Theory of Computing

Lecture 9

MAS 714

Hartmut Klauck

Minimum spanning trees

- **Definition**
  - A spanning tree of an undirected connected graph is a set of edges $E' \subseteq E$:
    - E' forms a tree
    - every vertex is in at least one edge in E'
  - When the edges of G have weights, then a *minimum* spanning tree is a spanning tree with the smallest sum of edge weights

MST

- Motivation: measure costs to establish connections between vertices
- Basic procedure in many graph algorithms
- Problem first studied by Boruvka in 1926

- Other algorithms: Kruskal, Prim
- Inputs: adjacency list with weights

MST

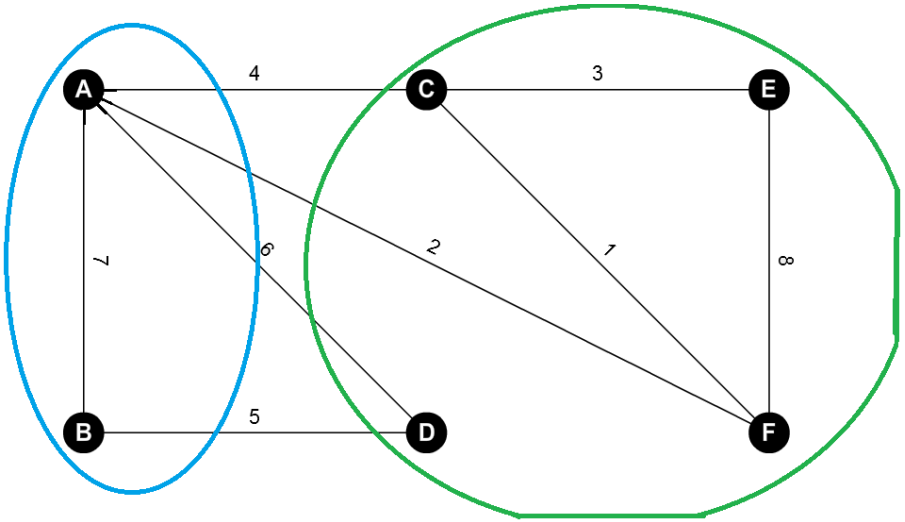
- Generic algorithm:
  - Start with an empty set of edges
  - Add edges, such that current edge set is always subset of a minimum spanning tree
  - Edges that can be added are called *safe*

Generic Algorithm

- Set A=∅
- As long as A is not (yet) a spanning tree add a safe edge e
- Output A

Safe Edges

- **Definition:**
  - A *cut* C=(S, V-S) is a partition of V
  - A set of edges *repects* the cut, if no edge crosses
  - An edge is *light* for a cut, if it is the edge with smallest weight that crosses the cut



Safe Edges

- **Theorem:**
  Let G be an undirected, connected, weighted graph. A a subset of a minimum spanning tree. C=(S, V-S) a cut that A respects.
  Then the lightest edge of C is safe.
- Proof:
  - T is an MST containing A
  - Suppose e is not in T (otherwise we are done)
  - Construct another MST that contains e

Safe Edges

- Inserting e={u,v} into T creates a cycle p in T∪{e}
- u and v are on different sides of the cut
- Another edge e‘ in T crosses the cut
- e‘ is not in A (A respects the cut)
- Remove e‘ from T (T is now disconnected into 2 trees)
- Add e to T (the two trees reconnect into one)
- W(e)=W(e‘), so T‘ is also minimal
- Hence A∪{e} subset of T ‘
- e is safe for A.

Which edges are not in a min. ST?

