Dijkstra's Shortest Path Algorithm

Kaminski / cs3310

PURPOSE: Find the Minimum Cost Path from a designated Start node to a designated Destination node in a Graph *(directed or undirected)*

ASSUMPTIONS:

- graph is already stored, including:
  1) `n`, which is the number of graph nodes (where node are numbered 0 to N-1, not 1 to N)
  2) `edgeWeight(s)` - stored as EITHER:
     - an adjacency matrix with 3 possible values:
       1) actual edge weights, 2) 0's in the diagonal, 3) "infinity" (i.e., MaxValue) for "no edge" node-pairs
       *[conventionally, **directed** graphs use rowNumber as the source ("from") & columnNumber as the sink ("to")]*
     - adjacency lists
- `GetWeight(a,b)` method returns the numeric `weight` value from the stored graph
  - for an INTERNAL AdjacencyMatrix, use direct address to get `edgeWeight[a,b]`
  - for an EXTERNAL AdjacencyMatrix, use direct address to read the `weight` value (after calculating offset, then doing a seek)
    [When calculating offset, allow for:
    the headerRec; nodeNumbers being 0 to N=1, not 1 to N; weights being int or short or long or double… (as specified)
    So at the start of the program, calculate sizeOfHeaderRec, sizeOfARow, and sizeOfAWeight once and for all
    to use in the offset calculation needed for each seek]
  - for an INTERNAL AdjacencyLists, search `a`'s linked list for node `b` *(not vice versa, so as to accommodate directed graph)*

ASSUMPTION: program already has gotten `start` & `destination` NUMBERS (integers from 0 to N-1)
(if user instead provides startName & destinationName, then these need to be converted into start & destination NUMBERS)

1) Initialize the 3 "working storage" arrays: *[INITIALIZATION MUST BE DONE EACH TIME BEFORE STEP 2 IS DONE]*

a) `included` – booleans *[Is **thisNode** **`included`** yet in the group of nodes already used to revise `distance` or not?]*
- all are initially set to false, except `included` at `start` is set to true

b) `distance` - integers (usually) *[What's the **`distance`** from `start` to **thisNode** SO FAR? (These are ceiling values that may go down).]*
- set each one's value to its `edgeWeight` value from the graph for `start` to **thisNode**
  which would be either: 1) the weight for an actual edge OR 2) 0 for `[start]` OR 3) "infinity" for no-edge cases

c) `path` - integers *[What's the nodeNumber of **thisNode**'s predecessor on the **path** from `start` to **thisNode**?*
(These are actual subscript values between 0 and N-1, corresponding to nodeNumbers).]
- set all to `-1's,` except put `start`'s nodeNumber when `distance[i]` has a valid edge weight (case #1 in distance above)

2) the main Search part: *[NOTE: indentation used below to indicate nesting].*
[Algorithm above handles "normal case" – check if changes are needed for "special case", e.g., start == destination]

```
while destination is NOT yet included
    - out of all nodes NOT yet included,
          choose target node as the node (number) with the minimum distance value
                (NOTE: target is a subscript not a distance)
    - target now becomes included (as having been evaluated to see it's effect)
    - check all distance values (ceilings) to see which ones can be lowered [i.e., loop: i = 0 to N-1]
        if included[i] is false                                  [i.e., GUARD against doing the BIG TEST unnecessarily]
            if edgeWeight from target to i is a valid edgeWeight     [another GUARD against doing …]
                    (i.e., edgeWeight isn't 0 or "infinity")
                                                  [Next line is the "BIG TEST" – i.e., should distance[i] ceiling be lowered?]
                if distance[target] + edgeWeight from target to i < distance[i]
                then: 1) distance[i] = distance[target] + edgeWeight from target to i
                      2) path[i] = target
```

3) Report the answer:

a) the **TOTAL DISTANCE** of the minimum path from `start` to `destination` is in `distance[destination]`

b) the **FINAL PATH** itself from `start` to `destination` is gotten from following the values in `path` array from `[destination]` to `-1`.
However, this gives the path in reverse order, from `destination` to `start`.
To fix this (and instead correctly report the path from `start` to `destination`), either use:
- recursion - printing the results on the way back UP recursion
- OR push answers on a stack instead of printing them, then pop them off the stack to print them
- OR store them to an array (incrementing, starting at 0), then print the array in reverse (decrementing, stopping at 0).