The University of Melbourne
School of Computing and Information Systems
COMP90054 AI Planning for Autonomy

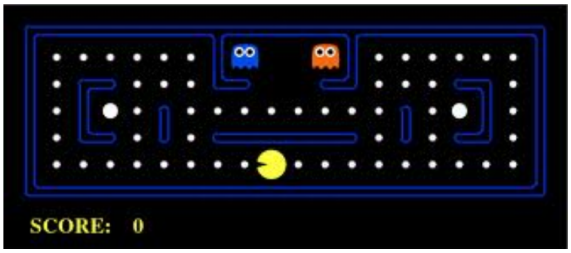
Project 1, 2021

Deadline: Thursday 1st April 18:00

This project counts towards 10% of the marks for this subject.
This project must be done individually.

Aims

The aim of this project is to improve your understanding of various search algorithms using the Berkely Pac Man framework.



https://inst.eecs.berkeley.edu/~cs188/sp21/project1/

Your task

Your tasks relate to the assignment at https://inst.eecs.berkeley.edu/~cs188/sp21/project1/.

Getting started

Before starting the assignment you **must** do the following:

- Create a github account at https://www.github.com if you don't already have one.
- Visit https://classroom.github.com/a/XQszkhjs and accept the assignment. This will create your personal assignment repository on github.
- Clone your assignment repository to your local machine. The repository contains the framework that you will need in order to complete the assignment.
- Complete the following form: https://forms.office.com/Pages/ResponsePage.aspx?id=z_NbDvQft0aRdlLFOMIqTTh0m8tXIJpNq6rJJHkp74NUREM3UFhNVjlNMDIwVjNJMUIzRlNNRzZUUS4u This allows us to link your University ID to your github ID so that we can mark your assignment.

Practice Task (0 marks)

To familiarise yourself with basic search algorithms and the Pacman environment, it is a good start to implement the breadth first search alrogithm at https://inst.eecs.berkeley.edu/~cs188/sp21/project1/; however, there is no requirement to do so. To help you understand how to interact with the framework, I have provided an implementation of the depth first search algorithm.

Part 1 (1 mark)

Implement the A* Algorithm described in lectures. You should be able to test the algorithm using the following command:

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

Other layouts are available in the `layouts` directory, and you can easily create you own!

Part 2 (3 marks)

The recursive best first search algorithm is designed to enable optimal solutions to be found while using less memory than A*. It may be used to find optimal solutions for memory-constrained problems. The algorithm is defined on page 99 of [2] (which you can access from the University library and is linked to in the Week 1 and 2 modules on Canvas) and reproduced here for your convenience:

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
  return RBFS(*problem*, MAKE-NODE(*problem*.INITIAL-STATE), ∞)

function RBFS(*problem*, *node*, *f_limit*) **returns** a solution, or failure and a new f-cost limit
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  successors $\leftarrow$ [ ]
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    add CHILD-NODE(*problem*, *node*, *action*) into *successors*
  if *successors* is empty **then return** *failure*, ∞
  for each *s* **in** *successors* **do** /* update f with value from previous search, if any */
    $s.f \leftarrow \max(s.g + s.h, node.f)$
  loop do
    best $\leftarrow$ the lowest f-value node in *successors*
    if $best.f > f\_limit$ **then return** *failure*, *best*.f
    alternative $\leftarrow$ the second-lowest f-value among *successors*
    result, *best*.f $\leftarrow$ RBFS(*problem*, *best*, $\min(f\_limit, alternative)$)
    if *result* $\neq$ *failure* **then return** *result*

Figure 3.26 The algorithm for recursive best-first search.

Note that this is a tree-search algorithm which does not consider repeated states. For example, moving Pacman down and then up again would produce a new state. As a result, the algorithm will expand a large number of search nodes in order to find solutions to problems. I encourage you to consider how you could modify the algorithm to reduce the number of nodes generated without requiring too much memory, but for the purposes of this question

you should implement the algorithm as described. You should be able to test the algorithm using the following command:

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

Other layouts are available in the `layouts` directory, and you can easily create you own!

Part 3 (4 marks)

We now consider a slight change to the rules of Pacman, specifically allowing non-uniform action costs. For the purposes of this question, moving Pacman to a square that is empty or contains food has a cost of 1, while moving Pacman to a square that contains a Capsule has a cost of 0. Note that once Pacman eats the capsule the square becomes empty so moving Pacman back to that square would incur a cost of 1.

We wish to solve the problem of eating all the food in the maze 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. You should already be able to solve this problem using your A* search implementation with the null Heuristic, but you will find that heuristic quite inefficient. As a reference, our implementation expands over 100,000 node to find a solution of length 25 for `task3search`.

