Sistemas Inteligentes

PacMan

João Peixoto

jpeixoto@ua.pt

PacMan | Search

- In this challenge, 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
- The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore
- You can download all the code and supporting files at **eLearning** platform: *Desafio 2*

PacMan | Search

- Files you'll edit:
  - *search.py*
    - Where all of your search algorithms will reside
  - *searchAgents.py*
    - Where all of your search-based agents will reside

PacMan | Search

- 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

PacMan | Search

- After downloading the code, 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*

Example of output:

```
Pacman emerges victorious! Score: 1692
('Average Score:', 1692.0)
('Scores:       ', '1692.0')
Win Rate:      1/1 (1.00)
('Record:       ', 'Win')
```

PacMan | Search

- 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*

```
Pacman emerges victorious! Score: 503
('Average Score:', 503.0)
('Scores:       ', '503.0')
Win Rate:      1/1 (1.00)
('Record:       ', 'Win')
```

PacMan | Search

- If we change *GoWestAgent* to *SearchAgent* (class available in the same file *searchAgents.py*)

  - *python pacman.py --layout testMaze --pacman SearchAgent*

```
*** Method not implemented: depthFirstSearch at line 90
```

PacMan | Search

- If we want the agent solve the problem
  - *python pacman.py -p SearchAgent -a fn=depthFirstSearch*

- To select an agent, use the '*-p*' option when running *pacman.py*

- Arguments can be passed to your agent using '*-a*' universidade de aveiro

deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

PacMan | Search

- Example:
  - *python pacman.py -p GoWestAgent*

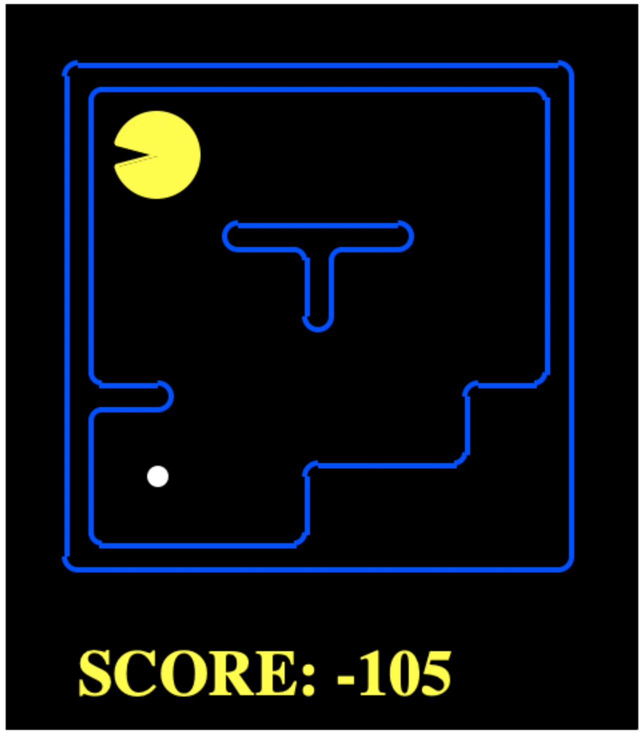
- Because the agent is stucked, the result has:

```
Pacman died! Score: -561
('Average Score:', -561.0)
('Scores:       ', '-561.0')
Win Rate:      0/1 (0.00)
('Record:       ', 'Loss')
```

- This is what we want to avoid ☺

PacMan | Search

- Another example:
  - *python pacman.py --layout tinyMaze --pacman GoWestAgent*



PacMan | Search

- Step 1
  - In *searchAgents.py*, we have 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 the challenge!**
  - First, test that the SearchAgent is working correctly by running:
    - *python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch*
    - The command tells the SearchAgent to use tinyMazeSearch as its search algorithm, which is implemented in *search.py*
    - Pacman should navigate the maze successfully

PacMan | Search

- Step 1
  - 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
  - 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)

PacMan | Search

- Step 1
  - Make sure to **use** the Stack, Queue and PriorityQueue data structures provided to you in *util.py*!
  - 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

PacMan | Search

- Step 1
  - 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*

PacMan | Search

- Step 1
  - 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?
  - If you use a Stack as your data structure, the solution found by your DFS algorithm for mediumMaze should have a length of 130
  - Is this a least cost solution? If not, think about what depth-first search is doing wrong

PacMan | Search

- Step 2
  - After the DFS algorithm, 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

PacMan | Search

- Step 2
  - If Pacman moves too slowly for you, try the option *--frameTime 0*
  - 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*

PacMan | Search

- Step 3
  - 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
    - 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*

PacMan | Search

- Step 3
  - Look through *util.py* for some data structures that may be useful in your implementation
  - 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*

PacMan | Search

- Step 3
  - 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)
- Step 4
  - Next step, 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

PacMan | Search

- Step 4
  - 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?

PacMan | Search

- Step 5
  - 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!
  - The shortest path through tinyCorners takes 28 steps

PacMan | Search

- Step 5
  - 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 complete the challenge, 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)

PacMan | Search

- Step 5
  - 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
  - The implementation of breadthFirstSearch expands just under 2000 search nodes on mediumCorners. However, heuristics (used with A* search) can reduce the amount of searching required.

PacMan | Search

- Step 6
  - Implement a non-trivial, consistent heuristic for the CornersProblem in cornersHeuristic.
    - *python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5*
  - 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.

PacMan | Search

- Step 6
  - 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

PacMan | Search

- Step 6
  - 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)

PacMan | Search

- Step 7
  - 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 challenge, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman

PacMan | Search

- Step 7
  - 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*
    - 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, the implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search nodes

PacMan | Search

- Step 7
  - 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

PacMan | Search

- Step 8
  - Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard
  - In these cases, we'd still like to find a reasonably good path, quickly
  - You must write an agent that always greedily eats the closest dot
  - ClosestDotSearchAgent is implemented for you in *searchAgents.py*, but it's missing a key function that finds a path to the closest dot
  - Implement the function findPathToClosestDot in *searchAgents.py*
    - Our agent solves this maze (suboptimally!) in under a second with a path cost of 350
    - *python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5*

PacMan | Search

- Step 8
  - 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. The solution should be very short!
  - Your ClosestDotSearchAgent won't always find the shortest possible path through the maze
  - Make sure you 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