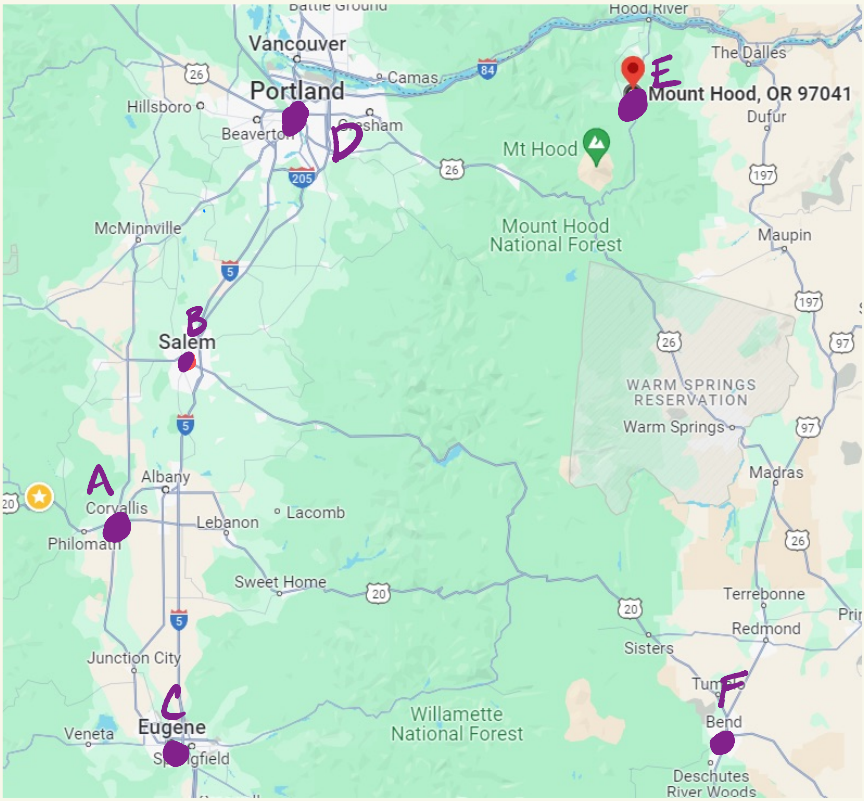
ADD NODES



Main.cpp

```
GraphNode A {"Corvallis"};
GraphNode B {"Salem"};
GraphNode C {"Eugene"};
GraphNode D {"Portland"};
GraphNode E {"Mt Hood"};
GraphNode F {"Bend"};
```

```
Graph. add_node(&A);
Graph. add_node(&F);
```

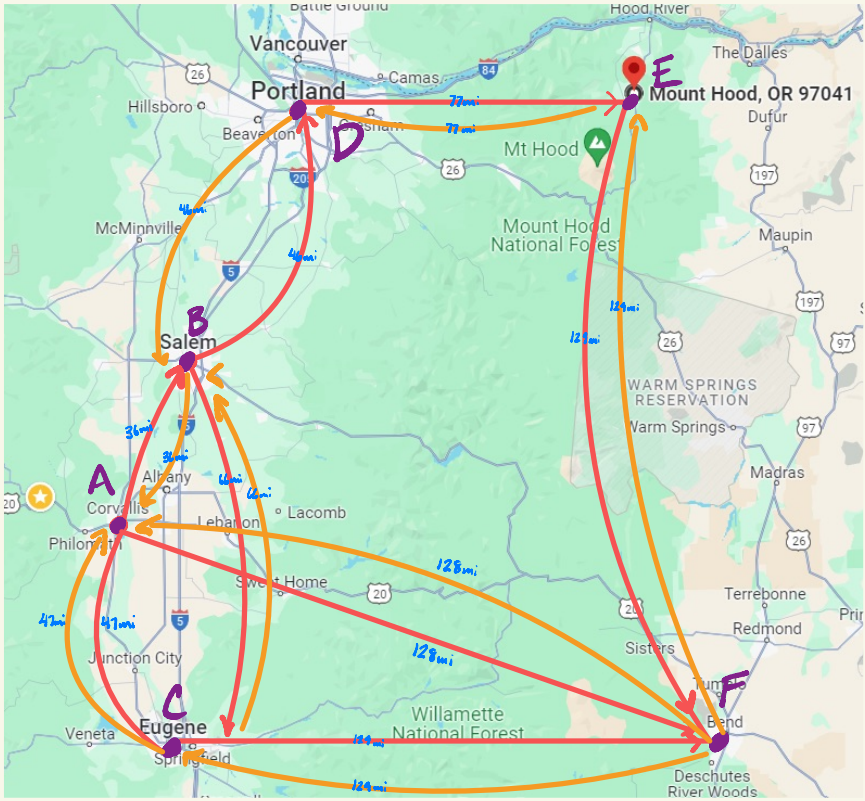
Graph.h

```
add_node (GraphNode *new_node) {
    nodes.push_back(new_node);
}
```

