بِسْمِ اللهِ الرَّحْمنِ الرَّحِيمِ

"In the name of Allah,
the entirely merciful, the especially merciful."

- Quran 1:1

HolyQuran.co

Project 1 - Search

By : Mojtaba Zolfaghari

mowjix@gmail.com

Professor : Dr.Ali Shakiba

http://alishakiba.ir

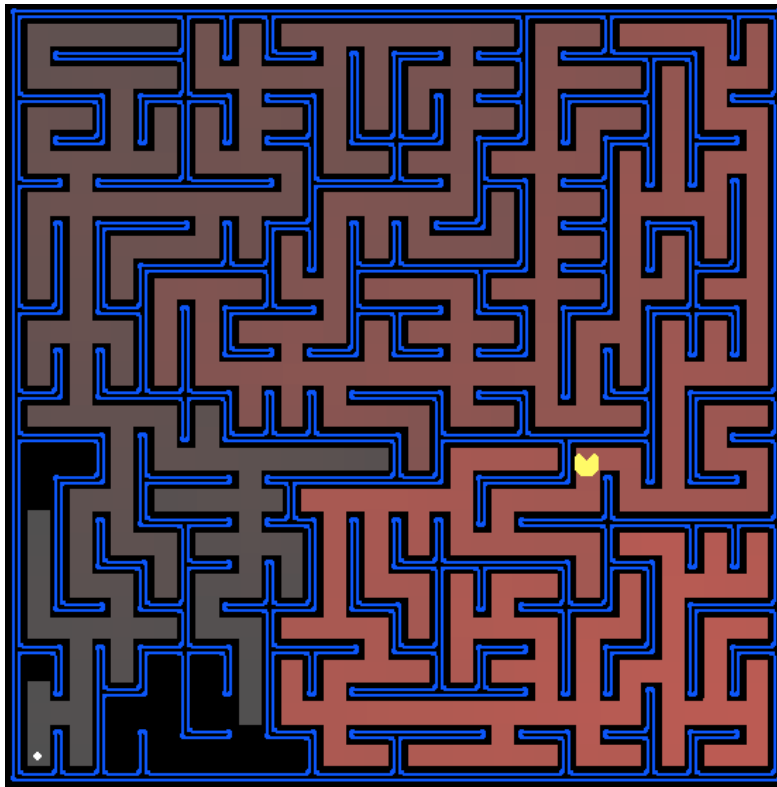Prepared in the Faculty of Mathematics and Computer Science

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 Updated: 04/21/2021.

All those colored walls,

Mazes give Pacman the blues,

So teach him to search.

Table of Contents

Title	Page

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.

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
```
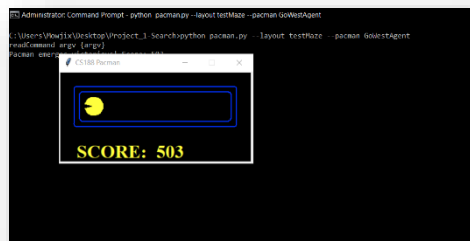
OUTPUT:

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
```

OUTPUT :

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

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

OUTPUT:

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 Here.

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
```

OUTPUT:

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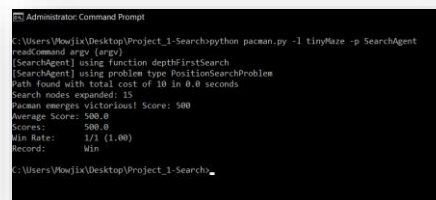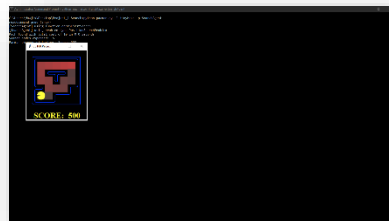
Hint If Pacman moves too slowly for you, try the option --frameTime 0.

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.
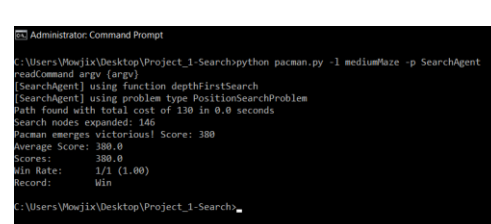
Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent
```

