# SFU CMPT-307 2008-2 Lecture: Week 13

**Ján Maňuch**

**E-mail: jmanuch@sfu.ca**

**Lecture on July 29, 2008, 5.30pm-8.20pm**

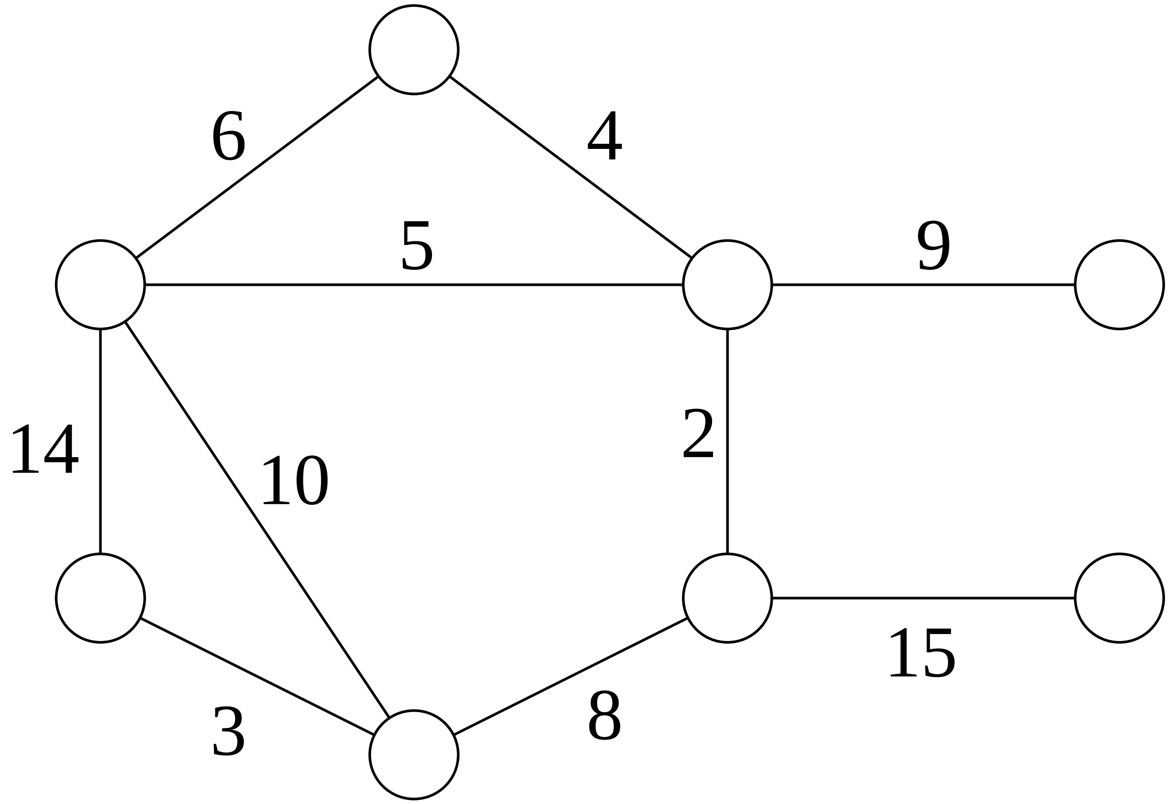
# Minimum spanning trees

One of the most famous greedy algorithms
(actually rather *family* of greedy algorithms).

- Given undirected graph $G = (V, E)$, connected
- Weight function $w : E \to \mathbb{R}$
- Spanning tree: tree that connects all vertices, hence $n = |V|$ vertices and $n - 1$ edges
- MST $T : w(T) = \sum_{(u,v) \in T} w(u, v)$ minimized

What for?

- Chip design
- Communication infrastructure in networks


6
4
5
9
14
10
2
3
8
15

# Growing a minimum spanning tree

First, "generic" algorithm. It manages set of edges $A$, maintains invariant:

**Prior to each iteration, $A$ is subset of some MST.**

At each step, determine edge $(u, v)$ that can be added to $A$, i.e. **without violating invariant**, i.e., $A \cup \{(u, v)\}$ is also subset of some MST. We then call $(u, v)$ a **safe edge**.

