Algorithms and Data Structures Part 4

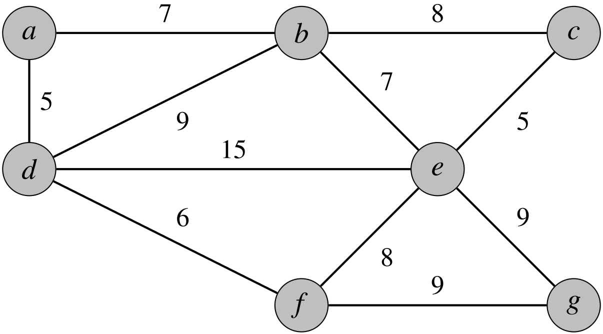
Lecture 9: Implementing MST Algorithms

George Mertzios

george.mertzios@durham.ac.uk

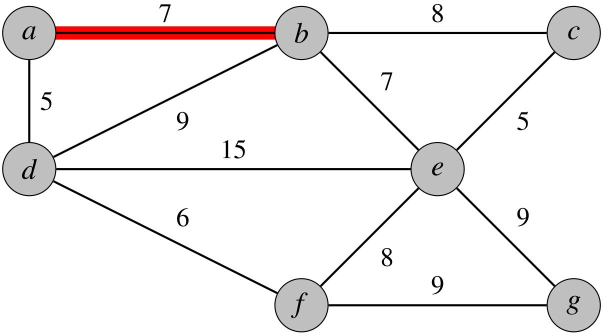
Prim's algorithm

Start at b:



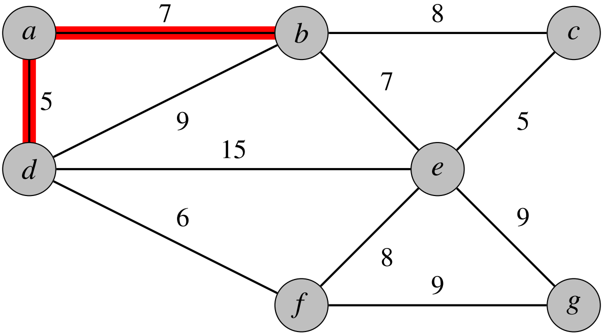
Prim's algorithm

Start at b:



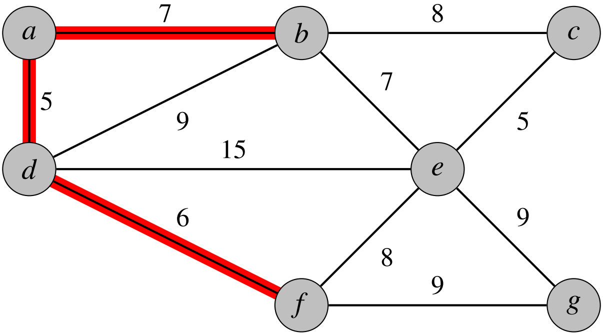
Prim's algorithm

Start at b:



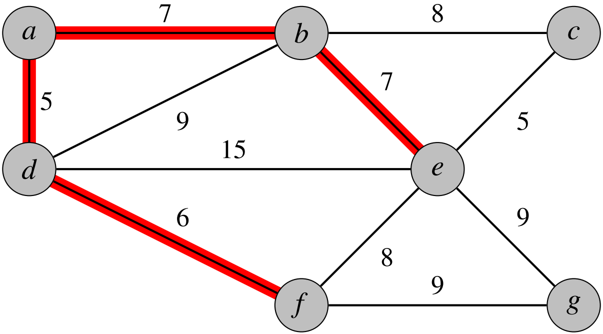
Prim's algorithm

Start at b:



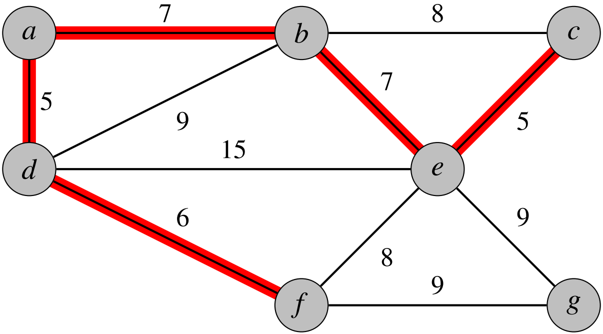
Prim’s algorithm

Start at b:



