Introduction to AI - A.Y. 2024/2025

Project 1: Search

Fall '24

Abstract

This project belongs to the projects series from the Introduction to AI Course by Berkeley University. Further information can be found at the link `http://ai.berkeley.edu`.
Modules and data are stored in the `Project1Search.zip` folder from the course page.
At the end of this project, students complete the course topic on Search. They should be able to implement, analyze, and compare search algorithms in simple-to-complex problems.

1 Introduction

1.1 Problem & Purpose

Students implement **depth-first**, **breadth-first**, **uniform cost**, **A***, and **sub-optimal search algorithms** in this project. These algorithms are used to solve navigation and traveling salesman problems in the Pacman game. More precisely, a Pacman agent should find paths through a given maze to reach a particular location and collect food efficiently. The project is subdivided into *Tasks* that students have to refer to. They mainly consist of code implementations of those search algorithms discussed during the course. To submit their work, students should refer to Section 10.

1.2 Project Folder & Modules

The code for this project consists of several Python files, some of which have to be read and understood to complete the assignment and some of which can be ignored by the students. All the modules belong to the `Project1Search.zip` folder from the course webpage.

Files to be edited: [1]

- `search.py`: search algorithms will reside here.

[1] *DO NOT* change the other files

- `searchAgents.py`: search-based agents will reside here.

Other project files are briefly described in Appendix A.

Running the project: After downloading the code, unzipping it, and changing the directory, one can play a game of Pacman by typing the following at the command line:

`python pacman.py`

This is the basic command students could refer to to understand whether they correctly set the working folder.

Once this one doesn't generate any issues, two other commands have to be used to ensure the right functioning of the starting code:

> `python pacman.py --layout testMaze --pacman GoWestAgent`: Pacman game launch with a specific built-in Pacman agent, *GoWestAgent* (this is a trivial reflex agent, as it always goes West), with the map layout *testMaze*.
>
> `python pacman.py --layout tinyMaze --pacman GoWestAgent`: the same command as the previous one, but now with a different map layout. This example shows the *GoWestAgent* is not intelligent enough, as it gets stuck in one of the map's borders.

It is important to know that `pacman.py` supports other commands too. Appendix B summarizes all the commands used in this project. Inside the project folder, they are also listed in `commands.txt`.

Search algorithms are needed for Pacman to move autonomously in its environment.

2 Task 1 (3 pts): Finding a Fixed Food Dot using Depth First Search

A *SearchAgent* is fully implemented in `searchAgents.py`. It plans a path through Pacman's world and executes it step-by-step. To formulate a plan, **students have to implement the depth-first search (DFS) algorithms in the *depthFirstSearch* function in `search.py`**.

Before starting to develop the algorithm, students should test that *SearchAgent* is working correctly by running:

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

Pacman should navigate the maze successfully. In such a case, the student can proceed with the DFS search algorithm.

Note 1: To make the algorithm *complete*, students should write the graph search version of DFS, which avoids expanding any already visited states.

Note 2: A search node must contain not only a state but also the information necessary to reconstruct the path (plan) that gets to that state.

Note 3: All the search functions need to return a list of *actions* that will lead the agent from the state to the goal. These actions must be legal moves (i.e., valid directions and no moving through walls).

Note 4: Students should use the *Stack*, *Queue*, and *PriorityQueue* data structures provided in `util.py`.

Note 5: Each algorithm is very similar. DFS, BFS, UCS, and A* differ only in how the fringe is managed. Once DFS is correctly built, the rest should be easier.