Current nodes in Graph:

Corvallis
Salem
Eugene
Portland
Mt Hood
Bend

ADD EDGES & WEIGHTS



MAIN.CPP

```
EDGE AF {128, &A, &F};
EDGE FA {128, &F, &A};
```

```
GRAPH.ADD_EDGE(&AF);
GRAPH.ADD_EDGE(&FA);
```

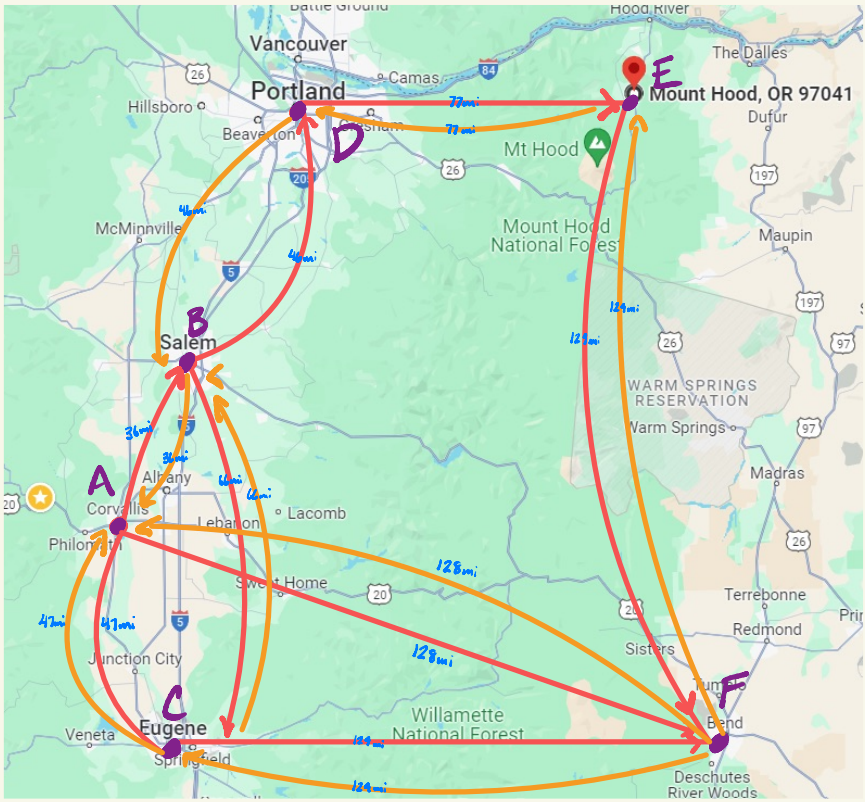
GRAPH.H

```
ADD_EDGE(EDGE *NEW_EDGE) {
NEW_EDGE -> SOURCE -> NEIGHBORS.push_back(NEW_EDGE);
}
```

MAIN TERMINAL

```
AF (WEIGHT, SRC->NAME, DST->NAME): (128, CORVALLIS, BEND)
FA (WEIGHT, SRC->NAME, DST->NAME): (128, BEND, CORVALLIS)
```

Shortest Path Algorithm (Dijkstra's)



Shortest Paths From Corvallis [mi]

Unvisited						Shortest Path						Visited
						A	B	C	D	E	F	
1	B	C	D	E	F	0	36	47	∞	∞	128	A
2		C	D	E	F	0	36	47	(46+36)	∞	128	B
3		C		E	F	0	36	47	82	(77+46+36)	128	D
4		C			F	0	36	47	82	159	128	E
5		C				0	36	47	82	159	128	F
6						0	36	47	82	159	128	C

Shortest Paths From Eugene [mi]

Unvisited						Shortest Path						Visited
						A	B	C	D	E	F	
1	A	B	D	E	F	47	66	0	∞	∞	129	C
2		B	D	E	F	47	66	0	∞	∞	129	A
3			D	E	F	47	66	0	(46+66)	∞	129	B
4				E	F	47	66	0	112	(77+46+66)	129	D
5					F	47	66	0	112	189	129	E
6						47	66	0	112	189	129	F

