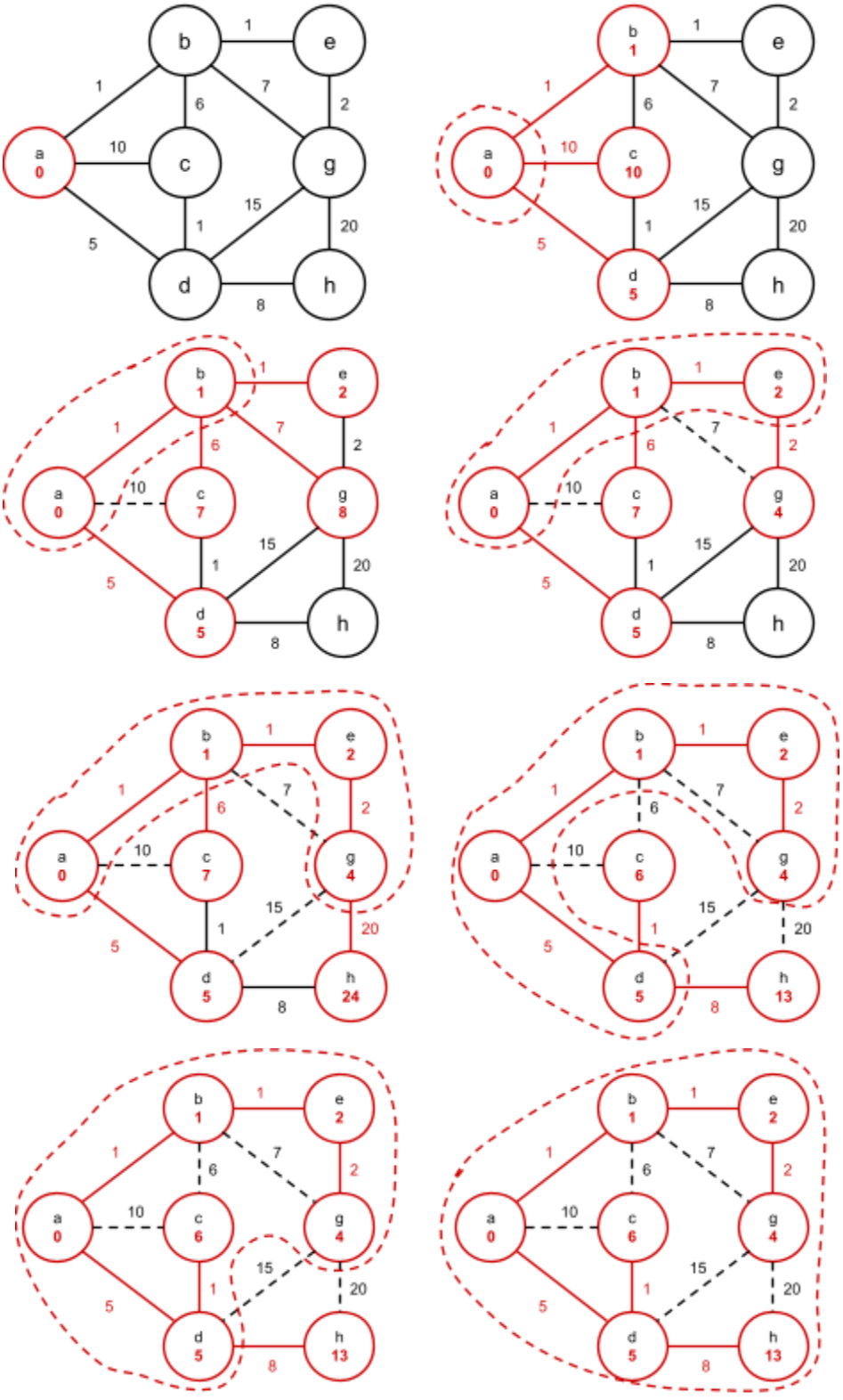
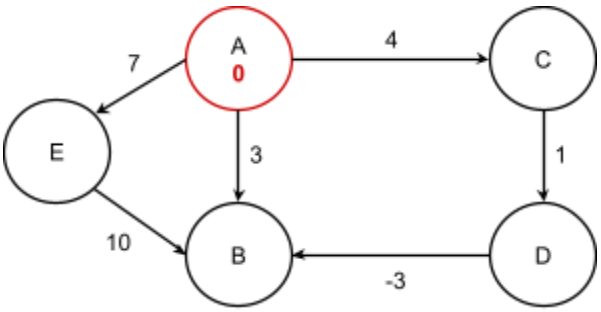
Problem #1 **Name:** Alexander Michelbrink

