Lecture 5b: Implementing MST Algorithms

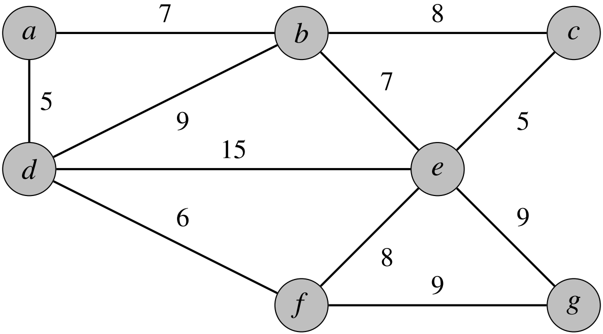
Amitabh Trehan

amitabh.trehan@durham.ac.uk

**Based on the slides of ADS-21/22 by Dr. George Mertzios*

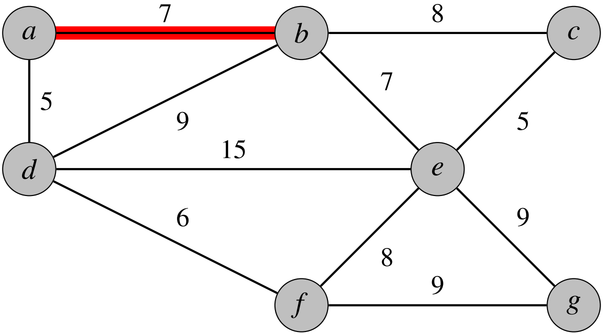
Prim's algorithm

Start at b:



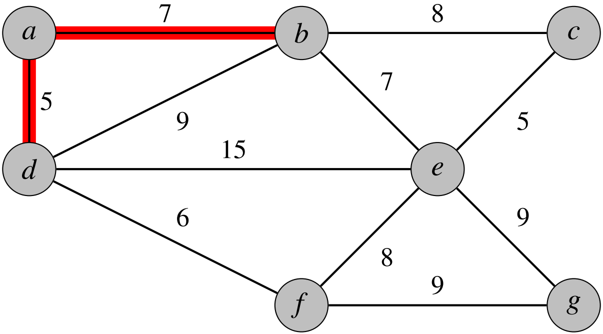
Prim's algorithm

Start at *b*:



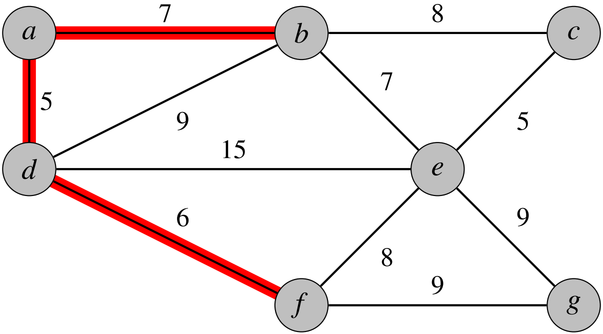
Prim's algorithm

Start at b:



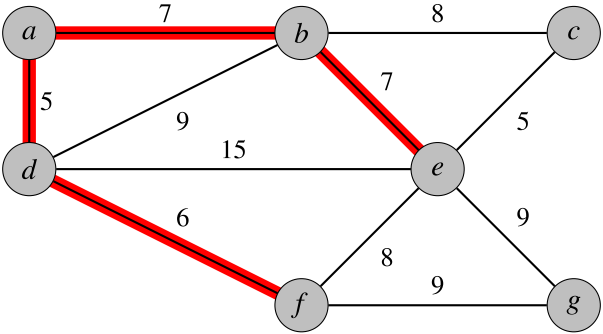
Prim's algorithm

Start at b:



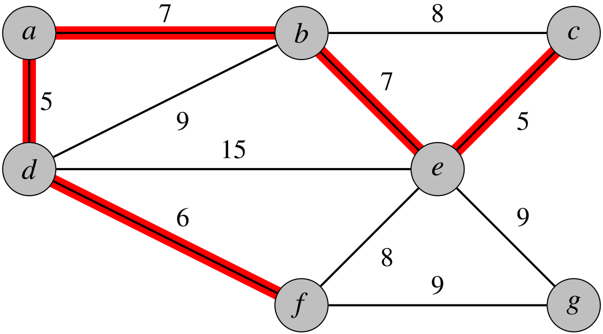
Prim's algorithm

Start at b:



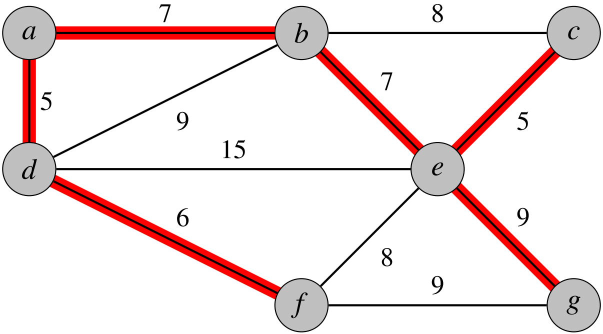
Prim's algorithm

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

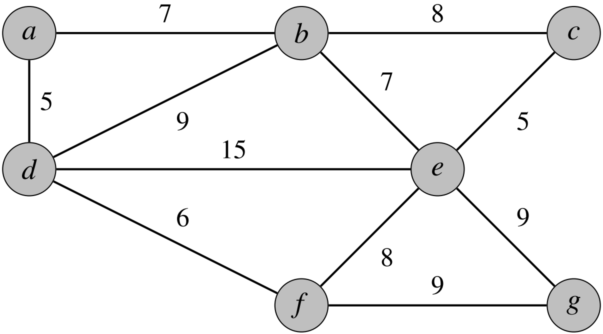
V is the set of vertices
E is the set of edges
$U = \{u\}$
A is the empty set (will add edges until it is MST)
for each vertex v except u **do**
    $B(v)$ is the least-weight edge from v to U
end for
while $U \neq V$ **do**
    choose v with minimum cost $B(v)$
    $A = A + e$
    $U = U + v$
    update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

Prim's Algorithm

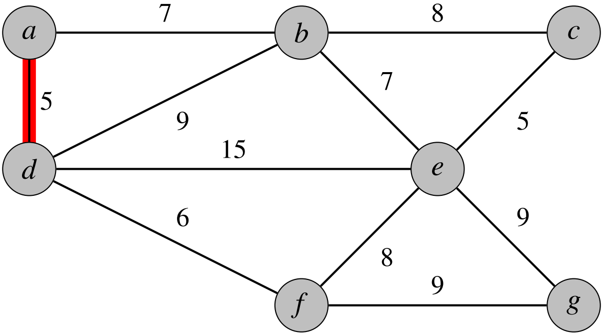
Implement the array using a Priority Queue (using a heap, for example).

- 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 $O(V \log V + E)$.

