
بِسْمِ اللهِ الرَّحْمنِ الرَّحِيمِ
"In the name of Allah,
the entirely merciful, the especially merciful."
-Quran 1:1
HolyQuran.co

# Project1-Search Of Artificial Intelligence Course

By: Mojtaba Zolfaghari

mowjix@gmail.com

Professor :Dr.Ali Shakiba

http://alishakiba.ir

Preparedin the Faculty of Mathematics and Computer Science

Vali-e-Asr University of Rafsanjan

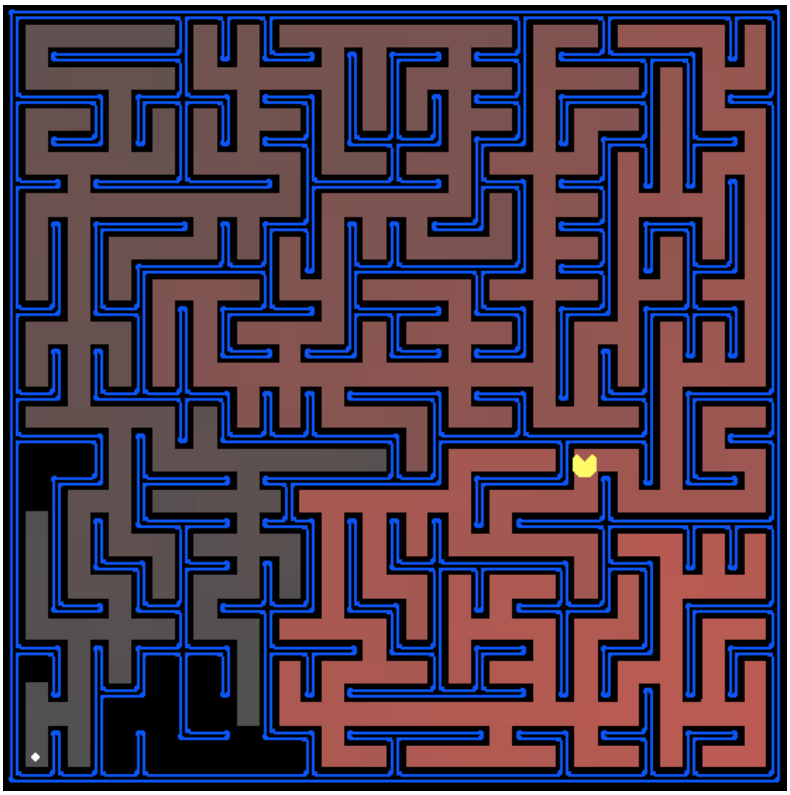
Apr. 2021

# Python3 port of Berkeley AI Pacman Search

(http://ai.berkeley.edu)

## Project 1: Search With Solutions

**Last Update: 04/21/2021.**



All those colored walls,

Mazes give Pacman the blues,

So teach him to search.

# Table of Contents

**Title** **Page**

Introduction ---------- 4

Q1: Depth First Search ---------- 7

Q2: Breadth First Search ---------- 11

Q3: Uniform Cost Search ---------- 13

Q4: A* Search ---------- 16

Q5: Corners Problem: Representation ---------- 18

Q6: Corners Problem: Heuristic ---------- 21

Q7: Eating All The Dots: Heuristic ---------- 23

Q8: Suboptimal Search ---------- 25

# PROJECT 1 : SEARCH

## Introduction

In this project, 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.

This project includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

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.

**Files you'll edit and submit:**

search.py Where all of your search algorithms will reside.

searchAgents.py Where all of your search-based agents will reside.

**Helpful file for running code (might edit, won't submit):**

run.py Use this file to run any commands in this readme. Look at the examples and make modifications as necessary. This file is helpful if you are not running from the command line, but you need to pass command line arguments to the code.

# PROJECT 1 : 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.

**Supporting files you can ignore:**

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 contents

autograder.py Project autograder

testParser.py Parses autograder test and solution files

testClasses.py General autograding test classes

test_cases Directory containing the test cases for each question

searchTestClasses.py Project 1 specific autograding test classes

## Welcome to Pacman

After downloading or cloning the code, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Note: First of all you have to go to the directory location then run above command.

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

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
```

But, things get ugly for this agent when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

Soon, your agent will solve not only `tinyMaze`, but any maze you want.

Note that `pacman.py` supports a number of options that can each be expressed in a long way (e.g., `--layout`) or a short way (e.g., `-l`). You can see the list of all options and their default values via:

```
python pacman.py -h
```

Also, all of the commands that appear in this project also appear in `commands.txt`, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with `bash commands.txt`.

Note: if you get error messages regarding Tkinter, see
(http://tkinter.unpythonic.net/wiki/How_to_install_Tkinter).

---

## Question 1 : Finding a Fixed Food Dot using Depth First Search

In `searchAgents.py`, you'll find 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 your job. As you work through the following questions, you might find it useful to refer to the object glossary (the second to last tab in the navigation bar above).

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

The command above tells the `SearchAgent` to use `tinyMazeSearch` as its search algorithm, which is implemented in `search.py`. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. 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.

***Important note:*** 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).

***Important note:*** Make sure to use the `Stack`, `Queue` and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder.

**Hint** 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. (Your implementation need **not** be of this form to receive full credit).

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
```

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?

