Nicholas Teresi, Pratik Mishra, Noble Ibe

A\* Search

This project was written in Python.

Part 0

## The Node class has 7 attributes. Value (1/0) represents a blocked or unblocked cell respectively. Point refers to the Node's coordinates.

## Parent denotes the parent cell of the Node. G is the cell's g value, h is the cell's h value, and s is the cell's search value. C is the cost

## to get to that cell.

class Node:

def \_\_init\_\_(self, value, point, parent, g, h, s, c):

self.value = value

self.point = point

self.parent = parent

self.g = g

self.h = h

self.s = s

self.c = c

## Compute path will compute a path from the given start node in the openset to the goal node. The tie\_desc variable will sort the openset in order

## of descending g values when True, so that when min() is called and there are multiple smallest f values, it will pick the first, which will also

## the largest g value. The rest of the code follows the given project pseudocode.

def computePath(maze, openset, closedset, goal, counter, tie\_desc):

if tie\_desc is True:

openset.sort(key = lambda o:o.g, reverse=True)

else:

openset.sort(key = lambda o:o.g)

while openset and goal.g > min(openset, key = lambda f:f.g + f.h).g + min(openset, key = lambda f:f.g + f.h).h:

curr = min(openset, key = lambda f:f.g + f.h)

openset.remove(curr)

closedset.append(curr)

for d in [[0, -1], [1, 0], [0, 1], [-1, 0]]:

neighbor = [curr.point[0]+d[0], curr.point[1]+d[1]]

if 0 <= neighbor[0] <= len(maze)-1 and 0 <= neighbor[1] <= len(maze)-1:

if maze[neighbor[0]][neighbor[1]] in closedset:

continue

if maze[neighbor[0]][neighbor[1]].s < counter:

maze[neighbor[0]][neighbor[1]].g = math.inf

maze[neighbor[0]][neighbor[1]].s = counter

if maze[neighbor[0]][neighbor[1]].g > curr.g+maze[neighbor[0]][neighbor[1]].c and maze[neighbor[0]][neighbor[1]].c < 2:

maze[neighbor[0]][neighbor[1]].g = curr.g+maze[neighbor[0]][neighbor[1]].c

maze[neighbor[0]][neighbor[1]].parent = curr

if maze[neighbor[0]][neighbor[1]] in openset:

openset.remove(maze[neighbor[0]][neighbor[1]])

openset.append(maze[neighbor[0]][neighbor[1]])

return openset

## This portion of code follows the Main() section of the given project pseudocode. When the openset becomes empty, then the problem is unsolvable. This

## function returns 0 if unsolvable, and 1 otherwise.

def aStar(start, goal, maze, heuristic, tie\_desc):

counter = 0

while start.point != goal.point:

counter += 1

start.g = 0

start.s = counter

goal.g = math.inf

openset = []

closedset = []

openset.append(start)

openset = computePath(maze, openset, closedset, goal, counter, tie\_desc)

if not openset:

print('Unsolvable')

return 0

path = []

curr = goal

while curr.point != start.point:

path.append(curr)

curr = curr.parent

path = path[::-1]

prev = curr

for p in path:

if maze[p.point[0]][p.point[1]].value == 0:

prev = p

continue

else:

maze[p.point[0]][p.point[1]].c += 1

break

start = prev

return 1

## This is the heuristic function which will create a secondary grid of only heuristic values, matching the main maze grid.

def ManhattanHeuristic(dim, maze):

heuristic = [[0] \* dim for i in range(dim)]

for i in range(len(maze)):

for j in range(len(maze[i])):

heuristic[i][j] = abs(dim-1-i)+abs(dim-1-j)

## for row in heuristic:

## print(' '.join([str(elem) for elem in row]))

return heuristic

Part 1

a) The thresshold where the majority of enviornments are solvable seems to be p <.25, and p > .25 seems to have majority unsolvable.

b) Best first search could give faster results since it does not take consideration the distance from the source to the current node. Therefore it can find a shortest path, though not the optimal path, whereas A\* would look for another path once it figures out it is not on the optimal path

Part 2

a)

|  |  |
| --- | --- |
| Heuristic | Runtime |
| Manhattan | 0.6554 s |
| Euclidian | 1.4518 s |
| Chebyshev | 1.4795 s |

Since the environment is a gridworld with uniform action costs, it makes perfect sense that Manhattan distance would yield the besst results.

b) Since h1 and h2 are both consistent heuristics, either the max or the min will also be consistent since there are only the two consistent heuristics to choose from in the first place.

Part 3

a) Tie-Breaking with higher g-values leads to an average solve time of 75 seconds, while tie-breaking with lower values resulted in an average time of 83 seconds. Higher g-values might be better because it picks which paths have traveled the most distance, and are therefore more likely to reach the end goal sooner.

b)

Part 4

a) Forward A\* had an average runtime of 76 s while backward A\* had an average runtime of 65s. The difference between the two largely depends on the randomly generated grid layouts, where one layout can favor one algorithm over the other. Therefore, the difference in runtimes likely doesn’t mean much.

b)

0:= unblocked

1:= blocked

A:= Agent

T:= Target

P:= Estimated Path

X:= Agent hits block

Time 0 - Start

0 0 0 0 0

0 0 1 0 0

0 0 1 1 0

0 0 1 1 0

0 0 T 1 A

Time 1 – Estimated Path

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 T P A

Time 1 – Path Check

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 T X A

Time 2 – Estimated Path

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 P P P

0 0 T 1 A

Time 2 – Path Check

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 X A

0 0 T 1 0

Time 3 – Estimated Path

0 0 0 0 0

0 0 0 0 0

0 0 P P P

0 0 P 1 A

0 0 T 1 0

Time 3 – Path Check

0 0 0 0 0

0 0 0 0 0

0 0 0 X A

0 0 0 1 0

0 0 T 1 0

Time 4 – Estimated Path

0 0 0 0 0

0 0 P P P

0 0 P 1 A

0 0 P 1 0

0 0 T 1 0

Time 4 – Path Check

0 0 0 0 0

0 0 X A 0

0 0 0 1 0

0 0 0 1 0

0 0 T 1 0

Time 5 – Estimated Path

0 P P P 0

0 P 1 A 0

0 P 0 1 0

0 P 0 1 0

0 P T 1 0

Time 5 – Path Check

0 0 0 0 0

0 0 1 0 0

0 0 0 1 0

0 0 0 1 0

0 0 A 1 0

Success

c)

S 0 0 1 0

0 1 1 0 0

0 0 1 0 0

0 1 0 1 0

0 0 0 0 G

S:= start state

G:= goal state

1:= blocked cell

0:= unblocked cell

In this scenario, using the Manhattan heuristic, the algorithm will first try to traverse the grid going right. However, since the right path leads to a dead end, the algorithm will have to backtrack through the start state to reach the goal, yielding a shortest path, but not an optimal path.