Algorithms and Data Structures

Minimal Spanning Trees

Ulf Leser

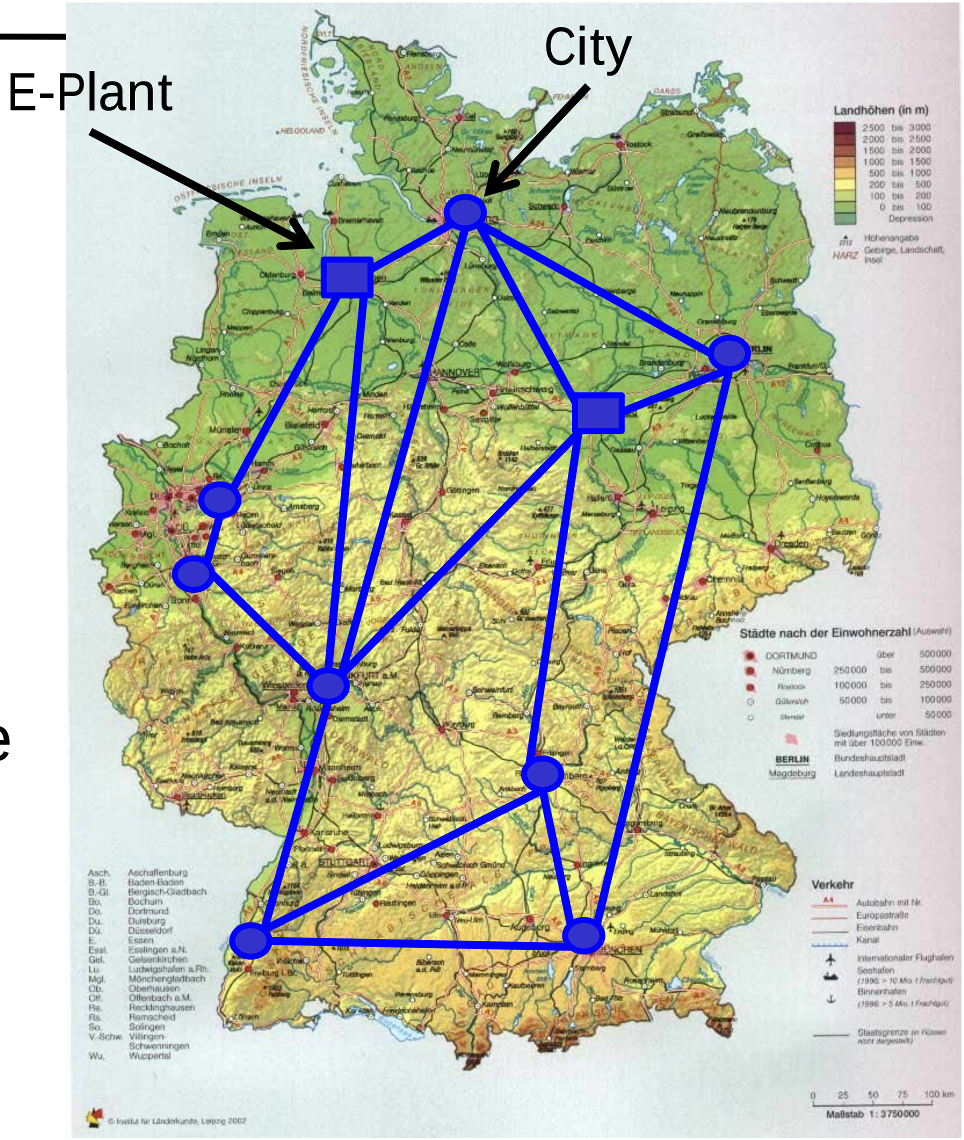
Die Energiewende

- Electricity is created in many more places than before
- Electricity is consumed in many places
- Places of production are not evenly distributed across the country
- We need to build new electricity highways

Source: http://www.deutsche-mittelgebirge.de/

Die Energiewende

- How can we do this as cheap as possible?
- Not all connections are possible
  - Mountains, rivers, ...
- Different connections have different costs