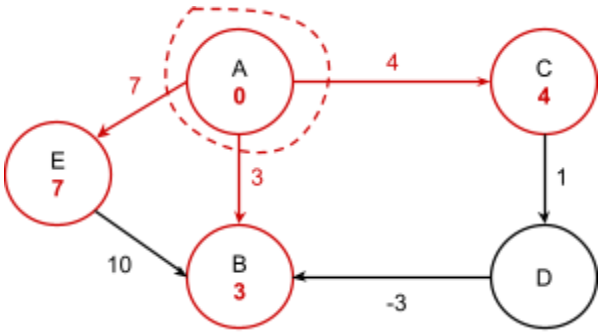
A.) _____

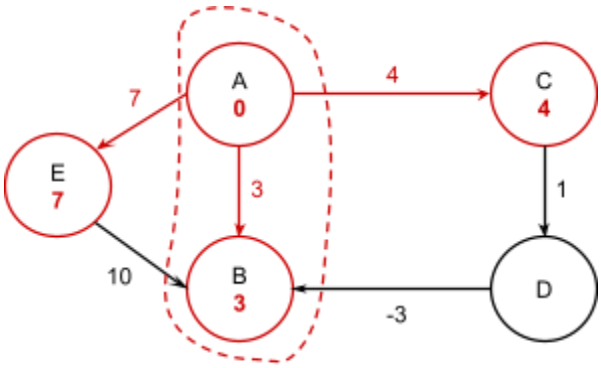


The shortest path tree is shown by the red edges and vertices.

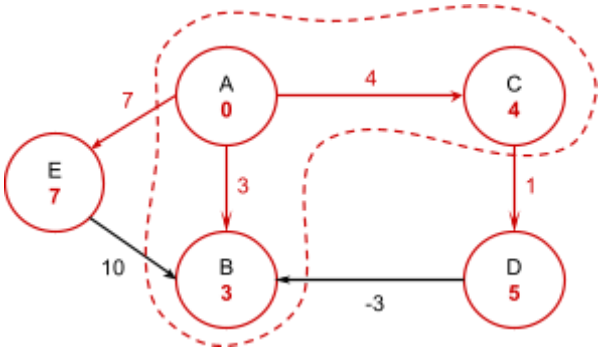
B.) _____

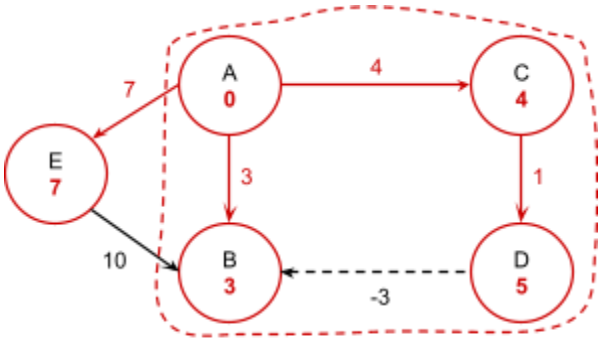




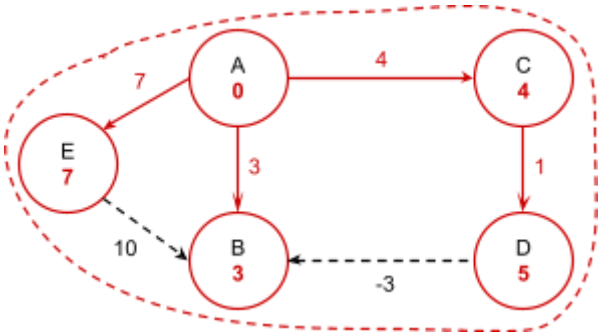


This is where things go wrong. Going from A to C to D to B ends up having a shorter distance (d = 2) than A to B (d = 3). But, B is already in the cloud with d = 3.



