IT University of Copenhagen, 2023

Foundations of Game AI

Exercise Session 3 (walkthrough)

Adjusting our game for the algorithms

Because both Dijkstra and A* are graph traversing algorithms, we need to modify our node system so that it can be easily traversed. We will define various functions that will make our job easier.

getListOfNodesVector()

Returns a list of all nodes in vector (i.e. xy coordinates) form.

getVectorFromLUTNode()

Given a node in its Look Up Table form (remember that the LUT was implemented in the first exercise session), this function returns the node in vector form.

getNeighborsObj()

Given a node in vector form, returns a list containing the node's neighbours in LUT form. If there is no neighbour in a direction, None is inserted in the list in its place.

getNeighbors()

Similar to *getNeighborsObj()*, but the neighbours returned are in vector form.

getNodes()

Returns a dictionary in which each of the nodes is a key, and each key has as value a 4-element list where each element corresponds to one of the four possible directions: if the node HAS a neighbour in that direction, the element is 1; if the node DOESN'T HAVE a neighbour in that direction, the element is None.

The dictionary looks like the following:
{node1 : [neighbour_up, neighbour_down, neighbour_left, neighbour_right],

node2 : [neighbour_up, neighbour_down, neighbour_left, neighbour_right],

node3 : [neighbour_up, neighbour_down, neighbour_left, neighbour_right],

… }

Where neighbour_X is a 1 if there is a neighbour in one direction or None if there isn't.

Dijkstra's Algorithm

Navigate to "algorithm.py". The function takes two arguments: "nodes" and "start_node". "nodes" is the node system we create in the "run.py" when starting the game, while "start_node" is the node from which we'll start the search.

```
3  def dijkstra(nodes, start_node):
```

The first thing we do is create the dictionary explained before: we will cast it into a list, so that we only keep the keys (i.e. the nodes). This will be our list of unvisited noted.

```
4      unvisited_nodes = list(nodes.costs)
```

We also initialise two empty dictionaries:

- "shortest_path"
  - to keep track of best-known cost of visiting each city in the graph starting from the "start_node". In the beginning, the cost starts at infinity, but we'll update the values as we move along the graph
- "previous_nodes"
  - to keep track of the current best known path for each node

```
5      shortest_path = {}
6      previous_nodes = {}
```

Afterwards, for each node we create an entry in the "shortest_path" dictionary, where the key is the node and the value is infinity (or, actually, the system's maximum possible value), except for the "start_node" whose shortest path to itself has 0 as value.

```
8      max_value = sys.maxsize
9      for node in unvisited_nodes:
10         shortest_path[node] = max_value
11     shortest_path[start_node] = 0
```

Now we can start the algorithm. Remember that Dijkstra's algorithm executes until it visits all the nodes in a graph, so we'll represent this as a condition for exiting the while-loop.

```
13     while unvisited_nodes:
```

Now, the algorithm can start visiting the nodes. We start by finding the node with the lowest value.

```
13      while unvisited_nodes:
14          current_min_node = None
15          for node in unvisited_nodes:
16              if current_min_node == None:
17                  current_min_node = node
18              elif shortest_path[node] < shortest_path[current_min_node]:
19                  current_min_node = node
```

Once that's done, the algorithm visits all node's neighbours that are still unvisited. If the new path to the neighbour is better than the current best path, the algorithm makes adjustments in the "shortest_path" and "previous_nodes" dictionaries.

```
21      neighbors = nodes.getNeighbors(current_min_node)
22      for neighbor in neighbors:
23          tentative_value = shortest_path[current_min_node] + 1 #nodes.value(current_min_node, neighbor)
24          if tentative_value < shortest_path[neighbor]:
25              shortest_path[neighbor] = tentative_value
26              # We also update the best path to the current node
27              previous_nodes[neighbor] = current_min_node
```

After visiting all of its neighbors, we can mark the current node as "visited" by popping it off the list of unvisited nodes.

```
29              # After visiting its neighbors, we mark the node as "visited"
30              unvisited_nodes.remove(current_min_node)
```

After the algorithm terminates its execution, we return the two dictionaries:

```
32      return previous_nodes, shortest_path
```