Code check: Thanks to their implemented codes, students should quickly find a solution to the following problems:

```
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 something expected? Does Pacman move physically to all the explored squares on his way to the goal? *Hint:* if using a *Stack* as a

data structure, the solution found by students' DFS algorithm for *mediumMaze* should have a length of 130 (provided that the algorithm pushes successors onto the fringe in the order provided by *getSuccessors*; one might get 246 when pushing them in the reverse order). Is this a least-cost solution? If not, what is depth-first search doing wrong?

3 Task 2 (3 pts): Breadth First Search

Students have to implement the breadth-first search (BFS) algorithm in the *breadthFirstSearch* function in `search.py`.

Note 1: As for the DFS, the student has to write a *graph search* algorithm that avoids expanding any already visited states. This makes the algorithm *complete*.

Code check: Similarly to DFS, the following commands are useful to verify the correctness of the implementation.

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

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

BFS should find a least-cost solution. If Pacman moves too slowly, one could try the option `--frameTime 0`.

4 Task 3 (3 pts): Varying the Cost Function

By changing the cost function in the maze, different paths can be found instead of the BFS' fewest actions path to the goal. For example, one may assign higher cost values for dangerous steps or lower for those steps in food-rich areas. A rational agent should respond by adjusting its behavior.

To introduce these costs, *mediumDottedMaze* and *mediumScaryMaze* are considered.

Students have to implement the uniform cost graph search (UCS) algorithm in the *uniformCostSearch* function in `search.py`.

Note 1: Students are encouraged to look through `util.py` for some data structures that may be useful in the implementation.

Code check: UCS agent should be successful in all the three following layouts:

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

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

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

UCS agents differ only in the cost function they use. Students should get very low and very high path costs for *StayEastSearchAgent* and *StayWestSearchAgent* respectively, due to their exponential cost functions (see `searchAgents.py` for more details).

5 Task 4 (3 pts): A* search

Students have to implement the A* graph search algorithm in the *aStarSearch* function in `search.py`.

Note 1: 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.

Note 2: The Manhattan distance heuristic is already implemented as *manhattanHeurisitc* in `search.py`.

Code check: To check the right A* implementation, the following command tests the original problem of finding a path to a fixed position through the maze *bigMaze* using the Manhattan distance heuristic:

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

A* should find the optimal solution slightly faster than uniform cost search (about 549 vs 620 search nodes expanded, but ties in priority may make numbers differ).

What happens on *openMaze* for the various search strategies?

6 Task 5 (3 pts): Finding All the Corners

The real power of A* comes out in more difficult problems. This part focuses on formulating a new problem and designing a heuristic for it. More precisely, in the so-called *corner mazes* there are four dots, one in each corner. The new search problem, *CornersProblem*, consists of finding the shortest path through the maze that touches all four corners (irrespective of whether the maze has food there or not).
Students have to implement the *CornersProblem* search problem in `searchAgents.py`.

Note 1: Make sure to complete *Task 2* before working on *Task 5*, because the latter builds upon the answer for the former.

Note 2: The student has to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached.

Note 3: Do not use a Pacman *GameState* as a search state: the only parts of the game state to be referenced in the implementation are the starting Pacman position and the location of the four corners.

Note 4: Depending on the maze, the shortest path does not always go to the closest food first. To verify this, consider that the shortest path through *tinyCorners* takes 28 steps.

Check code: The search agent should be successful when running the two following command lines:

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

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

The implementation of *breadthFirstSearch* expands just under 2000 search nodes on *mediumCorners*. However, A* search with heuristics can reduce the required searching.

7 Task 6 (3 pts): Corners Problem, Heuristic

Now, the focus is on solving the *CornersProblem* with informed search. **Students have to implement a non-trivial, non-negative, consistent heuristic for the *CornersProblem* in *cornersHeuristic***.

Note 1: Make sure to complete *Task 4* before working on *Task 6*, because the latter builds upon the answer for the former.

Note 2: The developed heuristic must be a non-trivial, non-negative, consistent heuristic. Be sure the heuristic returns 0 at every goal state and never returns a negative value.

Note 3: The trivial heuristics are (1) those returning zero everywhere (UCS) and (2) those which compute the exact cost. The built heuristic must be non-trivial.

Note 4: Heuristics are just functions that take search states and return numbers that estimate the cost to (nearest) goal. More effective heuristics will return values closer to the actual cost to (nearest) goal. To be *admissible*, the heuristic value must be a lower bound 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*.

Note 5: Remember that *admissibility* is not enough to guarantee correctness in graph search: the stronger condition of *consistency* is needed. However, admissible heuristics are usually also consistent. Therefore, it is usually easier to start by deriving admissible heuristics. Once an admissible heuristic is derived, one can check whether it is consistent too. The only way to guarantee consistency is with proof.

Note 6: If UCS and A* ever return paths of different lengths, the heuristic is surely inconsistent.

Code check: The search agent has to be successful when running the following command line:

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

where *AStarCornersAgent* is just a shortcut for

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

8 Task 7 (4 pts): Eating All the Dots

Similarly to the two previous points, we now focus on an even more complex problem and try to build a heuristic to solve it efficiently. The objective is to implement a search algorithm so that the Pacman eats all the Pacman food in as few steps as possible. For this, a new search problem formalizing the food-clearing problem is defined: *FoodSearchProblem* in `searchAgents.py`.
A solution will be a path that collects all of the food in the Pacman world. For the present project, solutions only depend on the placement of walls, regular food, and Pacman.
Students should fill in *foodHeuristic* in `searchAgents.py` with a non-trivial, non-negative, consistent heuristic for the *FoodSearchProblem*.

Before tackling the problem, students could find it useful to verify the correctness of their previously developed search methods by running the following command.

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

where *AStarFoodSearchAgent* is just a shortcut for