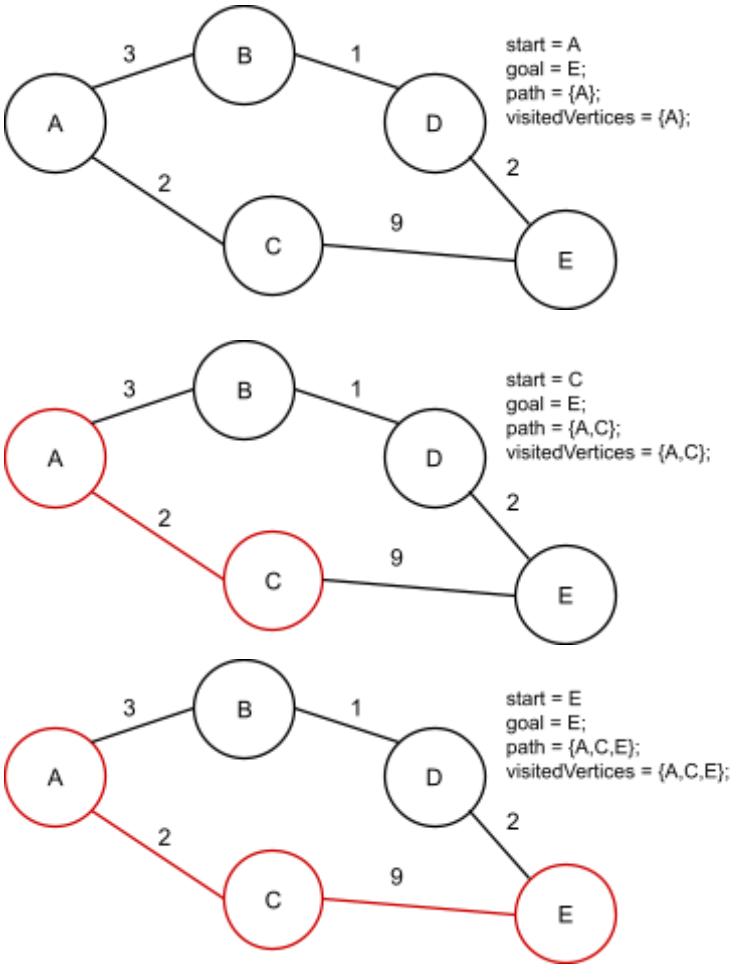


B's distance can't be updated (edge can't be relaxed) since B is already in the cloud (not in the queue).



C.) _____

The greedy strategy described in (C) does not always find the shortest path from start to goal. A counterexample where the strategy does not work can be seen below.



In this example, the shortest path from A to E is {A,B,D,E}, but following the strategy returns {A,C,E}. Although (A → C) is shorter than (A → B), the path that the strategy gets stuck on when choosing (A → C) over (A → B) ends up being longer than the path it would get stuck on when choosing (A → B) .

Additionally, had there not been an edge between C and E, the strategy would have never even reached and discovered E.

Problem #2

The situation can be modeled using a directed graph G, where each trading post is a vertex and each possible path between trading posts is a directed edge (from the starting post to the destination post). Therefore, we can use a modified version of the All-Pairs Shortest Path Algorithm to solve the problem of finding the minimum cost of a trip by canoe from each possible departure point i to each possible arrival point j. Although, translation of the problem into an actual graph is not required. This can be seen in the pseudocode below.

