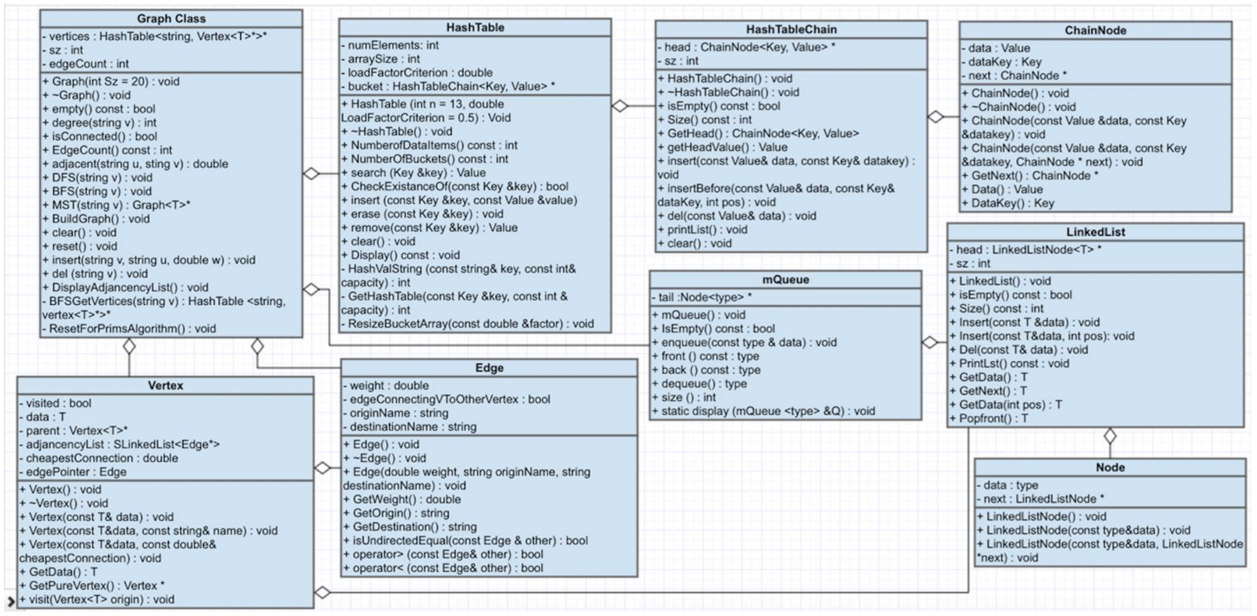
Team Project 4 Report - Graphs

by: Manuel Martinez , John Santoro , and Jim Bui

UML Diagrams


Graph Class
- vertices : HashTable<string, Vertex<T>*>*
- sz : int
- edgeCount : int
+ Graph(int Sz = 20) : void
+ ~Graph() : void
+ empty() const : bool
+ degree(string v) : int
+ isConnected() : bool
+ EdgeCount() const : int
+ adjacent(string u, sting v) : double
+ DFS(string v) : void
+ BFS(string v) : void
+ MST(string v) : Graph<T>*
+ BuildGraph() : void
+ clear() : void
+ reset() : void
+ insert(string v, string u, double w) : void
+ del (string v) : void
+ DisplayAdjancencyList() : void
- BFSGetVertices(string v) : HashTable <string, vertex<T>*>*
- ResetForPrimsAlgorithm() : void
HashTable
- numElements: int
- arraySize : int
- loadFactorCriterion : double
- bucket : HashTableChain<Key, Value> *
+ HashTable (int n = 13, double LoadFactorCriterion = 0.5) : Void
+ ~HashTable() : void
+ NumberofDataItems() const : int
+ NumberOfBuckets() const : int
+ search (Key &key) : Value
+ CheckExistanceOf(const Key &key) : bool
+ insert (const Key &key, const Value &value)
+ erase (const Key &key) : void
+ remove(const Key &key) : Value
+ clear() : void
+ Display() const : void
- HashValString (const string& key, const int& capacity) : int
- GetHashTable(const Key &key, const int & capacity) : int
- ResizeBucketArray(const double &factor) : void
HashTableChain
- head : ChainNode<Key, Value> *
- sz : int
+ HashTableChain() : void
+ ~HashTableChain() : void
+ isEmpty() const : bool
+ Size() const : int
+ GetHead() : ChainNode<Key, Value>
+ getHeadValue() : Value
+ insert(const Value& data, const Key& datakey) : void
+ insertBefore(const Value& data, const Key& dataKey, int pos) : void
+ del(const Value& data) : void
+ printList() : void
+ clear() : void
ChainNode
- data : Value
- dataKey : Key
- next : ChainNode *
+ ChainNode() : void
+ ~ChainNode() : void
+ ChainNode(const Value &data, const Key &datakey) : void
+ ChainNode(const Value &data, const Key &datakey, ChainNode * next) : void
+ GetNext() : ChainNode *
+ Data() : Value
+ DataKey() : Key
LinkedList
- head : LinkedListNode<T> *
- sz : int
+ LinkedList() : void
+ isEmpty() const : bool
+ Size() const : int
+ Insert(const T &data) : void
+ Insert(const T&data, int pos): void
+ Del(const T& data) : void
+ PrintLst() const : void
+ GetData() : T
+ GetNext() : T
+ GetData(int pos) : T