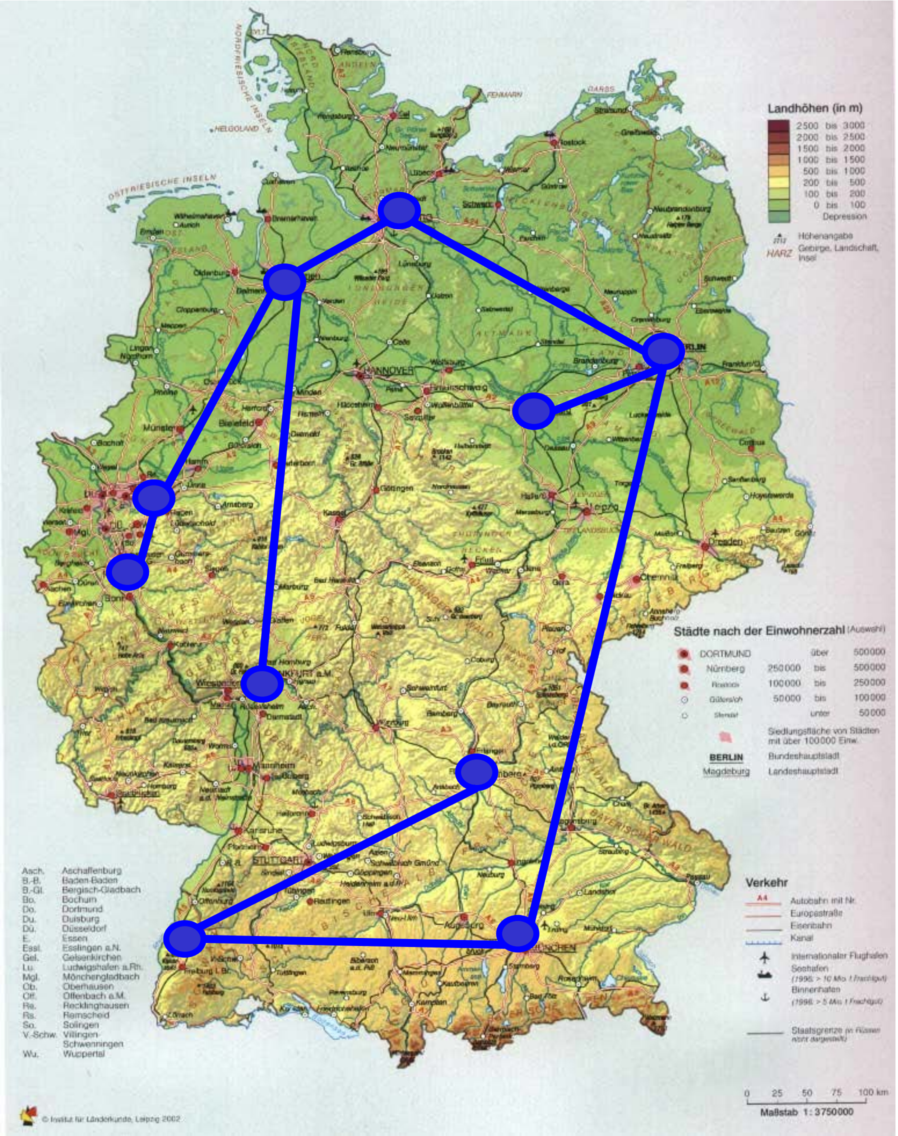


Die Energiewende

- Requirement for a solution: Every city and every plant must be connected to the network
  - We treat them uniformly
- One solution

Die Energiewende

- Another solution
- Of course, in real life we may build crossroads outside cities

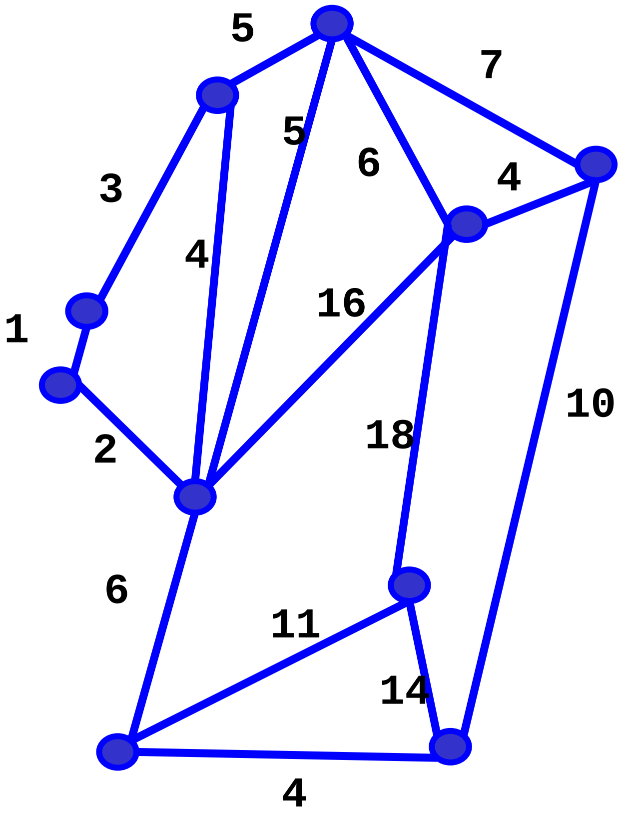


Die Energiewende

- Not considered valid here

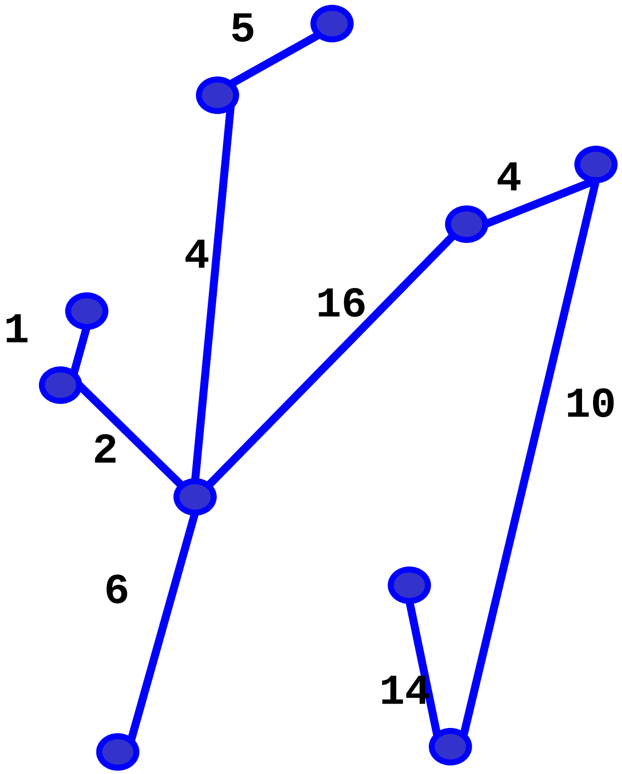
Abstraction

- Given an undirected, positively weighted, connected G=(V,E)
- Find a subset $E' \subseteq E$ such that cost(E') is minimal and G'=(V, E') is connected
  - Cost(E'): Sum of the edge weights
- E' (or G') is called a minimum spanning tree (MST) for G



