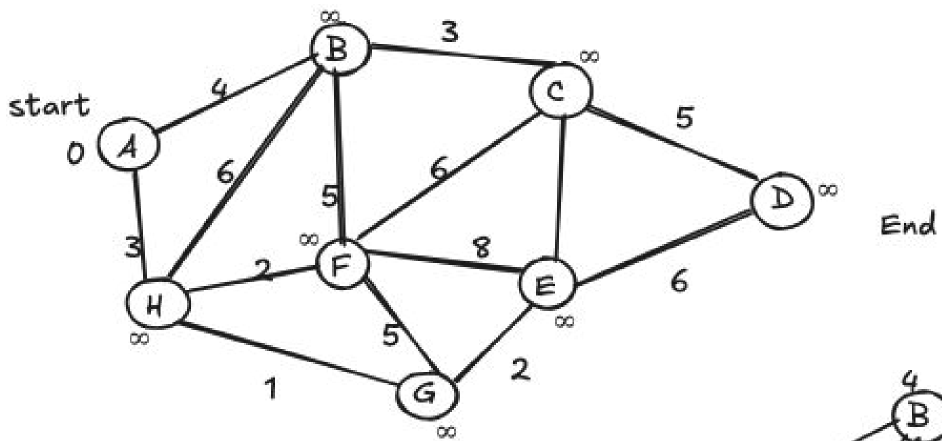
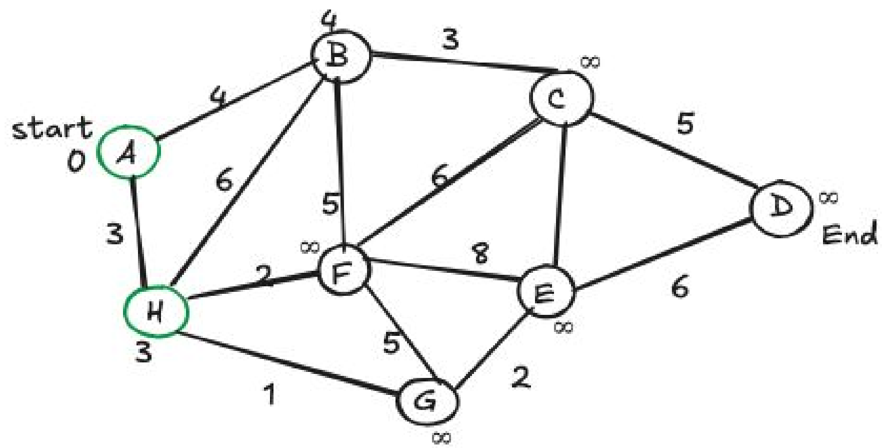
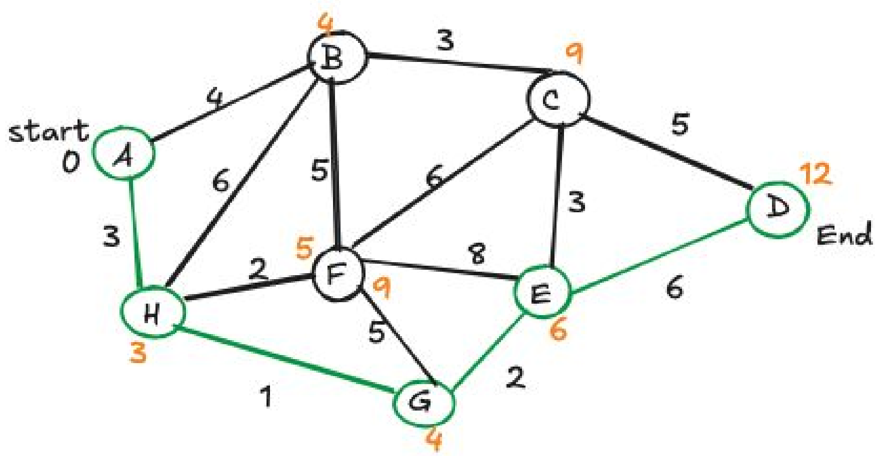
Assignment 14a-Shortest Path

R-7.1 Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Identify one vertex as a "start" vertex and illustrate a running of Dijkstra's shortest path algorithm on this graph