Pseudocode:

```
Algorithm allMinCosts(R)
    for i ← 1 to n
        for j ← 1 to n
            if (i = j)
                D0[i,i] ← 0
            else if (i ≤ j)
                D0[i,j] ← R[i,j]
            else
                D0[i,j] ← ∞
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                Dk[i,j] ← min{Dk-1[i,j], Dk-1[i,k]+Dk-1[k,j]}
    return Dn
```

The algorithm first cycles through all possible departure points (*i*) and all possible arrival points (*j*). It then sets all of the values (*i*,*j*) within the initial table D_0 to their values in the table *R[i,j]* (which includes the cost of a canoe picked up at a post *i* and dropped off at point *j*). If (*i*,*j*) is outside of the table (meaning $j > i$), its value in D_0 is set to ∞.

Next, we compare the cost of each possible way of traveling between trading posts (including stopping at multiple trading posts before reaching the final destination trading post). Once all three for-loops have concluded, $D_n[i,j]$ will hold the minimum possible cost for a trip between trading posts *i* and *j*. Therefore, we return D_n.

Assuming the table is implemented as an array of arrays (meaning accessing the element at any rank takes O(*1*) time), the two initial for-loops run in $O(n^2)$ time. Assuming the same implementation (meaning replacing the element at any rank takes O(*1*) time), the final three for-loops run in $O(n^3)$ time. Therefore, just like the All-Pairs Shortest Paths Algorithm, the above algorithm runs in $O(n^3)$ time.

Problem #3

I modified Dijkstra's Algorithm to solve this problem.

After inputting the graph representing the telephone network G and the two switching centers a and b, this algorithm will output the maximum bandwidth possible between the two centers.

Pseudo:

```
Algorithm maxBandwidth(G,a,b)
    if (a = b)
        throw sameCenterException
    Q ← new heap-based priority queue (maximum queue)
    for all e in G.edges()
        setLabel(e,"UNEXPLORED")
    for all v in G.vertices()
        if v = a
            setDistance(v,∞)
        else
            setDistance(v,-∞)
        l ← Q.insert(getDistance(v),v)
        setLocator(v,l)
    while (!Q.isEmpty())
        u ← Q.removeMax()
        if (u = b)
            return getDistance(u)
        for all e in G.incidentEdges(u)
            if (getLabel(e) = "UNEXPLORED")
                setLabel(e,"VISITED")
                z ← G.opposite(u,e)
                if (weight(e) < getDistance(u))
                    r ← weight(e)
                else
                    r ← getDistance(u)
                if (r > getDistance(z))
                    setDistance(z,r)
                    Q.replaceKey(getLocator(z),r)
```

Worst-Case Runtime: O(*mlog(n)*)

The worst-case runtime of the algorithm would occur when b is the last vertex to be removed from the queue. The following analysis assumes this worst-case, along with the assumption that the graph is represented with an adjacency structure.

First, labeling all edges as “UNEXPLORED” takes O(*m*) time. Like in Dijkstra’s Algorithm, setting and/or getting the distances, locators, labels, and weights of edges is done O(Σ_v*(deg(v))* = *2m*) times. Since each of these actions are O(*1*), this part of the algorithm runs in O(*m*) time. The incidentEdges() method is called once per vertex, totaling to a runtime of O(Σ_v*(deg(v))* = *2m*). Inserting and removing each vertex from the priority queue takes O(*nlog(n)*) time total, while updating the keys of vertices (occurring $\Sigma_v$deg(v) = 2m times at the most) can take up to O(*mlog(n)*) time. Adding all of these factors together, we get a total worst-case runtime of O((*n+m)log(n)*).

If we can assume that the entire network is connected, this means that $n - 1 \leq m$. Therefore, we can write the worst-case runtime as O(*mlog(n)*).

Problem #4

This problem can be solved with a slight modification to the partition-based implementation of Kruskal's Algorithm. The only alteration is that, instead of removing the edge with the minimum value each loop, we remove the edge with the maximum value. This works since, if there were a way to connect two stations (or two sets) with an edge with a larger bandwidth, this edge would have been dequeued before any smaller edges were.

In the end, this algorithm will produce a tree *T* containing all of the paths that would result in the maximum total bandwidth across *n-1* stations. This *T* is the maximum-spanning tree of *G*.