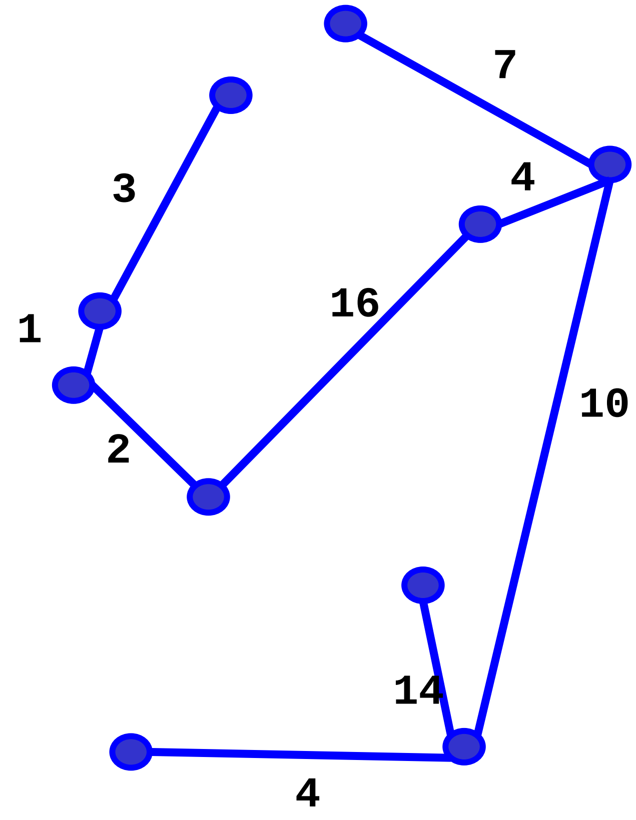
Example 1

- Cost = 62



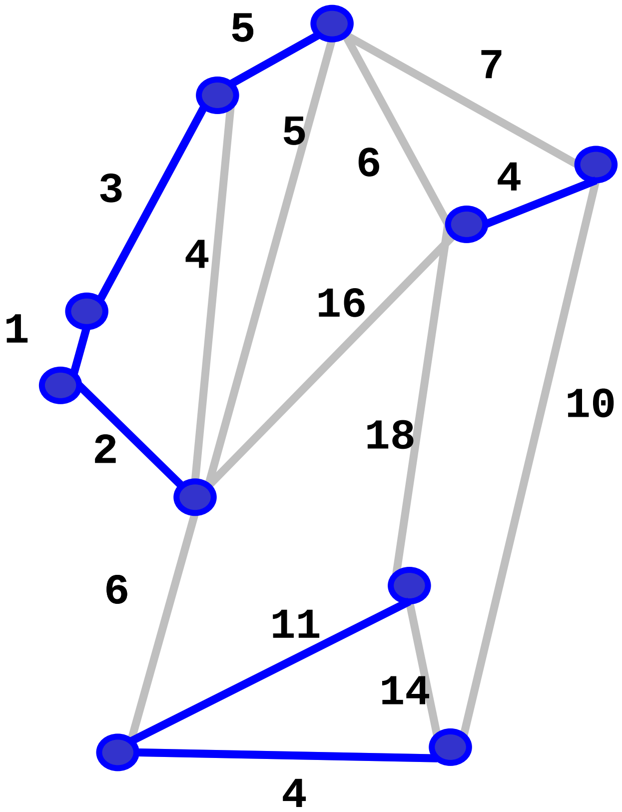
Example 2

- Cost = 61



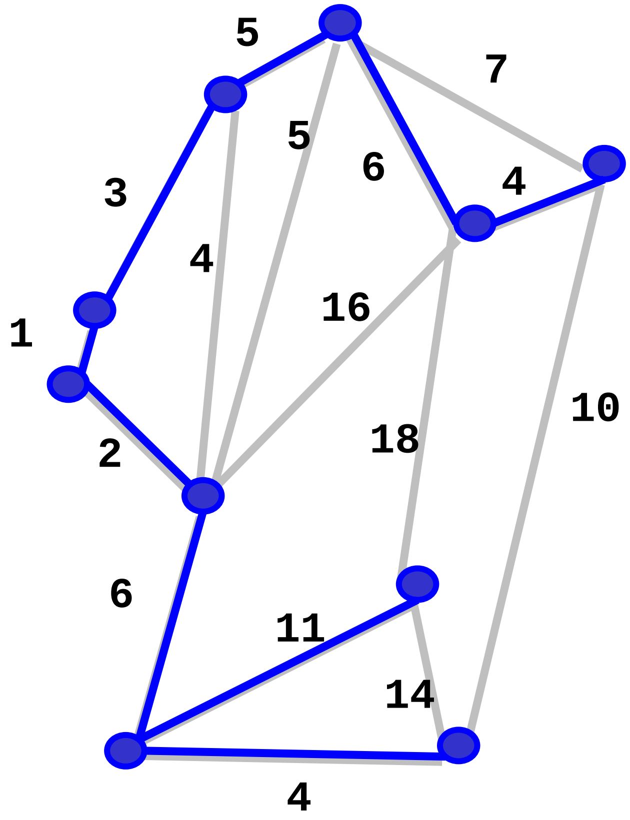
First Algorithm

- Let's be greedy
  - Sort edges by weight
  - Add edges to E' whenever a new node is getting connected to something
- Hmm



Second Algorithm

- Let‘s be greedy – another way
  - Sort edges by weight
  - Add cheapest edge to E’
  - Add all edges to E’ in ascending order such that every new edge adds a new node to the graph induced by E’
  - Repeat until E’ is complete
- Cost = 42
  - Is this optimal?
  - Does this always work?
  - How can we implement this algorithm efficiently?



Overview

- First algorithms for computing MST date back to the 1920s
- Algorithms are not very difficult; much research went into efficient implementations
- Actually, MSTs can be computed in a greedy manner
- Algorithms need not grow only one component; in general, we may have “connected islands” here and there that all gets connected to one component in the end
- In each step, one needs to decide which edge to add next to which island (or which edges not to add)
- What are criteria for adding / not adding edges?