1: $A \leftarrow \emptyset$
2: **while** $A$ does not form a spanning tree **do**
3: find an edge $(u, v)$ that is safe for $A$
4: $A \leftarrow A \cup \{(u, v)\}$
5: **end while**

We use an invariant to check that an MST is produced:

**Initialization.** After line 1, $A$ trivially satisfies invariant.

**Maintenance.** Loop in lines 2–5 maintains invariant by adding only safe edges.

**Termination.** All edges added to $A$ are in an MST, so $A$ must be an MST.

*Question:* How to recognize safe edges?

The following theorem provides a rule.

**Definition.** A *cut* $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of $V$.

**Definition.** An edge $(u, v)$ *crosses* a cut $(S, V - S)$ if one end point is in $S$, the other the other in $V - S$.

**Definition.** A cut *respects* a set $A \subseteq E$ if no edge in $A$ crosses the cut.

**Definition.** An edge is a *light edge* crossing a cut if its weight is the minimum of all edges crossing the cut.

**Theorem 1.** Let $G = (V, E)$ be connected, undirected graph with real-valued weight function defined on $E$. Let $A$ be a subset of $E$ that is included in some MST for $G$, let $(S, V - S)$ be any cut of $G$ that respects $A$, let $(u, v)$ be a light edge crossing $(S, V - S)$. Then, $(u, v)$ is safe for $A$.

*Proof of* ***Theorem 1****.*

- let $T$ be an MST that includes $A$ (by assumptions there is one)
- assume $T$ does not include $(u, v)$ (otherwise we are done)
- we will construct another MST $T'$ that includes $A \cup \{(u, v)\}$, showing that $(u, v)$ is **safe**
- $(u, v) \notin T$, so there exists a path

  $$p = \langle u = w_1, w_2, \ldots, w_\ell = v \rangle$$

  with $(w_i, w_{i+1}) \in T$ for all $1 \leq i < \ell$
- $u$ and $v$ are on opposite sides of the cut $(S, V - S)$, hence when going from $u$ to $v$ along the path $p$, at least one of the edges, say $(w_k, w_{k+1})$ on the path $p$ is crossing the cut
- $(w_k, w_{k+1})$ is not in $A$ because $A$ respects the cut
- $(w_k, w_{k+1})$ is on the unique path from $u$ to $v$, so removing $(w_k, w_{k+1})$ breaks $T$ into two components
- adding $(u, v)$ reconnects them to form a new spanning tree $T' = T - \{(w_k, w_{k+1})\} \cup \{(u, v)\}$

now, it's enough to show that $T'$ is an MST containing $A \cup \{(u, v)\}$:

- $(u, v)$ is a light edge crossing the cut $(S, V - S)$, and $(w_k, w_{k+1})$ also crosses this cut, therefore $w(u, v) \leq w(w_k, w_{k+1})$ and

$$W(T') = w(T) - w(w_k, w_{k+1}) + w(u, v) \leq W(T)$$

- since $T$ is an MST, i.e., $w(T) \leq w(T')$, we have $w(T') = w(T)$, and hence $T'$ is an MST too ✓
- $A \subseteq T$ and $(w_k, w_{k+1}) \notin A$ , so $A \subseteq T'$ also
- since $(u, v) \in T'$, we have $A \cup \{(u, v)\} \subseteq T'$ ✓

we are done.

**Exercise 1.**

Show that if for every cut of a graph there is a unique light edge crossing the cut, then the graph has a unique minimum spanning tree. Show that the converse is not true by giving a counterexample.

*Remark:* Do not assume that all weight edges are distinct.

**Observations:**

- as algorithm proceeds, $A$ is always **acyclic** (otherwise, the MST including $A$ would contain cycle)
- at any point, graph $G_A = (V, A)$ is a **forest** (each connected component is a *tree*)
- some components may contain just one vertex (initially, $A$ is empty, and forest contains $|V|$ trees, one for each vertex)
- any safe edge $(u, v)$ for $A$ connects two distinct components of $G_A$, since $A \cup \{(u, v)\}$ must be acyclic
- main loop is executed $|V| - 1$ times: each iteration adds 1 edge to the resulting MST and decreases number of components by 1

Let's generalize the definition of *light edge*.

**Definition.** An edge is a **light edge** satisfying a given property, if its weight is the minimum of all edges satisfying the property.

The following consequence of **Theorem 1** is going to be used to design 2 algorithms for constructing an MST.