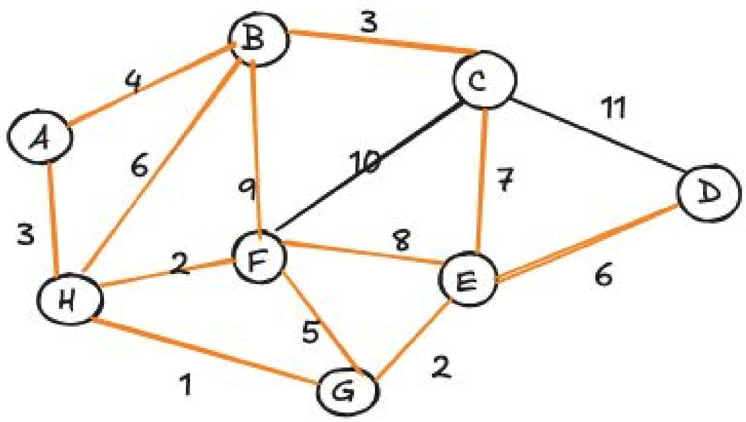




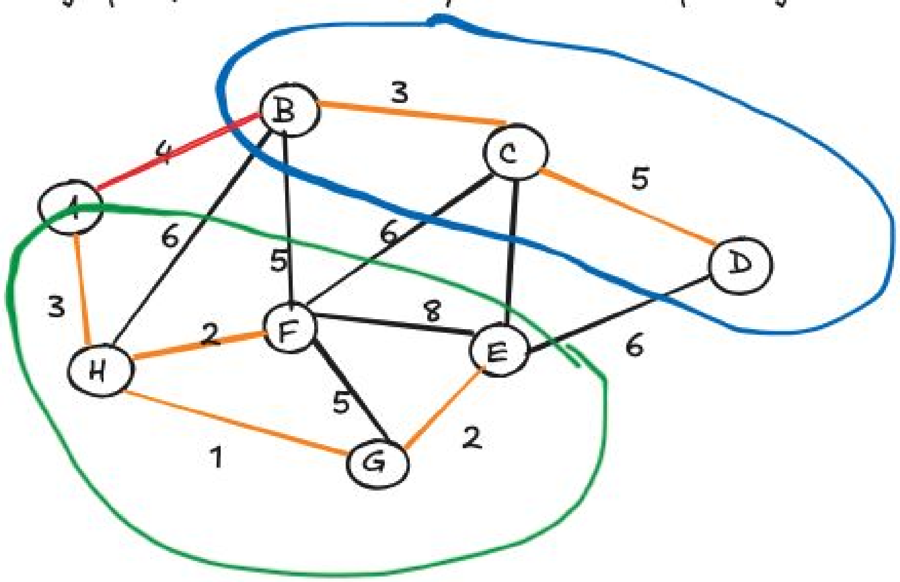


Assignment 14b – Minimum Spanning Tree

R-7-8 Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Illustrate the execution of Prim-Jarvik's algorithm on this graph. (Note there is only one minimum spanning tree for this graph.)



R-7-9 Draw a simple, connected, undirected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Illustrate the execution of Baruvka algorithm on this graph. (Note there is only one minimum spanning tree for this graph.)



Consider the following potential MST algorithms based on the generic MST algorithm. Which, if any, successfully computes a MST? Hint: to show that an algorithm does not compute an MST, all you need to do is find a counterexample. If it does, you need to argue why based on the cycle property and/or the partition property.

a) Algorithm MST-a(G, w)

```
T  edges in E sorted in nonincreasing order of edge weights
for each e in T do {each e is taken in nonincreasing order by weight}
if T − {e} is a connected graph then
T  T − {e} {remove e from T}
return T
```

Answer: Correct – Computes an MST

Reasoning (Cycle Property):

The algorithm starts with all edges, sorted in nonincreasing order (heaviest first).

It removes any edge if removing it does not disconnect the graph.

By the cycle property, the heaviest edge in any cycle is never part of an MST.

Thus, after removing all such heavy edges, what remains is a Minimum Spanning Tree.

b). Algorithm MST-b(G, w)
T <- { }
for each e in E do { e is taken in arbitrary order }
if T ∪ {e} has no cycles then
T <- T ∪ {e} {add e to T}
return T

Answer: Incorrect – Does NOT always compute an MST

Counterexample (proves it wrong):

Consider the graph:

Vertices: A, B, C
Edges:
A-B: 10
A-C: 2
B-C: 1

Correct MST = Edges (A-C, B-C), Total Weight = 3

MST-b Execution (bad order: [A-B(10), A-C(2), B-C(1)]):

Add A-B (10) → T = {A-B}

Add A-C (2) → T = {A-B, A-C}

Skip B-C (1) because it forms a cycle

Result: Total weight = 12, which is not minimum.

