Dijkstra’s Algorithm With Adjacency List Documentation

# Description

The assignment was to perform Dijkstra’s algorithm on a set of city-city pairs with both a cost and time between them. Then display the three best paths if they exist for another set of origin-destination city pairs that also specify whether to prioritize cost or time. The structure to store flights was required to be a linked list of linked lists. Additionally, an “iterative backtracking algorithm” was required to be implemented, along with a linked list and stack class. Sample flight data, requested flight plans, and expected solution output were provided.

I created a Vertex and a Linked List class such that each Vertex has a Linked List pointer within it and each Linked List has a standard head and tail Vertex pointer. This way, a single linked list is a sufficient adjacency list. After parsing the input data, I instantiated all vertices, linked lists, and their connections from the flight data and placed the flight plans in a vector of structs to loop through. I then implemented Dijkstra’s algorithm with maps for least cost data, predecessors, and unrelaxed edges. For the iterative backtracking, I used a breadth-first search and a stack emulating recursive calls to iteratively go through every possible path from a source to a destination, avoiding all cycles.

# Input

Input is given from two files: sampleFlightData.txt and sampleFlightPlans.txt. The former starts with the number of connections on the first line, then formats its data like so: OriginCity|DestinationCity|Cost|Time. A connection from Dallas to Houston costing $101 and lasting 51 minutes would appear as Dallas|Houston|101|51. The latter has a similar format – number of plans on the first line, then a format of OriginCity|DestinationCity|Letter. ‘Letter’ is either ‘T’ or ‘C’ to denote prioritizing time or cost, respectively. An example flight plan from Dallas to Houston prioritizing time would appear as Dallas|Houston|T.

# Output

All output is displayed on standard output. Each flight is shown with its initial origin, destination, and time or cost prioritization, followed by the three best paths if they exist.

# Adjacency List Organization Explanation

This is the main thing I would change if I were to do this all over again. First, a single linked list serves as the main reference for the adjacency list. Each vertex in this linked list also contains a linked list. However, the vertices in this second layer of linked lists have their linked lists set to nullptr. Thus, you cannot follow an endless loop of the head of a linked list pointing to a vertex containing another head, since an attempt to access a second layer linked list head would return an error because no linked list exists for that vertex.

One major caveat to my attempt to be clever is that there is no room in this structure for edge weights to connect vertices. So, the vertices in the second layer of linked lists must hold that information. The problem is that the vertices in the first linked list cannot be reused in other vertices’ linked lists since they aren’t supposed to hold edge weight information – that’s only supposed to be available in their adjacent vertices. We now have two sets of vertices implicitly defined – vertices in the first linked list with their own linked lists but no edge weight information, and the adjacent vertices in those second linked lists with no linked lists of their own but holding edge weight information. This implicit distinction causes problems for storing vertices in maps used in Dijkstra’s algorithm discussed later.

# Specific Method Explanations

## Class name

Method name

<Method explanation>

## DijkstraAlgorithm

runAlgorithm

Maps leastPathData, predecessors, and unrelaxedEdges are instantiated with vertices of the first linked list according to the algorithm requirements. Then a loop goes through unrelaxedEdges, the least edge from unrelaxedEdges is saved to currentLeastUnknownVertex, and is erased from unrelaxedEdges. After checking that this edge does have a list of adjacent vertices (is connected to the graph), its adjacent vertices are looped through. The pointer currentAdjacentVertex is exactly what its named, while currentAdjacentVertexForMaps is a pointer to the vertex in the first linked list with the same name as currentAdjacentVertex. It’s needed in case we need to add a vertex to a map since maps only deal with vertices in the first linked list (this is why I should’ve just made separate vertex and edge classes and two linked list types).

If the current adjacent vertex’s edge weight plus the stored least path data for the vertex before it in the path is less than the stored least path data for the current adjacent vertex itself, the first quantity replaces the second in the leastPathData map and both the current adjacent vertex’s predecessor and unrelaxedEdges map data are updated. The loop continues to the next adjacent vertex. When the current unrelaxed edge vertex (the vertex also represents an edge…I know) adjacency list is looped through, the next least unrelaxed edge is selected and its adjacency list is looped through until all vertices in the first linked list are looped through in order of decreasing edge weights.

printIterativeBacktracking

The struct pathWithState is defined in Stack.h and holds a vector of Vertex pointers (path) and the total cost and time of that path. The vector allCalls holds all pathWithState pointers allocated on the heap for later deletion. The vector paths holds all valid paths from source to destination. The pointer p is simply a reusable pointer for new pathWithState pointer instantiation when adding to paths. The stack unprocessedCalls functions as a call stack for a recursive method and holds pathWithState pointers. The idea is to traverse the graph with a breadth-first search – first adding two-city paths from the source vertex to all of its adjacent vertices, and following every possible path until each path either reaches a dead-end/cycle or the destination. A check for the former involves checking if, for the current path’s next step being evaluated, any of its adjacent vertices are already a part of its path. If it is, ignore it, and if it isn’t, keep following that unencountered path. Either way, the call is removed and replaced with more calls until all paths leading to the destination are stored in the paths vector and all others are pruned off.

Finding the three best paths involves searching the paths vector for the path with the least cost or time, removing it, and repeating until three iterations pass or the it’s empty.