Project 3 Design Document

Zhiyuan Zhang z667zhan

1. Overview of classes

Five classes are implemented in this design. The alNode class is used to represent nodes in the adjacency list, with each node having a vertex, a weight that represents the distance between nodes, and pointing to the next node.

The adList class represents the adjacency list of each vertex in the graph; they have a head pointer which points to the next node(alNode) that they are connected to, and a degree counter to record the degrees of vertices. Adjacency check, degree check, and edge insertion/deletion is performed in this class.

The graph class is used to represent the graph. It has a vertices counter and an edge counter to record these data, and has an array of adjacency list(adList) to record all edges of the graph. With this class one can create a graph.

The heapNode class represents node in the minimum heap. Each heapNode has a vertex and a key that will be used for heapify.

The heap class is used to find MST. It keeps track of the size of itself, has an integer array that tracks the position of nodes and an array of heapNodes that is used to find the shortest cut that can be added to the constructed MST.

2. Design decisions:

For alNode class:

The constructor sets the vertex and weight of alNode and makes it point to NULL. The node can be set to point to another alNode. Since the vertex and weight of a node will not change, ‘const’ can be used here for these parameters.

For adList class:

The constructor sets the head of adjacency list to NULL and degree to zero since at first all vertices are disconnected. New alNode can be added to the head of a list and all actions will be performed both on the source adjacency list and the destination adjacency list.

For graph class:

The constructor sets the initial number of vertices and edges to be zero. The destructor will delete the array of adjacency list to clear cache. With input, array of adjacency list can be created; since it will only create graph once, ‘const’ can be used here. After creation, all sorts of modifications can be performed for this graph.

For heapNode class:

The constructor sets the vertex and key of a heapNode.

For heap class:

The constructor sets the size of heap to be zero at the start, and create an array of integers for positions of heapNodes and an array of heapNodes to extract the heapNode with minimum key. The destructor will delete the position array and the heap array to free memory. Heapify(), modifyKey() and extractMin() are used here together to pop out the node with shortest cut to the current MST until all nodes are in the MST or the graph is found to be not connected.

3. Test strategy:

It should always create a graph with m>1.

For (inserting/ erasing/checking adjacent) edges and finding MST, it should throw illegal exceptions if there is illegal input.

It should always return the correct degree of vertex; it should always return the correct edge count of graph; after insertion/deletion of edges it should update both and can still check if two vertices are adjacent.

For clearing, it should always remove all edges from the graph.

For finding mst, it should decide whether the graph is connected or not; it should output the total weight of MST correctly with two digits of precision; it can be performed on a graph multiple times for different input vertex.

4. Performance analysis:

n(m): O(1) since it only creates an array of empty adjacency list with size m.

i(u;v;w) & e(u;v): O(V) if all nodes are connected and the edge expected to be erased/updated is at the tail of an adjacency list; it has to examine the nodes one by one.

adjacent(u;v;w): O(V) for the same reason as above.