Thus, MST-b fails.

c) Algorithm MST-c(G, w)

```
T <- { }
for each e in E do { e is taken in arbitrary order }
T <- T U {e} {add e to T}
if T now has a cycle C then
if e' is the edge of C with the maximum weight then
T <- T - {e'} {remove e' to T}
return T
```

Answer: Correct – Computes an MST

Reasoning (Cycle Property):

The algorithm adds edges in any order but removes the heaviest edge in every cycle.

By the cycle property, the heaviest edge in a cycle cannot belong to any MST.

Therefore, the remaining edges form a Minimum Spanning Tree.

A.

```
BFS(G):
    for each vertex v in G.vertices():
        v.setLabel(UNEXPLORED)
        v.distance <- INF
    for each edge e in G.edges():
        e.setLabel(UNEXPLORED)
    for each vertex v in G.vertices():
        if v.getLabel() == UNEXPLORED:
            BFSComponent(G, v)

BFSComponent(G, s):
    s.setLabel(VISITED)
    s.distance <- 0
    Q <- new Queue()
    Q.enqueue(s)

    while not Q.isEmpty():
        v <- Q.dequeue()
        startVisit(v)
        for each edge e incident to v:
            if e.getLabel() == UNEXPLORED:
                w <- opposite(v, e)
                if w.getLabel() == UNEXPLORED:
                    e.setLabel(DISCOVERY)
                    traverseDiscovery(e, v, w)
                    Q.enqueue(w)
                else:
                    e.setLabel(CROSS)
                    traverseCross(e, v, w)
        finishVisit(v)


startVisit(v):
    // Nothing extra needed

traverseDiscovery(e, v, w):
    w.setLabel(VISITED)
    w.distance <- v.distance + 1

traverseCross(e, v, w):
    // Nothing extra needed

finishVisit(v):
    // Nothing extra needed
```

B.

```
BFS Template with Subtree Check
isSubtree(G, T):
    for each vertex v in G.vertices():
        v.setLabel(UNEXPLORED)
        v.parent <- null

    countVertices <- 0
    countEdges <- 0
    hasCycle <- false

    pick any vertex s that is in T
    BFSSubtree(G, s, T)

    if hasCycle: return false
    if countEdges/2 != countVertices - 1: return false
    return true

BFSSubtree(G, s, T):
    s.setLabel(VISITED)
    Q <- new Queue()
    Q.enqueue(s)
    s.parent <- null

    while not Q.isEmpty():
        v <- Q.dequeue()
        startVisit(v)
        for each edge e incident to v:
            if e not in T: continue
            w <- opposite(v, e)
            if w.getLabel() == UNEXPLORED:
                e.setLabel(DISCOVERY)
                traverseDiscovery(e, v, w)
                Q.enqueue(w)
            else if w != v.parent:
                e.setLabel(BACK)
                traverseBack(e, v, w)
        finishVisit(v)
```

Override Hook Methods

```
startVisit(v):
    global countVertices
    countVertices <- countVertices + 1

traverseDiscovery(e, v, w):
    global countEdges
    countEdges <- countEdges + 1
    w.setLabel(VISITED)
    w.parent <- v

traverseBack(e, v, w):
    global hasCycle, countEdges
    countEdges <- countEdges + 1
    hasCycle <- true

finishVisit(v):
    // Nothing extra needed
```

C-5.1

1\. minStopsRoad(wateringHoles, k):

```
stops <- 0
currentPos <- 0

while currentPos < destination:
    nextPos <- furthest watering hole reachable from currentPos within k miles
    if nextPos == currentPos:
        return "Impossible"
    if nextPos != destination:
        stops <- stops + 1
    currentPos <- nextPos

return stops
```

C-5.1

2\. minStopsDesert(G, start, goal, k):

```
for each vertex v in G.vertices():
    v.visited <- false
    v.stops <- ∞
start.visited <- true
start.stops <- 0

Q <- new Queue()
Q.enqueue(start)

while not Q.isEmpty():
    v <- Q.dequeue()
    for each neighbor w of v:
        if distance(v, w) <= k and w.visited == false:
            w.visited <- true
            w.stops <- v.stops + 1
            Q.enqueue(w)
if goal.stops == ∞:
    return "Impossible"
return goal.stops - 1
```