CSCE 420 - Spring 2023

Programing Assignment 1
due: Thurs, Feb 16, 2023, 5:00pm

Overview

The goal of the project is to implement A* search algorithm in python, and then test it on a version of *pacman* that involves finding and eating all the food pellets (with no ghosts to avoid).

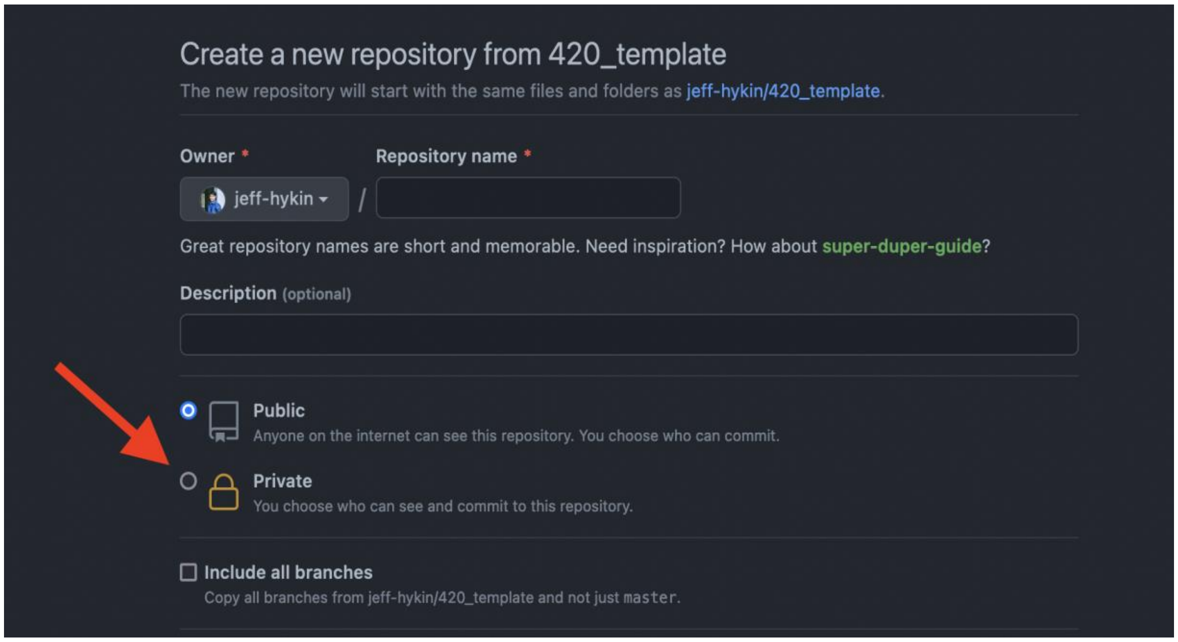
Requirements

You will need a github account, and a basic understanding of git
You will need python3
You will also need the following python packages: future, heapq
Make sure your python has tkinter (it probably does)
If your python does not have tkinter, here is how to install it:

On Linux run: `sudo apt install python3-tk`
On MacOS, get homebrew then run: `brew install python-tk`
On Windows, make sure the "tcl/tk and IDLE" checkbox is checked when installing python from the python website. Reinstalling python and checking the box is probably the easiest way to enable it.

Once you have a github account

1. Go to https://github.com/jeff-hykin/420_template
2. Click "Use Template -> Create a new repository" (use whatever name you like)
3. Make sure to make the repository private


Create a new repository from 420_template

The new repository will start with the same files and folders as jeff-hykin/420_template.

Owner * jeff-hykin / Repository name *

Great repository names are short and memorable. Need inspiration? How about super-duper-guide?

