CMSC 170 MACHINE PROBLEM 1 **Total: 70 points** (15% of grade)
Pacman Search **Deadline**: March 22, 2019 (Friday)

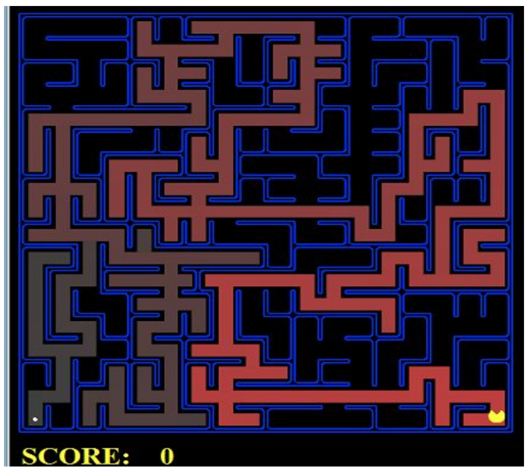
The first Machine Problem is one of the *Pacman Projects* from UC Berkeley's CS 188 (Artificial Intelligence) class (http://ai.berkeley.edu/search.html), which will test your abilities and understanding of *search problems*, *state representations*, *uninformed* and *informed search algorithms,* and *heuristics.* You will be working **in pairs** for this project, and you are allowed to choose your own partner.

Files and software. All the codes and supporting files needed for this project are included in the zip file (credits to UCB CS 188 for making this available to the public), including the original instructions from UC Berkeley CS 188. You will need **Python 2.7** installed to run the code.

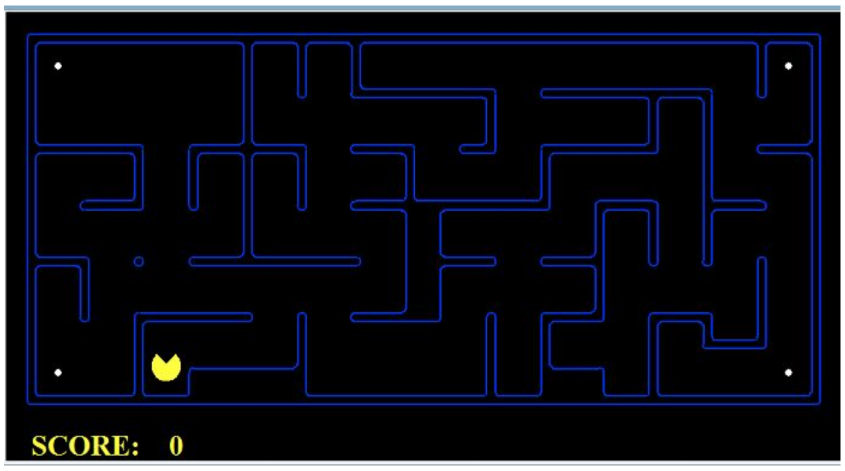
Tasks. In this project, there are *three search problems* set in the Pacman environment, that involves reaching a particular location in the maze, and collecting food efficiently. Your tasks include setting appropriate *state representations* of the search problem, implementing *general search algorithms* that apply to a variety of scenarios, and designing *heuristics* to make the search more efficient.

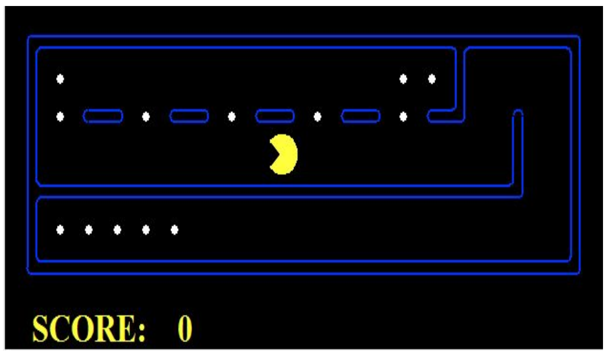
You only need to edit two files from the given codes: **search.py** and **searchAgents.py**. For extra function definitions, please write them in a new file named **extra.py**, and import as needed.

