National University of Singapore
School of Computing
CS3243 Introduction to Artificial Intelligence

Project 1: Search

Issued: 26 August 2020 Due: 16 September 2020, 2359hrs

Overview

In this project , you will implement search algorithms and heuristics as part of the Pacman[1] game. Your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios. Project 1 is divided into 6 Tasks:

1. Implement **depth-first graph search (DFS)** algorithm to find a fixed food dot in the maze.
2. Implement **breadth-first graph search (BFS)** algorithm to find a fixed food dot in the maze.
3. Implement **uniform-cost graph search (UCS)** algorithm to find a fixed food dot in the maze.
4. Implement **A* graph search algorithm** to find a fixed food dot in the maze.
5. Formulate a **search problem** to find the shortest path through the maze that touches all four corners *(CornersProblem)*.
6. Implement a **non-trivial, consistent heuristic** for the *CornersProblem*.

This project is worth 10% of your module grade.

General Project Requirements

The general project requirements are as follows:

- Group size: **2 students**
- Submission Deadline: **16 September 2020** (Wednesday), **2359 hours** (local time)
- Submission Platform: **LumiNUS** > **CS3243** > **Projects** > **Project 1 Submission Folder**

[1]Note that this part of the project is based on ***UC Berkeley's CS188 Pac-Man Projects***. We have their permission to use this project. However, it should be noted that your solutions should **NOT** be distributed or published

- Submission Format: One standard (non-encrypted) **zip file**[1] containing only the necessary project files. In particular, it should contain exactly two `.py` files (`search.py` and `searchAgents.py`). Make only one submission per group.

As to the specific project requirements, you must complete and submit the following:

- Task 1 (DFS): Enter your DFS algorithm in `depthFirstSearch` method in `search.py`.
- Task 2 (BFS): Enter your DFS algorithm in `breathFirstSearch` method in `search.py`.
- Task 3 (UCS): Enter your UCS algorithm in `uniformCostSearch` method in `search.py`.
- Task 4 (A* Search): Enter your A* Search algorithm in `aStarSearch` method in `search.py`.
- Task 5 (A* Search): Complete the code in `CornersProblem` class in `searchAgents.py`.
- Task 6 (Heuristic): Enter the code in `cornersHeuristics` method in `searchAgents.py`.

Academic Integrity and Late Submissions

Do note that any material used does not originate from you (e.g., is taken from another source), should not be used directly. You should do up the solutions on your own. Failure to do so constitutes plagiarism. In any case, you may not share materials between groups.

For projects submitted beyond the submission deadline, there will be a 20% penalty for submitting after the deadline and within 24 hours, 50% penalty for submitting after 24 hours past the deadline but within 48 hours, and 100% penalty for submitting more than 48 hours after the deadline. For example, if you submit the project 30 hours after the deadline, and obtain 92%, a 50% penalty applies and you will only be awarded 46%.

Background: Pacman

Pacman is a maze chase game. Our character, Pacman, navigates in the grid-like maze, collecting food pellets whilst avoiding ghosts along the way. However, we have simplified the game of Pacman for you for project purposes.

- For **Task 1-4**, the objective is to find a fixed food dot located at any position of the grid in the maze during the search algorithms taught in class.

[1]Note that it is the responsibility of the students in the project group to ensure that this file may be accessed by conventional means.

