2022310853 박연우

알고리즘개론 Assignment 1

<Dynamic Minimum Spanning Tree>

1. Objective

The objective of this assignment was to design and implement an algorithm to dynamically maintain the minimum spanning tree (MST) of an initially empty undirected graph, as edges are inserted, deleted, or their weights are changed over time.

There were five assumptions in total:

1. The graph is initially empty and undirected. Nodes are numbered from 1 to N, where N is the total number of nodes in the graph, specified in the first line of the input file 'mst.in.' ($N \leq 500$).
2. The total number of `insertEdge`, `findMST`, `deleteEdge`, and `changeWeight` operations is less than or equal to 10,000.
3. Edge weights are positive integers.
4. If the graph is connected, `findMST` should print the total weight sum of the MST to the output file 'mst.out.' If the graph is disconnected, it should print 'Disconnected.'
5. If an operation is not valid (e.g., inserting an existing edge, deleting a non-existent edge), it should be ignored.

2. Implementation

Since edges are continuously inserted, changed, and deleted, I implemented a 2D array of integers where the index of the array represents the vertexes and the value in the array represents the weights. This 2D array, which I will refer to as the adjacency matrix from now on, only really needs the upper triangle values since the matrix is symmetrical due to the undirected edges. However, I still decided to use the full 2D array. This adjacency matrix is largely used to check if an edge exists.

While initially I only used the adjacency matrix to traverse the edges, I encountered a problem when the array became too sparse. It took significant time to traverse each row of the adjacency matrix. Hence, I also implemented an adjacency list. The adjacency list contains edges as a form of linked lists to a single 1D array. (You can think of the nodes of hash key, and edges as the values being hashed). This way, I could reduce the time it took to traverse the edges that each node had.

Furthermore, I had an array to hold instructions until the `findMST` instruction was called. All instructions before `findMST` are stored into an array, and once the `findMST` instruction is called, all the instructions that were stored are completed one by one. Then, the `findMST` function is called and the MST value is printed to the output file "mst.out".

In order to find the MST of a given graph, I used Prim's Algorithm. A binary heap was used as a priority queue to find the minimum key vertex. Then, for the minimum key vertex, its edges were traversed, and vertices which are not already included in the MST had their key values changed if there was a smaller edge. If the key value is changed, then the minimum key value becomes the parent of the node on the other side of that edge. After going through all the nodes, if there is a node—except the starting node 1—without a parent, then the tree is disconnected. In this case, the prim will return `-1` as the MST value, and the `findMST` function will print "Disconnected" to the file When the MST does exist, `prim` will return the MST value and `findMST` function will print that value.

3. Performance Analysis

Analyzing the time complexity involves examining the time complexity of individual operations. (|V| is the number of vertices, and |E| is the number of edges). To summarize the time complexity for each component is as shown in <Table 1>. Detailed explanation per function is below the table.

	Heap Operations	**Prim's Algorithm**	**Heap Initialization**	**Instruction Functions**	**Main Function**
Time Complexity	O(log |V|)	O(|E| log|V|)	O(|V|)	O(n)	O(|E| log |V|) + O(n)

<Table 1> Time complexity of each operation in big O notation

1. **Heap Operations**
    a. These operations include `heapifyDown`, `heapifyUp`, and `extractMin`. They are called while the `prim` function looks for the MST. The time complexity for each heap operation is O(log |V|) since the while loop in `heapifyUp` and `heapifyDown` iterates log(|V|) times at most.

2. **Prim's Algorithm**
    a. The time complexity of the `prim` function is O((|V|+|E|)log|V|) = O(|E| log|V|)[1] since an adjacency list is used with a binary heap. In the worst case, each edge is considered, and for each edge, a heap operation takes O(log |V|) time.

3. **Heap Initialization**
    a. The `initHeap` function takes a time complexity of O(|V|) since it needs to initialize for every vertex.

4. **Instruction Functions**
    a. These functions include `insertEdge`, `changeWeight`, and `deleteEdge`. Thanks to the adjacency matrix, it takes O(1) time to check if the edge already exists. Iterating over the adjacency list takes a time complexity of O(n), where n is the number of instructions. Updating the adjacency matrix and list takes constant time as well.

5. **Main Function**
    a. The overall time complexity is dominated by the Prim's algorithm and the loop iterating over instructions, resulting in O(E log V) + O(n).

[1] "Prim's Algorithm." Wikipedia, 15 Nov. 2023, en.wikipedia.org/wiki/Prim%27s_algorithm.

4. Code Explanation

Structures:

1. **Heap Structure**

   The following are the structures used for the binary heap:

```
typedef struct HEAP{
    int size; //size of heap
    int* pos; //stores position of vertex in heap
    Vertex **vertex; //stores vertices in heap
}Heap;
```

   This was the basic outline of the binary heap. It has an array named `pos` in order to store the positions of the `vertices` inside the heap (this helps access the vertices in O(1) time in the `prim` function). The vertex array stores the vertices inside the heap.

```
typedef struct VERTEX{
    int name; //name of vertex
    int key; //key value of vertex
    int inMST; //checks if vertex is in MST
    struct VERTEX* parent;
} Vertex;
```

   This is the structure for each vertex. The `key` stores the actual value of each vertex, used to compare with the adjacency matrix values and compute the MST. The attribute `inMST` checks if the vertex has been included in the MST yet. The `parent` points to the parent vertex of the node. If the `parent` is null for a node that is not the starting node, the graph is disconnected.