- **Theorem:**
  - G a graph with weights W.
    All edge weights distinct.
    C a cycle in G and e={u,v} the largest edge in C.
  - Then e is in no minimum spanning tree.
- Proof:
  - Assume e is in a min. ST T
  - Remove e from T
  - Result is two trees (containing all vertices)
  - The vertices of the two trees form a cut
  - Follow C-{e} from u to v
  - Some edge e‘ crosses the cut
  - T-{e}∪{e‘} is a spanning tree with smaller weight T

Algorithms

- We complete the algorithm „skeleton“ in two ways
  - Prim: A is always a tree
  - Kruskal: A starts as a forest that joins into a single tree
    - initially every vertex its own tree
    - join trees until all are joined up

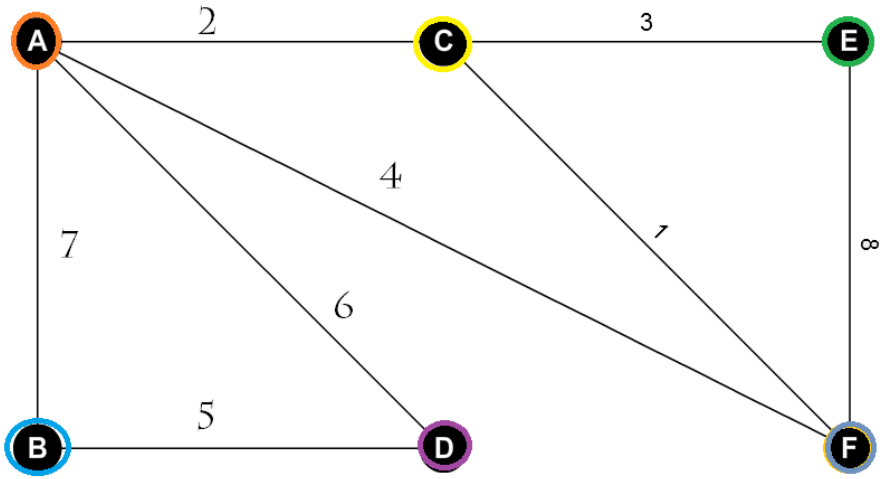
Data structures: Union-Find

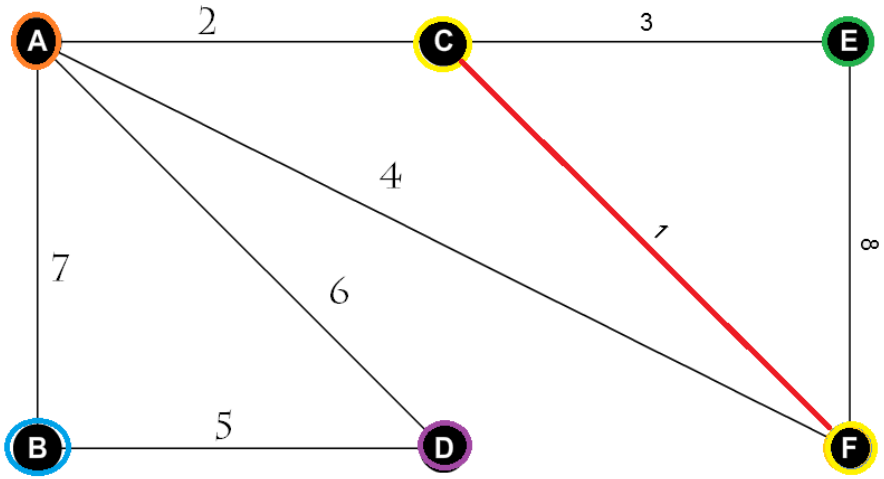
- We need to store a set of disjoint sets with the following operations:
  - Make-Set(v):
    generate a set {v}. Name of the set is v
  - Find-Set(v):
    Find the name of the set that contains v
  - Union(u,v):
    Join the sets named u and v. Name of the new set is either u or v
- As with Dijkstra/priority queues the running time will depend on the implementation of the data structure