Content of this Lecture

- Minimal Spanning Trees
- Basic Properties
  - Tree
  - Cuts
  - Cycles
- Algorithms
- Implementation

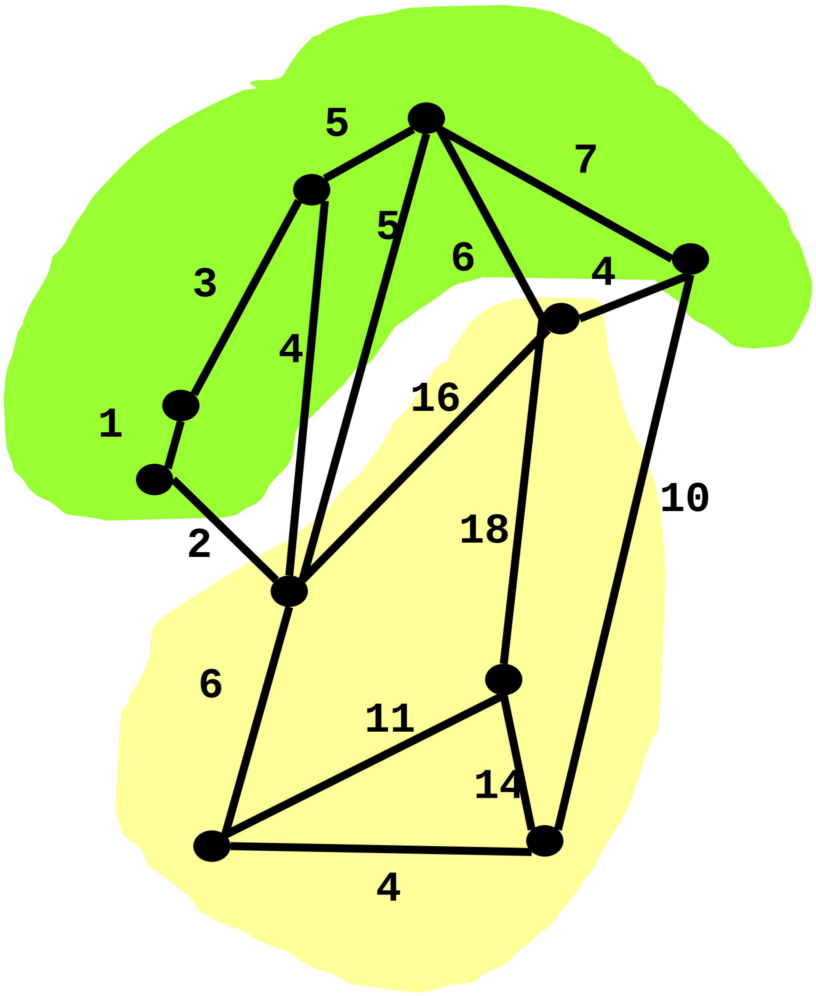
Mimimal Spanning Tree

- Lemma
  Let G=(V, E) and E'⊆E be the subset of E' with minimal cost such that G', the graph induced by E', is connected. Then G' is a tree.
- Proof
  - Recall: A tree is a undirected, connected acyclic graph
  - By definition, G' is connected
  - Imagine G' had a cycle. Then G' cannot have minimal cost, because removing any of the edges on the cycle from E' would create a subset E'' that has less cost, and the induced subgraph would still be connected
- Note: If all edge weights are distinct, the MST is unique

