STMC coding team Training

Lesson 7: More On Problem Solving

Tsai Yun Chen

April 19, 2024

Goal today

Today we will continue to look into solving some real life problems using all the thing taught so far.

- Maze Game
- Maze Game with Shortest Path
- Maze Game with Weighted Shortest Path

A brief recap

- We solved two intersting problems last week
  - the Hanoi Tower
  - the (0-1) Knapsack Problem
- We learnt some techniques in solving those problems
  - Greedy Algorithm
  - Dynamic Programming
- Today we will see some more intersting application of these two techniques

The Maze Game

- Consider you are in a large field of $m \times n$ grids
- you standing at the top-left corner
- the exit is at the bottom-right corner
- There are obstacles on some of the grids
- you can only move right or down, one grid at a time
- Problem: Can you find the way out?

Task Description

- We would like to program an algorithm that tell you either the way out or there is no way out
- Input: The maze, expressed as a list of list
  - your position is marked as u, initially at the top-left corner
  - obstacles are marked as o, can 0, 1, or more than 1
  - exit is marked a e, initially at the bottom-right corner
- Output: The list of grids to be visited, or “NO WAY OUT” if there is no way to reach the end

First Trial

- Sometimes simple is the best
- Since we have at most two choices at each grid, we can try all of them
- Let's try to code it!

```python
def MazeSolver(maze):
  #the maze is expressed as a list of list
  ###your code start below###

#Then we ask the user to provide the number of pile to be solved
   , and call the function we just implemented.
SampleMaze = [['u',' ',' '],[' ','o',' '],[' ',' ','e']]
MazeSolver(SampleMaze)
```

- Problem: How long does it takes to try all? Let's say for a 3×3 field with obstacle in the middle.

Simple Improvement

- We are looking into the same grids more than once!
- How can we avoid that?
- Observation: If we can reach exit from a certain grid, then we must already returned the solution!
- So all visited grids have no solution, therefore no need to check again
- Idea: Create a boolean table, same as the size of the maze, to mark whether the grid has been checked or not
- Problem: How to implement it? Let's try!

Shortest Path Problem

- What we have just done is called the Depth-First Search (DFS), it's also kind of Greedy Algorithm
- Just as last time, it's simple to think, and simple to implement
- But… It's not optimal, also become slow when it becomes more complicated
- Let's assume the exit no longer stay at the bottom right corner
- Also we are now free to move in all four direction
- Consider now we are not only to find the way out, we want to find the shortest way possible
- What can we do?

What if there is more than one people?

- Let's assume, instead of only you, your friends are in the maze as well
- Another assumption, if one of you get out, all of you are free
- Can we do better?
- We can try multiple path at the same time!
- Problem: How can we simulate that in our computer?
- We can't really compute various things at the same time (actually we can, but that's hard)
- But … we can simulate that using looping

Looping with Visit Queue

- To achive that, we maintain a list named VisitQueue
- Initially the VisitQueue contains only the initial position
- At each loop, it is simulating a group of people on a certain grid
- Then as we need to send people to other grid
- We add all possible direction of walking to the end of the VisitQueue
- As soon as the loop visited the target, we are done

Implementation

```
def MazeSolver(maze):
  m = len(maze)
  n = len(maze[0])
  Visited = [[False * n]*m]
  VisitQueue = [(0,0)]
  for pos in VisitQueue:
    Visited[pos[0]][pos[1]] = True
    if maze[pos[0]][pos[1]] == 'e':
      return "Found the way out!"
    if pos[0]>0 and not Visited[pos[0]-1][pos[1]] and maze[pos[0]-1][pos[1]] != 'o':
      VisitQueue.append((pos[0]-1,pos[1]))
    if pos[1]>0 and not Visited[pos[0]][pos[1]-1] and maze[pos[0]][pos[1]-1] != 'o':
      VisitQueue.append((pos[0],pos[1]-1))
    if pos[0]<m-1 and not Visited[pos[0]+1][pos[1]] and maze[pos[0]+1][pos[1]] != 'o':
      VisitQueue.append((pos[0]+1,pos[1]))
    if pos[1]<n-1 and not Visited[pos[0]][pos[1]+1] and maze[pos[0]][pos[1]+1] != 'o':
      VisitQueue.append((pos[0],pos[1]+1))
  return "No Way Out!"
```

Challenge I

- The code in checking whether we can append a certain position into the VisitQueue is quite annoying and redundant
- Can you write a simple function and replace those lines with the function?
- Design your own function, include naming and parameter needed
- try to make it as simple as possible

Challenge II

- Recall in the original version of the task, we need to obtain the path to the exit
- Here we only answer whether there is a way out
- Can you modify the code so that it also return the code whenever the exit is reachable?
- Idea: Instead of a single position, maintain a list of position in the queue

Challenge III

- Now consider the exit is locked
- You need to find the key before moving to the exit, which is marked as ’k’ in the field
- How can we find the new shortest path for this?

Challenge IV

- Continue with the previous setting, the exit is locked
- Now the floor of each grid will disappear after you walked over it, there is lava below it
- This means you cannot walk through the same grid twice
- Now how do you find the new shortest path?