```
-p SearchAgent -a fn=astar, prob=FoodSearchProblem, heuristic=foodHeuristic
```

If one has written the general search methods correctly, A* with a null heuristic (that is, uniform cost search) should quickly find an optimal solution to *testSearch* with no code change, with a total cost of 7 (run command above). UCS should start to slow down even for the simple *tinySearch*. Indeed, by trying the command

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

a path of length 27 would take approximately 2.5 seconds after expanding 5057 nodes.

Note 1: Make sure to complete *Task 4* before *Task 7*, as the answer to the latter builds upon the former.

Note 2: The developed heuristic must be a non-trivial, non-negative, consistent heuristic. Be sure the heuristic returns 0 at every goal state and never returns a negative value.

Code check: Students should verify the performance of the coded agent in the *trickySearch* board through the following command:

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

As a reference benchmark for the heuristic implementation, the UCS agent should find the optimal solution in around 13 seconds, exploring over 16000 nodes.

9 Task 8 (3 pts): Suboptimal Search

Sometimes, finding the optimal path through all the dots is hard, even with A* and a good heuristic. In such cases, one may accept to find suboptimal paths, quickly.
This part focuses on building an agent that always greedily eats the closest dot. *ClosestDotSearchAgent* is implemented in `searchAgents.py`, but it is missing a function that finds a path to the closest dot.
Students have to implement the function *findPathToClosestDot* in `searchAgents.py`.

Code check: Students should find a solution to the following problem in a few seconds:
`python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5`

Note 1: The quickest way to complete *findPathToClosestDot* is to fill in the *AnyFoodSearchProblem*, which is missing its goal test. Then, solve that problem with an appropriate search function.

Note 2: The *ClosestDotSearchAgent* won't always find the shortest possible path through the maze. The student should understand why and try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots.

10 Submission

To submit their work, students must upload their folder on the course page on Microsoft Teams. To check their implementations in advance, students can try the following command:

```
python autograder.py
```

The purpose of module `autograder.py` is to briefly evaluate students' work. Using the autograder is helpful for both students and instructors. In particular, there is no limit to the number of times it can be used.

A Further Files: Info

- `pacman.py`: this is the main module that runs Pacman games. This file describes a Pacman *GameState* type, that is used in this project.
- `game.py`: this file describes several supporting types like *AgentState*, *Agent*, *Direction*, and *Grid*. Such a module contains the logic behind the Pacman world.
- `util.py`: it contains useful data structures for implementing search algorithms.
- `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 contests.
- `autograder.py`: project autograder.
- `testParser.py`: parses autograder test and solution files.
- `testClasses.py`: general autograding test classes.
- `searchTestClasses.py`: project 1 specific autograding test classes.
- `test_cases/`: directory containing the test cases for each question.

B Project Commands

Below, a list of all the commands that appear in this project is given:

- `python pacman.py`: launch Pacman game with default settings. This is a simple initial check on the imported modules.
- `python pacman.py --layout testMaze --pacman GoWestAgent`: Pacman game launch with a specific Pacman agent, *GoWestAgent*, with the map layout *testMaze*. This is another initial check to verify that everything works properly.
- `python pacman.py --layout tinyMaze --pacman GoWestAgent`: same command as the one above, with the map layout *tinyMaze*.
- `python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch`: w.r.t. the previous command, a new search agent is applied, with a new search strategy.

- `python pacman.py -l tinyMaze -p SearchAgent`: this command tells the *SearchAgent* to navigate *tinyMaze*
- `python pacman.py -l mediumMaze -p SearchAgent`: same command as the one above, with the only difference that now the maze is *mediumMaze*.
- `python pacman.py -l bigMaze -z .5 -p SearchAgent`: now the maze is *bigMaze* and it is zoomed in.
- `python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs`: *SearchAgent* has to navigate *mediumMaze* with *breadth-first search* as the selected search algorithm.
- `python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5`: *SearchAgent* has to navigate *bigMaze* with *breadth-first search*. The visualization is zoomed in.
- `python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs`: in the *mediumMaze*, the *searchAgent* navigates according to *uniform cost search*.
- `python pacman.py -l mediumDottedMaze -p StayEastSearchAgent`: the environment is not *mediumDottedMaze* and the agent *StayEastSearchAgent* (see `searchAgents.py` for more details).
- `python pacman.py -l mediumScaryMaze -p StayWestSearchAgent`
- `python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar, heuristic=manhattanHeuristic`:
  differently from the previous commands, now A* search method is applied with *Manhattan Distance* as heuristic.
- `python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`: Differently from the previous commands, here the problem to be tackled by the algorithm is specified.
- `python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`
- `python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5`
- `python pacman.py -l testSearch -p AStarFoodSearchAgent`
- `python pacman.py -l trickySearch -p AStarFoodSearchAgent`
- `python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5`
- `python pacman.py -l bigSearch -p ApproximateSearchAgent -z .5 -q`

where the options supported by `pacman.py` can each be expressed both in a *short* and in a *long* way. Below, we list some of them:

- `-n`: `--numGames` (the number of games to play, default 1).

- `-l`: `--layout` (the layout file from which to load the map layout, default *'mediumClassic*).
- `-p`: `--pacman` (the agent type in the `pacmanAgents.py` module to use, default *KeyboardAgent*).
- `-a`: `--algorithm` (specifies the algorithm to be used).
- `-t`: `--textGraphics` (display output as text only).
- `-q`: `--quietTextGraphics` (generate minimal output and no graphics).
- `-k`: `--numghosts` (number of ghosts in the maze).

While coding, one can see the list of options and their default values via: `python pacman.py -h`