Finding The Shortest Route, Reading

What we *really* want to do with a road map graph is to find the shortest route from any selected node to any other selected node, like Google Maps and Apple Maps do. Part of the challenge is getting the answer *fast*, but another part is even being able to get an answer at all!

But before you get too excited, note that this reading covers the *shortest route,* which in graph terminology refers to the minimum number of *steps* from a start node to an end node. The distance between nodes or the cost or time to get from one to another is *not* considered. Each link costs the same. If San Francisco's neighbors include Sacramento and Los Angeles, both are considered to be one *step* away, period.

Unweighted Edges

This is a good place to introduce the graph terminology for a connection between two nodes -- it's an **"edge"**. Shortest route solutions are based on "unweighted" edges -- that is, each edge has the same cost or distance or time.

This is also a good place to introduce another graph term for the cost or distance or time associated with an edge -- it's generic term is **"cost"**. Unweighted edges are said to have "unit cost" -- that is, a cost of 1.

A Shortest Route Algorithm

The difference between this solution and BFS is that (1) we want to keep track of the shortest path to each node, and (2) we should stop once we find the end node. Once a path to a node is found, it's guaranteed to be the shortest, because in the fanning out process, the later found paths to already-visited nodes will only be longer.

To track the path that gets us to any node, we only have to store the *index* of it's predecessor. It's kind of like a "bread crumb" so we can find our way back to the start node. That's going to require another tracking variable in **struct Node**:

```
  int prev; // index of the preceding node in the route, -1 for none or blank
```

The BFS algorithm is then altered as follows:

```
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
    set neighbor's prev to the index of the node under consideration
    push neighbor's index into the "to-do" list
    mark neighbor as visited
    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
```

Note that a return queue is not created at the start and built up in the loop, to be returned at the end. Instead, since we have the bread crumbs, we can *back out* the solution at the end. That means building it in reverse order, so we'll use a LIFO stack so that the first-pushed end node is the last one out of the stack.

The Shortest Route Function

The Shortest Route Function

There are two differences between this and the BFS and DFS: (1) it returns a stack, and (2) there needs to be another input parameter: the *end* node's index. Here's the stand-alone prototype:

```
stack<int> shortestRoute(int startNodeIndex, int endNodeIndex, vector<int>& database);
```

Here's how the function would be called from the main program and its results used:

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