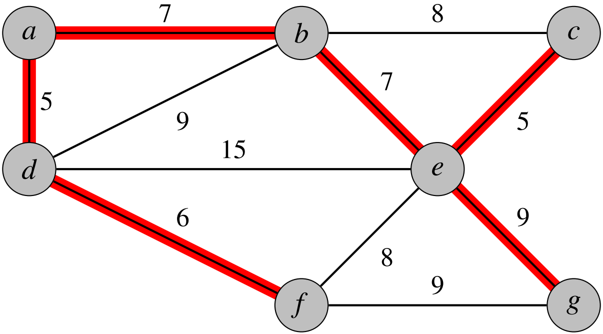
Prim's algorithm

Start at b:



Prim's algorithm

Start at *b*:



Prim's Algorithm: simple implementation

V is the set of vertices
E is the set of edges
$U = \{u\}$
A is the empty set (will add edges until it is MST)
while $U \neq V$ **do**
    choose $e = (v, w)$ in E such that $v \in U$, $w \notin U$,
        and e has min cost
    $A = A + e$
    $U = U + w$
end while
return A

Prim's Algorithm: simple implementation

V is the set of vertices
E is the set of edges
$U = \{u\}$
A is the empty set (will add edges until it is MST)
while $U \neq V$ **do**
    choose $e = (v, w)$ in E such that $v \in U, w \notin U$,
        and e has min cost
    $A = A + e$
    $U = U + w$
end while
return A

- Iterate through while loop once for each vertex.
- Need to check every edge each time.
- Naive implementation: running time $O(VE)$

A better implementation

- We want to avoid checking all the edges.
- For each vertex v not yet in U, we only want to know the least-weight edge from v to a vertex in U.

A better implementation

- We want to avoid checking all the edges.
- For each vertex v not yet in U, we only want to know the least-weight edge from v to a vertex in U.
- So we maintain an array to record these values — then we just have to check this array to find which edge to pick next (and then maybe perform some updates).

Prim's Algorithm: improved implementation

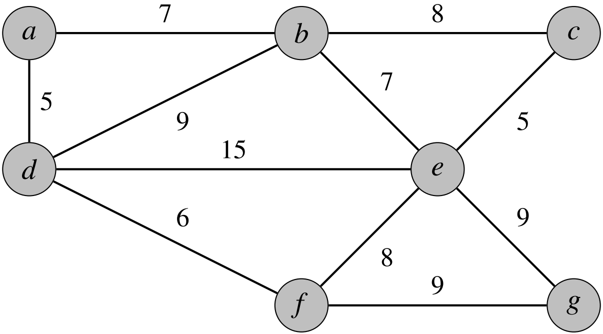
V is the set of vertices
E is the set of edges
$U = \{u\}$
A is the empty set (will add edges until it is MST)
for each vertex v except u **do**
    $B(v)$ is the least-weight edge from v to U
end for
while $U \neq V$ **do**
    choose v with minimum cost $B(v)$
    $A = A + e$
    $U = U + v$
    update B
end while
return A

Prim's Algorithm

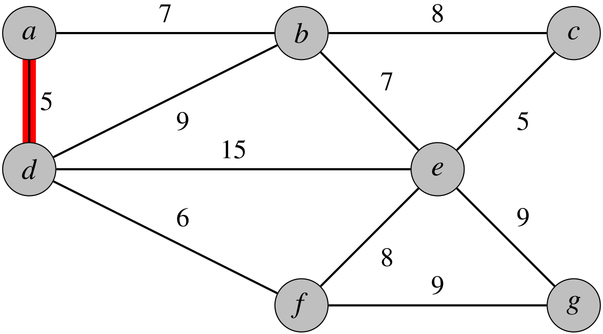
Implement the array using a Priority Queue (using a heap, for example).

- To initialize, all edges considered.
- Iterate through While loop once for each vertex.
- Extracting the minimum cost edge and performing updates take $O(\log V)$ time.
- Running time $O(V \log V + E)$.