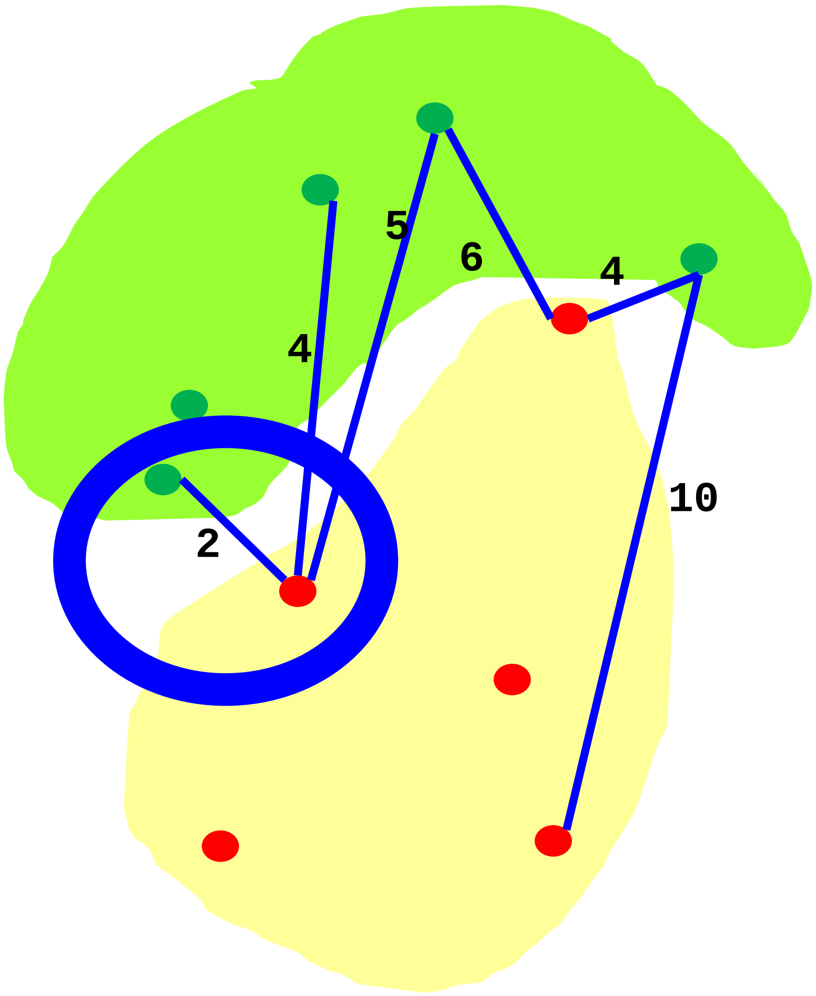
Cuts

- Definition
  Let G=(V, E). A cut is a binary partition of V into sets V_1, V_2 such that $V_1 \cap V_2 = \emptyset$ and $V_1 \cup V_2 = V$.
- Lemma
  Let G=(V, E) and V_1, V_2 be a cut of V. Let F be the set of all edges going from any node in V_1 to any node in V_2. Let F' be those edges of F with minimal weight. Then any MST G' of G must contains one edge of F', and every edge of F' is contained in at least one MST of G
- Remarks
  - This holds for arbitrary cuts – a very powerful statement
  - Edges in F are called crossing edges

Example



Example

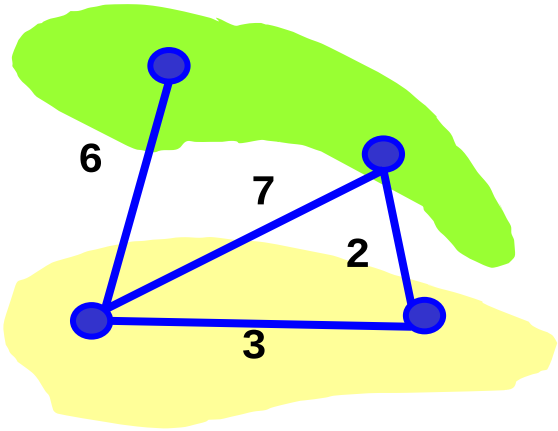


Proof

- Every MST G' contains one $f \in F'$
  - Imagine G' has no such f. Still, G' must be connected, so it must contain at least one of the crossing edges for V_1, V_2. Assume it contains only one such edge, f'. f' must have a higher weight than f because $f' \notin F$. Further, V_1 and V_2 must be connected in themselves. Then G' cannot be minimal, because removing f' and adding some $f \in F$ would create a cheaper MST – contradiction.
  - Same argument holds if G' contains more than one crossing edge, all of which are not minimal
- Every $f \in F'$ is contained in at least one MST
  - Imagine f is not contained in any MST. Let G' be a MST and f' be the edge in G' connecting V_1 and V_2. f' must be in F, or G' is not minimal. Thus, the MST formed by removing f' and adding f also is a MST – contradiction

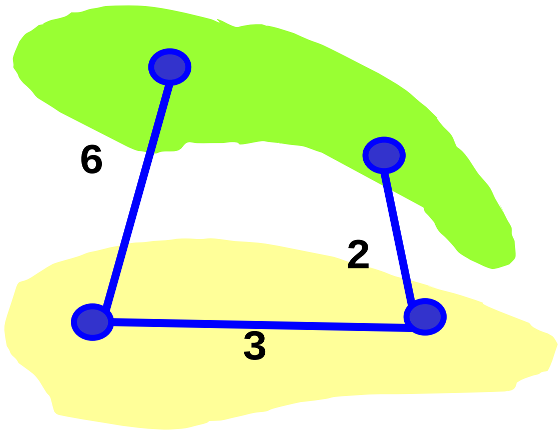
Beware

- For a cut V_1, V_2, a MST G‘ may (have to) contain more than one crossing edge (but one must have minimal weight)



Consequences

- The cut property is a strong help for computing MSTs
- Lemma (cut property)
  Let G=(V, E) and G'=(V, E') be a MST of G. Then every $e \in E'$ has minimal cost among all crossing edges of the cut V_1, V_2 formed by removing e from G'.
- Proof
  - Clearly, every edge from E' “cuts” G
  - Rest follows from previous lemma
- Can be used to check whether a given E' is a MST