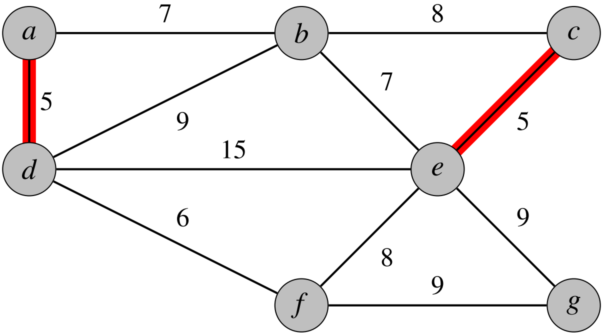
Kruskal's algorithm



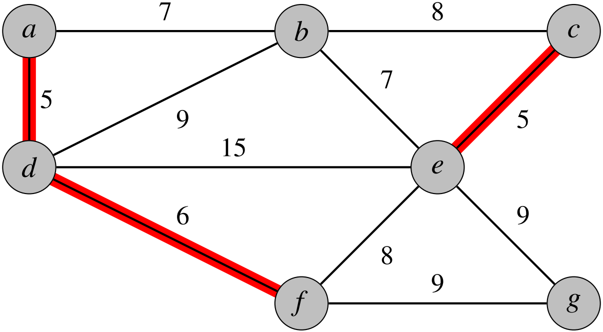
Kruskal's algorithm



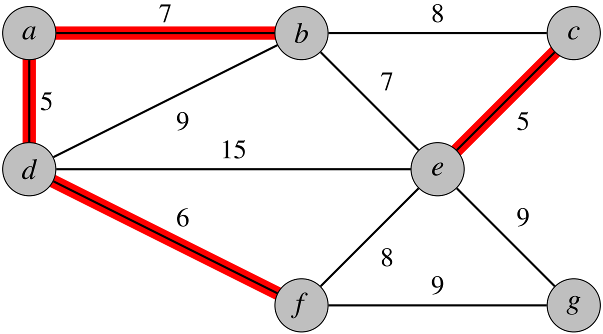
Kruskal's algorithm



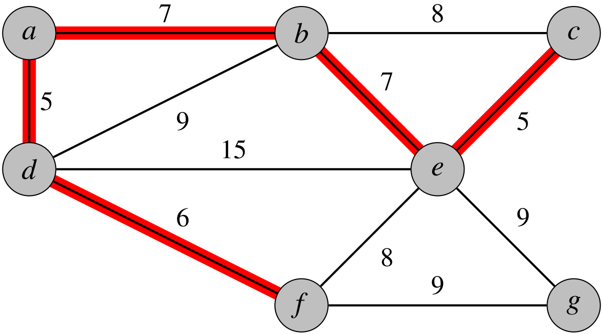
Kruskal's algorithm



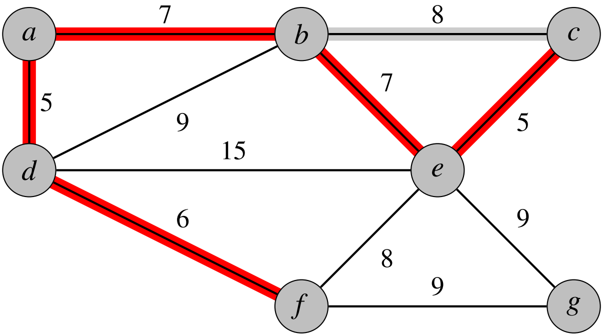
Kruskal's algorithm



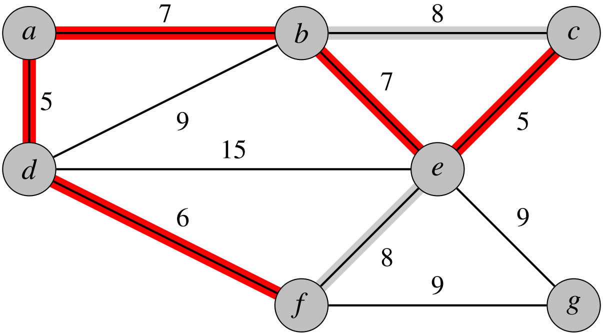
Kruskal's algorithm



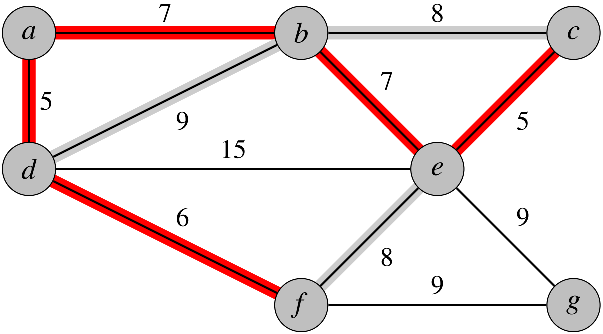
Kruskal's algorithm



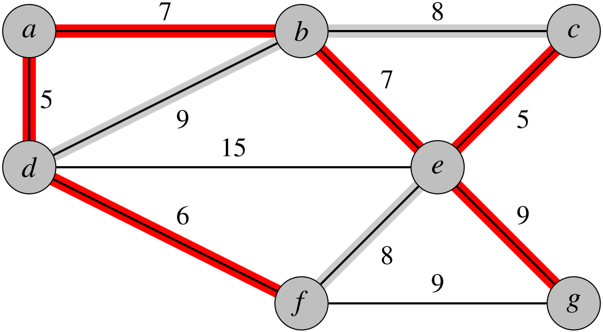
Kruskal's algorithm



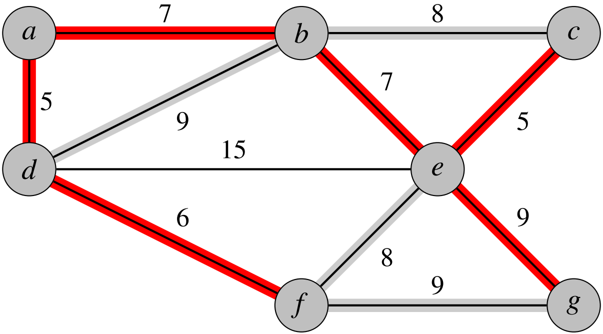
Kruskal's algorithm



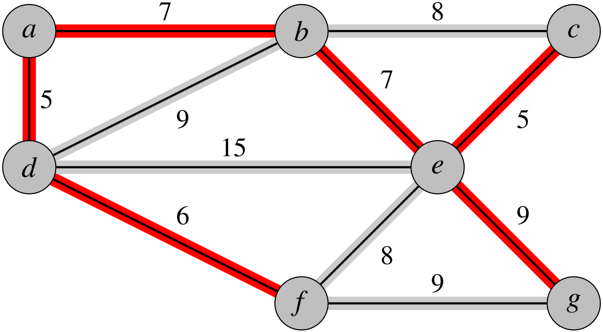
Kruskal's algorithm



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 $O(E \log V)$

The Union-Find Data Structure

- Kruskal's algorithm, like many other algorithms in Computer Science, requires a dynamic partition of an n-element set S into a collection of disjoint subsets $S_1, S_2, \ldots, S_k$.
- After being initialised as a collection of n one element subsets, we perform union and find operations on the collection.
- The union operation joins two subsets into a single set. The number of such operations is bounded by $n - 1$, since we have n elements in total.

The Union-Find Data Structure

- We thus have an abstract data type of a collection of disjoint subsets of a finite set with the following operations:
  - makeset(x) - creates a one element set x;
  - find(x) - returns a subset containing x;
  - union(x, y) - constructs the union of the disjoint subsets S_x and S_y containing x and y respectively and replaces S_x and S_y by this union.

An example

- As an example, let $S = \{1, 2, 3, 4, 5, 6\}$.
- Then makeset(i) creates the set $\{i\}$ and applying this six times gives:

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$$

- Performing union(1, 4) and union(5, 2) yields:

$$\{1, 4\}, \{5, 2\}, \{3\}, \{6\}$$

- If followed by union(4, 5) and union(3, 6), it yields:

$$\{1, 4, 5, 2\}, \{3, 6\}$$

Representatives