**Corollary.** Let $G = (V, E)$ be a connected undirected graph with a real-valued weight function defined on $E$. Let $A$ be a subset of $E$ that is included in some MST for $G$, let $C = (V_C, E_C)$ be a connected component (tree) in forest $G_A = (V, A)$.
If $(u, v)$ is a *light edge* connecting $C$ to some other component in $G_A$, then $(u, v)$ is safe for $A$.

*Proof.* The cut $(V_C, V - V_C)$ respects $A$ ($A$ *defines* the components of $G_A$), and $(u, v)$ is a light edge for this cut. Therefore, $(u, v)$ is safe for $A$.

We will describe Kruskal's and Prim's algorithms. They differ in how they specify rules to determine safe edges.

In Kruskal's algorithm, $A$ is a **forest**; while in Prim's algorithm, $A$ is a **single tree** (other components are single vertices).

## Kruskal's algorithm

Finds a safe edge to add to growing forest by finding minimum-weight edge $e$ that connects any two trees (directly using **Corollary**).

If $C_1, C_2$ denote the two trees that are connected by $(u, v)$, then since $(u, v)$ must be light edge connecting $C_1$ to some other tree, the corollary implies that $(u, v)$ is safe for $A$.

Kruskal's is **greedy** because at each step it adds an edge of least possible weight.

We will use `Disjoint-Set` data structure.

Each set contains the vertices in a tree of the current forest.

We will use the following operations:

- `Make-Set`$(u)$ initializes a new set containing just vertex $u$.
- `Find-Set`$(u)$ returns representative element from set that contains $u$ (so we can check whether two vertices $u, v$ belong to same tree).
- `Union`$(u, v)$ combines two sets (the one containing $u$ with the one containing $v$).

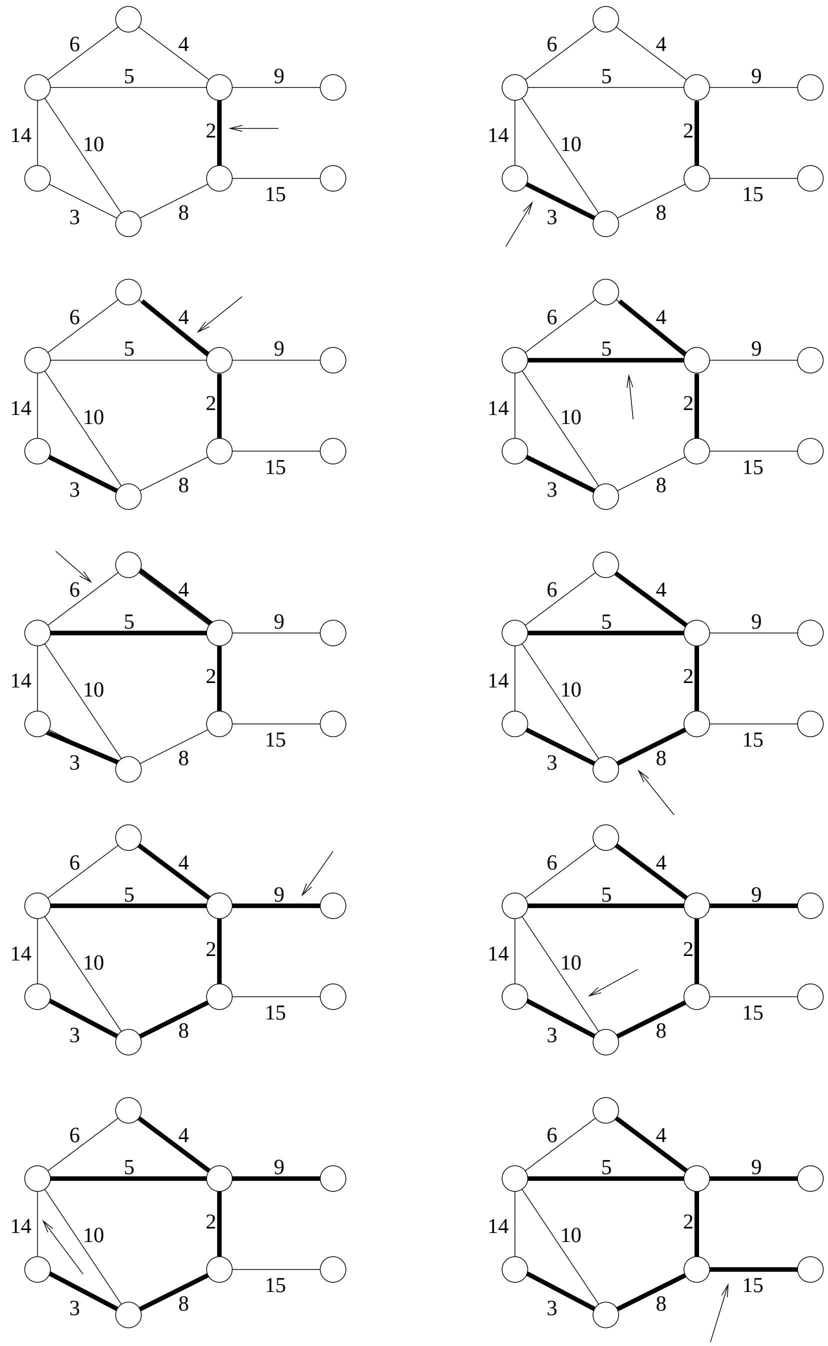
Time complexity depends on the actual implementation of Disjoint-set data structure. Implementation described in Section 21.3-4 of the textbook requires $\alpha(n)$ time, where $n$ is the number of elements and $\alpha$ is a very slowly growing function, hence, $\alpha(n) = \mathcal{O}(\log n)$.