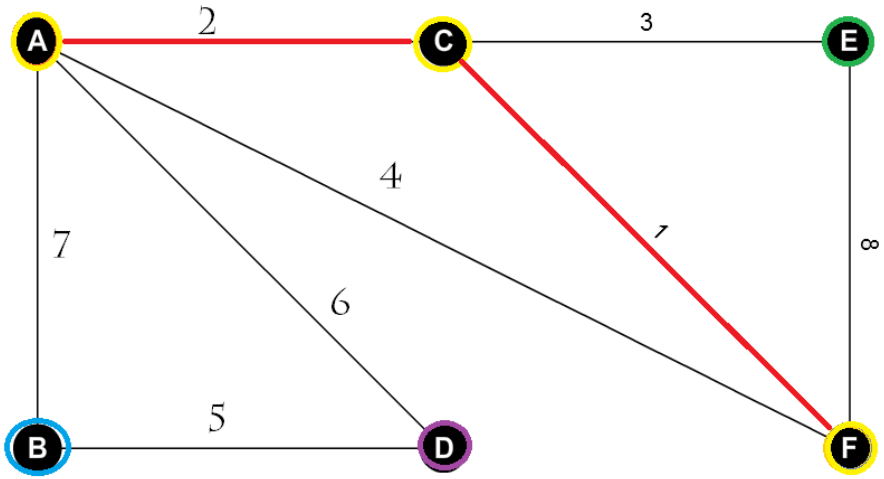
Kruskal

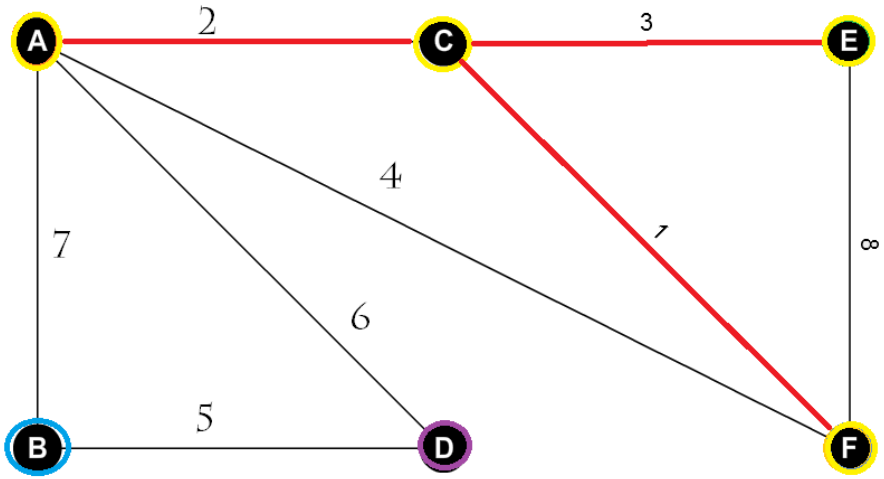
1. Input: graph G, weights $W:E \mapsto R$
2. $A=\emptyset$
3. For each vertex v:
   a) Make-Set(v)
4. Sort edges in E (increasing) by weight
5. For all edges {u,v} (order increasing by weight):
   a) a=FindSet(u), b=FindSet(v)
   b) If $a \neq b$ then
      $A:=A \cup \{\{u,v\}\}$
      Union(a,b)
6. Output A