+ Popfront() : T
mQueue
- tail :Node<type> *
+ mQueue() : void
+ IsEmpty() const : bool
+ enqueue(const type & data) : void
+ front () const : type
+ back () const : type
+ dequeue() : type
+ size () : int
+ static display (mQueue <type> &Q) : void
Vertex
- visited : bool
- data : T
- parent : Vertex<T>*
- adjancencyList : SLinkedList<Edge*>
- cheapestConnection : double
- edgePointer : Edge
+ Vertex() : void
+ ~Vertex() : void
+ Vertex(const T& data) : void
+ Vertex(const T&data, const string& name) : void
+ Vertex(const T&data, const double& cheapestConnection) : void
+ GetData() : T
+ GetPureVertex() : Vertex *
+ visit(Vertex<T> origin) : void
Edge
- weight : double
- edgeConnectingVToOtherVertex : bool
- originName : string
- destinationName : string
+ Edge() : void
+ ~Edge() : void
+ Edge(double weight, string originName, string destinationName) : void
+ GetWeight() : double
+ GetOrigin() : string
+ GetDestination() : string
+ isUndirectedEqual(const Edge & other) : bool
+ operator> (const Edge& other) : bool
+ operator< (const Edge& other) : bool
Node
- data : type
- next : LinkedListNode *
+ LinkedListNode() : void
+ LinkedListNode(const type&data) : void
+ LinkedListNode(const type&data, LinkedListNode *next) : void

Prim's Algorithm with Undirected Graph

Summary

Prim's algorithm can be implemented using a computer, a mouse, a keyboard, and an human / dolphin brain. Prim's algorithm takes is used on an undirected weighted graph. It finds the MST (minimum spanning tree) for the graph, which is the set of edges that connects the graph with the least weight.

Data Members

All data members are private.

HashTable<string, Vertex<T>*>* vertices:

This hash table stores all of the vertices in the graph by their name. Calling search([vertex name]) returns a particular vertex in the graph, along with all associated edges. Usually O(1); see project 2 report.

Int sz:

This integer stores the maximum number of vertices in the graph. Upon initialization, this value will be set to at least 20, meaning that the graph can store at least 20 vertices at one time.

Int edgeCount:

This integer stores the number of edges currently in the graph. Since edges are undirected, there will actually be double the number of edge objects in the graph. For instance, an "edge" between V1 and V3 is represented as V1 → weight → V3 (stored in V1) and V3 → weight → V1 (stored in V3).