Find Fixed Dot

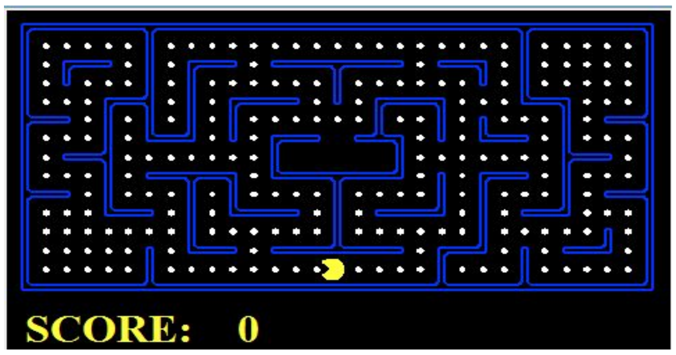


Corners Problem





Eat-all-Dots *(Few)*



Eat-all-Dots (*Many*)

Instructions: Please read the original instructions (**pacman_search.pdf**) for a detailed discussion of each problem. The ff. will just serve as a supplementary guide to help you solve the problems.

GENERAL TIPS

- You can check the files **pacman.py**, **game.py**, and **util.py** to have better understanding of how the code works. You can just ignore the rest of the files.
- During testing, if you find Pacman's speed too slow, add **--frameTime 0** to the command to make Pacman run faster (e.g. python pacman.py -l mediumMaze -p SearchAgent --frameTime 0).
- During testing, if the maze doesn't fit entirely on your screen, add **-z 0.5** to the command to make the maze display smaller (e.g. python pacman.py -l bigSearch -p ClosestDotSearchAgent -z 0.5)

PROBLEM #1: FIND FIXED DOT

This problem involves Pacman starting in some location on the maze and the goal is to find the single food dot present in the maze. You will be implementing graph search algorithms to make Pacman plan its route through the maze to find the dot.

Task 1. *Implementing Depth First Search and Breadth First Search* **6 points**

- Implement the **graph search** version of **Depth First Search** in **search.py**
- Implement the **graph search** version of **Breadth First Search** in **search.py**
- Use appropriate *data structures* from **util.py**
- Codes for the two are very similar; only differs in a few lines
- Expected **input** = problem object
- Expected **output** = list of actions (Directions)
- Useful methods to explore: problem.*getStartState*(), problem.*isGoalState*(state), problem.*getSuccessors*(state)
- Use a **set** to keep track of explored states (graph search)
- Make sure you create a *new action list* for the next state; don't overwrite the current state's action list.
- If no solution found, **return [Directions.STOP]**.
- You need to keep track of both the state and the actions taken to reach that state. You can implement your own search node object or you can use a tuple.
- *Scoring multiplier:* **x 1.0**; the total score you will get from the autograder for Q1 and Q2 (maximum = 6) will be your score in Task 1.

Task 2. *Implementing Uniform Cost Search and A* Search* **9 points**

- Implement the **graph search** version of **Uniform Cost Search** in **search.py**
- Implement the **graph search** version of **A* Search** in **search.py**
- Use appropriate *data structures* from **util.py**
- Codes for the two are similar to the codes for task 1; only differs in the usage of the data structure used
- Codes for the two are very similar; only differs in the computation of priority
- Expected **input** for **A* search** = problem, heuristic
- *heuristic* function takes two arguments: *state, problem*
- Useful methods to explore: problem.*getCostOfActions(*actions)
- *Scoring multiplier:* **x 1.5**; the total score you will get from the autograder for Q3 and Q4 (maximum = 6) will be multiplied by 1.5 for your score in Task 2 (*maximum = 9 points*).

PROBLEM #2: CORNERS PROBLEM

This problem involves Pacman starting in some location on the maze and the goal is to find all four food dots located in the corners of the maze. You will decide the state representation and design a consistent heuristic for this problem. The aim is for Pacman to plan the shortest path through the maze that touches the 4 corners. *Note:* The shortest path doesn't always go to the closest food first.