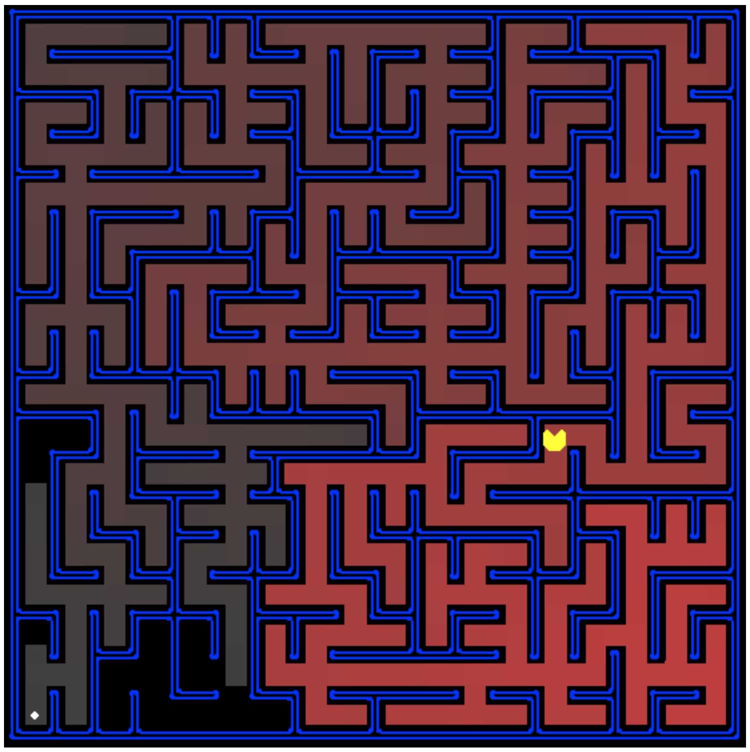

Figure 1: A layout of Pacman game with a food dot at the bottom left corner in the maze.

- For **Task 5**, the objective is to find the shortest path to the corners of the maze using A* Search.
- For **Task 6**, the objective is to come up with a consistent heuristic for the problem in **Task 5**.

Figure 1 shows an example of a configuration of the Pacman game.

Autograder

This project includes an autograder for all subtasks. You can run the following command to grade yourself:

```
python autograder.py
```

It can also be run for one particular task, such as question 2, by:

```
python autograder.py -q q2
```

For all tasks, the autograder grades your work out of 3 marks. We will re-scale it to the marks as reflected in the table at the end of this document.

File Specifications

The code for this project contains the following files:

Files you will **edit and submit**:	
`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 **read but not edit**:	
`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.
Supporting files you can **ignore**:	
`graphicsDisplay.py`	Graphics for Pacman
`graphicsUtils.py`	Support for Pacman graphics
`textDisplay.py`	ASCII graphics for Pacman
`ghostAgents.py`	Agents to control ghosts
`keyboardAgents.py`	Keyboard interfaces to control Pacman
`layout.py`	Code for reading layout files and storing their contents

The remaining files are not as important and need not be reviewed.

Submission Specifications

- For this part of the project, you will need to submit two files: `search.py` and `searchAgents.py`.
- Make only one submission (i.e., one set of two files) per group.
- Do not modify the file name.
- Place your two files in a folder named `CS3243_P1_XX`, where `XX` is your group number (if your group number is a single digit, prepend it with 0). For example, the folder `CS3243_P1_03` should contain the `search.py` and `searchAgents.py` files for Group 03.
- Points will be deducted for not following the naming convention, please follow it closely.

Getting Started

After downloading the code package, unzipping it and navigating to the folder directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Note: if you get error messages regarding `python-tk`, use your package manager to install `python-tk`, or see this page for more detailed instructions. 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` in the folder, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with bash `commands.txt`.

Finding a Fixed Food Dot using Search Algorithms (Task 1 - 4)

The objectives of this part of the project are to:

1. Gain exposure in implementing search algorithms taught in class
2. Learn to recognize the differences and usages of each search algorithms.
3. Learn to recognize the effects that the different search methods (uninformed and informed search) have on the performance and efficiency of the solutions.

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 need to refer to the glossary of objects in the code (at the end of this document). 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! 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).

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, 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).

Hint: Make sure to check out the `Stack`, `Queue` and `PriorityQueue` types provided to you in `util.py`!

Task 1 (2 points) Depth First Search

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). 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 244 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.

Task 2 (1 points) 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.

Your code should quickly find a solution for:

```
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`.

Task 3 (1 points) Uniform Cost Search

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):

Your code should quickly find a solution for:

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

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

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).

Task 4 (2 points) A* Search

Implement **A* graph search** in the `aStarSearch` function in `search.py`. A* Search 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). What happens on `openMaze` for the various search strategies?

Finding All the Corners (Task 5 - 6)

The objectives of this part of the project are to:

1. Formulate search problems and design heuristics to them
2. Learn to implement heuristics in A* Search Algorithms
3. Learn to recognize the importance of heuristics and the value of consistent heuristics.

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.

Task 5 (2 points) Formulating a Search Problem

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.

Task 6 (2 points) Consistent Heuristic

Implement a **non-trivial, consistent heuristic** for the `CornersProblem` in `cornersHeuristic` found in `searchAgents.py`.