2. **Graph Structure**

```
typedef struct GRAPH{
    int num_nodes; //number of vertices
    int num_edges; //number of edges
    AdjList* adj_list; //adjacency list
    int **adj_matrix;  //adjacency matrix
} Graph;
```

   This structure is the entire graph, which counts the number of nodes and edges. The `adj_list` contains the linked list of edges, and the `adj_matrix` contains the 2D array to indicate the weight and existence of edges.

```
typedef struct EDGE{
    int dest; //destination vertex
    int weight; //weight of edge
    struct EDGE* next; //next edge
}Edge;
```

   This structure represents a single edge. The only one value `dest` is needed because the other node will be used as the index for `adj_list`. There is a pointer to connect to the next edge.

```
typedef struct ADJLIST{
    Edge* head; //head of adjacency list
} AdjList;
```

This structure contains the head for each adjacency list.

3. **Other Structures**

```
typedef struct INSTR{
    char instruction;
    int node1;
    int node2;
    int weight;
}Instr;
```

The sole purpose of the `Instr` structure is to store the instruction in a single array, not having to maintain two arrays to store the instruction and the edge information.

Functions:

The functions for this Priority Queue consists of four parts: main function, heap functions, Prim's algorithm function, and instruction functions.

1. **Main function**

   The main function takes in the number of nodes and initializes the `graph`. It creates a list of instructions. For `insertEdge`, `deleteEdge`, and `changeWeight` instructions, the instruction is stored to `instr_list`. When `findMST` is called, the stored instructions are executed one by one until the most recent `findMST` call. When no more instructions exist, the main function terminates.

2. **Heap functions**

   The following are the functions used to create and maintain a binary heap. It was created in reference to Wikipedia's binary heap description[2].

void swap_int(int* a, int* b)

- Swaps two integer values. Used to swap the values in the `pos` array.

void swap_vertex(Vertex a, Vertex** b)**

- Swaps two vertices when reorganizing the heap.

void heapifyDown(Heap* heap, int index)

- Responsible for maintaining the min heap property. Ensures that the heap remains valid after the minimum is extracted.

[2] "Binary Heap." *Wikipedia*, 12 Oct. 2023, en.wikipedia.org/wiki/Binary_heap.

void heapifyUp(Heap* heap, int index)

- Responsible for maintaining the heap property. Ensures that the heap remains valid after the key value of a vertex is changed.

Vertex* extractMin(Heap* heap)

- Fetches the minimum element of the heap, which is at the root. It decreases the heap size and calls `heapifyDown`().

void initHeap(int num_nodes, Heap* heap)

- Initializes the heap. By allocating the necessary memory space for each attribute.

3. Prim's algorithm

There is a single function called `prim`, which completes prim's algorithm.

int prim(Graph* graph, Heap* queue)

It resets all values in the queue. The initial vertex is node 1 (represented by index 0). While there are elements remaining inside the queue, the minimum vertex is extracted. If the minimum vertex is not the starting node and it does not have a parent, the graph is disconnected. Hence, return -1. Otherwise, the minimum vertex is then added to the MST (by marking `inMST` = 1). Its key value is added to the `total_weight` of the MST.

Using a while loop, the adjacent edges of the minimum vertex are traversed using the `adj_list`. If the current key is greater than the weight of the edge (recorded in `adj_matrix`), and the vertex is not already included in the MST, the vertex's key value is changed. The vertex's parent is set to the minimum vertex, and the `key` is set to the weight from the `adj_matrix`. Lastly, heapifyUp is called to maintain the heap property. The process repeats for each adjacent edge. Lastly, return the `total_weight` of the MST.

4. Instruction functions

The following functions are used to complete the instructions from the file.

void insertEdge(Instr instr, Graph* graph)

- If the edge does not already exist (checked with the `adj_matrix`), insert the new edge to both the `adj_matrix` and `adj_list`.

void changeWeight(Instr instr, Graph* graph)

- If the edge exists (checked with the `adj_matrix`), change the weight in both the `adj_matrix` and `adj_list`.

void deleteEdge(Instr instr , Graph* graph)

- If the edge exists (checked with the `adj_matrix`), delete the edge from both the `adj_matrix` and `adj_list`.

void findMST(Graph* graph, FILE* output_file)

- Calls on the `prim` function to find the MST value. Write the values to the output file.

5. Conclusion

By using Prim's algorithm with an adjacency list and binary heap, the weight of dynamic minimum spanning tree could be found in O(|E| log |V|) + O(n)

The Priority Queue described in the assignment uses a Min-Max heap to store the greater half of values and a Max-Min heap to store the lesser-half of the values. It is able to efficiently find the minimum, median, and maximum in constant time. Inserting an element and deleting the minimum, median, or maximum is completed in O(log n) time.

6. Work Cited

[1] "Prim's Algorithm." Wikipedia, 15 Nov. 2023, en.wikipedia.org/wiki/Prim%27s_algorithm.

[2] "Binary Heap." Wikipedia, 12 Oct. 2023, en.wikipedia.org/wiki/Binary_heap.