Lab 5

Document 5-1, Problem 1

For the non-incremented graph g, to find the MST, we will compare any two edge weights w_i and w_j to determine the next edge to be added to the MST. Suppose the result of comparing w_i to w_j was $w_i < w_j$.

If we transformed each weight by incrementing them by 1, we must then reevaluate the previous expression, utilizing w_i' to w_j', where $w_i' = w_i + 1$, and $w_j' = w_j + 1$.

$w_i' < w_j' = (w_i + 1) < (w_j + 1) = w_i < w_j$.

Thus, there will be no effect on the MST.

Document 5-1, Problem 2

Prim's algorithm chooses the lightest adjacent edges to the tree and is solely concerned with minimizing the total weight. Thus, it does not matter if there exist negative edges, they will not mar the growth of the tree.

Document 5-1, Problem 4

If the priority queue were to be implemented as an unordered list, instead of utilizing a heap, then the heap root removal algorithm and drift down algorithm usually used, would in turn have to be implemented as linear search. Thus, for every edge, we must complete |V| operations as opposed to log(|V|) operations, and thus, the complexity of Prim's would be O(|E|*|V|).

Document 5-1, Problem 5

Suppose we compare any two edge weights before and after the transformation. Call those weights w_i and w_j. Suppose before the transformation, the result of comparing w_i to w_j was $w_i < w_j$.

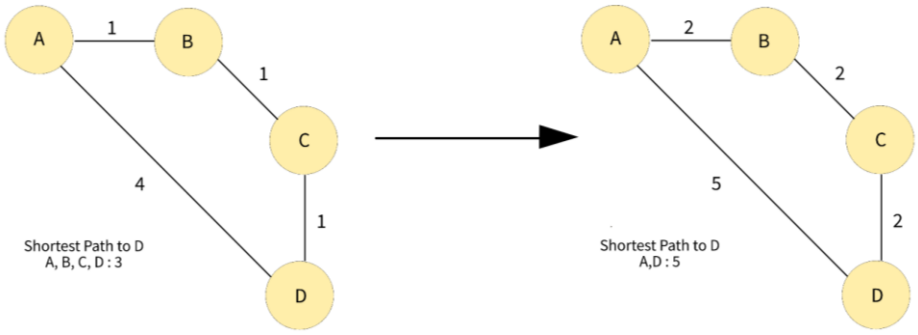
If we transformed each weight by squaring them, we must then reevaluate the previous expression, utilizing w_i' to w_j', where $w_i' = w_i^2$, and $w_j' = w_j^2$

$w_i' < w_j' = w_i^2 < w_j^2 \rightarrow w_i < w_j$.

By taking the square root of both sides of the inequality, we are given the original inequality used to evaluate that same choice on the untransformed graph for the MST. Since the all edges are known to be positive and distinct, all such inequalities shall evaluate the same way. Thus the MST T will still be an MST after the transformation.

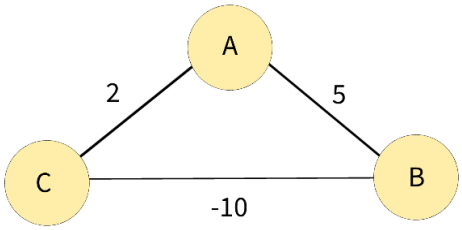
Document 5-2, Problem 8

Counter Example: Dijkstra's will one the first graph, recognize the path A-B-C-D (Path weight 3) as the shortest path to vertex D from the source. However, after the transformation, the shortest path to D will become A-D (Path weight 5) changing the MST after the transformation.



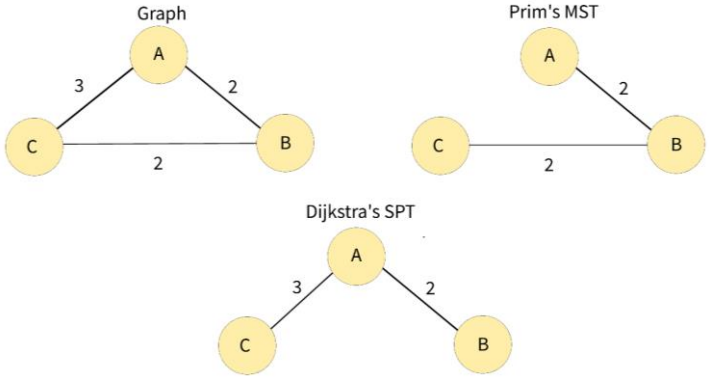
Document 5-2, Problem 9

Counter Example: Since Dijkstra's is greedy, it will not look in retrospect at the paths that may be lighter. Thus, negative edges making previously weighty paths lighter cannot be accounted for, and Dijkstra's breaks down. For example, Dijkstra's would say the edge A-C contains the shortest path to C (Path weight 2) when the path A-B-C (Path weight -5) is actually the shortest.



Document 5-2, Problem 10

Counter Example: Since Prim's is always looking for the relatively closest node to the tree, it will choose A-B then B-C, giving the MST below. Meanwhile Dijkstra's will choose the shortest path, and since the path to node C through A-B-C gives a path weight of 4, versus the path A-C which gives path weight 3, Dijkstra's will give the PST shown below.



Document 5-2, Problem 12

The issue with negative edge weights anywhere else in the graph was the possibility that those edge weights would retroactively affect the total path weights from the source to a node by making it lighter. Since Dijkstra's is a greedy algorithm, it cannot retroactively account for the new, lighter paths being discovered. However, if the only negative edges were those at the source, then the remaining edge weights could never retroactively produce a new, lighter path since the paths could only increase in weight. Thus, Dijkstra's algorithm works for graphs who source has entirely negative edge weights.

Problem 13 on the following page…

Document 5-2, Problem 13

Algorithm
//Assuming Source exists in G

totalDistinctPaths = 0; //Used to count all total distinct paths.

findDistinctShortestPathsCount(Graph G, source s){

```
        for vertex in G
                vertex.level = 0;
                vertex.pathCount = 0;

        s.level = 0;

        s.pathCount = 1;

        for vertex in G.vertices
                if(vertex.visisted == false)

                        bfs(vertex);

        return totalDistinctPaths-1;
```

}

bfs(Vertex v){

```
        Queue Q;
        Q.add(v);

        do{

                currentVertex = Q.pop();

                for vertex in currentVertex.adjacenyList{

                        if(vertex.visited == false){
                                vertex.visited = true;
                                vertex.level = currentVertex.level+1;
                                vertex.pathCount += currentVertex.pathCount;
                                totalDistinctPaths += currentVertex.pathCount;
                                Q.add(vertex);

                        }else if(vertex.level == currentVertex.level+1){
                                vertex.pathCount += currentVertex.pathCount;
                                totalDistinctPaths += currentVertex;

                        }

        while(Q.isEmpty() != true)
```

}

Efficiency is the same as breadth first search, since this is performing breadth first search with minimal other operations, the algorithm performs no more operations than breadth first search. Thus, the efficiency is O(|E|).