**Hint** If you use a `Stack` as your data structure, the solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

## Solution

For implementing `def depthFirstSearch(problem):` we have to search the deepest nodes in the search tree first. Search algorithm needs to return a list of actions that reaches the goal. For this we implement a graph search algorithm.

We describe the solution step by step, so let's go :

***Step 1)*** First of all we have to generate start state for search problem:

```
    startingNode = problem.getStartState()
```

***Step 2)*** Check if start state is goal state therefore we have to return empty action list:

```
if problem.isGoalState(startingNode):
    return []
```

***Step 3)*** For next part we generate a stack and a empty list for visited nodes then we have to push startingNode and empty list(which represents empty action list) in generated stack:

```
myQueue = util.Stack()
visitedNodes = []
# (node,actions)
myQueue.push((startingNode, []))
```

***Step 4)*** It's time to implement main part on our `myQueue` and find the solution.

1) pop an element from `myQueue`.

```
currentNode, actions = myQueue.pop()
```

2) Check if pop element not in `visitedNodes` then insert it at the end of `visitedNodes`.

```
    if currentNode not in visitedNodes:
        visitedNodes.append(currentNode)
```

3) If previous condition was true then check if pop element is goal state then return `actions` list.

```
if problem.isGoalState(currentNode):
        return actions
```

4) If previous `2)` condition was true and if pop element was not goal state then according to `getsuccessors` method, we have to find next node and next action then finaly push them in `myQueue`.

```
for nextNode, action, cost in problem.getSuccessors(currentNode):
    newAction = actions + [action]
    myQueue.push((nextNode, newAction))
```

5) Repeat above steps until `myQueue` is not empty yet.

```
while not myQueue.isEmpty():
```

***Full Implementation:***

```
def depthFirstSearch(problem):

        startingNode = problem.getStartState()
        if problem.isGoalState(startingNode):
        return []

        myQueue = util.Stack()
        visitedNodes = []
        # (node,actions)
        myQueue.push((startingNode, []))

        while not myQueue.isEmpty():
        currentNode, actions = myQueue.pop()
        if currentNode not in visitedNodes:
```

```
                visitedNodes.append(currentNode)

            if problem.isGoalState(currentNode):
                return actions

            for nextNode, action, cost in problem.getSuccessors(currentNode):
                newAction = actions + [action]
                myQueue.push((nextNode, newAction))

        util.raiseNotDefined()
```

---

## Question 2 : Breadth First Search

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.

**Hint** If Pacman moves too slowly for you, try the option `--frameTime 0`.

***Note:*** 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
```

## Solution

For implementing `def depthFirstSearch(problem):` we have to search the shallowest nodes in the search tree first. Solution of this part is exactly same as previous solution but instead of `Stack` we have to use `Queue`.

***Implementation:***

```
def breadthFirstSearch(problem):

    startingNode = problem.getStartState()
    if problem.isGoalState(startingNode):
        return []

    myQueue = util.Queue()
    visitedNodes = []
    # (node,actions)
    myQueue.push((startingNode, []))

    while not myQueue.isEmpty():
        currentNode, actions = myQueue.pop()
        if currentNode not in visitedNodes:
            visitedNodes.append(currentNode)

            if problem.isGoalState(currentNode):
                return actions

            for nextNode, action, cost in problem.getSuccessors(currentNode):
                newAction = actions + [action]
                myQueue.push((nextNode, newAction))

    util.raiseNotDefined()
```

## Question 3 : Varying the Cost Function

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. For example, 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`. We encourage you to look through `util.py` for some data structures that may be useful in your implementation. You should now 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
```

***Note:*** 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).

## Solution

For implementing `def uniformCostSearch(problem):` we have to search the node of least total cost first. Search algorithm needs to return a list of actions that reaches the goal. For this we implement a graph search algorithm.

