Derek Rodriguez, Derek Caprio
COP 4530 Project 4
Report and UML dialgrams

Report:

This project demonstrates graph data structures and functions that are common to them. This includes depth first search (DFS) and breadth first search (BFS) on both directed and undirected weighted graphs, as well as Dijkstra's shortest path algorithm on the directed graph, and Prim's minimum spanning tree (MST) algorithm on the undirected graph. This project also uses a hash table to store the vertices of the graph. Initial graph data is loaded from a file defined at run time (graph.txt), after this the graph can be modified using menu functions during program execution.

Edges are implemented with sourceVertex and targetVertex data members. Both Graph and DirectedGraph have an edge list as a data member which stores a linked list of the edges in the graph. The difference between the undirected graph and the directed graph is that during the insert funciton, the undirected graph sets source and target accordingly for both vertices on the edge, whereas the directed graph only sets source and target in one direction.

The DFS and BFS functions are implemented in similar for both directed and undirected graphs. The DFS algorithm is recursive and repeatedly travels "down" the graph as long is there a new targetVertex to explore. After this is done, the recursion continues as the search repeats iteself down each unexplored branch. The BFS algorithm is non-recursive. Instead, it uses a queue to "remember" which vertices it needs to go back to after it repeatedly searches the next vertex down the graph. The difference between these two searches is that DFS repeatedly searches the target of a target, whereas BFS repeatedly searches for all targets of the current vertex before moving on to the next target.

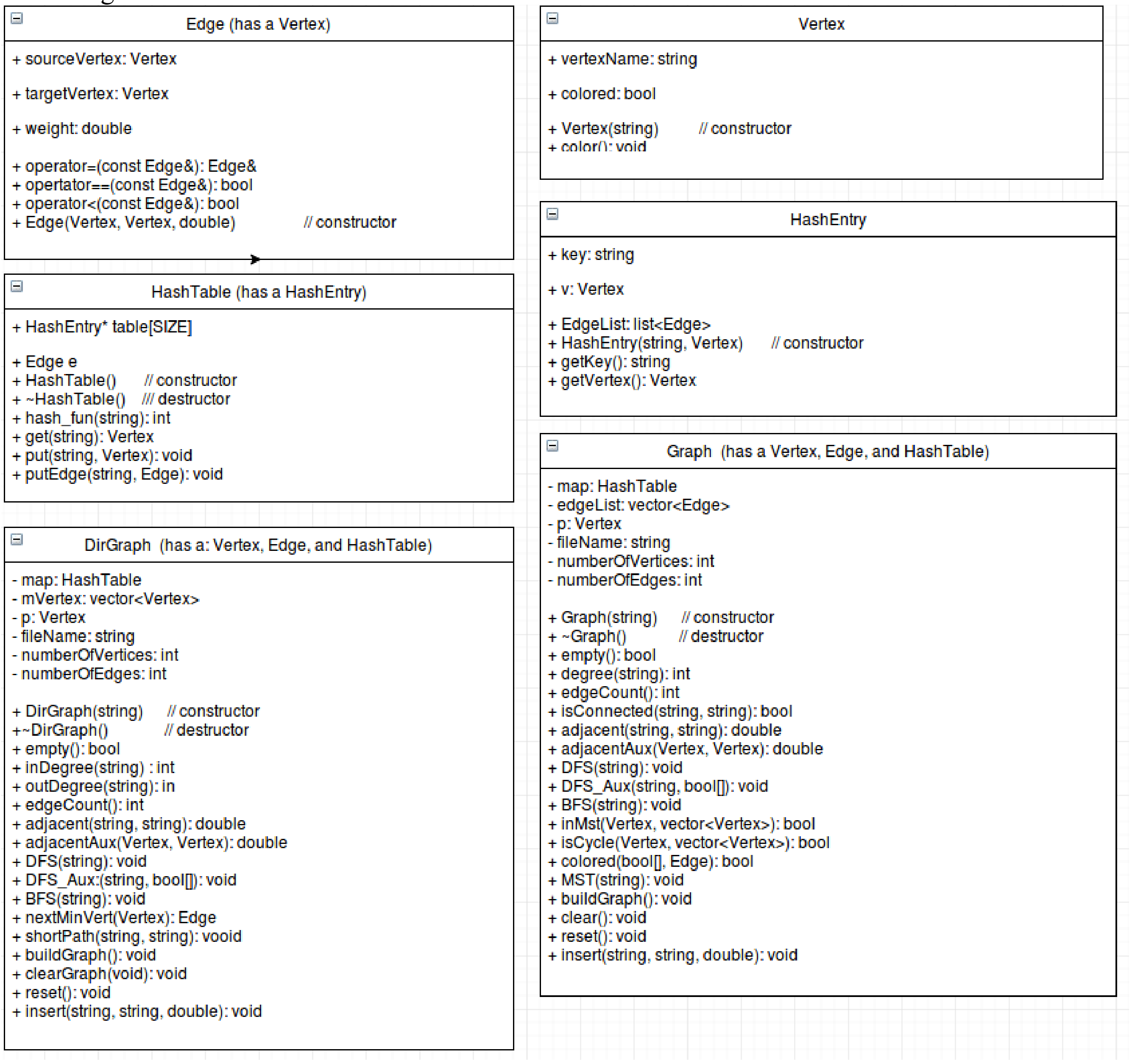
Dijkstra's algorithm is implemented on the directed graph by using 2-dimensional array which serves as an adjacency table, a 1-dimensional array which stores tentative minimum distances to each vertex in the graph from the source, and a 1-dimensional array to store the predecessor of each vertex in the graph during algorithm execution. As the graph is searched in a BFS-like manner, if the length of the current edge plus the current shortest distance to a vertex is less than the value currently in the tentative minimum distance array for that vertex, the value there is updated with the new shortest distance. When the traversal is finished, the distance array stores the shortest distance to all vertices in the graph.