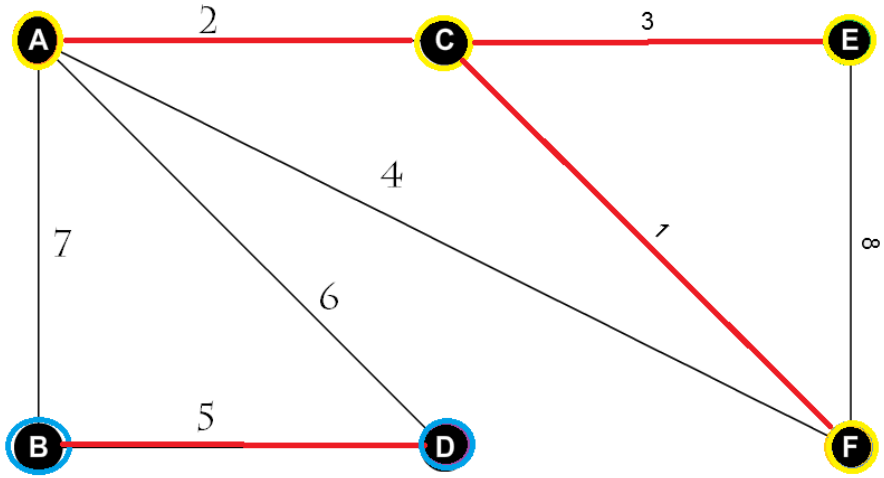
Example

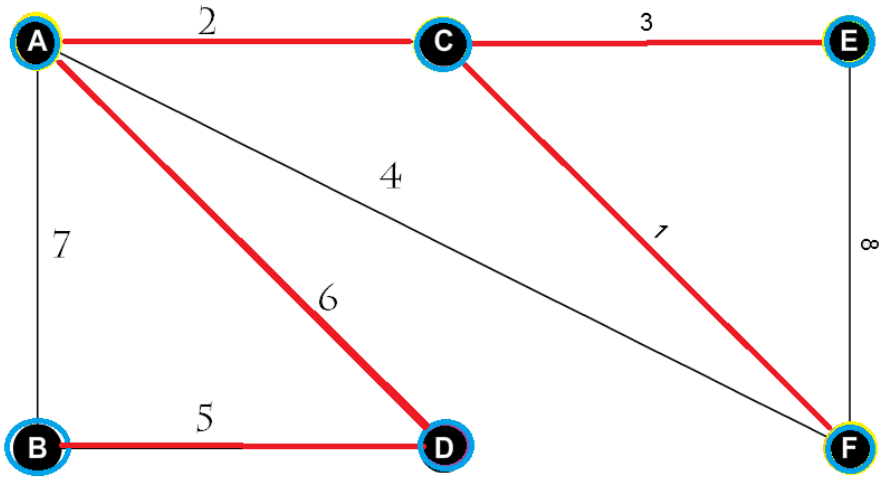



A
B
C
D
E
F
2
3
4
7
6
5
1
8


A
B
C
D
E
F
2
3
4
7
6
5
1
8


A
B
C
D
E
F
2
3
4
7
6
5
1
8


A
2
C
3
E
4
7
7
8
6
5
B
D
F


A
B
C
D
E
F
2
3
4
5
6
7
8

Kruskal

1. Input: graph G, weights $W:E \mapsto R$
2. $A=\emptyset$
3. For each vertex v:
   a) Make-Set(v)
4. Sort edges in E (increasing) by weight
5. For all edges {u,v} (order increasing by weight):
   a) a=FindSet(u), b=FindSet(v)
   b) If $a \neq b$ then
      $A:=A \cup \{\{u,v\}\}$
      Union(a,b)
6. Output A

Running time Kruskal

- We will have:
  - until 3: O(n) times Time for Make-Set
  - 4: O(m log n)
  - 5: O(m) time Time for Find/Union

  - Total will be O(n+m log n)

Correctness

- We only have to show that all edges inserted are safe
  - Choose a cut respected by A, which is crossed by the new edge e
  - e has minimum weight under all edges forming no cycle, hence e has minimum weight among all edges crossing the cut
  - Hence e must be safe for A

Data structures: Union-Find

- We need to store a set of disjoint sets with the following operations:
  - Make-Set(v):
    generate a set {v}. Name of the set is v
  - Find-Set(v):
    Find the name of the set that contains v
  - Union(u,v):
    Join the sets named u and v. Name of the new set is either u or v
- As with Dijkstra/priority queues the running time will depend on the implementation of the data structure

Implementation Union Find

- Universe of n elements
  - Use array M with n entries
  - Sets are represented as trees, by pointers towards the roots

Implementation Union Find

- MakeSet(v) for all v: M(v)=v for all v
- Union(u,v): set M(v)=u, if the set of u is larger than the set of v (important!)
- Find(v): follow the pointers from M(v) until M(u)=u, output u
  - We need to store for each v that is a root also the size of its set
    - Beginning with 1, update during Union

Running times Union Find

- MakeSet: O(1) time to make a singleton set
  - O(n) to make n singleton sets
- Union: O(1)
- Find: corresponds to maximum depth of trees
- Claim: Depth is O(log n)

Depth of the trees

- **Claim:**
  Trees with size g have depth at most log g:
  Proof:
  - This is true when trees are generated
  - Union: Sets u,v join with sizes a,b and depths $q \leq \log a$ and $r \leq \log b$, wlog $b \leq a$
  - New tree has size a+b
  - $r < q$, then new depth is $q \leq \log a \leq \log (a+b)$
  - $r = q$, then new depth is q+1, but $a \geq 2^q$, $b \geq 2^q$, $a+b \geq 2^q+2^q = 2^{q+1}$, hence new depth $<= \log (a+b)$
  - $r > q$, then the new depth is r+1. $r \leq \log b$, $b \leq a$, hence $r+1 \leq \log(2b) \leq \log(a+b)$