## PROJECT 1 : SEARCH

We describe the solution step by step... so let's go :

***Step 1)*** First of all we have to generate start state for search problem:

```
startingNode = problem.getStartState()
```

***Step 2)*** Check if start state is goal state then we have to return empty action list:

```
if problem.isGoalState(startingNode):
    return []
```

***Step 3)*** For next part we generate a priority queue and a empty list for visited nodes then we have to push startingNode as coordinate/node, empty list(which represents action to current node), cost to current node and priority in generated priority queue:

```
    pQueue = util.PriorityQueue()
    visitedNodes = []
    #((coordinate/node , action to current node , cost to currentnode),priority)
    pQueue.push((startingNode, [], 0), 0)
```

***Step 4)*** It's time to implement main part on our `pQueue` and find the solution:

1) pop an element from `pQueue`.

```
currentNode, actions, prevCost = pQueue.pop()
```

2) Check if pop element not in `visitedNodes` then insert it at the end of `visitedNodes`.

```
    if currentNode not in visitedNodes:
        visitedNodes.append(currentNode)
```

3) If previous condition was true then Check if pop element is goal state then return `actions` list.

```
if problem.isGoalState(currentNode):
        return actions
```

4) If previous `2)` condition was true and if pop element was not goal state then according to `getsuccessors` method, we have to find next node, next action and next priority then finally push them in `pQueue`.

```
for nextNode, action, cost in problem.getSuccessors(currentNode):
        newAction = actions + [action]
        priority = prevCost + cost
        pQueue.push((nextNode, newAction, priority),priority)
```

5) Repeat above steps until `pQueue` is not empty yet.

```
while not pQueue.isEmpty():
```

***Full Implementation:***

```
def uniformCostSearch(problem):

        startingNode = problem.getStartState()
        if problem.isGoalState(startingNode):
            return []

        pQueue = util.PriorityQueue()
        visitedNodes = []
    #((coordinate/node ,action to current node , cost to current node),priority)
        pQueue.push((startingNode, [], 0), 0)

        while not pQueue.isEmpty():

            currentNode, actions, prevCost = pQueue.pop()
            if currentNode not in visitedNodes:
                visitedNodes.append(currentNode)

                if problem.isGoalState(currentNode):
                    return actions
```

```
                    for nextNode, action, cost in problem.getSuccessor(currentNode):
                        newAction = actions + [action]
                        priority = prevCost + cost
                        pQueue.push((nextNode, newAction, priority),priority)

            util.raiseNotDefined()
```

---

## Question 4 : A* search

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.

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?

## Solution

For implementing `def aStarSearch(problem, heuristic=nullHeuristic):` we have to search the node that has the lowest combined cost and heuristic first. Search algorithm needs to return a list of actions that reaches the goal. For this we implement a graph search algorithm.

Solution of this part is exactly same as previous solution but in ***Step 4)*** at part 4 we have to do some changes:

***Step 4)***

4) If previous `2)` condition was true and if pop element was not goal state then according to `getsuccessors` method, we have to find next node, new action, next cost to node and determine heuristic cost then finally push them in `pQueue`.

```
for nextNode, action, cost in problem.getSuccessors(currentNode):
        newAction = actions + [action]
        newCostToNode = prevCost + cost
        heuristicCost = newCostToNode + heuristic(nextNode,problem)
        pQueue.push((nextNode, newAction, newCostToNode),heuristicCost)
```

***Full Implementation:***

```
def uniformCostSearch(problem):

        startingNode = problem.getStartState()
        if problem.isGoalState(startingNode):
            return []

        pQueue = util.PriorityQueue()
        visitedNodes = []
    #((coordinate/node ,action to current node , cost to current node),priority)
        pQueue.push((startingNode, [], 0), 0)

        while not pQueue.isEmpty():

            currentNode, actions, prevCost = pQueue.pop()
            if currentNode not in visitedNodes:
                visitedNodes.append(currentNode)

                if problem.isGoalState(currentNode):
                    return actions

                for nextNode, action, cost in problem.getSuccessor(currentNode):
                newAction = actions + [action]
                newCostToNode = prevCost + cost
                heuristicCost = newCostToNode + heuristic(nextNode,problem)
                pQueue.push((nextNode, newAction, newCostToNode),heuristicCost)

        util.raiseNotDefined()
```

## Question 5 : Finding All the Corners

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!

**Hint** the shortest path through `tinyCorners` takes 28 steps.

***Note:*** Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.

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 receive full credit, 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).

**Hint** 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.

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

## Solution

For implementing `class CornersProblem(search.SearchProblem):` Some of part predefined. In this solution we will discuss about parts that not implemented.

***1)*** In `def isGoalState(self, state):` we have to returns the start state.

***Implementation:***

```
def getStartState(self):

return (self.startingPosition, [])
    util.raiseNotDefined()
```

***2)*** In `def isGoalState(self, state):` we have to returns whether this search state is a goal state of the problem.

***Implementation:***

```
def isGoalState(self, state):
    node = state[0]
    visitedCorners = state[1]

if node in self.corners:
        if not node in visitedCorners:
            visitedCorners.append(node)
        return len(visitedCorners) == 4
    return False
```

***3)*** In `def getSuccessors(self, state):` we have to returns successor states, the actions they require, and a cost of 1. Notice that 'successor' is a successor to the current state.