Task 3. *State Representation for Corners Problem* **6 points**

- *Prerequisite:* You need to solve Task 1 before you can work on this task
- Finish the **implementation** of **CornersProblem** class in **searchAgents.py**
- Choose an appropriate **state representation** that encodes all information necessary to detect whether all four corners have been reached
- Methods to edit:
    - **__init__** - decide state representation, set ***self.startState***
    - **getStartState** - return ***self.startState***
    - **isGoalState** - does the given state pass the goal test → are all 4 corners visited?
    - **getSuccessors** - add (*next_state, action, cost*) to **successors** list
- Check *PositionSearchProblem* class in the same file as a reference for what each method needs to do
- Check the commented-out code snippet in **getSuccessors** to help you in completing this method
- *Common error:* list is unhashable; *solution:* use tuples instead
- *Scoring multiplier:* **x 2.0;** the score you will get from the autograder for Q5 (maximum = 3) will be multiplied by 2.0 for your score in Task 3 (maximum = 6).

Task 4. *Designing a Consistent Heuristic for Corners Problem* **9 points**

- *Prerequisite:* You need to solve Tasks 2 and 3 before you can work on this task
- Implement a **non-trivial, consistent heuristic** for the *CornersProblem*
- *Goal:* use this heuristic to reduce number of nodes expanded by A* search
- Fill in **cornersHeuristic** in **searchAgents.py**
- Expected **input** = state, problem
- Expected **output** = score that estimates how close state is to the goal (lower number = closer)
- Return 0 for goal states; <u>never</u> return *negative values*
- *How to design the heuristic?* Think of a relaxed version of the original problem → finding the shortest path from the current location that visits all (unvisited) corners.
- Autograder score depends on the number of nodes expanded = the fewer, the better (refer to *pacman_search.pdf**);*** however, inconsistent heuristics will receive 0 points
- *Scoring multiplier:* **x 3.0;** the score you will get from the autograder for Q6 (maximum = 3) will be multiplied by 3.0 for your score in Task 4 (maximum = 9).

PROBLEM #3: EAT-ALL-DOTS

This problem involves Pacman starting in some location on the maze and the goal is to eat all the food dots in the maze. You will design a consistent heuristic for this problem, and using A* search the aim is for Pacman to plan how to eat all the dots in as few steps as possible.

However, for test cases with a lot of food dots, A* search will take a long time to find a solution. We might be better off using a Greedy agent that immediately gets to work by repeatedly eating the closest food available. We will get a *reasonably good* path. Although it might not be optimal, this agent will finish quicker than A*.

Task 5. *Desigining a Consistent Heuristic for FoodSearch Problem* **12 points**

- *Prerequisite:* You need to solve Task 2 before you can work on this task
- Implement a **non-trivial, consistent heuristic** for the *FoodSearchProblem*
- *Goal:* use this heuristic to reduce number of nodes expanded by A* search
- Fill in **foodHeuristic** in **searchAgents.py**
- Expected **input** = state, problem
- Expected **output** = score that estimates how close state is to the goal (lower number = closer)
- Return 0 for goal states; never return *negative values*
- *How to design the heuristic?* Think of a relaxed version of the original problem → finding the shortest path from the current location that eats all uneaten food dots.
- state → (pacmanPosition, foodGrid)
- foodGrid.asList() → sorted list of food coordinates
- *Common error:* Heuristic failed non-triviality test → your heuristic is returning the same value whatever the given state is; try to incorporate the current state in computing your heuristic value.
- Autograder score depends on the number of nodes expanded = the fewer, the better (refer to *pacman_search.pdf***);** however, inconsistent heuristics will receive 0 points
- *Scoring multiplier:* **x 3.0;** the score you will get from the autograder for Q7 (maximum = 4) will be multiplied by 3.0 for your score in Task 5 (perfect score = 12, maximum score = 15 (with extra credit)).

Task 6. *Implement a Greedy agent* **9 points**

- Complete the **implementation** of **ClosestDotSearchAgent** in **searchAgents.py**
- Fill in the **findPathToClosestDot** method in **ClosestDotSearchAgent**
- You can easily solve *findPathToClosestDot* by using an existing search algorithm you implemented in earlier tasks (mini-search inside a search problem)
- *Hint:* I used Uniform Cost Search, but try to see what BFS and DFS does as well
- The **problem** object in *findPathToClosestDot* is an **AnyFoodSearchProblem** object
- **AnyFoodSearchProblem** → problem of searching for *any* food in the grid; which food it will look for depends on the goal test definition; in our case, we want to find the closest food dot
- Fill in **AnyFoodSearchProblem**'s **isGoalState** method to complete the problem definition → in our case, we want to find the closest food dot from Pacman's current position; we have found the closest dot if it is in the same position as Pacman's position.
- *Scoring multiplier:* **x 3.0;** the score you will get from the autograder for Q8 (maximum = 3) will be multiplied by 3.0 for your score in Task 4 (maximum = 9).
- *Hint:* Finish this task before Task 5; maybe it will give you an *idea* how to design your heuristic.

