A* Pathfinding Algorithm for Mario Game

```
By Pranshu Acharya
```

This Notebook implements the A* pathfinding algorithm to find the shortest path between two points on a grid-based map.

Requirements

1. Python
2. Jupyter
3. heapq
4. random

```
pip install notebook #For jupyter notebook
```

Levels

The levels are stored in the "*levels*" folder where I have created some levels. These are labelled as level1.txt, level2.txt, etc. Levels other than those can be added but needs to follow the format like below:

Example of level1.txt

```
s . . o .
p o . p o
. . . . .
o p . o .
. . p . g
```

Difference between Deterministic and Probabilistic

The difference between the two is in the ==mapPipes()== function.

1. Deterministic: The user is asked for the pipe destinations.
2. Probabilistic: The pipe destinations are assigned randomly.

Assumptions

1. If mario goes to a pipe, he has to enter the pipe. So an appropriate heuristic would be the heuristic(Manhattan distance to the goal) of the destination pipe.
2. If mario is teleported to a pipe, he cannot reenter the same pipe immediately.

Problem

- Suppose pipe distance of P1 -> Goal = 7
- Suppose pipe distance of P2 -> Goal = 2
- Suppose pipe distance of P3 -> Goal = 7
  - Mapings: P1->P2 ; P2->P3
  - Heuristics: P1 = 2; P2 =7
- However, if we are exiting from P2, our heuristic should be 2 *NOT* 7.

Solution

Keep track of how you reached the pipe; if you reached the pipe from a pipe (teleported) or a cell (walked).

How To Run

1. Extract all files including the 'levels' folder and open the terminal in the folder where the ipynb is.

2. Open the notebook as a jupyter notebook.

   ```
   jupyter notebook
   ```

3. Select the program you want to run.

4. In the loadLevel() cell, you can change the level if you want. There are 5 test levels labelled as level1,level2,...level5.

   ```
   with open('levels/level3.txt','r') as file:
   #Change this line to select your desired level.
   ```

5. Run all the cells and go to the ==Main Method== at the end. There input the desired pipe destinations when asked. Example:

   ```
   What should be the destination for pipe (1, 0) :3,1
   ```

Functions Explanation

1. loadLevel()

```
Reads the level layout, removes whitespaces and special characters (\n)
Returns the level as a 2d array.
```

2.getStartGoal

```
Searches the loaded level array and find the start ('s') and goal ('g') positions.
Returns these positions as a list [start, goal].
```

3. getPipePositions

```
Finds and stores the positions of pipes ('p') in the level array.
Returns a list of tuples, where each tuple is the (row, column) of a pipe.
```

4. mapPipes

- Deterministic:

  ```
  Asks the user the destination of each pipe.
  The user inputs as row,column; which are stored in a dictionary 'p_dict'.
  Returns p_dict.
  ```

- Probabilistic:

  ```
  A random pipe destination is assigned to each pipes, which are stored in a dictionary 'p_c
  Returns p_dict.
  ```

5. calculateHeuristic

```
Returns heuristic values for each cell as a 2d array.
```

6. getNeighbour

```
Finds the neighbors a cell and adds them to the list of neighbors.
If the current cell is a pipe,only includes the destination pipe as a neighbor.
Returns the list of neighbors.
```

7. getMoveCost

```
Determines the cost of moving from the current cell to a neighbor.
```

8. isWalkable

```
Check if a cell is not an obstacle and is within the level.
Returns True or False.
```

9. a_star

```
Implementation of A*.
```

10. reconstructPath

```
Reconstructs the path from a disctionary.
Starts from the goal and traces back to the start, appending each cell to the path list.
```

Code Explanation

1. First we load the level by reading a text file.

```
with open('levels/level.txt','r') as file:
    for line in file:
    c=line.split(' ')
```

2. We then find the start and goal positions. We do this by simply scanning through the level until we find the 's' and 'g' values and storing its positions.

```
if(x=='s'):
        start=(level.index(y),y.index(x))
elif(x=='g'):
        goal=(level.index(y),y.index(x))
```

3. Like storing the start and goal, we also store the positions of 'p' / pipes.

```
if(level[i][j] == 'p'):
        position.append((i,j))
```