Your task is to implement `foodHeuristic` in order to improve the efficiency of A* search for this problem. Recall that in order to find optimal solutions to the problem, your heuristic must be admissible.

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 admissible heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands on `task3search`, you'll be graded:

Number of nodes expanded	**Grade**
Less than 20,000	1/4
Less than 10,000	2/4
Less than 5,000	3/4
Less than 2,500	4/4

You can check the performance of your algorithm by running the following command:

```
python pacman.py -l task3Search -p AStarFoodSearchAgent
```

Since this computation can take some time for less efficienct heuristics, you may wish to start by verifying that your heuristic finds an optimal solution for `task3Small` and `task3Medium`.

Part 4 (2 marks)

Challenge Question
Note that this is a much more difficult question that requires you to interpret and implement an algorithm from a research paper. Learning to implement it successfully will give you great experience in solving a modern AI planning problem and experience in self-directed learning – something that is valuable in general, but particularly with contemporary AI techniques, but is not necessary in order to do well in the subject. As a result there is only a small mark allocation for this question.

Deceptive path-planning involves finding a path to a goal that makes it difficult for an outside observer to guess what that goal might be. This paper by Masters and Sardina describes a number of algorithms for deceptive path-planning [1]. The main idea is that an agent has a true goal as well as one or more false goals. The agent plans a path to the true goal designed to make it difficult for an observer to figure out whether it is trying to reach the true goal or one of the false goals.

Your task is to implement two of the strategies described in [1] using the Pacman framework. For each of the deceptive path planning layouts, assume that the true goal is represented by the Capsule and that the false goals are represented by Food.

a) {1 mark} Implement an agent that uses the π_{d2} strategy. You can call your agent using the following command

```
python pacman.py --layout deceptiveMap --pacman pid2DeceptiveSearchAgent
```

b) {1 mark} Implement an agent that uses the π_{d3} strategy. You can call your agent using the following command

```
python pacman.py --layout deceptiveMap --pacman pid3DeceptiveSearchAgent
```

NOTE: You should not change any files other than `search.py` and `searchAgents.py`. You should not import any additional libraries into your code. This risks being incompatible with our marking scripts.

Checking your submission

Run the command:

```
python autograder.py
```

We have provided some tests, called `task1`, `task2`, `task3`, `task41` and `task42`. It is important that you are able to run the autograder and have these tests pass, otherwise, our marking scripts will NOT work on your submission. While the final tests will be similar to those provided, we will vary the final test layouts to prevent hardcoded action paths from achieving any marks.

Marking criteria

This assignment is worth 10% of your overall grade for this subject. Marks are allocated according to the breakdown listed above, based on how many of our tests the algorithms

pass. No marks will be given for code formatting, etc.

Submission

Ensure that you have pushed your files to the repository and completed the form in the 'Getting Started' section before the due date.

The master branch on your repository will be cloned at the due date and time.

From this repository, we will copy *only* the files: `search.py` and `searchAgents.py`. Do not change any other file as part of your solution, or it will not run. Breaking these instructions breaks our marking scripts, delays marks being returned, and more importantly, gives us a headache.

Note: Submissions that fail to follow the above will be penalised.

Originality Multiplier

We will be using a code similarity comparison tool to ensure that each student's work istheir own. For code that is similar to another submission or code found online, an originality multiplier will be applied to the work. For example, if 20% of the assessment is deemed tohave been taken from another source, the final mark will be multiplied by 0.8.

Late submission policy

Submissions that are late will be penalised 1 mark per day, up to a maximum of 5 marks.

Academic Misconduct

The University misconduct policy[1] applies. Students are encouraged to discuss the assignment topics, but all submitted work must represent the individual's understanding of the topic. The subject staff take academic misconduct seriously. In the past, we have prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.

Important: As part of marking, we run all submissions via a code similarity comparison tool. These tools are quite sophisticated and are not easily fooled by attempts to make code look different. In short, if you copy code from classmates or from online sources, you risk facing academic misconduct charges.

But more importantly, the point of this assignment is to have you work through a series of foundational search algorithms. Successfully completing this assignment will make the rest of the subject, including other assessment, much smoother for you. If you cannot work out solutions for this assignment, submitting another person's code will not help in the long run.

[1] See https://academichonesty.unimelb.edu.au/policy.html

References

[1] Masters, P., and Sardina, S. Deceptive path-planning. pp. 4368–4375.

[2] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall Press, USA, 2009.