OUTPUT:


```
python pacman.py -l mediumMaze -p SearchAgent
```
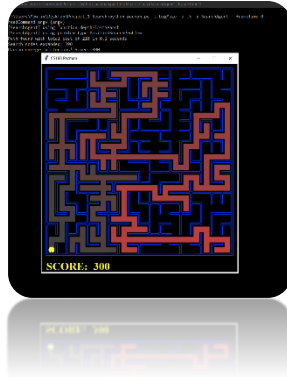
OUTPUT:

```
python pacman.py -l bigMaze -z .5 -p SearchAgent --frameTime 0
```

OUTPUT:

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 in the end of the 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 finally 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()
```
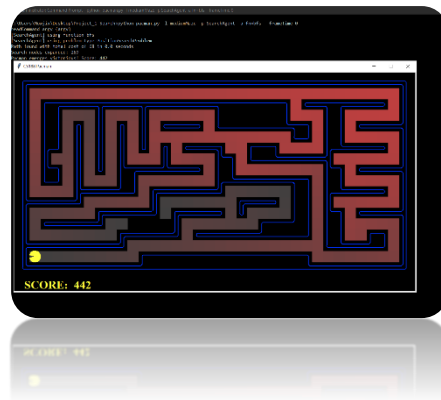
Question 2 : Breadth First Search

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 --frameTime 0
```
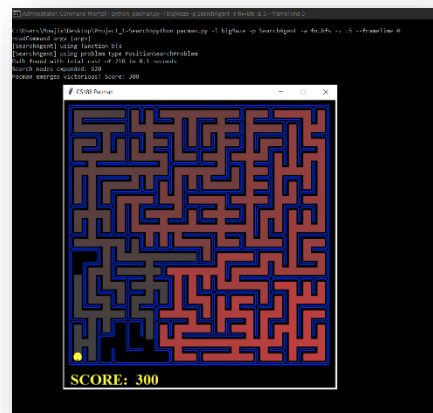
OUTPUT:


```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime 0
```

OUTPUT:
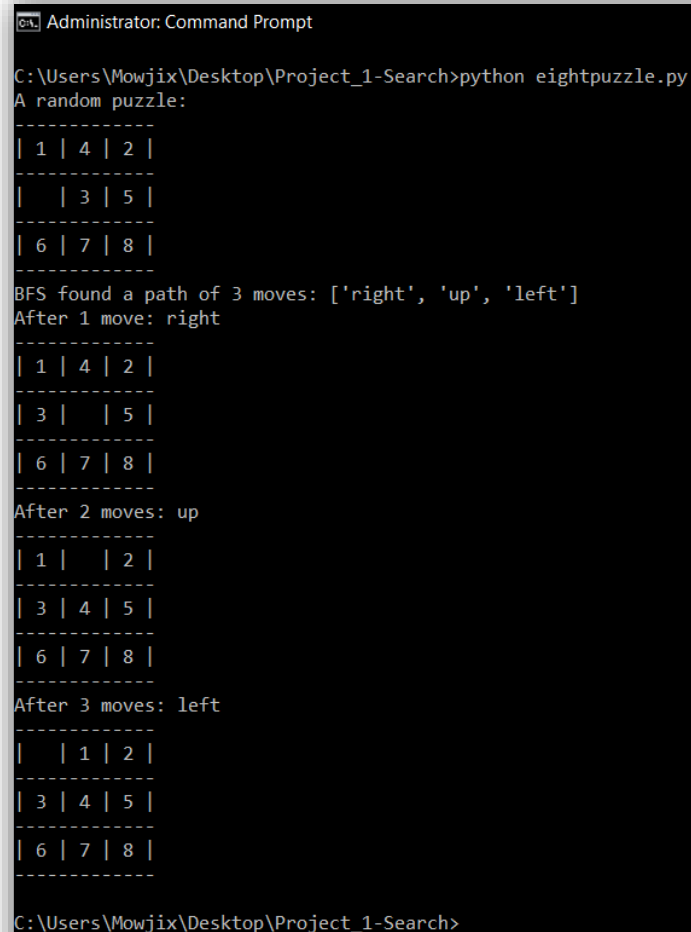

Does BFS find a least cost solution? If not, check your implementation.

