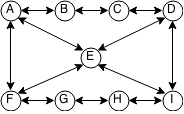
Exercise – Dijkstra’s Shortest Path

This path finding algorithm is ideal for finding the closest node from a list of potential end nodes. For example, a Dijkstras pathfinding algorithm could be used to find the closest health pack within a game where the location of the health packs are unknown.

The Dijkstras pathfinding algorithm is a slight modification to the classic Breadth First Search (BFS) with a focus on distance, rather than degrees of separation.

Given the following graph  
  
let’s use a BFS and Dijkstra algorithm to find a path from node ‘A’ to node ‘D’

The BFS will focus on getting to the target via the least number of traversed nodes, whereas Dijkstras will focus on getting to the target with the lowest cost, where cost comes from the links between the nodes, also known as the edges.

|  |  |
| --- | --- |
| BFS | DIJKSTRA |
| C:\Users\Administrator\Downloads\Untitled Diagram (1).png | C:\Users\Administrator\Downloads\Untitled Diagram (2).png |

What’s the Cost?

When we traverse from one node to another node we need to calculate a total running cost from the start node. For us, the cost to traverse between nodes will usually be represented as a distance, but sometimes edges have additional costs, such as a cost for traveling through a swamp compared to a cost for traveling along a road.

This cost is used to calculate the “G” score for each node when it is processed and added to the Open List. For example, Node C’s GScore will be B’s GScore + the cost of traveling from B to C.

Exercise:

For this session you are to implement Dijkstra’s Shortest Path algorithm.

You will need to create classes to represent a graph structure. For this you will most likely need a Node class and an Edge class.

Edges will contain a cost value, and point to a Node that that link to. They do not need a starting node as Edges are typically one way. For example, an Edge might lead through a portal, but the portal might not lead back, so the Edge is one way. For example:

struct Edge {

Node\* target;

float cost;

};

Nodes will contain a collection of Edges to the Nodes it connects to. For traversal purposes it is useful if the Node contains a pointer to the Node that lead to this node, and a total cost from the start Node in a search to the current node. Although not necessarily needed, a Node might have some form of location, which may be a position. For example:

struct Node {

Vector2 position;

float gScore;

Node\* parent;

std::vector< Edge > connections;

};

With the above Node structure, the parent should be set to nullptr and gScore set to 0 for all nodes before a Dijkstras search is performed.

As a node is added to the open list, its parent should be set to the “currentNode” and the gScore should be calculated. See Pseudo code below:

Procedure dijkstrasSearch(startNode, endNode)

Let openList be a List of Nodes

Let closedList be a List of Nodes

Set startNode.parent to null

Add startNode to openList

While openList is not empty

Sort openList by Node.gScore

Let currentNode = first item in openList

// Process the node, do what you want with it. EG:

if currentNode is endNode

break out of loop

remove currentNode from openList

Add currentNode to closedList

for all connections c in currentNode

Add c.target to openList if not in closedList

c.target.gScore = currentNode.gScore + c.cost

c.target.parent = currentNode

// Calculate Path, in this example as positions

Let path be a Stack of Vector2

Let currentNode = endNode;

While currentNode is not null

Add currentNode.position to path

currentNode = currentNode.parent

Return path

Implement the above pseudo code.