Test Program Against These Values

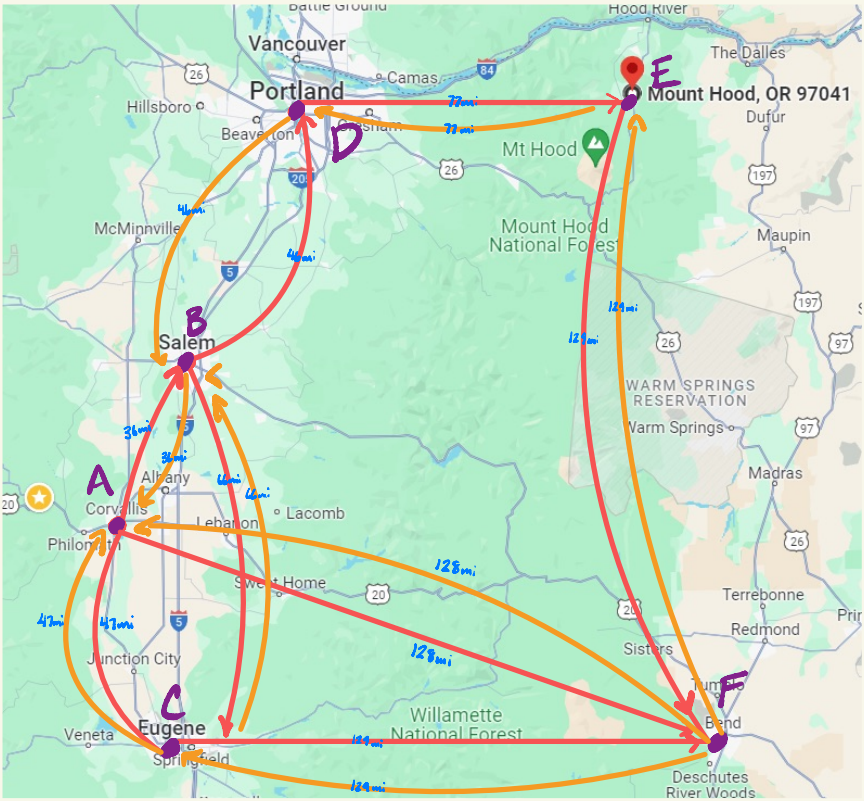
FINDING SHORTEST PATH:

- SET DISTANCES FOR ALL NODES TO INFINITY
- SET DISTANCE TO THE START NODE TO ZERO (DISTANCE TO ITSELF)
- STORE UNVISITED NODES (CITIES) & WEIGHTS (DISTANCES) IN A QUEUE (Q)
- SET THE SOURCE NODE (DISTANCE = 0) ON TOP OF THE Q
- WHILE Q IS NOT EMPTY:
    - GET CURRENT NODE (SMALLEST DISTANCE) FROM TOP of Q
    - POP() CURRENT NODE OFF Q
    - IF CURRENT NODE → DISTANCE > DISTANCE → CURRENT NODE:
        - * SKIP THIS NODE, ALREADY HAVE A KNOWN SHORTER PATH
    - FOR ALL NEIGHBORING NODES of CURRENT NODE (CONNECTED BY EDGE):
        - * DETERMINE NEW DISTANCE TO ALL NEIGHBORING NODES
        - IF NEW DISTANCE < KNOWN DISTANCE:
            - * UPDATE SHORTEST DISTANCE TO NEIGHBOR
            - * PUSH() NEW DISTANCE & NEIGHBOR ONTO THE Q
    - UPDATE NEW CURRENT NODE
- STORE EACH NODE & SHORTEST DISTANCE IN A VECTOR
- RETURN RESULT VECTOR

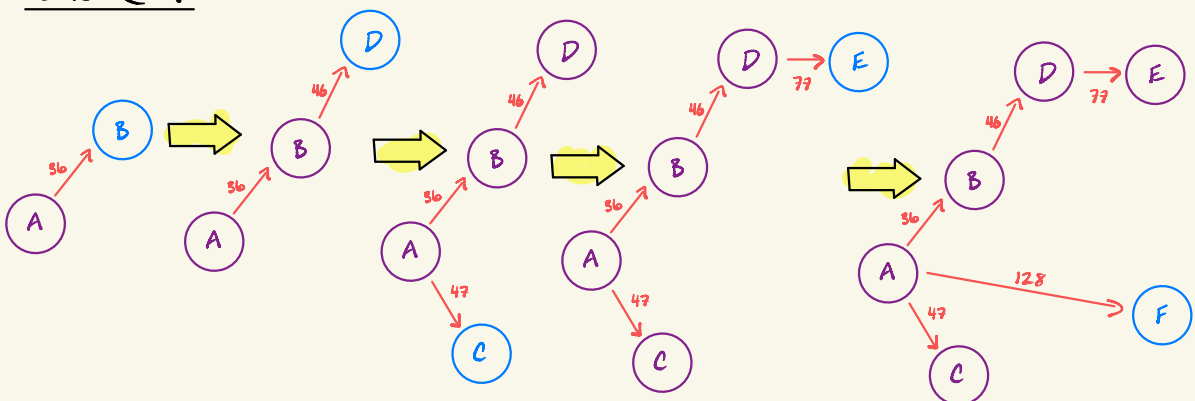
- CONSIDER HASH TABLE TO STORE DISTANCES?
    - KEY AS GRAPHNODE* & VALUE IS SHORTEST DISTANCE
        - * STILL HAVEN'T TURNED IN ASSIGNMENT #7...