Methods

Private

HashTable<string, Vertex<T>*>* BFSGetVertices(string v):

This helper method gets all vertices associated with a BFS on the vertex denoted by v. It is used to determine what vertices will be included in the MST obtained from running Prim's Algorithm. Like BFS, the running time is O(|V| + |E|), where V is the number of vertices and E is the number of edges.

Void ResetForPrimsAlgorithm():

This helper method resets all the variables from all the vertices and edges in the graph after a call to prim's algorithm. Note that the following values can be modified: cheapestConnection in Vertex and edgeConnectingVToOtherVertex in Edge. Runtime: O(|V| + |E|) because all the vertices and all the edges are traversed once.

Public

Graph(int Sz = 20):

The constructor initializes a new graph object to have a size of 20 (unless otherwise specified by the user and 0 edges. O(1) is the running time.

~Graph():

The destructor clears the graph and frees all memory associated with the vertices and edges. O(|V| + |E|) is the runtime because all objects must be traversed before they are deleted.

bool empty() const:

This method returns whether or not there are 0 vertices in the graph through the numElements variable in the hash table containing the vertices. O(1) is the running time since only one operation is performed.

int degree(string v):

This method gets the number of edges going into and out from the vertex denoted by v. Usually O(1) (see project 2 document → hash table for exceptions). The method returns the size of the edge list in vertex v.

bool isConnected():

This method performs a BFS traversal (O(|V| + |E|) due to adjacency lists) to determine if the graph is connected. If the number of vertices traversed == the number of vertices in the graph, it is; otherwise, it isn't.

int EdgeCount():

This method returns the number of edges currently in the graph. Edges are bidirectional (see Data Members >> int edgeCount), so the actual number of edge objects will be double this value. O(1) since a constant number of operations are performed.

double adjacent(string u, string v):

This method returns the weight of an edge connecting vertex U with vertex V. To do so, it iterates through the edge list of vertex U and checks whether the destination of each edge is equal to V. The worst case running time will be O(|V|) and will occur when there is a list of edges containing links to every other vertex, and the edge to be found is at the end of the list.

void DFS(string v):

This method performs a depth-first search starting at vertex v and ending once all reachable vertices have been traversed. O(|V| + |E|) is the running time because all of the vertices and edges can be traversed at least once. Note that the method is recursive and a default stack is used to implement it.

void BFS(string v):

This method performs a breadth-first search starting at vertex v and ending once all reachable vertices have been traversed. O(|V| + |E|) is the running time because all of the vertices and edges can be traversed at least once. Note that the method requires a queue containing vertex objects to function properly.

Graph<T>* MST(string v):

This method calculates the minimum spanning tree starting from the vertex entered by the user. To do so, it first acquires all vertices connected to v by doing a breadth-first search (O(|V| + |E|), see BFS above) and stores these in a temporary hash table for faster access. While the temporary hash table is not empty, the program determines the least expensive connections between the edge and its corresponding vertices, and then adds that edge to the MST. This method has a running time of O((|V| + |E|)|V|) because no complex edge sorting algorithm is used.

void BuildGraph():

This method builds a graph from a text file.* The runtime is O(|V| + |E|), where V is the number of vertices and E is the number of edges. All edges and vertices must be inserted into the existing graph.

void clear():

This method clears all vertices from the graph and sets its size to 0. Runtime is O(|V| + |E|) because all memory associated with both the vertices and their edges is freed.

void reset():

This method resets all of the flags in the vertices indicating that they had been visited during a BFS or DFS. Note that the runtime is O(|V|) because each vertex must be traversed.

void insert(string u, string v, double w):

This method inserts an edge between vertices u and v of weight w. If an edge between the vertices already exists, its weight is replaced. Due to the way edges are stored, when an edge from u to v is added, another object from v to u with the same weight must also be added. As a result, the runtime constant is doubled. In the worst case scenario, an insertion will take O(|V| + |E|), but in most cases, insertion will be less expensive.

void del(string v):