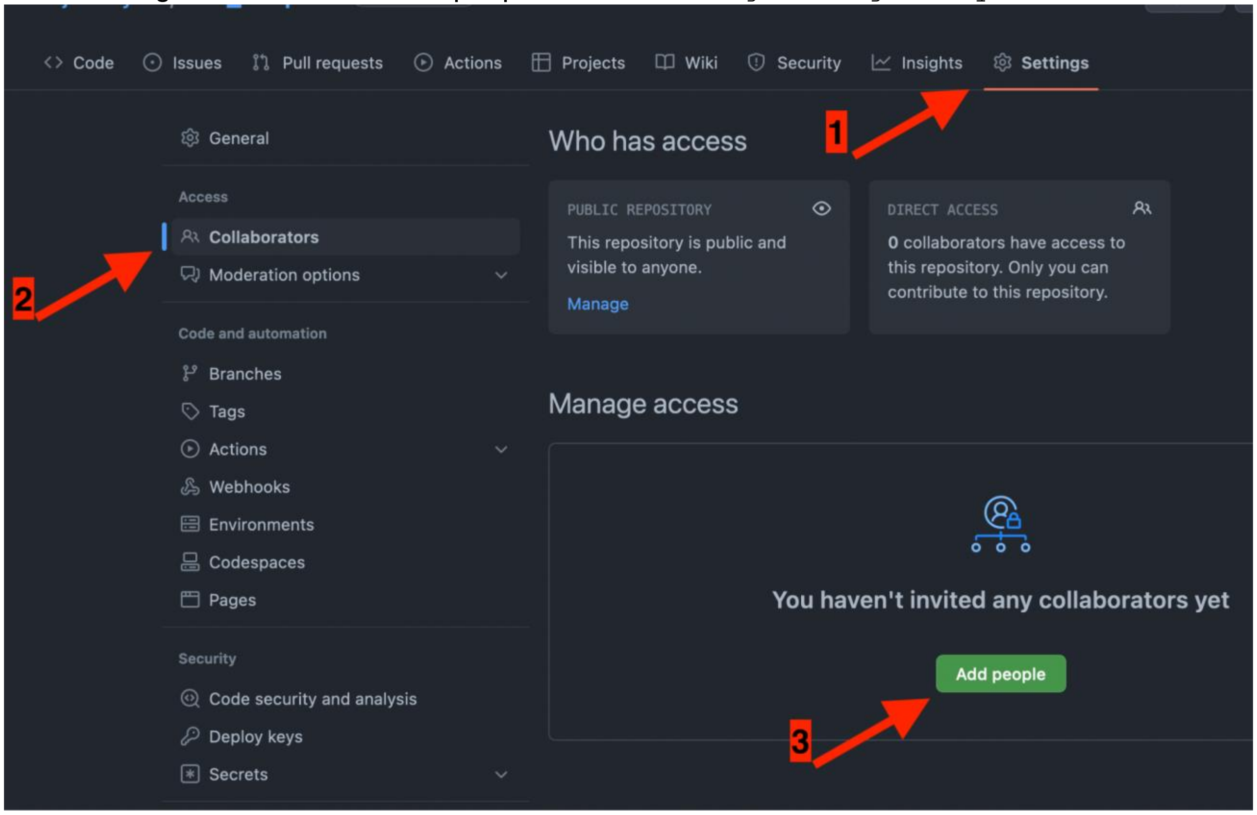
Description (optional)

[✓ Public] Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Include all branches Copy all branches from jeff-hykin/420_template and not just master.

4. Go to settings, Collaborators, Add people, and add `ioerger`, and `jeff-hykin`



Who has access

PUBLIC REPOSITORY This repository is public and visible to anyone. Manage

DIRECT ACCESS 0 collaborators have access to this repository. Only you can contribute to this repository.

Manage access

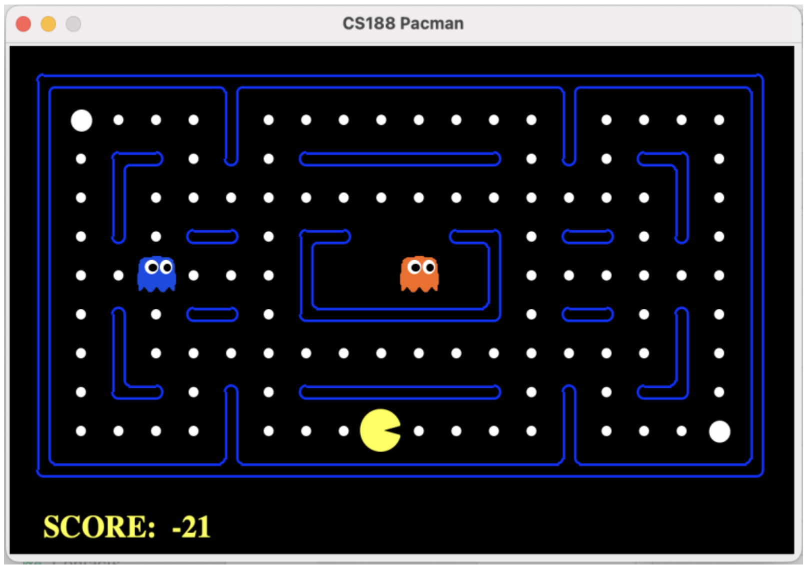
You haven't invited any collaborators yet

5. Then use git to clone your repository somewhere on your computer

Pacman

You can play the game yourself (controlled using arrow keys or ‘wasd’) by cd’ing into the ‘main/’ subdirectory and running pacman.py:

```
> cd YOUR_REPO/programming_assignment_1/main
> python pacman.py
```



There are different game layouts. The one above is called "medium_classic" and you can change it using `--layout` or `-l`. For example, to use the `big_safe_search` layout

```
python pacmac.py --layout big_safe_search
```

You can also speed up or slow down time:

```
python pacmac.py --frame_time=0.5
python pacmac.py --frame_time=2.0
```

Automated Pacman

If you want to control pacman with a program, you can use -p.
There are some built-in agents, for example:

```
python pacmac.py -p LeftTurnAgent
```

That agent will do exactly what you expect it to do; turn left.
If you want to have some fun, you can try making up your own agents, but it is NOT required for this assignment.