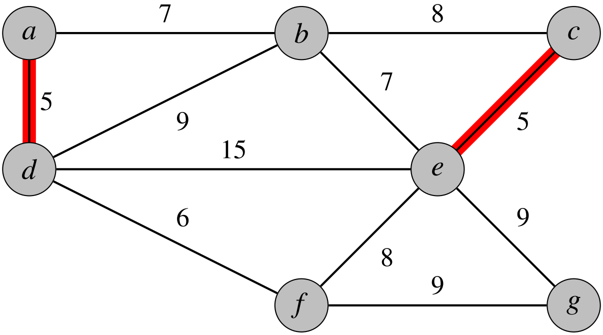
Kruskal's algorithm



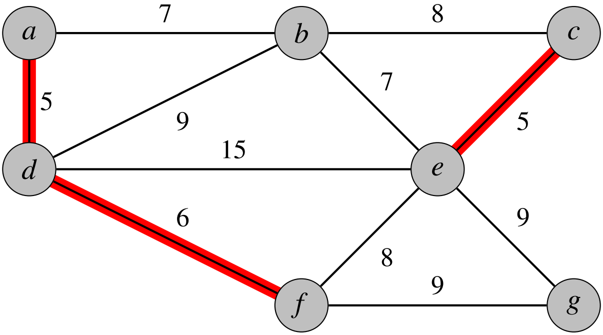
Kruskal's algorithm



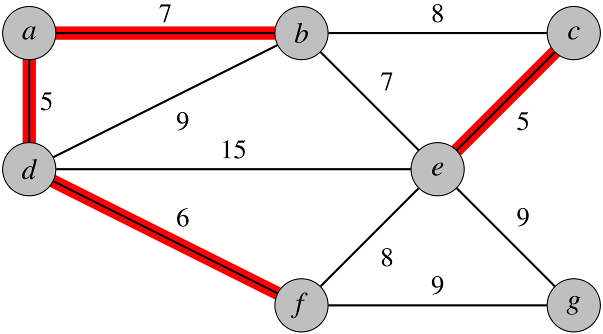
Kruskal's algorithm



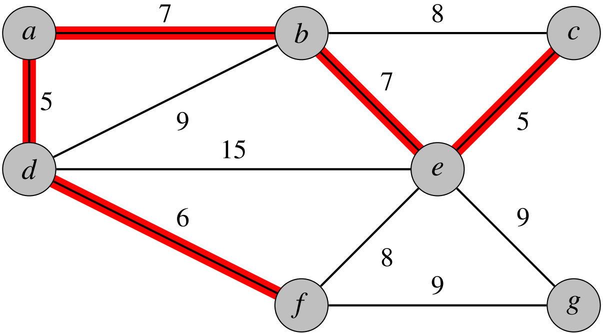
Kruskal's algorithm



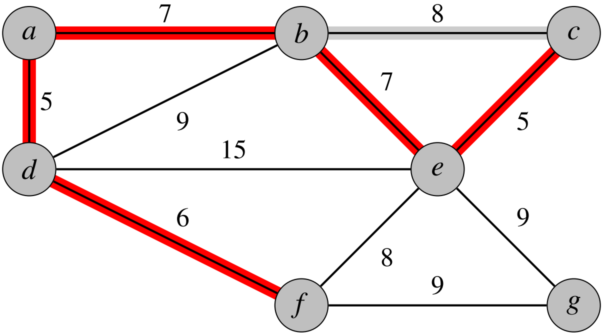
Kruskal's algorithm



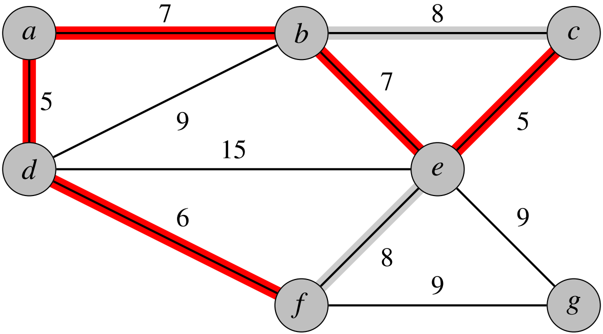
Kruskal’s algorithm



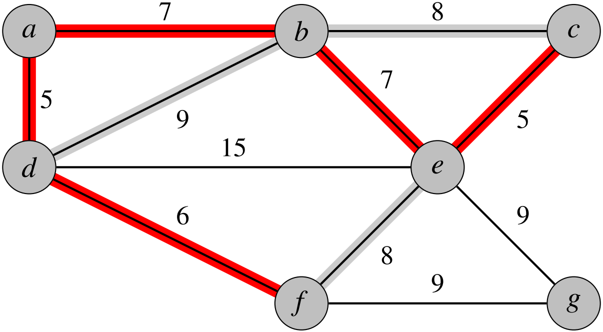
Kruskal's algorithm



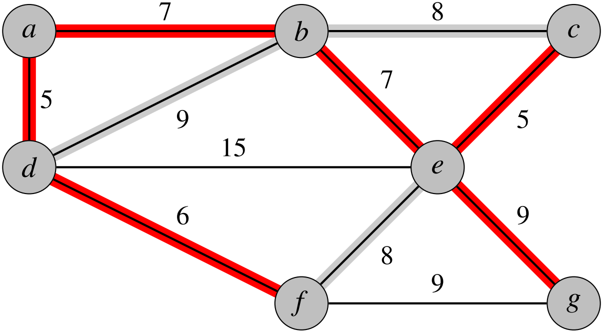
Kruskal's algorithm



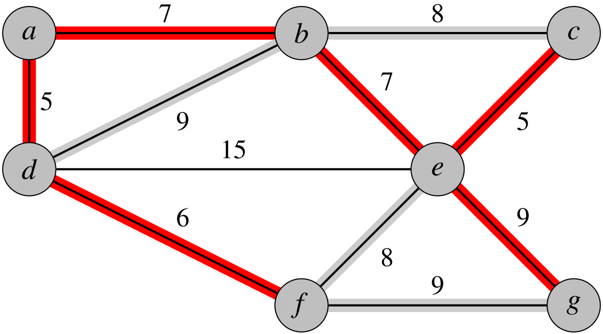
Kruskal's algorithm



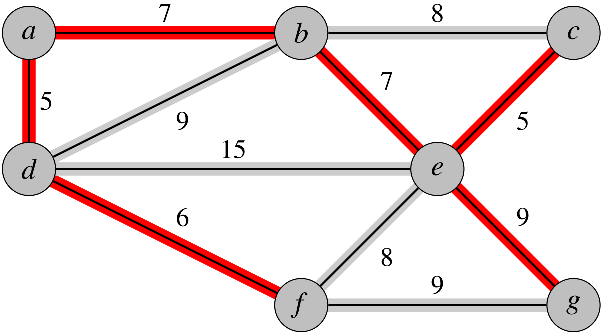
Kruskal's algorithm



Kruskal's algorithm



Kruskal's algorithm



Kruskal's Algorithm: simple implementation

```
V is the set of vertices, E is the set of edges
A is the empty set (will add edges until it is MST)
sort E
while E is not empty do
    choose e in E with min cost
    if A + e contains no cycle then
        add e to A
    end if
end while
return A
```

Kruskal's Algorithm: simple implementation

```
V is the set of vertices, E is the set of edges
A is the empty set (will add edges until it is MST)
sort E
while E is not empty do
    choose e in E with min cost
    if A + e contains no cycle then
        add e to A
    end if
end while
return A
```

- Sorting initially takes time $O(E \log E) = O(E \log V)$.