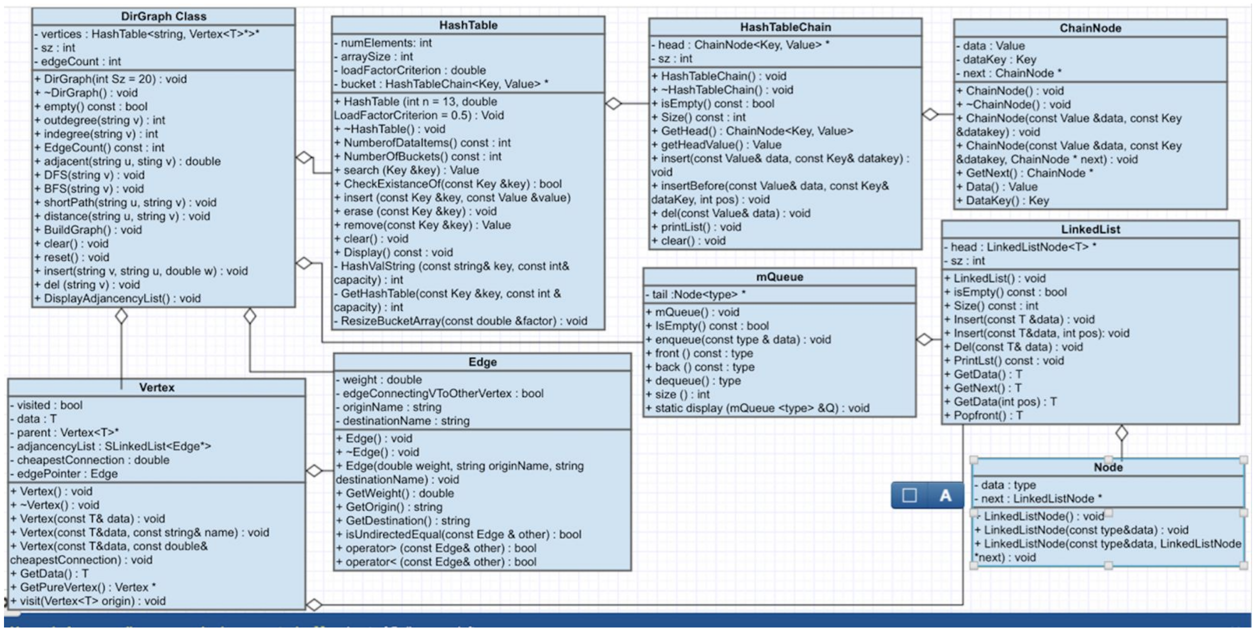
This method deletes a vertex and all edges associated with it. This operation is expensive, as all the vertices of all the edges connecting to v must be checked to ensure that one of their edges does not connect to v. As a result, the running time is O(|V| + |E|), where V is the number of vertices and E is the number of edges.

void DisplayAdjacencyList():

What do you think this does? O(|V| + |E|) is the running time because, well, I said so.

* Appears to read in a text file, but in reality, all information pertaining to the graph is hard-coded because of multiple errors when switching from Visual Studio to Linux based machines and vice versa.