Individual Report

- Each student will write an individual report that will contain the ff:

  Name: ____ Self-Rating: _________
  Pair: ____ Rating: _____________
  1. Which tasks in MP1 did you solve?
  2. How did your codes for DFS, BFS, UCS, and A* vary?
  3. Why is graph search more appropriate than tree search in this scenario?
  4. ClosestDotSearchAgent (from task 6) doesn't always find the optimal path through the maze. Why do you think, in some cases, it ignores some nearby dots and goes back to them at the tail-end of the solution? (see demo video posted on Facebook group)
  5. For task 6, try to run A* search on the bigSearch test case. Compare its performance with ClosestDotSearchAgent. Which one is "better" in this case?
- Individual reports will be submitted in PDF format, through email (see instructions below). Please name your PDF file using this format: 170_mp1_*lastname*.pdf (example: 170_mp1_Daradal.pdf)

Autograder. The files include an autograder that you can use to automatically check if your answer is correct. The autograder runs your solution against different scenarios to discourage hardcoding of solutions. The same autograder will be used to check and grade your final output, although we will be using a different scoring system (see Scoring section below).

If you run **python autograder.py** on the command line, it will run the test cases for all problems (q1-q8). If you only want to test for a particular problem (e.g. q1), you can run **python autograder.py -q q1.**

Please use the *data structures* (Stack, Queue, PriorityQueue) found in **util.py**, and do not edit *other files* besides **search.py** and **searchAgents.py**; otherwise, you will encounter problems with the autograder.

Scoring. The scoring system we will use will be different from the original one (see details in the table below).

	Problems	**Original Score**	**Multiplier**	**Perfect Score**
Task 1	q1, q2	6	x 1.0	6
Task 2	q3, q4	6	x 1.5	9
Task 3	q5	3	x 2.0	6
Task 4	q6	3	x 3.0	9
Task 5	q7	4	x 3.0	12
Task 6	q8	3	x 3.0	9
Comments	-	-	-	5
References	-	-	-	2
Punctuality	-	-	-	2
Individual Report	-	-	-	10
Total				**70**

Academic Dishonesty. Work on this project as a pair. Hopefully the hints and tips given above will be sufficient to help you solve the problems. Please do not submit code that you copied from another group or from a source you found online (and you slightly modified). Copied codes will automatically be given a score of zero. Remember UP's motto - Honor and Excellence (honor first, before excellence).

Comments and References. To make sure that you understand the code you are submitting, please write *comments* on your code explaining the major steps in your solution (**5 points**). Please also list down all the *references* you used in this project in ***references.txt*** **(2 points).**

Submission

1. Create a **zip file** containing the ff:
    a. **search.py** -- contains your code for Tasks 1 and 2; sufficiently *commented*
    b. **searchAgents.py** -- contains your code for Tasks 3,4,5,6; sufficiently *commented*
    c. **extra.py** -- *optional*; contains extra function definitions; sufficiently *commented*
    d. **references.txt** -- list of references you used
2. Please follow this *filename format* for the zip file: *lastname1_lastname2.zip* (e.g. Garcia_Santos.zip).
3. Email your output to jrdaradal@up.edu.ph on or before **March 22, 2019 (Friday), 11:59 PM**.
4. Please use this format as the subject: **CMSC 170 - MP 1 - Lastname1 / Lastname2**
5. Late submissions will not receive **2 points** for punctuality.
6. Submissions after **March 25, 2019 (Monday)** will no longer be accepted.
7. Submit your **individual reports** through email.
8. Failure to submit individual report = assumed that you didn't help in the machine problem (rating = 0)

Maximize your expected utility.