Dykstra's Algorithm, Reading

This solution finds the *cheapest* route from any node in the network to any other. It's based on the *shortest* route solution, but the problem with *that* solution to *this* problem is mainly that when a node is reached for the first time by fanning out BFS-style, "*shortest*" calls that the minimum path to that node. That's because any alternate path that may be found later in the fanning out *cannot* be better. But not so with *weighted* edges, because a later solution with more steps could actually be better if those steps add up to be a lower cost. For example, two steps of cost 10 each is not as good as four steps of cost 3 each. In "*shortest*", two steps is better than four. But in "*cheapest*", cost 12 is better than cost 20.

Setting the **isVisited** tracking variable to true when a node is reached and ignoring it thereafter is not going to work. Instead, we'll have to also track the *cost* of getting to that node. And instead of ignoring it just because it's been visited, we can no longer ignore it, because if the next found path to the node from a different direction costs less, we **change our result** for the best way to that node.

That means the end of the process does not occur when the end node is reached for the first time. Instead, we have to clear the to-do list until there is nothing else to compare. It would be something like this:

```
for all nodes: set cost to zero, prev=-1, and mark all as not visited
create a queue of ints to store the "to-do" list
mark start node as "visited" and push it onto the "to-do" list
while the "to-do" list is not empty
  peek/pop a node's index from the "to-do" list -- call it the "node under consideration"
  for each of that node's neighbors
    ~~if neighbor has been visited, "continue;"~~
    ~~set neighbor's cost to 1+cost of node under consideration~~
    calculate "new cost" = cost to the node under consideration PLUS the neighbor's edge cost
    if neighbor's been visited AND the new cost is not less than the current cost, "continue;"
    set neighbor's cost to the new cost
    set neighbor's prev to the index of the node under consideration
    mark neighbor as visited
    push neighbor's index' into the "to-do" list
    ~~if neighbor node contains the index of the end city~~
      ~~empty the "to-do" list (so that the while-loop will end)~~
      ~~exit for-loop~~
the route's cost equals the end node's cost
build a stack of entries, starting from the end node, back towards the start
```

This works, but if you think about the effect of taking out that last if-statement block that ends the loop before the to-do list is done, it's *way* inefficient. We have to basically look at all possible routes. Given how fast Google and Apple and Garmin find directions, that's probably *not* how they do it.

Applying A Priority Queue

Instead of using a *queue* for the to-to list, how about if we use a *vector* -- then instead of blindly taking the first item in the list, what if we searched the list for the the index of a visited node with the lowest cost, and use that index for the next cycle of the while-loop. Then when that index matches the end node index, we have a solution, because any other solution will only cost more (because all the remaining nodes in the to-do list already cost more, and they still have more steps to get to the end).

That's better, but at the same time, we introduced a O(n) for-loop to search the to-do list.

We saw this before. Remember the service queue simulation that was the subject of two lab assignments? In the first version, we checked each server each minute to see if they were busy and if their service was ending in that minute -- O(n) where "n" was the number of servers. But in the second version, we maintained a lo-to-hi priority queue of end-of-service times and we just checked the top of that list each minute instead of "bothering" each server. We can apply similar time-saving logic here, by using a lo-to-hi priority queue for the cheapest route's to-do list!

That solves one problem, but leaves us with another, very subtle one -- the same index can be in the to-do list multiple times because of all the different ways to get to its corresponding node (because we took out the **if-continue** statement). They are ordered in the priority queue based on cost. But when ways to get to its corresponding node (because we took out the **if-continue** statement). They are ordered in the priority queue based on cost. But when a lower cost path to a node is found and updates the node, the priority queue would then be out of order. You may remember that the peek function in our priority queue template returns a *copy* of the front value and not a mutable reference, for just this same reason.

The solution is to create a new struct-based object to store the index *and* the cost of getting to it *and* the index preceding it in the path to it. Since the object will be used in a priority queue, an operator less-than function is needed. So we get this:

```
struct Terminus // the path to a node and the cost of that path
{
  int index; // the node's index in the database array
  int prev; // the index of the preceding node in the path
  double cost; // the cost of getting to index's node from prev's node
};

bool operator<(const Terminus& a, const Terminus& b)
{
  return b.cost < a.cost; // reverse logic for lo-to-hi
}
```

Dykstra's Algorithm

Now with this new object type for the priority queue, we can write **Dykstra's algorithm** (https://en.wikipedia.org/wiki/Dijkstra) :

```
reset cost to zero, prev=-1 for all database nodes, isVisited to false
create an STL priority queue of Terminus objects as the to-do list
create a Terminus object for the start node with prev=-1, cost=0
push the start node's Terminus object onto the priority queue to-do list
while the priority queue to-do list is not empty
  peek/pop a Terminus object from the priority queue to-do list
  if contained object's been visited, "continue;"
  mark contained object as visited
  set its cost to that of its Terminus
  set its prev to that of its Terminus
  if Terminus' node is the end node, exit while-loop
  for each of that node's unvisited neighbors
    create a Terminus object for neighbor
    set its cost to that of the node, plus edge cost
    set its prev to the node's index
    push the Terminus object onto priority queue to-do list
build a stack of entries, starting from the end node, back towards the start
```

when the stack is returned to the main program, get the cost of the cheapest route from the stack's last node, like this:

```
  int start = ... // start node's index
  int end = ... /end node's index
  double cost = 0;
  for (stack<int> result = cheapestRoute(start, end, database); !result.empty(); result.pop())
  {
    int nodeIndex = result.front;
    cost = database[nodeIndex].cost;
    ...database[nodeIndex]... // do something with the node
```

```
}
cout << "Total cost is " << cost << endl;
```

Since the last index to come off of the stack is the one for the end node of the route, the update of **cost=database[nodeIndex].cost;** in its cycle -- the last cycle -- is the cost of getting to the end node. That's the cost of the cheapest route from start to end.