- Implementations of this data structure use one element from a subset as its representative
- We usually use the smallest element in the set (assumed to be integers)
- We can use an array indexed by the elements of the underlying set with values to represent the representative of that element
- A linked list can be used to store the elements in each subset (indexed by representative), together with the number of elements in the subset

Representatives

- To represent the following union-find data structure

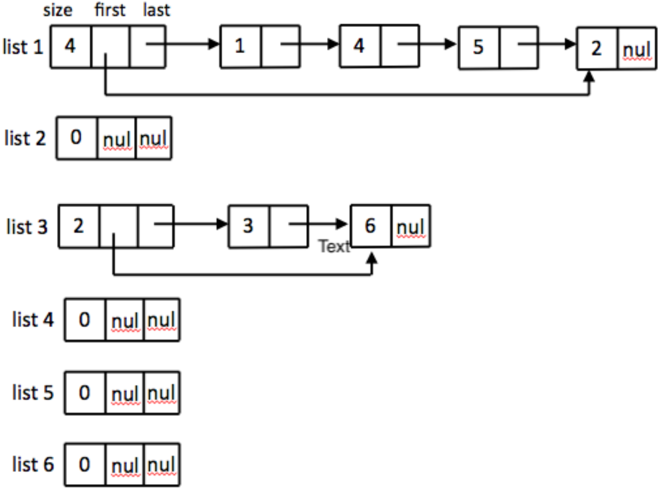
$$\{1, 4, 5, 2\}, \{3, 6\}$$

- We would use the following array

Element Index	1	2	3	4	5	6
Representative	1	1	3	1	1	3

- Together with the linked list of the following page

Representation of a Union-Find Algorithm



Operations on the Union-Find data structure

- Executing makeset(x) requires assigning x to the element at position x of the representative array and initialising the corresponding linked list to a single node with the x value -this takes O(1) operations.
- Performing find(x), to find the representative for an element x, also requires just $O(1)$ operation; we simply need to look at the element in index x of the representative array
- Performing union(x, y) requires us to append y's list to the end of x's list, update the information about the represenatives for the y list and then delete the y list from the collection (requires $O(n^2)$ in the worst case, but can be improved to $O(n \log n)$)

Back to Kruskal's Algorithm

- How does the Union-Find data structure help with Kruskal's algorithm?
- Store each vertex as a separate integer (i.e. makeset(x))
- Each time we want to add an edge (i, j) to the MST, we need to determine if adding edge (i, j) to the MST would create a cycle
- To do this, we simply need to determine if find(i) = find(j)
- If so, both vertices i and j are in the same subset and we can't add an edge between them without creating a cycle
- If we can add an edge to the graph, then we perform union(i, j) to update the data structure