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
```

OUTPUT:


```
Administrator: Command Prompt

C:\Users\Mowjix\Desktop\Project_1-Search>python eightpuzzle.py
A random puzzle:
-------------
| 1 | 4 | 2 |
-------------
|   | 3 | 5 |
-------------
| 6 | 7 | 8 |
-------------
BFS found a path of 3 moves: ['right', 'up', 'left']
After 1 move: right
-------------
| 1 | 4 | 2 |
-------------
| 3 |   | 5 |
-------------
| 6 | 7 | 8 |
-------------
After 2 moves: up
-------------
| 1 |   | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
After 3 moves: left
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------

C:\Users\Mowjix\Desktop\Project_1-Search>
```

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:

```python
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()
```
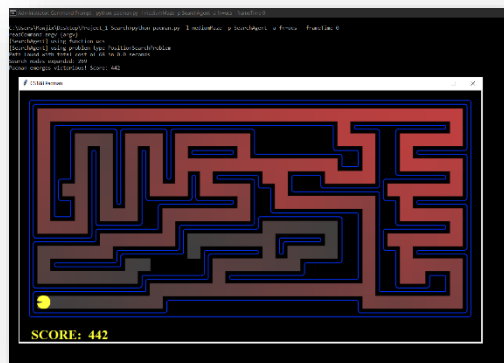
Question 3 : Varying the Cost Function

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 --frameTime 0
```
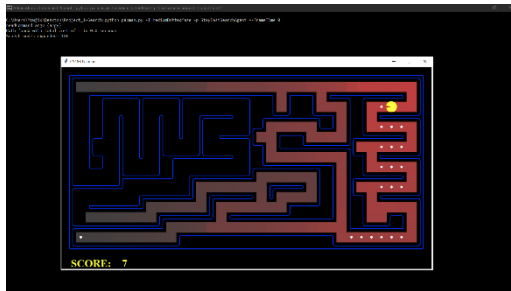
OUTPUT:

```
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent --frameTime 0
```

OUTPUT:


```
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent --frameTime 0
```

OUTPUT:
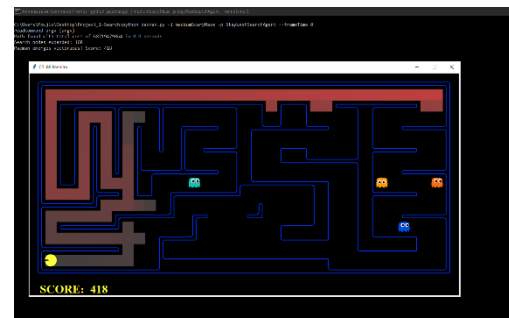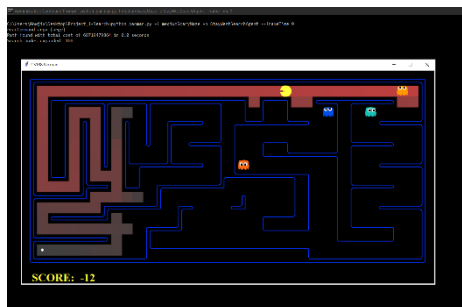
__Note:__ You should get very low and very high path costs for the StayEastSearchAgent and StayWestSearchAgent respectively, due to their exponential cost functions (see searchAgents.py for details).

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.

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 --frameTime 0
```

OUTPUT:

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 the-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()
```

...The End...