*Given:* graph $G = (V, E)$, weight function $w$ on $E$

**MST-Kruskal**$(G, w)$

1: $A \leftarrow \emptyset$
2: **for** each vertex $v \in V[G]$ **do**
3:     `Make-Set`$(v)$
4: **end for**
5: sort edges of $E$ into nondecreasing order by weight $w$
6: **for** each edge $(u, v) \in E$, taken in the order **do**
7:     **if** `Find-Set`$(u) \neq$ `Find-Set`$(v)$ **then**
8:         $A \leftarrow A \cup \{(u, v)\}$
9:         `Union`$(u, v)$
10:     **end if**
11: **end for**
12: **return** $A$

— in the loop 6–11, for each edge we check whether it belongs to the same component (tree); if not: it's a cheapest edge (= save edge) connecting 2 components (edges are sorted, hence all consecutive edges have a weight at least the weight of the current edge)


6
4
5
9
14
10
2
15
3
8

## Running time

We assume that all `Disjoint-Set` operations, can be done in $\mathcal{O}(\log |V|)$ time

- initializing $A$ takes $\mathcal{O}(1)$
- sorting edges takes $\mathcal{O}(|E| \log |E|)$; since $|E| \leq |V|^2$, we have $\log |E| = \mathcal{O}(\log |V|)$; hence sorting takes: $\mathcal{O}(|E| \log |V|)$
- the initialization loop performs $|V|$ `Make-Set` operation;
  the main **for** loop performs $\mathcal{O}(E)$ `Find-Set` and `Union` operations;
  together it takes $\mathcal{O}((|V| + |E|) \log |V|)$
- since $G$ is connected, $|E| \geq |V| - 1$, so `Disjoint-Set` operations take $\mathcal{O}(|E| \cdot \log |V|)$
- *the total running time is* $\mathcal{O}(|E| \log |V|)$

**Exercise 2.**

Show that for each minimum spanning tree $T$ of $G$, there is a way to sort the edges of $G$ in Kruskal's algorithm so that the algorithm returns $T$.

*Remark:* Do not assume that all weight edges are distinct.

# Prim's algorithm

- Like Kruskal's, a special case of the generic algorithm.
- The set $A$ always forms a **single tree** (as opposed to a forest in Kruskal's).
- The tree starts from a single (arbitrary) vertex $r$ (root) and grows until it spans all of $V$.
- At each step, a light edge is added to tree $A$ that connects $A$ to isolated vertex of $G_A = (V, A)$ (a cheapest edge crossing the cut $(A, V - A)$).
- By corollary, this adds only edges safe for $A$, hence on termination, $A$ is an MST.
- Strategy is greedy, always pick a cheapest possible edge.