Pseudocode:

```
Algorithm maxBandwidth(G)
    P ← partition of the vertices in V, where each vertex v
        forms a separate set.
    Q ← priority queue storing each edge e in E, sorted by
        their weights w(e)(which represent their bandwidths).
    T ← a new, initially empty tree
    while (!Q.isEmpty())
        (u,v) ← Q.removeMaxElement()
        if (P.find(u) != P.find(v))
            Add (u,v) to T
            P.union(u,v)
    return T
```

Worst-case Runtime: O(*(n+m)log(n)*)

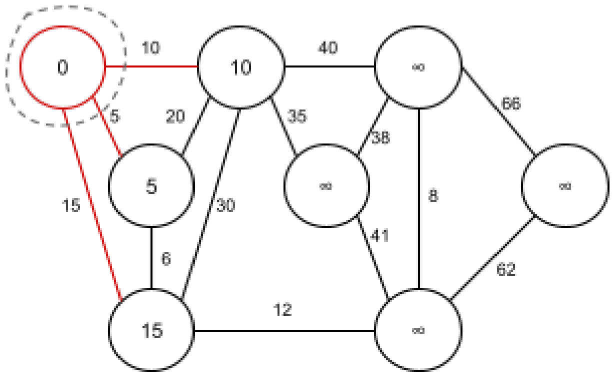
The priority queue *Q* must be initialized, which can be done in O(*m*) time using bottom-up heap construction. Each edge must also be removed from the priority queue, as dictated in the while-loop. Each removal takes O(*log(m)*) time, making all removals take O(*mlog(m)*) time. Since *G* is simple (each pair of stations has one bandwidth), this is equivalent to O(*mlog(n)*). Since each vertex can be moved to a new set, at most, *log(n)* times, the total number of set unions (P.union(*u*,*v*)) is less than or equal to *nlog(n)*.

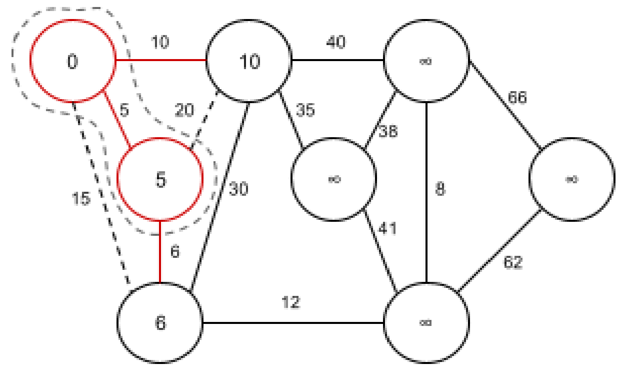
Adding together all of this information, we can conclude that the algorithm has a worst-case runtime of O(*(n+m)log(n)*).

