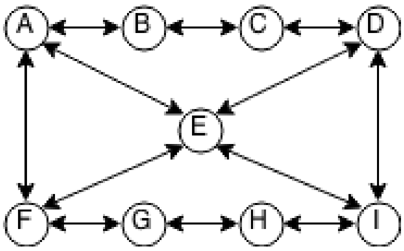
Exercise – Dijkstras Shortest Path

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 Breadth First Search with a focus on distance, rather than degrees of separation.

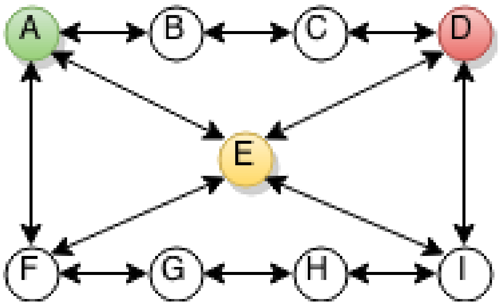
Given the following graph



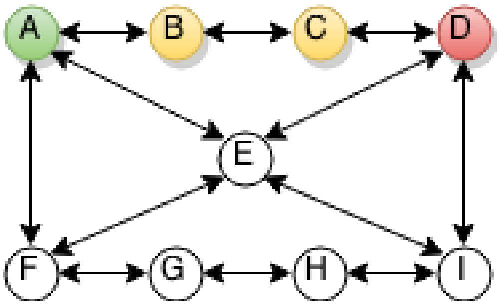
let's use a BFS and Dijkstra algorithm to find a path from node 'A' to node 'D'

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.

BFS



DIJKSTRA



What's 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.

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.

Sorting the Open list

In the last exercises for BFS, we modified the algorithm to sort the Open List by degrees of separation. For Dijkstras, instead, we need to sort the open list by GScore.

Calculating a path to the end Node

When we remove a node from the open list, we need to check if it's one of our potential end nodes. If it is, we can break out of the loop. We can then record the position of each node by backtracking through each nodes parent until we get to the start node. (See pseudo code below)

Modifying the Graph Node and Edges Structure

```
struct Node
{
	Vector2 pos;

	float gScore;
	Node *parent;

	std::vector< Graph::Edge > connections;

	// default constructor
	Node() : pos(0,0), gScore(0), parent(nullptr) {}

	// overloaded constructor
	Node(Vector2 a_pos) : pos(a_pos), gScore(0), parent(nullptr)  {}

	// overloaded constructor
	Node(Vector2 a_pos, float a_gScore, Node *a_parent) :
	     pos(a_pos), gScore(0), parent(a_parent)  {}

};
```

```
struct Edge
{
	Node *connection;

	float cost;

	// default constructor
	Edge () : connection(nullptr), cost(0) {}

	// overloaded constructor
	Edge (Node a_connection) :
		connection (a_ connection), cost(0){}

	// overloaded constructor
	Edge (Node a_connection, float a_cost) :
		connection(a_ connection), cost(a_cost){}

};
```

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 FindPathDijkstras(startNode, List of potentialEndNodes)

    Let openList be a List of Nodes
    Let closedList be a List of Nodes

    Let endNode be a Node set 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 one of the potentialEnd
            endNode = currentNode
            break out of loop

        remove currentNode from openList
        Add currentNode to closedList

        for all connections c in currentNode
            Add c.connection to openList if not in closedList
            c.connection.gScore = currentNode.gScore + c.cost
            c.connection.parent = currentNode


    // Calculate Path
    Let path be a List of Vector2
    Let currentNode = endNode;
    While currentNode != NULL
        Add currentNode.position to path
        currentNode = currentNode.parent
```

Exercise 1:

Implement the above pseudo code into your graph class within a function called “FindDijkstrasPath”

As an optional addition, consider using templates to define graph data type.

See below for the basic Graph Class Interface.

```
class Graph
{
public:

	struct Edge
	{
		/* insert Edge stuff */
	};

	struct Node
	{
		/* insert Node stuff */
	};

	// Default Constructor
	Graph();

	// destructor
	virtual ~Graph();

	// Add's a node at the given location, return the created node.
	Node *AddNode( float xPos, float yPos );

	// this function connects the 2 nodes by adding an edge
	// cost is automaticly set to the distance between n1 and n2
	void AddConnection( Node *n1, Node *n2);

	// Searches the graph starting from the "start" node untill one of
	// the "potential end node's" are found.
	// the resulting path is added to the "outPath" list.
	void FindDijkstrasPath(Node *start,
	                       const std::list<Node *> &potentialEndNodes,
	                       std::list<Node *> &outPath );

	// Helper function, populates “outNodes” with nodes that are within
	// a circular area (xPos, yPos, radious)
	void FindNodesInRange(std::vector<Node *> &outNodes,
	                          float xPos, float yPos, float radious);


};
```

Exercise 2

The problem with the above implementation is that our Path finding algorithm and Graph data are tightly coupled. It's entirely possible to define a graph which does not require pathfinding, or will require numerous different algorithms. For example, there will be scenarios where a Dijkstras pathfinding algorithm will be required and other times an AStar pathfinding algorithm may be required.

In addition, with this implementation, it's imposable to preform 2 independent pathfinding operations over the same graph at the same time. Our current implementation requires that the graph data be reset before preforming a second path find.

The solution to this problem is to create a "PathFinder" class. This class should remove all the pathfinding information from the graph nodes and edges and record that information separately. This will enable us to preform multiple Pathfinding operations at the same time over the graph, without needing to reset the graph data between searches.

Consider the following for reference on how you can separate the graph data and Pathfinding algorithms.

```
// create an instance of our graph
Graph *pGraph = new Graph();

// TODO: populate the graph with nodes and edges.

Graph::Node *startNode; // TODO: assign to a starting node

// TODO: add an end node to this list
std::list< Graph::Node * > potentialEnd;

// this list will contain our found path
std::list< Graph::Node * > path;

//-----------------------------------------------------------------------------
// as you can see our pathfinding information is seporate from the graph
//-----------------------------------------------------------------------------
Pathfinder *pPathfinder = new Pathfinder();
pPathfinder->Dijkstras( startNode, potentialEnd, path ); // returns when path is found.

// don't forget to delete the pGraph and pPathfinder objects when done…
```

The path finder class still needs to record information for each node as adds and removes nodes from the open list and closed list. To keep that data separated from our graph class, create a separate "node" class for storing this data, with a pointer to the node in the graph.

Continues on next page

```
class PathFinder
{
public:

	// constructor
	PathFinder();

	// destructor
	virtual ~PathFinder();

	// the path finding function that we had in our graph class previously.
	void Dijkstras(	Graph::Node *start,
				const std::list<Graph::Node *> &potentialEnd,
				std::list<Graph::Node *> &outPath );
private:

	struct Node
	{
		// pointer to the origional node in the graph
		Graph::Node *pNode;

		// information required for pathfinding
		BFS_Node	*pParent;
		int		 degreesOfSeporation;
		float		 gScore;
	};
};
```

Your task is to decouple the Pathfinding algorithm from the Graph object.