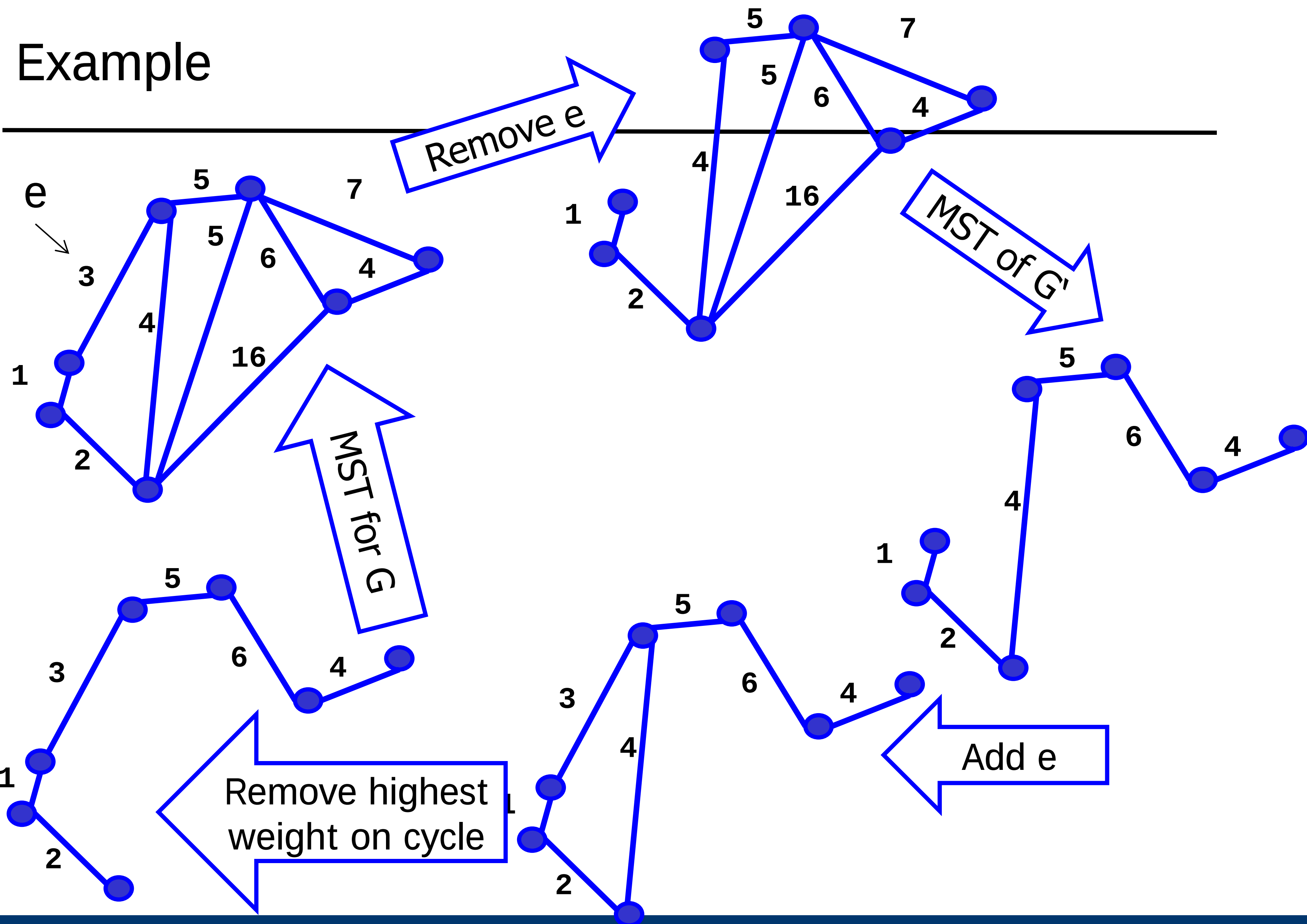
Content of this Lecture

- Minimal Spanning Trees
- Basic Properties
  - Tree
  - Cuts
  - Cycles
- Algorithms
- Implementation

Cycles

- Lemma (cycle property)
  Let G=(V, E) and G'=(V, E') with E'=E\e for some edge e such that G' still is connected. Let T' be a MST for G'. When we add e to T' and remove the edge with the highest weight on the then introduced cycle in T', forming T, then T is a MST for G.
- Proof idea
  – Adding e must introduce cycle
  – Removing any of the edges on the cycle still leaves a connected tree
  – Removing the most expensive one leaves the minimal tree

Example



e
Remove e
MST of G'
Add e
Remove highest weight on cycle
MST for G

Implications

- Note that T' is a MST for G without e
- Imagine we would enumerate edges by some order
- Taking into account e allows us to replace an edge in T' with a cheaper one, creating a "better" MST for G
  - If e is not the edge with the highest weight on the cycle
- This means that an edge with maximal weight on a cycle in G cannot be part of any MST of G

Content of this Lecture

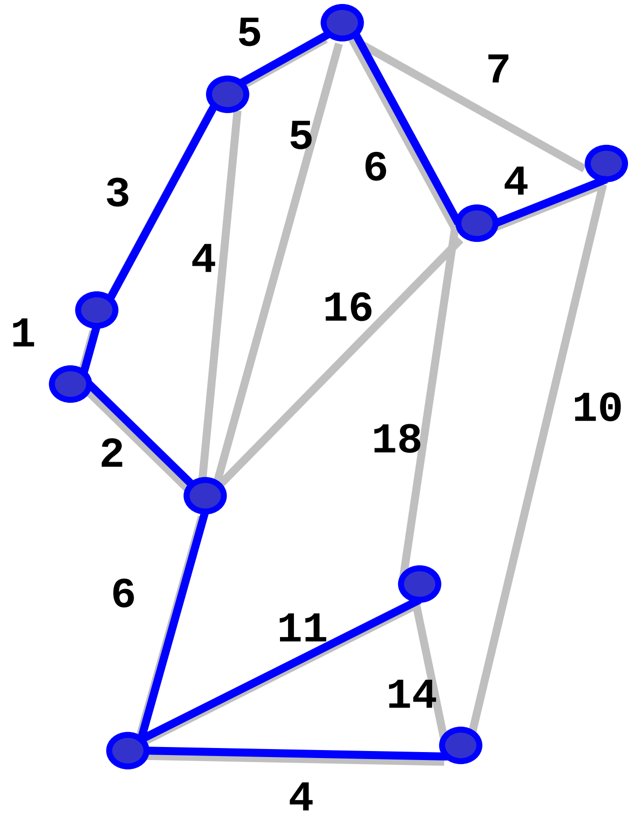
- Minimal Spanning Trees
- Basic Properties
- Algorithms
  - R.C. Prim: Shortest connection networks and some generalizations. Bell System Technical Journal, 1957
    - Also Jarnik, Prim, Dijkstra: Jarník, 1930 – Prim, 1957 – Dijkstra , 1959
  - J. Kruskal: On the shortest spanning subtree and the traveling salesman problem. Proc. of the American Mathematical Soc., 1956
  - Otakar Borůvka: O jistém problému minimálním (Über ein gewisses Minimierungsproblem), 1926
  - [Wikipedia, OW93]
- Implementation

Prim's Algorithm

Greedy; we never make mistakes

- Recall cut property. Every edge e in a MST is a minimal cost edge among the 2 partitions created by removing e
- Prim's Algorithm
  Start with an empty tree T. Continue adding the edge e with the lowest cost to T such that e connects T with a new node until all nodes of G are in T. Then T is a MST
- Proof
  - Consider, at each stage, nodes in T as one partition V_1 and all other nodes as the other partition V_2
  - By cut property, the cheapest crossing-edge between V_1 and V_2 must be in the MST
  - Since we only add those edges, T finally must be a MST