What you need to Automate

1. We are not going to be playing with any ghosts. So don't worry about your algorithm needing to avoid ghosts.
2. The code needs to generate a plan; a python-list of strings, where each string represents an action
3. The goal is 0 remaining food pellets, and pacman should take the shortest possible path for eating all the food pellets.
4. You can assume there will always be at least one food pellet

Normally we would need an agent to execute our plan, but lucky for us there is some built-in code that does it for us (the `SearchAgent` class). For example, if you have a function called `a_star_search` and you save it inside of main/search.py, you can tell `SearchAgent` to run that function (which generates a plan), and then execute that plan like this:

```
python pacman.py -p SearchAgent -a fn=a_star_search
```

The default map has ghosts though so you probably want to run it on a different map:

```
python pacman.py -l medium_maze -p SearchAgent -a fn=a_star_search
```

If doing a map that involves collecting food make sure to add `-a prob=FoodSearchProblem`

```
python pacman.py -l food_search_1 -p SearchAgent -a prob=FoodSearchProblem
```

What this is doing under the hood is calling the function, getting a complete list of actions and then blindly executing those actions one after another, regardless of what the game itself is doing.

How can a plan be made without playing the game?

Inside of the `a_star_search` function you have access to helpers such as `problem.get_successors()` which you can think of as a "what-if" calculation.
For example, the code below would be equivalent to asking the game
"If I took the 0th action, what options would be available after that?"

```
start = problem.get_start_state()
next_steps      = problem.get_successors(start)
next_next_steps = problem.get_successors(next_steps[0].state)
```

Exploring many "what-if" scenarios will let you generate the entire sequence of actions before pacman ever makes a single move. This kind of plan-generation really only works in perfectly deterministic, and perfectly emulatable games, which is why pacman without ghosts is perfect.

How to Automate & Test

Open up `search.py` and find `def a_star_search`. The comments inside the function will explain how to access data and what values need to be returned. Remove the `util.raise_not_defined()` and then the rest of the function is up to you.

1. There is a UI test

```
python pacman.py -l food_search_1 -p SearchAgent -a prob=FoodSearchProblem
python pacman.py -l medium_maze -p SearchAgent
```

2. There is an autograder

```
python autograder.py a_star_only
```

Individual tests of the autograder can also be run, as follows

```
python autograder.py a_star_only -q q1
python autograder.py a_star_only -q q2
```

When running on the food layouts, here are some scores you can expect.
(Expect these to change depending on your heuristic)

food_search_1.lay done with a score of around -500
food_search_2.lay done with a score of around -1500
food_search_3.lay done with a score of around -2500

Tips for implementing A* search

- Trying to implement BFS is usually the best place to start

- Although BFS usually represents the frontier with a queue, using a *priority-queue* version of BFS will make it easy to convert to A* later, where the priority score is the path cost g(n) (depth, or number of steps/moves/actions to each node from the start). You can import the *heapq* package, which implements priority queues in python.

- As you're exploring different possible actions, if you find a goal state, you'll need to **return a list of what actions lead to the goal**. One way of doing this is by creating a wrapper (a "Node" class) around each game-state, and storing the list of actions required to get to that game-state as an attribute on the node class. A more efficient way (depending on the tree depth) is to have a "parent_node" attribute on each node (think linked-list), and then, once the goal is found, traverse those parent connections to generate/recreate a list of actions.

- Once you have BFS implementation working, switch the priority queue values to be depth + heuristic value (e.g. f(n) = g(n) + h(n)).

- One of the challenges of this assignment is that you will have to come up with a good **heuristic function** h(n). This might take some creativity, as well as trial-and-error, to find one that produces efficient pacman behavior. Remember that a heuristic is supposed to estimate the number of steps remaining to reach the goal.

Grading

- 50% - Does the code make sense / is it an A star algorithm (as described by the book)?
- 30% - Does it pass the given/public test cases? (proportion correct from autograder.py)
- 20% - Does it perform well on our hidden/private test cases?
  (NOTE: the hidden tests are not difficult, and are not tricky/edge-case layouts. They're just basic tests, to further disincentivize hard-coding a solution.)

What to Turn-in:

Nothing needs to be emailed or uploaded to Canvas, just follow the steps below to create and push the git tag for this assignment.

1. Save the autograder output to result.txt

```
python autograder.py a_star_only > result.txt
```

2. Add and commit your changes

```
git add -A && git commit -m "your message"
```

3. Push your changes

```
git push
```

4. Create & push a git tag using the following command

```
git tag "pa1" && git push origin "pa1"
```

5. If you want to resubmit, simply delete the old git tag, and then recreate it on your latest commit. Here's a command for deleting the old tag:

```
git push --delete origin "pa1" && git tag --delete "pa1"
```