The crucial point is **efficiently selecting new edges**. In this implementation, we store all vertices that are **not** in the tree, in a min-priority queue $Q$. We have to assign priorities (**keys**) to vertices:

For $v \in V$,

- key$[v]$ is

  $\boxed{\text{the minimum weight of any edge connecting } v \text{ to a vertex in tree } A}$

  key$[v] = \infty$ if there is no such edge.

- $\pi[v]$ is the parent of $v$ in tree.

During the algorithm, the set $A$ from generic algorithm is kept implicitly as

$$A = \{(v, \pi[v]) : \ v \in V - \{r\} - Q\}$$

When the algorithm terminates, the min-priority queue $Q$ is empty, hence $A$ contains an MST for $G$:

$$A = \{(v, \pi[v]) : \ v \in V - \{r\}\}$$

**Given:** graph $G = (V, E)$, weight function $w$, root vertex $r \in V$

**MST-Prim**$(G, w, r)$

```
for each u ∈ V do
    key[u] ← ∞
    π[u] ← NIL
end for
key[r] ← 0
Q ← V    /* Build-Min-Heap */
while Q ≠ ∅ do
    u ← Extract-Min(Q)
    for each v ∈ adj[u] do
        if v ∈ Q and w(u, v) < key[v] then
            π[v] ← u
            key[v] ← w(u, v)    /* Decrease-Key */
        end if
    end for
end while
```

*Lines 1–6*

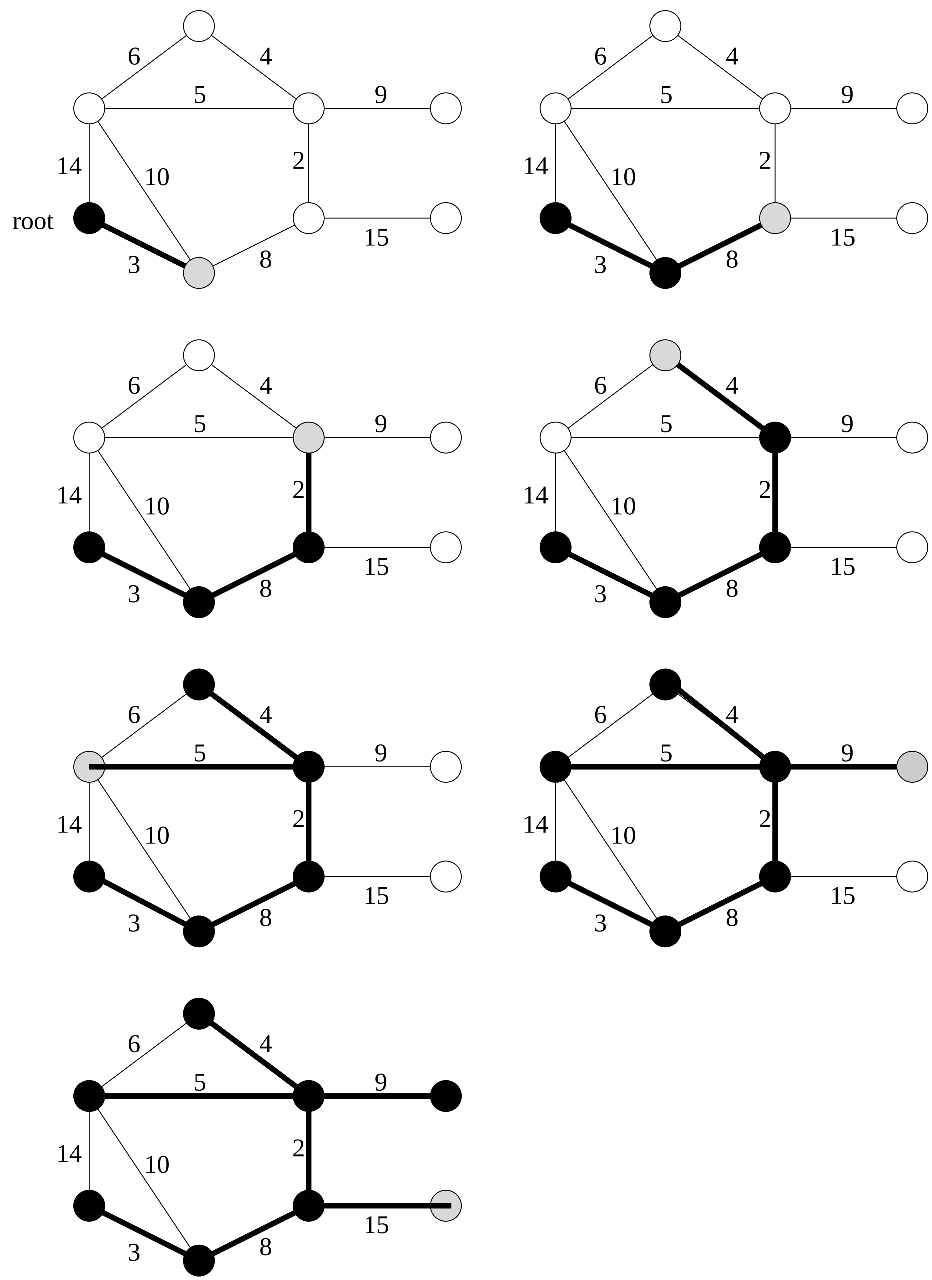
- set the key of each vertex to $\infty$ (except root $r$ whose key is set to $0$ so that it will be processed first)
- set parent of each vertex to NIL
- initialize min-priority queue $Q$ (all vertices)