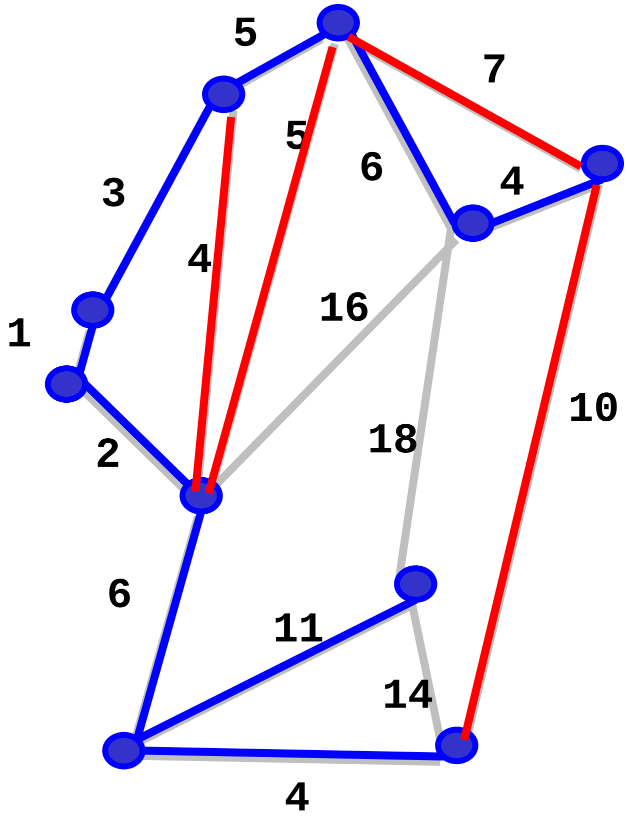
Example



Kruskal’s Algorithm

- Kruskal‘s Algorithm
  Start with an empty forest F. Continue “adding” edges e to F in order of increasing cost until F becomes a tree. Adding an edge e=(v, w) to F proceeds as follows:
  - *If F contains a tree containing both v and w, then e is dropped*
  - *If no tree in F contains either v or w, then a new tree formed by e is added to F*
  - *If F contains a tree T containing either v or w and neither T nor any other tree in F contains the other node, then e is added to T*
  - *If F contains a tree T containing either v or w and a tree T’ containing the other node, then T, T’ and e are merged into one tree*

Example



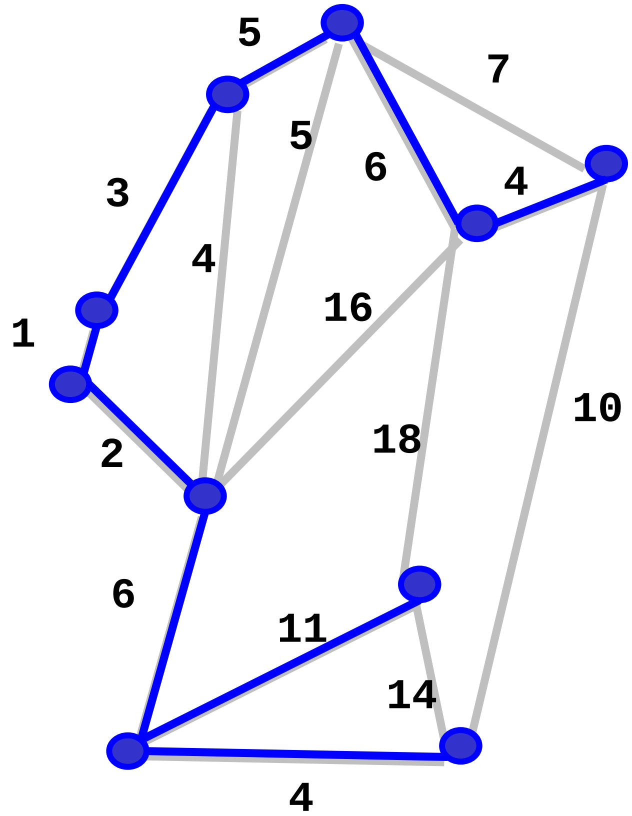
Proof

- By induction (only central idea)
  - We show that all trees in F are MST of subgraphs of G
  - Claim is true at the beginning (F empty)
  - Assume claim holds when we consider the next edge e=(v, w)
  - Case 1: Claim holds. E would introduce a cycle, and e has the highest cost on this cycle (all cheaper edges were considered before). Thus, e cannot be in an MST for G
  - Case 2: Claim holds because e is the cheapest edge connecting v and w, and thus the new tree is a MST (for v and w)
  - Case 3: Claim holds because e is the cheapest edge connecting v (or w) and T, and thus the new tree is a MST
  - Case 4: Claim holds because e is the cheapest edge connecting T and T', and thus the new tree is a MST

Boruvka‘s Algorithm

- Boruvka‘s Algorithm
  Start with an empty forest F. Add all edges (at once) that connect a node with its “cheapest” neighbor (edge with least cost) – taking care of not introducing cycles. Then consider each pair of trees in F and add cheapest crossing-edge until F becomes a unique tree.
- Proof (and details) omitted; see [Sed04]

Example



Communalities

- All three algorithms iteratively choose an edge by the cut property or reject an edge by the cycle property
  - Prim: Growing T is one partition, all other nodes the other (isolated nodes)
  - Kruskal: Each T that grows is one partition, all other nodes the other (islands of mini-MSTs)
  - Boruvka: Each T that grows is one partition, all other nodes the other (islands of mini-MSTs)
- Difference is the order in which edges are chosen – there are always many candidates
- Differences are the data structures that these algorithms need to maintain – efficient implementations are not trivial