- NEED TO ORGANIZE Q, SHORTEST DISTANCE ALWAYS ON TOP

(Prim's) MST Algorithm – Test #1

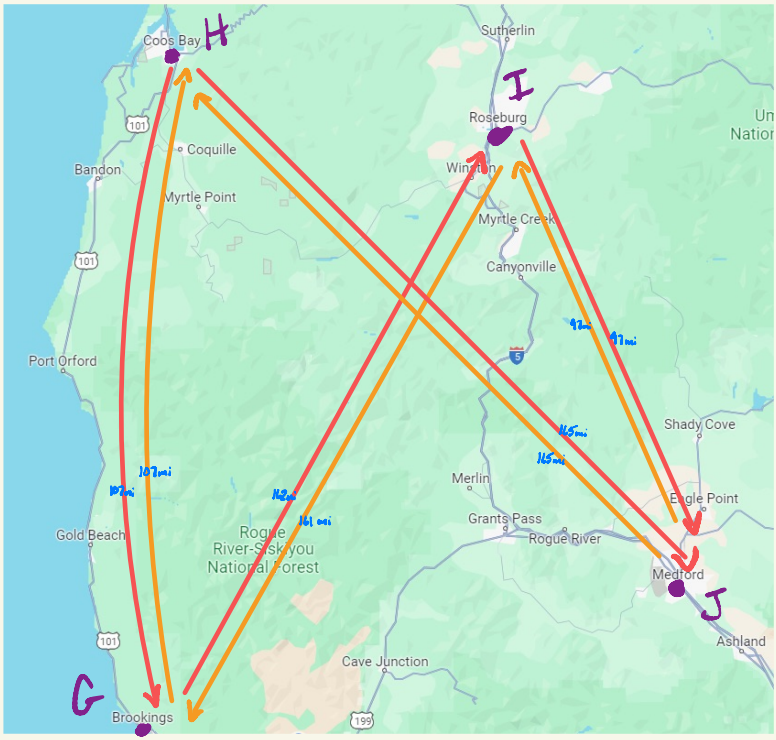


START @ 'A'

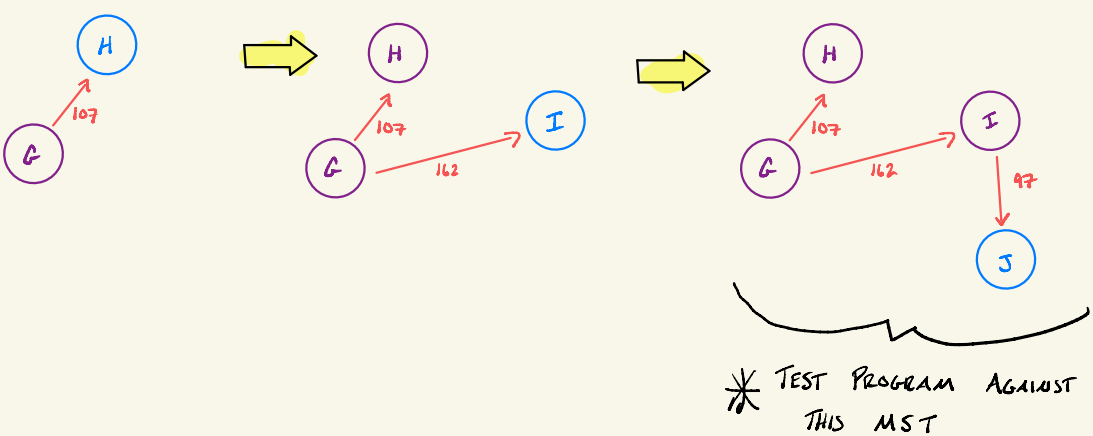


* TEST PROGRAM AGAINST THIS MST

(Prim's) MST Algorithm – Test #2



START @ 'G'



* TEST PROGRAM AGAINST THIS MST

Finding MST:

- Vector to store edges of MST
- Q to store edge lengths & Edge* (ordered)
- Hash table to track if nodes are in vector? (Key -> GraphNode*, Value -> T/F) ?
  - * Still need assignment 7 ...
- Add current node's edges to Q
- While Q is not empty & Vector contains < (nodes.size() - 1) edges:
  - Find edge with shortest distance in Q
  - Pop() that edge from Q
  - If destination node of this is already in vector:
    - * Skip this edge, cannot create a cycle
  - Add edge to the vector
  - Add edge to the Q
  - Return Vector

- Need another function to iterate over neighbors, check if edges are in vector (T/F), and add missing (F) edges to Q
- Need a way to sort Q (shortest -> longest edges)