Algorithm maintains the following **loop invariant**:

1. $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$
2. Vertices already placed into MST are those in $V - Q$
3. For all $v \in Q$, if $\pi[v] \neq$ NIL, then key$[v] < \infty$ and key$[v]$ is the weight of a light edge $(v, \pi[v])$ connecting $v$ to some vertex already placed into MST

*Line 8* identifies $u \in Q$ incident to a light edge crossing cut $(V - Q, Q)$, expect in first iteration, in which $u = r$ due to line 5.
Removing $u$ from $Q$ adds it to set $V - Q$ of vertices in the tree, adding $(u, \pi[u])$ to $A$.

The **for** loop of *lines 9–14* updates the *key* and $\pi$ fields of every vertex $v$ adjacent to $u$ but **not** in the tree. This maintains the third part of the loop invariant.


root
6
4
5
9
14
10
2
15
3
8

## Running time

Depends on how the min-priority queue $Q$ is implemented. If as *a binary min-heap*, then

- can use `Build-Min-Heap` for initialization, time $\mathcal{O}(|V|)$
- body of the **while** loop is executed $|V|$ times, each `Extract-Min` takes $\mathcal{O}(\log|V|)$, hence total time for all calls to `Extract-Min` is $O(|V|\log|V|)$
- **for** loop in lines 9–14 is executed $\mathcal{O}(E)$ times altogether, since sum of lengths of all adjacency lists is $2|E|$
- test for membership in $Q$ on line 10, can be implemented in constant time $\mathcal{O}(1)$ (keeping a membership bit for every vertex)
- line 12 performs `Decrease-Key` operation, each takes $\mathcal{O}(\log|V|)$ time, hence the total time spent here is $\mathcal{O}(|E|\log|V|)$
- **the total time:** $\mathcal{O}(|V|\log|V| + |E|\log|V|) = \mathcal{O}(|E|\log|V|)$

However, when using *Fibonacci heaps* implementation of the min-priority queue (Chapter 20), we get running time $\mathcal{O}(|E| + |V|\log|V|)$.

# Approximation algorithms

**Intuitive definition 1.** A problem is in P if there is a polynomial time algorithm to solve it (optimally, in case of optimization problems).

**Intuitive definitio 2.** A problem is NP-complete (hard) if it is "very unlikely" that there is a polynomial time algorithm to solve it (optimally, in case of optimization problems) but it's solvable in exponential time. Plus: the corretness of the solution can be verified in polynomial time.

**Approximation algorithms** compute **near-optimal** solutions.

Consider an *optimization problem*.

Each potential solution has a **positive cost**.

Algorithm has an **approximation ratio** of $\rho$, if for any input the cost $C$ of its solution is **within the factor** $\rho$ of cost of optimal solution $C^*$, i.e.:

For **maximization** problems, $0 < C \leq C^*$, thus we require $C^*/C \leq \rho$.

For **minimization** problems, $0 < C^* \leq C$, thus we require $C/C^* \leq \rho$.

Approximation ratio is **never** less than one.

An algorithm with guaranteed approximation ration of $\rho$ is called a **$\rho$-approximation algorithm**.