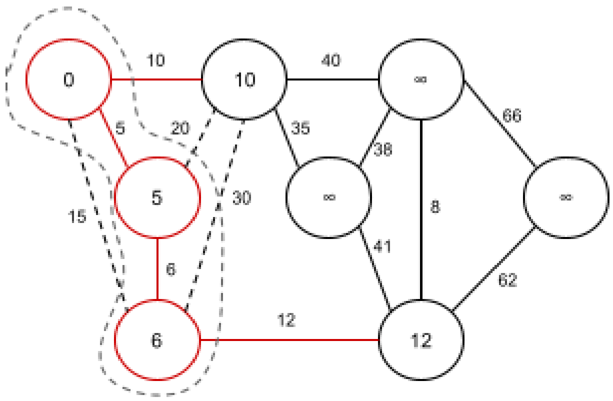
Problem #5

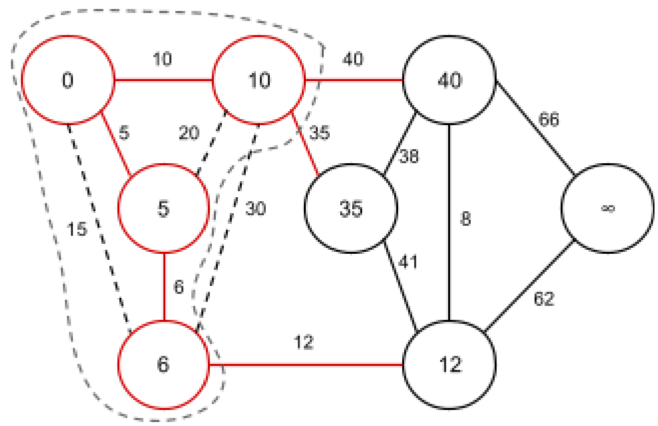
A.) ____

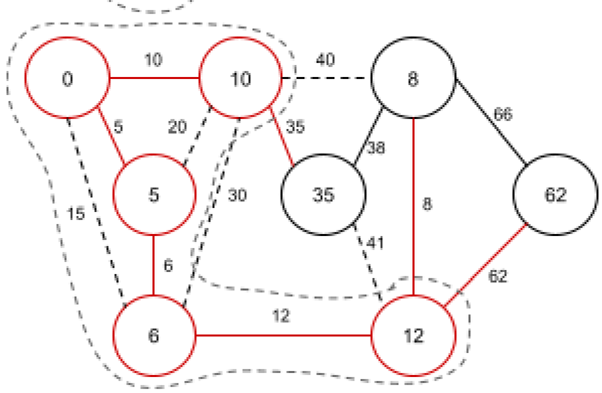
Note: The D values of each vertex are shown within each vertex.

