mmSearch Data Structures

Node class represents a node on the graph and has a node id, a disk id that it belongs to, a 3d position and list of node links
NodeLink class represents a link between 2 nodes, it has a source node, a destination node, a disk for the source node, and a weight.

NodeSearchData class represents the data required to perform the search algorithms for meet in the middle.
It has the following properties.
SearchData start - This is the search data used when searching from the start node to find the meeting node.
SearchData end - This is the search data used when searching from the end node to find the meeting node.

SearchData class contains the data required to perform the search algorithms for Dijkstra or A*.
It has the following properties
visited - This is set to true when a node is on the closed list
path_node - This is the previous node that is on the path to reach this node. It is used for rebuilding the path.
given_cost - This is the g cost which is the sum of the link weights to get to this node in the optimal path.
heuristic - This is the euclidean distance between the node and the goal node (either start node or end node).
priority - This is just the sum of given_cost and heuristic used in the Open List for finding the next best node to look at.

UpdatablePriorityQueue
This is a priority queue that uses the index as the node id and the value as the priority cost.

std::deque
The double ended queue that contains the resulting path of the searches.

mmSearch Algorithm

Search for path
The mmSearch function takes two parameters, the start node and the end node ids. The algorithm will return the optimal path found between these nodes.

First the search data for each nodes is cleared and updated with the heuristic information which is the distance between the nodes position and the goal nodes position. This is done in both search directions, start to end and end to start. The function resetSearchDataWithHeuristics does this step.

The Open List 1 (queue_start) and 2 (queue_end) are defined as updatable priority queues. The index is the node id and the value is the priority (f = g + h)

The start node is added to open list 1
The end node is added to open list 2

While the open list 1 and open list 2 are not empty we will loop

In the loop we will peek at the nodes in both Open list 1 and open list 2.
Whichever has a lower priority we will pop that node off and it will become the current node.

the current node search data object will be the data of the node in the direction we are searching (from the start or from the end).
This search data will be marked as visited, which means it is on the closed List in that direction.
I also set a flag to remember which queue the node was popped off of for later use

Now we loop through every single node link of the current node

We grab the node link's search data depending on which queue we popped of off (queue_start or queue_end)
If this node was visited (is on closed list) in this direction we can skip it
If the given cost of the current node given cost + link node weight is less than the link nodes given cost then we have found a more optimal path.
If it is more optimal we will update given cost, priority, and path node (the path node for the linked node will be the current node we came from).
Then we update the correct priority queue, queue_start or queue_end depending on which queue we popped the current node from.

When the outer while loop breaks we have found a path.
We will mark the current node (which is the meet in the middle node) as visited from both directions. Meaning It is on both closed lists.

Now we rebuild the path.

Build the path
The getmmPath function takes a node start id, node meeting id, and node end id.

The path is a double ended queue.

starting at the meeting node id
we loop through the nodes in the starting node direction and look at the path node, this is the node we came from to get to this node.

These are the nodes that build the path.
They are pushed onto the front of the path until we encounter the starting node
So the middle node is pushed first and starting nodes are pushed last.
This builds the first half of the path.

Starting at the meeting node id again.
We loop through the nodes in the ending node direction and look at the path node.
They are pushed onto the back of the path until we encounter the end node.
We make sure not to push on the middle node twice.
The next node after middle is pushed on first and the end node is pushed on the back last.

The path has been rebuilt and it is returned.