Helper function

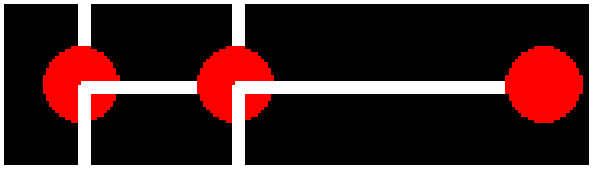
Lastly, we need to create a function that prints out the results. This function will take the two dictionaries returned by the "dijskstra_algorithm" function, as well as the names of the beginning and target nodes. It'll use the two dictionaries to find the best path and calculate the path's score.

```
35  def print_result(previous_nodes, shortest_path, start_node, target_node):
36      path = []
37      node = target_node
38
39      while node != start_node:
40          path.append(node)
41          node = previous_nodes[node]
42
43      # Add the start node manually
44      path.append(start_node)
45
46      print("We found the following best path with a value of {}.".format(shortest_path[target_node]))
47      print(path)
```

Home exercise (optional)

On line 23, we used the “+1” at the end to specify that all edges to neighbouring nodes have weight 1.

This, however, might not necessarily give the best result because there are situations in which different values might give a better result, like in the case shown below:



As an exercise, try to think of another way of implementing the different weights values into the Dijkstra’s algorithm.

A* Algorithm

We implement a Manhattan heuristic. Given two nodes in vector form, we simply compute and return the sum of the squares of their components.

```
51  # A*
52  def heuristic(node1, node2):
53      # manhattan distance
54      return abs(node1[0] - node2[0]) + abs(node1[1] - node2[1])
```

We then implement the same Dijkstra algorithm, with the difference that on line 78 we also add the computed heuristic, passing the current minimum node and the analysed neighbour as parameters.

```
77      if a_star:
78          tentative_value = shortest_path[current_min_node] + heuristic(current_min_node,neighbor)
```

Home exercise (optional)

In this implementation we used the Manhattan heuristic. Try changing the code so that we use the Euclidian distance hieuristic instead.

Can you think of how this would impact the performance of the algorithm in our PacMan game?

Which hieuristic do you think would yield the best results, and why?

Implementing the algorithms in the game

getDijkstraPath()

Navigate to “ghost.py”. The method “getDijkstraPath()” computes the target node (node that Pacman is travelling to) and the start node (node that ghost is travelling to).

```
27      def getDijkstraPath(self, directions):
28          lastPacmanNode = self.pacman.target
29          lastPacmanNode = self.nodes.getVectorFromLUTNode(lastPacmanNode)
30          ghostTarget = self.target
31          ghostTarget = self.nodes.getVectorFromLUTNode(ghostTarget)
```

We then run the Dijkstra algorithm, storing the returned dictionaries.

```
34      previous_nodes, shortest_path = dijkstra_or_a_star(self.nodes, ghostTarget, a_star=False)
```

We then implement the same logic used for the “print_result()” function, and reverse the obtained list at the end. We then return the computed path.

```
35      path = []
36      node = lastPacmanNode
37      while node != ghostTarget:
38          path.append(node)
39          node = previous_nodes[node]
40      path.append(ghostTarget)
41      path.reverse()
42      # print(path)
43      return path
```

goalDirectionDij()

This is the method that, given the valid directions, executes the method we just implemented and determines which direction to navigate to next.

It starts by computing the path to the target.

```
48              path = self.getDijkstraPath(directions)
```

We then append the current ghost’s target to the list of nodes that lead to the target (since our character hasn’t reached that target yet).

We then examine the second node in the list and determine in which direction the character has to turn (from the ghost’s target node) to reach the second node in the list.

```
50              ghostTarget = self.target
51              ghostTarget = self.nodes.getVectorFromLUTNode(ghostTarget)
52              path.append(ghostTarget)
53              nextGhostNode = path[1]
54              if ghostTarget[0] > nextGhostNode[0] and 2 in directions : #left
55                  return 2
56              if ghostTarget[0] < nextGhostNode[0] and -2 in directions : #right
57                  return -2
58              if ghostTarget[1] > nextGhostNode[1] and 1 in directions : #up
59                  return 1
60              if ghostTarget[1] < nextGhostNode[1] and -1 in directions : #down
61                  return -1
```

When the character is extremely close to the target (i.e. both characters have the same node as target), this implementation is likely to crash due to not having any second node in the list (there is just the target node). This situation only arises in this particular situation.

As such, one simple solution is to simply look at pacman’s direction, and use its reverse for the ghost. This will lead the ghost to take the direction that will make it collide with pacman.

```
62          else:
63              print(self.pacman.direction)
64              print(directions)
65              if -1 * self.pacman.direction in directions:
66                  return -1 * self.pacman.direction
67              else:
68                  return choice(directions)
```