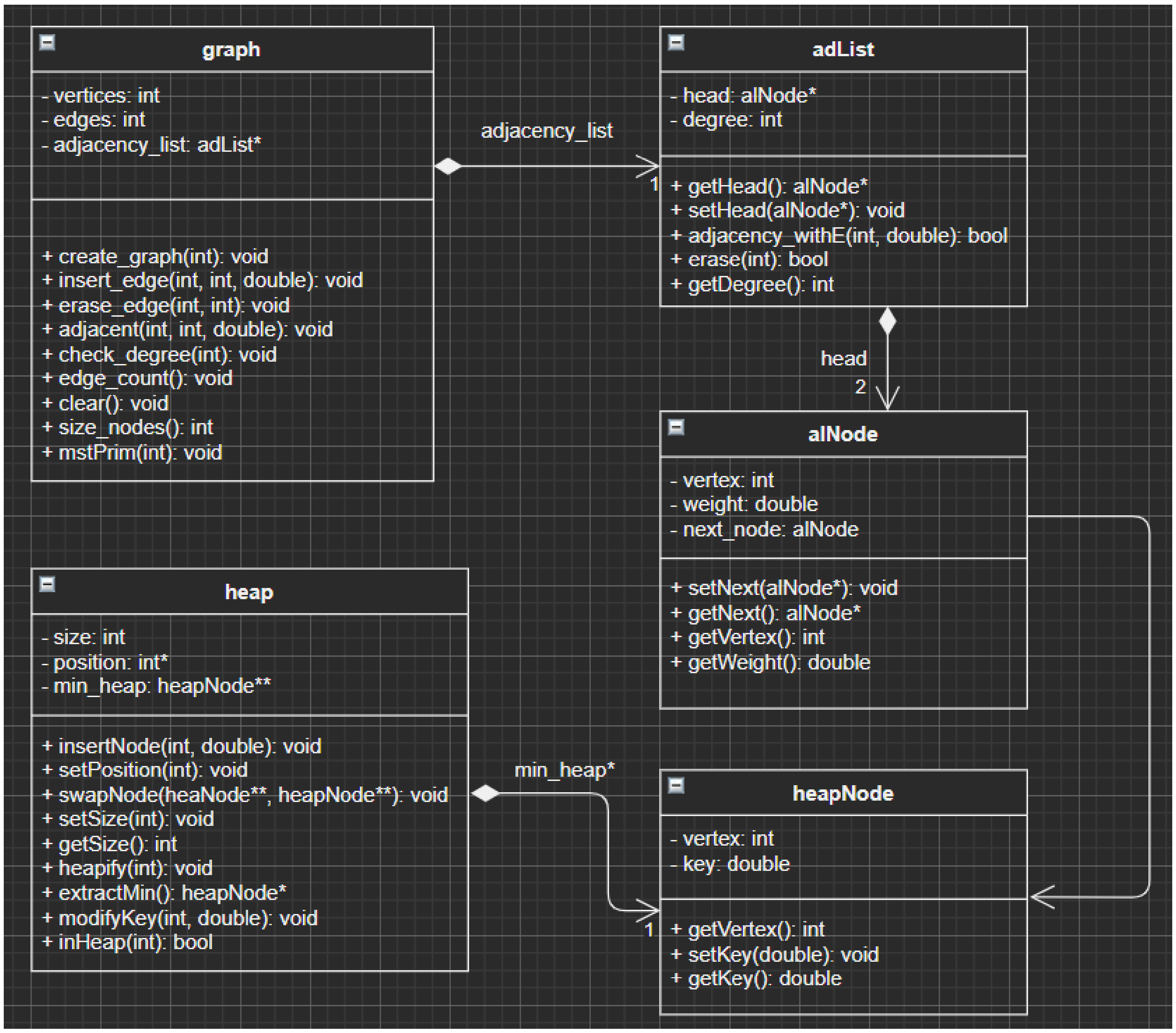
degree(u) & edge_count: O(1) since adjacency list has a counter that keeps track of degree and graph has a counter that keeps track of number of edges.

clear(): O(V) for it has to clear the adjacency list of every node.

mst(u): O(E*lgV) because a binary heap is used to find the mst; modifyKey() takes O(lgV) and extractMin() also takes O(lgV) to complete. Since extractMin() is in the outer loop which will run

O(V) times and modifyKey() is in the inner loop which will run O(E) times in total, the total time complexity of mst-prim's is O(V*lgV + E*lgV) = O(E*lgV).

5. UML class diagram:


graph
- vertices: int
- edges: int
- adjacency_list: adList*
+ create_graph(int): void
+ insert_edge(int, int, double): void
+ erase_edge(int, int): void
+ adjacent(int, int, double): void
+ check_degree(int): void
+ edge_count(): void
+ clear(): void
+ size_nodes(): int
+ mstPrim(int): void
adjacency_list
1
adList
- head: alNode*
- degree: int
+ getHead(): alNode*
+ setHead(alNode*): void
+ adjacency_withE(int, double): bool
+ erase(int): bool
+ getDegree(): int
head
2
alNode
- vertex: int
- weight: double
- next_node: alNode
+ setNext(alNode*): void
+ getNext(): alNode*
+ getVertex(): int
+ getWeight(): double
heap
- size: int
- position: int*
- min_heap: heapNode**
+ insertNode(int, double): void
+ setPosition(int): void
+ swapNode(heaNode**, heapNode**): void
+ setSize(int): void
+ getSize(): int
+ heapify(int): void
+ extractMin(): heapNode*
+ modifyKey(int, double): void
+ inHeap(int): bool
min_heap*
1
heapNode
- vertex: int
- key: double
+ getVertex(): int
+ setKey(double): void
+ getKey(): double