Content of this Lecture

- Minimal Spanning Trees
- Basic Properties
- Algorithms
- Implementation
  - Prim's, Kruskal's

Implementing Prim's Algorithm

- ChooseCheapest: Choose cheapest edge connecting a node in T to a node not yet in T
- Brute force: Search all such edges in every step
- More clever
  - Maintain a PQ of nodes reachable with 1 edge from T sorted by cost
  - When adding a new node to T, look at its neighbors and add them to the PQ (if not reachable before) or update costs (if now there is a cheaper edge reaching them)

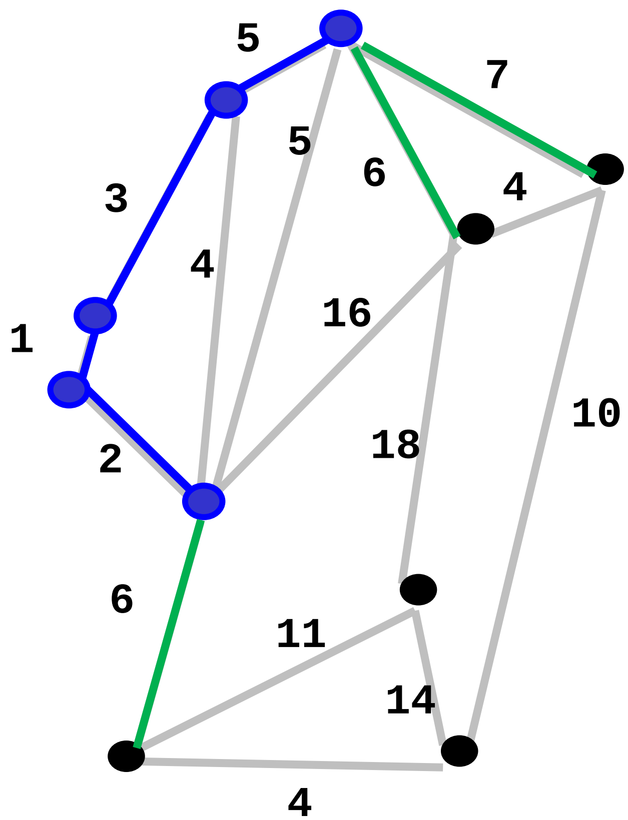
```
G := (V, E);
T := ∅;
R := E;
for i = 1 to |V|-1 do
  e := chooseCheapest( T, R);
  T := T ∪ e;
  R := R \ e;
end for;
```

Example

A
B
C
D
E
F

G
H

I,J



- T = {A, F, E, B, G}
- PQ = {(D,6), (I, 6), (C, 7)}

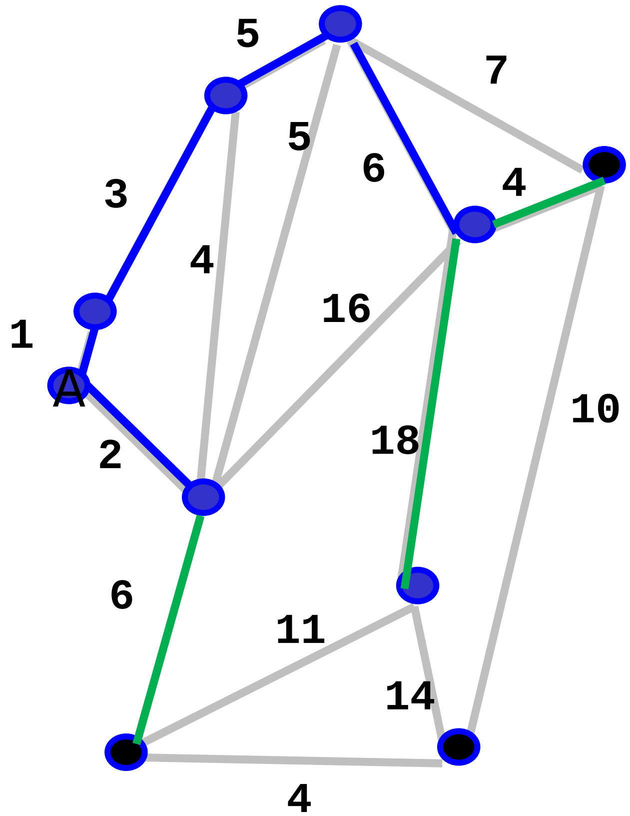
- Choose (A-D, 6)

Example

A
B
C
D
E
F

G
H

I,J



- T = {F, E, B, G}
- PQ = {(D,6), (I, 6), (C, 7)}

- Choose (A-D, 6)
- New T: {A, F, E, B, G, D}
- PQ = {(C,4), (I, 6), (H, 18)}

Complexity

- n=|E|, m=|V|

- Prim' algorithm runs in O((n+m)*log(n))
  - n times through the loop, performing altogether at most m PQ-operations in log(n)
- In dense graphs (m~n^2), this means O(m*log(n))

Implementing Kruskal‘s Algorithm

- ChooseCheapest: Simply choose cheapest edge in E
  - I.e., sort E at the beginning
- This is called a UNION-FIND data structure
  - Maintains a set of sets (all trees T)
  - Needs a method for quickly finding the set containing a given element (find)
  - Needs a method for quickly merging two sets (union)
- Can be implemented in O(m*log(n))

```
G := (V, E);
for i = 1 to |V| do
  T[i] := {i};
end do:
repeat
  (v,w) := chooseCheapest( E);
  E := E \ (v,w);
  T := find( v);
  T' := find (w);
  if T≠T' then
    T := T ∪ T';
  end if;
until |T|=|V|;
```