```
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
```

Grading (by autograder – which will then be scaled back to out of 2 points):

- **0 point**: Inconsistent or trivial heuristics.
- **1 point**: Non-trivial consistent heuristics but expanded more than 1600 nodes.
- **2 points**: Non-trivial consistent heuristics that expanded fewer than 1600 nodes.

- **3 points**: Non-trivial consistent heuristics that expanded fewer than 1200 nodes.

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. Morevoer, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent.

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).

Additionally, any heuristic should always be non-negative, and should return a value of 0 at every goal state (technically this is a requirement for admissibility!). We will **deduct 1 point** for any heuristic that returns negative values, or doesn't behave properly at goal states.

Object Glossary

Here's a glossary of the key objects in the code base related to search problems, for your reference:

`SearchProblem (search.py)`

- A SearchProblem is an abstract object that represents the state space, successor function, costs, and goal state of a problem. You will interact with any SearchProblem only through the methods defined at the top of `search.py`

`PositionSearchProblem (searchAgents.py)`

- A specific type of SearchProblem that you will be working with — it corresponds to searching for a single pellet in a maze.

`CornersProblem (searchAgents.py)`

- A specific type of SearchProblem that you will define — it corresponds to searching for a path through all four corners of a maze.

Search Function

- A search function is a function which takes an instance of SearchProblem as a parameter, runs some algorithm, and returns a sequence of actions that lead to a goal. Example of search functions are `depthFirstSearch` and `breadthFirstSearch`, which you have to write. You are provided `tinyMazeSearch` which is a very bad search function that only works correctly on `tinyMaze`

`SearchAgent`

- `SearchAgent` is a class which implements an Agent (an object that interacts with the world) and does its planning through a search function. The `SearchAgent` first uses the search function provided to make a plan of actions to take to reach the goal state, and then executes the actions one at a time.

Marking Rubrics (Pacman - 10 marks)

Requirements (Marks Allocated)	Total Marks
• Correctly implement Depth First Search Algorithm (2) • Correctly implement Breadth First Search Algorithm (1) • Correctly implement Uniform Cost Search Algorithm (1) • Correctly implement A* Search Algorithm (2) • Correctly implement the `CornersProblem` search problem (2) • Correctly implement a non-trivial, consistent heuristic for the `CornersProblem` (2)	10

Autograder

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 score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

For all tasks, the autograder grades your work out of 3 marks. We will re-scale it to the marks as reflected above in the computation of the scores.

To run the autograder on the whole project, run the following command

```
python autograder.py
```

You can also run it on a particular question (for example, q2 – or Task 2):

```
python autograder.py -q q2
```

CodePost (Platform for Running Autograder)

We provide an alternative method for you to test your code on the autograder provided. The result should be no different from that when run locally by yourself, but just in case you experience some unknown issue that would prevent you from running the autograder locally.

Register for the CS3243 course on CodePost.io to gain access to the autograder.

Logging in for the first time - Invite link: `https://codepost.io/signup/join?code=JAKZPL57LX` (Remember to check your spam/junk mail for the activation email (it may go there). Contact the course staff if you don't receive it after 30 minutes.)

Subsequent access: `https://codepost.io/student/CS3243%20Introduction%20to%20AI/AY20%2F21%20Semester%201/`

1. click on "Upload assignment" under "Project 1" and upload the two files `search.py` and `searchAgents.py` (do NOT rename the files).
2. Refresh the page and select "View feedback" after the submission have been processed.
3. You may check your output returned by our autograder under `_tests.txt` option (this is also the default view).

You may submit your files as many times as you like. Note that this platform is run by an external organisation – we are not responsible for any downtime. We are merely using it as an alternative method for you to run the autograder on your work.