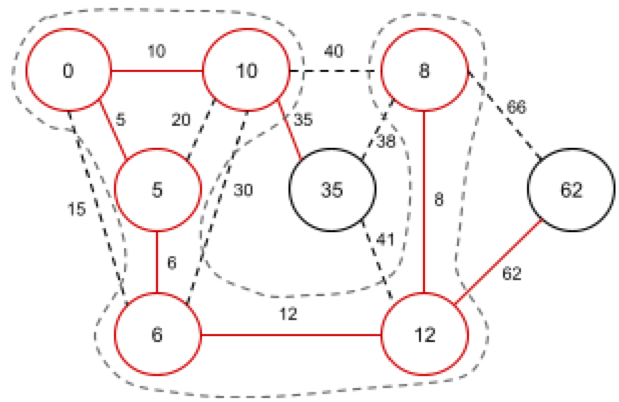


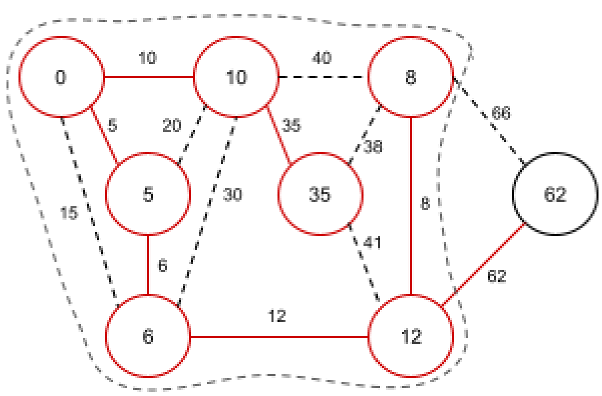


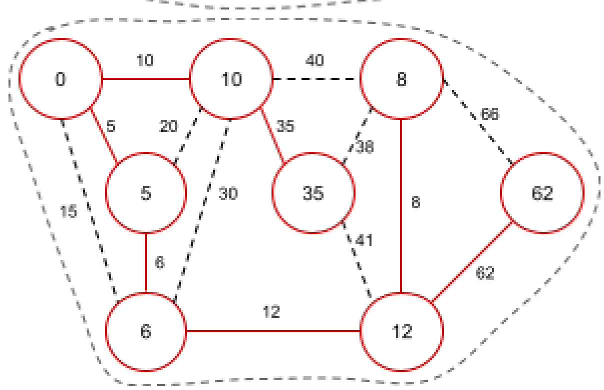




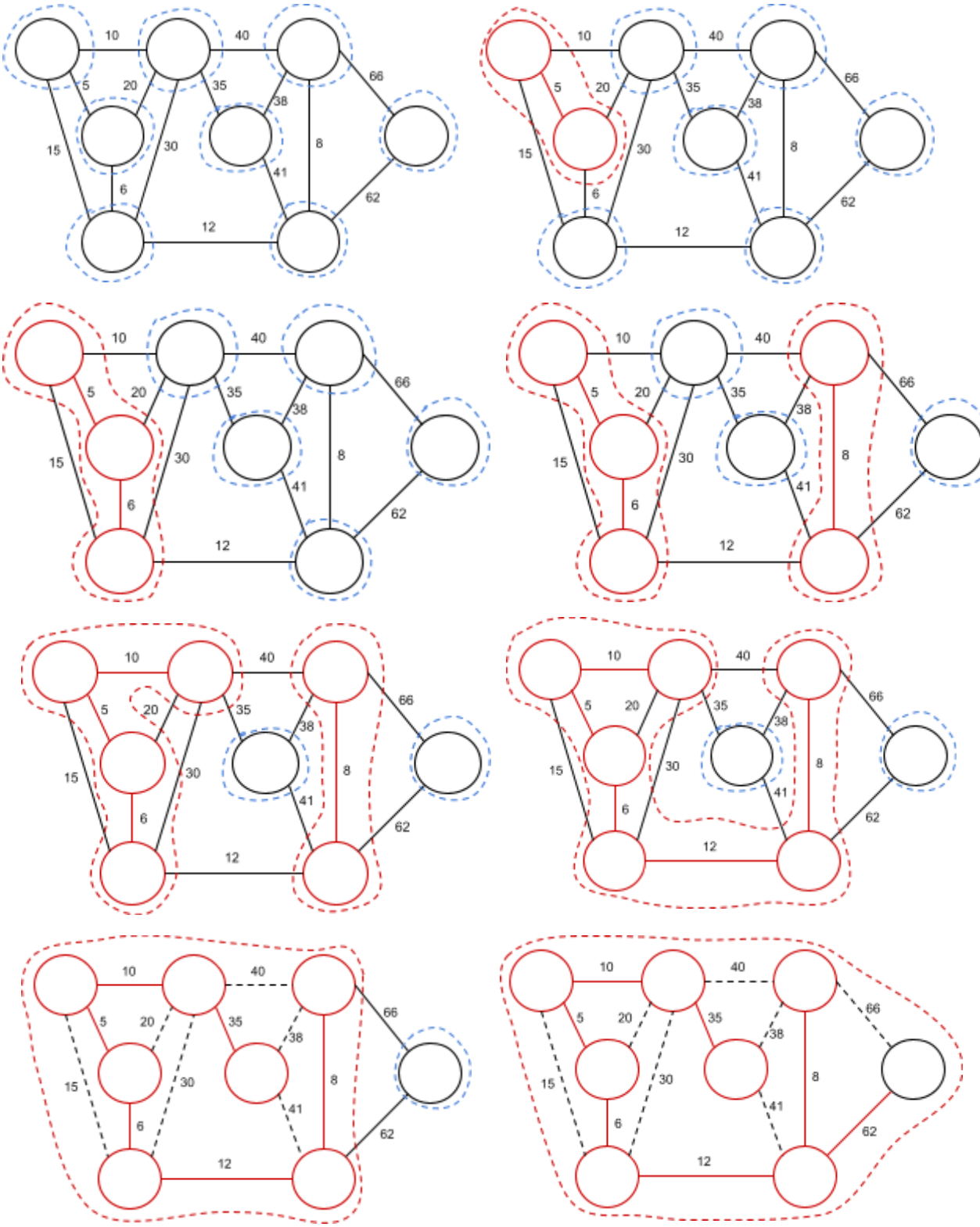








B.) _____



Order of Edges Added to the MST: 5, 6, 8, 10, 12, 35, 62