an abstract state representation that does not encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman `GameState` as a search state. Your code will be very, very slow if you do (and also wrong).
Hint: The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Our implementation of `breadthFirstSearch` expands just under 2000 search nodes on `mediumCorners`. However, heuristics (used with A* search) can reduce the amount of searching required.

10 Question 6: Corners Problem: Heuristic

Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.
Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Note: AStarCornersAgent is a shortcut for

```
-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be admissible, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be consistent, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most c.
Remember that admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!
Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).
Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to be accepted. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Your solution will be accepted if the number of nodes you will expand is at most 1600. **Describe your heuristic in a comment on top of the `cornersHeuristic` method.**

11 Question 7: Eating All The Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem:

`FoodSearchProblem` in `searchAgents.py` (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, A* with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to `testSearch` with no code change on your part (total cost of 7).

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Note: `AStarFoodSearchAgent` is a shortcut for:
`-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic`.
You should find that UCS starts to slow down even for the seemingly simple `tinySearch`. As a reference, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search nodes.
Note: Make sure to complete Question 4 before working on Question 7, because Question 7 builds upon your answer for Question 4.
Fill in `foodHeuristic` in `searchAgents.py` with a consistent heuristic for the `FoodSearchProblem`. Try your agent on the `trickySearch` board:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Our UCS agent finds the optimal solution in about 13 seconds, exploring over 16,000 nodes.
Any non-trivial non-negative consistent heuristic will be accepted. Make sure that your heuristic returns 0 at every goal state and never returns a negative value.
Remember: If your heuristic is inconsistent, you will receive no credit, so be careful! Can you solve `mediumSearch` in a short time? If so, we're either very, very impressed, or your heuristic is inconsistent.

IQ: Describe your heuristic used in the `foodHeuristic` method.