Dijkstra’s Algorithm with Directed Graph



DirGraph Class
- vertices : HashTable<string, Vertex<T>*>*
- sz : int
- edgeCount : int
+ DirGraph(int Sz = 20) : void
+ ~DirGraph() : void
+ empty() const : bool
+ outdegree(string v) : int
+ indegree(string v) : int
+ EdgeCount() const : int
+ adjacent(string u, sting v) : double
+ DFS(string v) : void
+ BFS(string v) : void
+ shortPath(string u, string v) : void
+ distance(string u, string v) : void
+ BuildGraph() : void
+ clear() : void
+ reset() : void
+ insert(string v, string u, double w) : void
+ del (string v) : void
+ DisplayAdjacencyList() : void
HashTable
- numElements: int
- arraySize : int
- loadFactorCriterion : double
- bucket : HashTableChain<Key, Value> *
+ HashTable (int n = 13, double LoadFactorCriterion = 0.5) : Void
+ ~HashTable() : void
+ NumberofDataItems() const : int
+ NumberOfBuckets() const : int
+ search (Key &key) : Value
+ CheckExistanceOf(const Key &key) : bool
+ insert (const Key &key, const Value &value)
+ erase (const Key &key) : void
+ remove(const Key &key) : Value
+ clear() : void
+ Display() const : void
- HashValString (const string& key, const int& capacity) : int
- GetHashTable(const Key &key, const int & capacity) : int
- ResizeBucketArray(const double &factor) : void
HashTableChain
- head : ChainNode<Key, Value> *
- sz : int
+ HashTableChain() : void
+ ~HashTableChain() : void
+ isEmpty() const : bool
+ Size() const : int
+ GetHead() : ChainNode<Key, Value>
+ getHeadValue() : Value
+ insert(const Value& data, const Key& datakey) : void
+ insertBefore(const Value& data, const Key& dataKey, int pos) : void
+ del(const Value& data) : void
+ printList() : void
+ clear() : void
ChainNode
- data : Value
- dataKey : Key
- next : ChainNode *
+ ChainNode() : void
+ ~ChainNode() : void
+ ChainNode(const Value &data, const Key &datakey) : void
+ ChainNode(const Value &data, const Key &datakey, ChainNode * next) : void
+ GetNext() : ChainNode *
+ Data() : Value
+ DataKey() : Key
LinkedList
- head : LinkedListNode<T> *
- sz : int
+ LinkedList() : void
+ isEmpty() const : bool
+ Size() const : int
+ Insert(const T &data) : void
+ Insert(const T&data, int pos): void
+ Del(const T& data) : void
+ PrintLst() const : void
+ GetData() : T
+ GetNext() : T
+ GetData(int pos) : T
+ Popfront() : T
mQueue
- tail :Node<type> *
+ mQueue() : void
+ IsEmpty() const : bool
+ enqueue(const type & data) : void
+ front () const : type
+ back () const : type
+ dequeue() : type
+ size () : int
+ static display (mQueue <type> &Q) : void
Edge
- weight : double
- edgeConnectingVToOtherVertex : bool
- originName : string
- destinationName : string
+ Edge() : void
+ ~Edge() : void
+ Edge(double weight, string originName, string destinationName) : void
+ GetWeight() : double
+ GetOrigin() : string
+ GetDestination() : string
+ isUndirectedEqual(const Edge & other) : bool
+ operator> (const Edge& other) : bool
+ operator< (const Edge& other) : bool
Vertex
- visited : bool
- data : T
- parent : Vertex<T>*
- adjancencyList : SLinkedList<Edge*>
- cheapestConnection : double
- edgePointer : Edge
+ Vertex() : void
+ ~Vertex() : void
+ Vertex(const T& data) : void
+ Vertex(const T&data, const string& name) : void
+ Vertex(const T&data, const double& cheapestConnection) : void
+ GetData() : T
+ GetPureVertex() : Vertex *
+ visit(Vertex<T> origin) : void
Node
- data : type
- next : LinkedListNode *
+ LinkedListNode() : void
+ LinkedListNode(const type&data) : void
+ LinkedListNode(const type&data, LinkedListNode *next) : void
A

Summary

Dijkstra's algorithm is little bitty trickier: In addition to requiring everything needed for Mr. Prim's algorithm, Mr. Dijkstra's algorithm requires the solar energy powered by a Dyson sphere around a pulsar star to function properly.

In truth, Djikstra's algorithm finds the shortest path on a directed weighted graph from one point to another. It is useful for many applications such as flight planning and path-finding in games.

The algorithm relies on modified max heap trees. The weights are stored inverted , so it behaves like a min heap tree. It first visits the first node, and creates a list of all adjacent lists. It then selects the closest node and travels to that. It then updates all the nodes adjacent to that node, and updates distances as needed. It keeps going until all nodes are visited. It then returns the distance traveled and the path taken.

Data Members

All data members are private.