Speeding up Union Find

- Suppose we search for the set of u
  - We traverse the path from u to the root
- We can save time in the future if we hang all vertices on the path directly under the root!
  - Find will be faster in the future
- But size does not reflect anymore how bad a tree is

Speeding up Union Find

- Rank of a tree:
  - Tree with one vertex has rank 0
  - In Union, put the tree with smaller rank under the tree with larger rank
  - If both have the same rank increase rank of root by 1
  - Otherwise the new rank is the maximum of the two ranks
- Rank reflects depth

Speeding up Union Find

- Find(x):
  - if parent(x)≠ x:
    - y=Find(parent((x))
    - parent(x)=y
    - return y

- Claim:
  - The amortized running time of Find operations is now O(α(n))
- α(n) is the inverse of A(n,n), the Ackermann function

- A(4,4)= $2^{2^{65536}} - 3$

Algorithms

- We complete the algorithm „skeleton“ in two ways
  - Prim: A is always a tree
  - Kruskal: A starts as a forest that joins into a single tree
    - initially every vertex its own tree
    - join trees until all are joined up

Prim

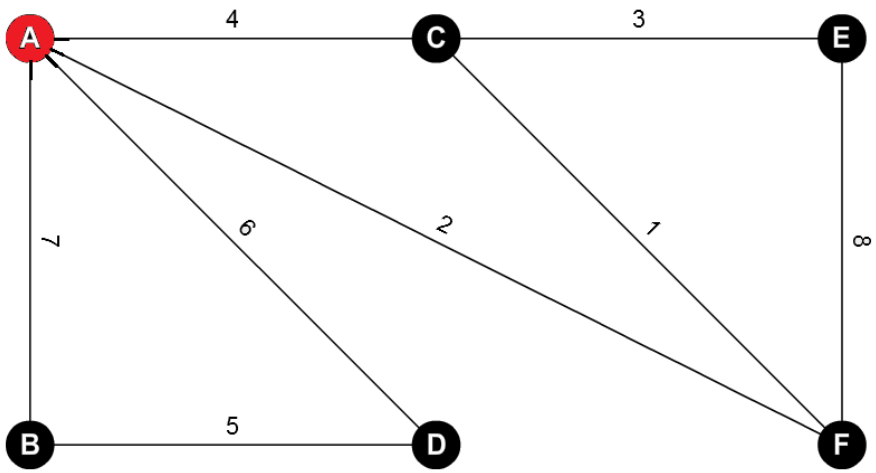
- In Prim's algorithm edges in A always form a tree
- Use a priority queue as in Dijkstra:
  - Operations:
    - Init: initialize empty priority queue
    - Insert(v,k): insert v with key k
    - Build Heap: Make Heap with n elements
    - ExtractMin: Find and remove element with min. key
    - DecreaseKey(v,k): reduce the key of v to k
  - Implementation with Heap:
    - Build Heap: O(n) for n elements inserted
    - Extract Min: O(log n) per Operation
    - DecreaseKey: O(log n) per Operation