First of all we have to get current position and also get visited corners of PacMan:

***Implementation:***

```
    x, y = state[0]
    visitedCorners = state[1]
```

In the next step we have to set initial value in the successors:

```
    successors = []
```

According to this function hints in `getSuccessors` Here's a code snippet for figuring out whether a new position hits a wall:

```
for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
            successorVisitedCorners = list(visitedCorners)
            next_node = (nextx, nexty)
            if next_node in self.corners:
                if next_node not in successorVisitedCorners:
                    successorVisitedCorners.append(next_node)
            successor = ((next_node, successorVisitedCorners), action, 1)
            successors.append(successor)
```

***Full Implementation:***

```
  def getSuccessors(self, state):

    x, y = state[0]
    visitedCorners = state[1]
    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
            successorVisitedCorners = list(visitedCorners)
            next_node = (nextx, nexty)
            if next_node in self.corners:
                if next_node not in successorVisitedCorners:
                    successorVisitedCorners.append(next_node)
            successor = ((next_node, successorVisitedCorners), action, 1)
            successors.append(successor)

    self._expanded += 1
    return successors
```

## Question 6 : Corners Problem: Heuristic

***Note:*** Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.

Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

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

***Note:*** `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. 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. 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).

***Grading:*** Your heuristic must be a non-trivial non-negative consistent 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, you'll be graded:

| number of nodes expanded | Grade |
|---|---|
| over 2000 | 0/3 |
| at most 2000 | 1/3 |
| at most 1600 | 2/3 |
| at most 1200 | 3/3 |

**Remember:** If your heuristic is inconsistent, you will receive **no** credit, so be careful!

## Solution

1) In `def cornersHeuristic(state, problem):` first of all we have to obtain corner coordinated and walls of the maze as Grid.

```
corners = problem.corners
walls = problem.walls
```

2) Find which corners are left to reach GoalState:

```
visitedCorners = state[1]
cornersLeftToVisit = []
for corner in corners:
    if corner not in visitedCorners:
        cornersLeftToVisit.append(corner)
```

3) While not all corners are visited find via manhattanDistance the most efficient path for each corner and finally returns totalCost:

```
  totalCost = 0
    coordinate = state[0]
    curPoint = coordinate
    while cornersLeftToVisit:
        heuristic_cost, corner = \
            min([(util.manhattanDistance(curPoint, corner), corner) for corner in cornersLeftT
oVisit])
        cornersLeftToVisit.remove(corner)
        curPoint = corner
        totalCost += heuristic_cost
    return totalCost
```

---

## Question 7 : Eating All The Dots

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 project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) 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
```

***Note:*** `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, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search nodes.

***Note:*** Make sure to complete Question 4 before working on Question 7, because Question 7 builds upon your answer for Question 4.

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.
Any non-trivial non-negative consistent heuristic will receive 1 point. 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, you'll get additional points:

| Number of nodes expanded | Grade |
|---|---|
| more than 15,000 | 1/4 |
| at most 15,000 | 2/4 |
| at most 12,000 | 3/4 |
| at most 9,000 | 4/4 |
| at most 7,000 | 5/4 (bonus point) |

**Remember**: If your heuristic is inconsistent, you will receive **no** credit, so be careful! Can you solve `mediumSearch` in a short time? If so, we're either very, very impressed, or your heuristic is inconsistent.

## Solution

Below Code gets 2/4 on autograder but expands 12517 nodes in ~9" :

```
    return len(foodGrid.asList())
```

But below code gets 0/4 on autograder but expands 6126 nodes in ~3" :

```
    foodToEat = foodGrid.asList()
    totalCost = 0
    curPoint = position
    while foodToEat:
        heuristic_cost, food = \
            min([(util.manhattanDistance(curPoint, food), food) for food in foodToEat])
        foodToEat.remove(food)
        curPoint = food
        totalCost += heuristic_cost

    return totalCost
```

---

## Question 8 : Suboptimal Search

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. In this section, you'll 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
```

**Hint** 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.

## Solution

Path cost per algorithm :

1) DFS with 5324 path cost :

```
return search.dfs(problem)
```

2) BFS with 350 path cost :

```
return search.bfs(problem)
```

3) UCS with 350 path cost :

```
    #return search.ucs(problem)
```

4) DFS with 350 path cost :

```
    #return search.astar(problem)
```

In `class AnyFoodSearchProblem(PositionSearchProblem):` we have to add some code in `def isGoalState(self, state):` to find greedily the closest dot to eat (= goal).

***Implementation:***

```
        distance,goal = min([(util.manhattanDistance(state, goal), goal) for goal in self.food.asList()])
        if state == goal:
            return True
        else:
            return False
```


...The End...