HashTable<string, Vertex<T>*>* vertices:

This hash table stores all of the vertices in the graph by their name. Calling search([vertex name]) returns a particular vertex in the graph, along with all associated edges. Usually O(1); see project 2 report.

Int sz:

This integer stores the maximum number of vertices in the graph. Upon initialization, this value will be set to at least 20, meaning that the graph can store at least 20 vertices at one time.

Int edgeCount:

This integer stores the number of edges currently in the graph. Since edges are undirected, there will actually be double the number of edge objects in the graph. For instance, an "edge" between V1 and V3 is represented as V1 → weight → V3 (stored in V1) and V3 → weight → V1 (stored in V3).

Methods

DirGraph():

This is the constructor for directed graphs. It just creates a new hash table of a specific size. Initially, it will be empty. This will be O(1)

~DirGraph():

The destructor clears the graph and frees all memory associated with the vertices and edges. O(|V| + |E|) is the runtime because all objects must be traversed before they are deleted.

empty():

Checks if the graph is empty. Will be O(1).

outdegree():

Returns the number of edges going out from a vertex. Will be O(1).

indegree():

Returns the number of edges going to the vertex. Will be O(1).

EdgeCount():

This method returns the number of edges currently in the graph. Edges are bidirectional (see Data Members >> int edgeCount), so the actual number of edge objects will be double this value. O(1) since a constant number of operations are performed.

adjacent(string u, string v):

This method returns the weight of an edge connecting vertex U with vertex V. To do so, it iterates through the edge list of vertex U and checks whether the destination of each edge is equal to V. The worst case running time will be O(|V|) and will occur when there is a list of edges containing links to every other vertex, and the edge to be found is at the end of the list.

DFS(string v):

This method performs a depth-first search starting at vertex v and ending once all reachable vertices have been traversed. O(|V| + |E|) is the running time because all of the vertices and edges can be traversed at least once. Note that the method is recursive and a default stack is used to implement it.

BFS(string v):

This method performs a breadth-first search starting at vertex v and ending once all reachable vertices have been traversed. O(|V| + |E|) is the running time because all of the vertices and edges can be traversed at least once. Note that the method requires a queue containing vertex objects to function properly.

shortPath()

This is the implementation of Djikstra's Algorithm. It will take in two values: a starting node and the ending node. It will output one of two things, depending on a boolean value also entered: either the distance traveled, or the path taken. It first checks to see if both nodes exist. If not, it will exit. Otherwise it will continue. The distance to all other nodes are set to an arbitrarily high number (say infinity and beyond), except for the starting node, which be zero. It then creates two heap trees: one to store the edges in a node, and the other to queue up the nodes that need to be visited. It updates the distances from the current node to the adjacent nodes if it is less. If that node has not been visited, then it is also added to the other heap tree. This process is repeated for all items in the visit queue tree. It will then return the value generated from the distances.

BuildGraph():

This method builds a graph from a text file.* The runtime is O(|V| + |E|), where V is the number of vertices and E is the number of edges. All edges and vertices must be inserted into the existing graph.

Clear():

This function clears the directed graph by deleting all of the vertices. It also sets the size of the graph back to 0. Runtime will be O(|V| + |E|).

reset():

This method resets all of the flags in the vertices indicating that they had been visited during a BFS or DFS. Note that the runtime is O(|V|) because each vertex must be traversed.

insert(string u, string v, double w):

This method inserts an edge between vertices u and v of weight w. If an edge between the vertices already exists, its weight is replaced. In the worst case scenario, an insertion will take O(|V| + |E|), but in most cases, insertion will be less expensive.

del(string v):

This method deletes a vertex and all edges associated with it. This operation is expensive, as all the vertices of all the edges connecting to v must be checked to ensure that one of their edges does not connect to v. As a result, the running time is O(|V| + |E|), where V is the number of vertices and E is the number of edges.

DisplayAdjacencyList():

This function displays the adjacency list. This function is only used to test code. O(|V| + |E|) is the running time.