Prim

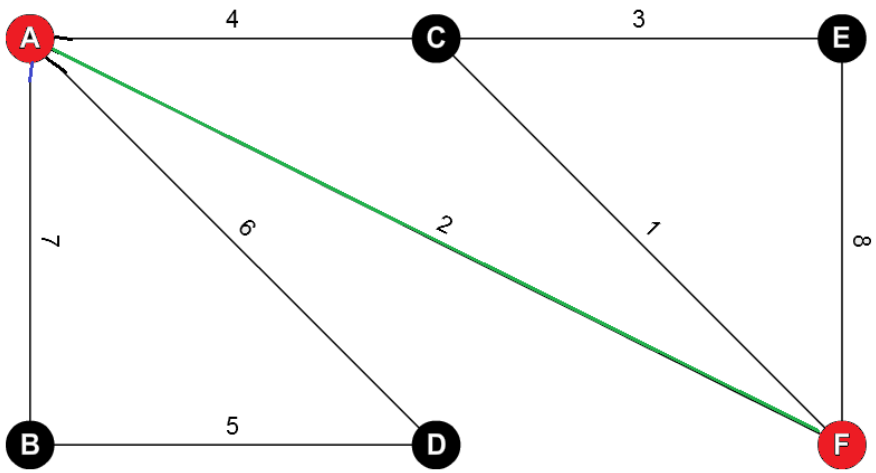
- Inputs: graph G, weights W, root r
- Output: minimum spanning tree as set $A=\{(v, \pi(v))\}$ of predecessor pointers

1. for all $v \in V$ set $key(v)=\infty$ and $\pi(v)=NIL$
2. $key(r)=0$, $S=\emptyset$
3. Init: BuildPriorityQueue Q of all (v,key(v))
4. While Q not empty:
   a) u=ExtractMin, $S=S\cup \{u\}$
   b) For all neighbors v of u:
      i. If v not in S and $key(v) > W(u,v)$ then $\pi(v)=u$ and $key(v):=W(u,v)$

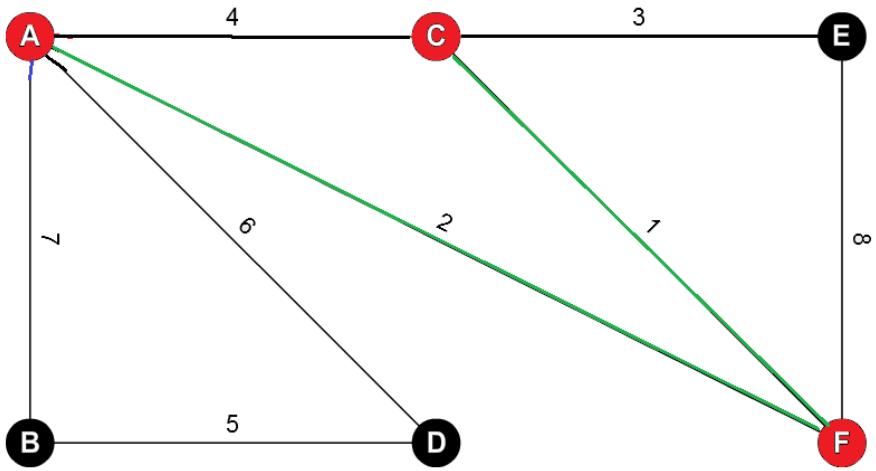
Example



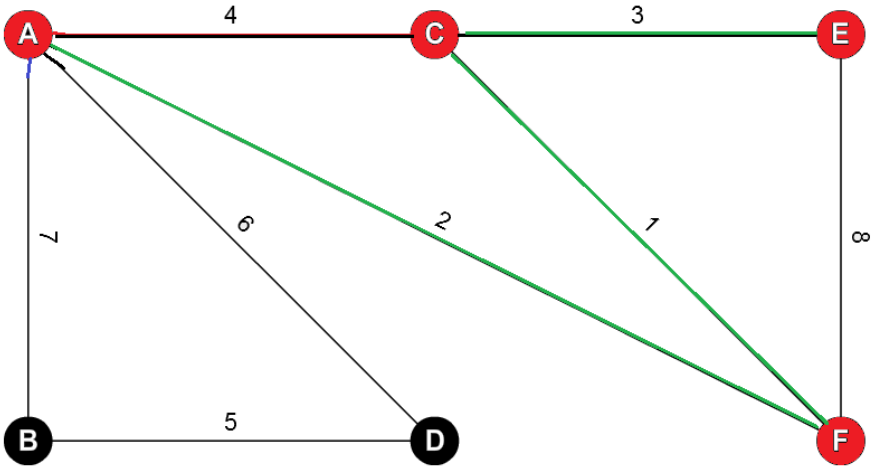
key(F)=2, key(C)=4, key(D)=6,key(B)=7, key(E)=∞