- Iterate through while loop once for each edge.
- Need to check every time for a cycle using, for example, depth-first search — takes time $O(V + E)$.
- Running time $O(E \log V) + O(E(V + E))$

A better implementation

- We want to avoid checking for cycles all the time.

A better implementation

- We want to avoid checking for cycles all the time.
- For each vertex v, if we could look-up which component of the partially built tree it belongs to, then . . .
- . . . we could decide quickly whether two vertices can be joined by an edge to the same component — if not, we can add the edge.
- So we maintain an array to record this.

Kruskal's Algorithm: improved implementation

V is the set of vertices, E is the set of edges
A is the empty set (will add edges until it is MST)
for each vertex v **do**
    $C(v) = \{v\}$ (each vertex in component by itself)
end for
sort E
while E is not empty **do**
    choose $e = (u, v)$ in E with min cost
    if $C(u) \neq C(v)$ **then**
        add e to A
        for each vertex w in $C(u)$ and $C(v)$ **do**
            update $C(w)$ with $C(u) \cup C(v)$
        end for
    end if
end while
return A

Kruskal's Algorithm

Implement the array using Union-Find data structure.

- Sorting initially still takes time $O(E \log V)$.
- Union-Find operations also take time $O(E \log V)$.
- So total running time $O(E \log V)$