4. Two cases:

   - Deterministic: We ask the user for the the destination of all pipes. The user will input in the format *row, column*. We then store the source and destination of the pipe in a dictionary.

```
value = input("What should be the destination for pipe " + str(pos) + " :")
spl=(value.split(','))
tpl=[]
for a in spl:
    tpl.append(int(a))
p_dict[pos] = tuple(tpl)
```

   - Probabilistic: We assign the destination of all pipes randomly. We then store the source and destination of the pipe in a dictionary.

```
p_dict = {}
available_pipes = list(positions)  #Creating a list of available pipe positions

for pos in positions:
#Randomly selecting another pipe position from the list
    dest_pos = random.choice(available_pipes)
    p_dict[pos] = dest_pos
    available_pipes.remove(dest_pos)
```

5. We calculate the heuristic value for all the cells in the level.

   - It uses the Manhattan distance from each cell to the goal ('g') as the heuristic value.

```
if level[y][x] == '.':
        heuristic_values[y][x] = abs(x - goal_position[0]) + abs(y - goal_position[1]
```

   - For pipe cells ('p'), it calculates heuristic values based on the destination pipe, considering previous pipe heuristics.

```
if level[y][x] == 'p':
    temp = (y, x)
    destination = p_dict.get(temp)
    if destination:
        b, a = destination  #Destination position
        if previous_pipe_heuristic is not None:
            heuristic_values[y][x] = previous_pipe_heuristic  # Use previous pipe's he
        else:
            heuristic_values[y][x] = abs(int(a) - goal_position[0]) + abs(int(b) - go
```

6. The neighbors are then calculated to determine where to go next.

   - If the current cell is a pipe, the only neighbor it has is the destination (assumption). (In the next iteration, the neighbor's adjacent cells are added)

     ```
     if level[y][x] == 'p':
             pipe_destination = p_dict.get(current)
             if pipe_destination:
                 nbrs = [pipe_destination]
     ```

   - If the current cell is not a pipe, then the adjacent cells are its neighbors.

     ```
     if isWalkable(y + 1, x) and y + 1 < len(level):
         nbrs.append((y + 1, x))
     if isWalkable(y - 1, x) and y - 1 >= 0:
         nbrs.append((y - 1, x))
     if isWalkable(y, x + 1) and x + 1 < len(level[0]):
         nbrs.append((y, x + 1))
     if isWalkable(y, x - 1) and x - 1 >= 0:
         nbrs.append((y, x - 1))
     ```

7. We get the Move Costs. If the cell is an obstacle 'o', then the move cost is infinite. Else the move cost is 1.

   ```
   if cell_type == 'p':
       return 1  #Here the cost to enter = 1 and cost to exit = 1 making it 2 in total
   elif cell_type == 'o':
       return float('inf')  # Infinite cost for obstacles
   else:
       return 1  #Default cost for other cells
   ```

8. We use a priority queue that stores: (f_score,cell) and is always sorted based on the lowest f_score.

   ```
   open_set = []  #Priority queue for open nodes
   heapq.heappush(open_set, (0, start))
   ```

   - It pops the cell with the lowest f_score from the queue and assigns it to current.

     ```
     f, current = heapq.heappop(open_set)
     ```

   - It gets the neighbours and calculates the f_score for them and insertes into the priority queue.

```
f_score = g_score[neighbor] + heuristic[neighbor[0]][neighbor[1]]
heapq.heappush(open_set, (f_score, neighbor))
```

- This process is repeated until we reach 'g'.

```
if current == goal:
    return reconstructPath(came_from, current), cost_so_far[goal]
```

9. While changing the current from a cell to another, we keep track of where each cell comes from in a dictionary. This is then used to reconstruct the path.

```
came_from[neighbor] = current
```

```
path = [current]
while current in came_from:
    current = came_from[current]
    path.append(current)
path.reverse()
```

10. Lastly, we use the same dictionary to print a '*' indicating the path.

```
for i in range(len(level)):
    for j in range(len(level[i])):
        if (i, j) == start:
            print("S", end=" ")
        elif (i, j) == goal:
            print("G", end=" ")
        elif (i, j) in path:
            print("*", end=" ")
        else:
            print(level[i][j], end=" ")
    print()
```