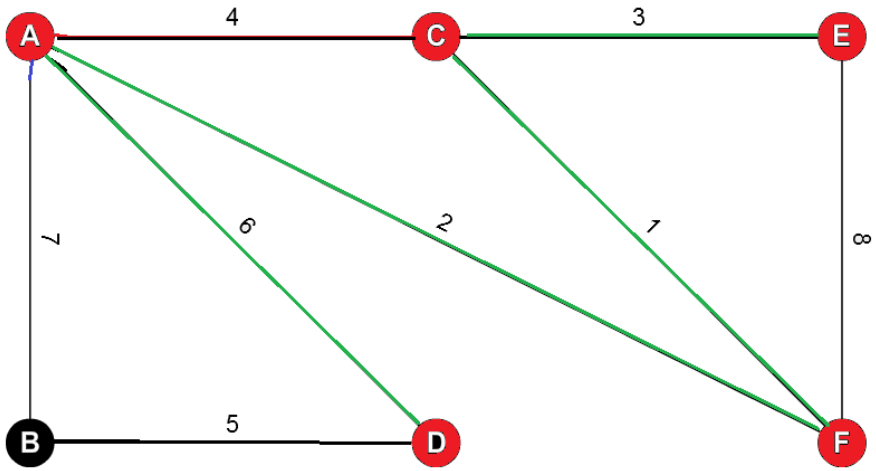
key(C)=1, key(D)=6, key(B)=7, key(E)=8



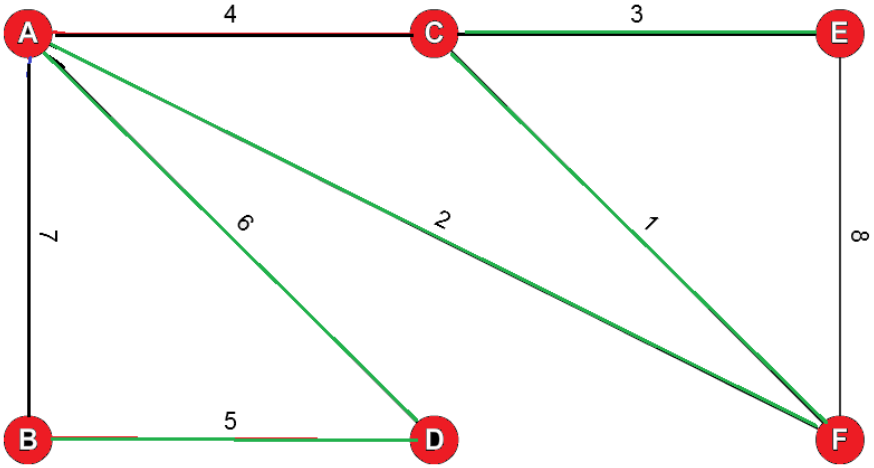
key(E)=3, key(D)=6,key(B)=7



key(D)=6,key(B)=7



key(B)=5


A
C
E
B
D
F
4
3
7
6
2
1
8
5

Running Time Prim

- Initialize, n times ExtractMin and m times DecreaseKey
- In total O((n+m) log n) time using Heaps
- Implementing the priority queue such that m DecreaseKey operations take time O(m) [amortized analysis]:
  - Time O(n log n + m)
- Linear time for m>n log n

Correctness

- Claim: for all v in Q the value key(v) is the weight of a lightest edge from v into S (if key(v)<∞) and A is always a tree
  - True in the beginning
  - When u is removed from Q then key(u) is the weight of a lightest edge by induction hypothesis
  - Edges in A form a tree (vertex set S), u is not in S, so still a tree after adding u into S
  - Update of neighbors v of u:
    - key(v)<∞ then key(v) is weight of a lightest edge {w,v} to S-{u}, if edge (u,v) is better: update, now key(v) is still weight of a lightest edge into S
- Hence edges added to A are always light for the cut (S, V-S) and thus safe

More about MST

- There is an algorithm that runs in time $O(n+m\alpha(n))$
- No deterministic linear time algorithm is known
- There is a randomized algorithm that runs in time $O(n+m)$
  - Contains a complicated algorithm to verify a spanning tree