To find the minimum spanning tree using Prim's algorithm, we began with two vectors, one storing vertices currently in the MST set (initially empty) and the other storing vertices not currently in the MST set (initially all vertices). First the starting vertex is loaded into the MST set and removed from the not MST set, after this, the algorithm repeatedly finds the shortest edge that has not yet been included that connects a non MST set vertex to a MST set vertex and that does not form a loop until the not MST set vector is empty. When this is done, the MST set is built.

We split work evenly over all parts of this project instead of dividing it up like we have on past assignments. This project took over 20 hours each to complete.

[UML diagrams on next page]

UML diagrams:



Edge (has a Vertex)

+ sourceVertex: Vertex
+ targetVertex: Vertex
+ weight: double
+ operator=(const Edge&): Edge&
+ opertator==(const Edge&): bool
+ operator<(const Edge&): bool
+ Edge(Vertex, Vertex, double) // constructor

Vertex

+ vertexName: string
+ colored: bool
+ Vertex(string) // constructor
+ color(): void

HashTable (has a HashEntry)

+ HashEntry* table[SIZE]
+ Edge e
+ HashTable() // constructor
+ ~HashTable() /// destructor
+ hash_fun(string): int
+ get(string): Vertex
+ put(string, Vertex): void
+ putEdge(string, Edge): void

HashEntry

+ key: string
+ v: Vertex
+ EdgeList: list<Edge>
+ HashEntry(string, Vertex) // constructor
+ getKey(): string
+ getVertex(): Vertex

DirGraph (has a: Vertex, Edge, and HashTable)

- map: HashTable
- mVertex: vector<Vertex>
- p: Vertex
- fileName: string
- numberOfVertices: int
- numberOfEdges: int

+ DirGraph(string) // constructor
+~DirGraph() // destructor
+ empty(): bool
+ inDegree(string) : int
+ outDegree(string): in
+ edgeCount(): int
+ adjacent(string, string): double
+ adjacentAux(Vertex, Vertex): double
+ DFS(string): void
+ DFS_Aux:(string, bool[]): void
+ BFS(string): void
+ nextMinVert(Vertex): Edge
+ shortPath(string, string): vooid
+ buildGraph(): void
+ clearGraph(void): void
+ reset(): void
+ insert(string, string, double): void

Graph (has a Vertex, Edge, and HashTable)

- map: HashTable
- edgeList: vector<Edge>
- p: Vertex
- fileName: string
- numberOfVertices: int
- numberOfEdges: int

+ Graph(string) // constructor
+ ~Graph() // destructor
+ empty(): bool
+ degree(string): int
+ edgeCount(): int
+ isConnected(string, string): bool
+ adjacent(string, string): double
+ adjacentAux(Vertex, Vertex): double
+ DFS(string): void
+ DFS_Aux(string, bool[]): void
+ BFS(string): void
+ inMst(Vertex, vector<Vertex>): bool
+ isCycle(Vertex, vector<Vertex>): bool
+ colored(bool[], Edge): bool
+ MST(string): void
+ buildGraph(): void
+ clear(): void
+ reset(): void
+ insert(string, string, double): void