# The traveling-salesman problem

Problem: given complete, undirected graph $G = (V, E)$ with non-negative integer cost $c(u, v)$ for each edge, find cheapest Hamiltonian cycle of $G$.

Consider two cases: with and without **triangle inequality**.

$c$ satisfies triangle inequality, if it is always cheapest to go directly from some $u$ to some $w$; going by way of intermediate vertices can't be less expensive.

Finding an optimal solution is NP-complete in both cases.

# TSP with triangle inequality

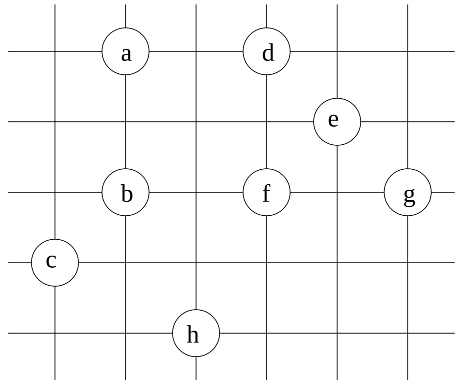
We compute a **minimum spanning tree** whose weight is lower bound for length of optimal TSP tour.

We use function MST-PRIM$(G, c, r)$, which computes an MST for $G$ and weight function $c$, given some arbitrary root $r$.

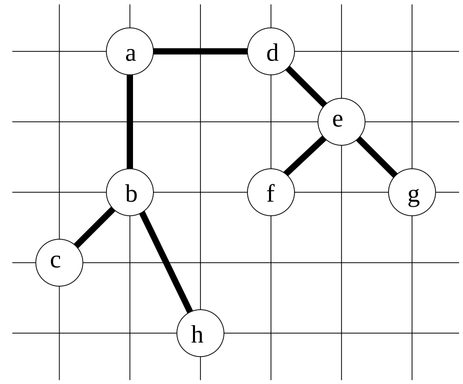
Input: $G = (V, E)$, $c : E \to \mathbb{R}$

APPROX-TSP-TOUR

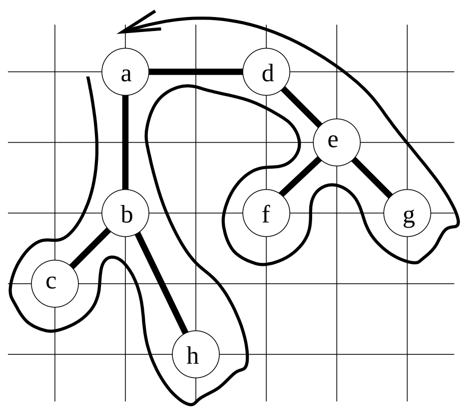
1: Select arbitrary $v \in V$ to be "root"
2: Compute MST $T$ for $G$ and $c$ from root $r$ using MST-PRIM$(G, c, r)$
3: Let $L$ be list of vertices visited in pre-order tree walk of $T$
4: Return the Hamiltonian cycle that visits the vertices in the order $L$



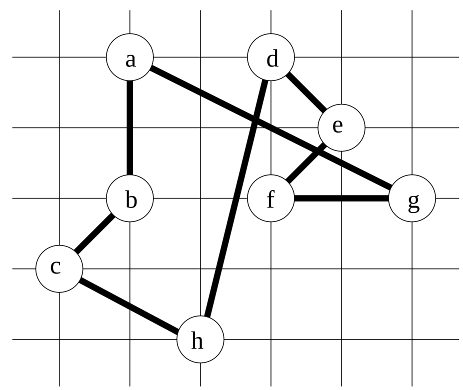

Set of points, lie in grid



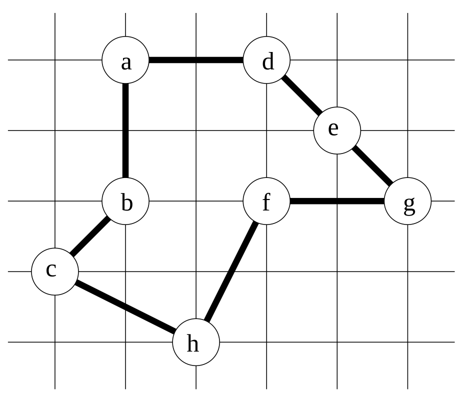

MST, root a




Pre−order walk




Resulting tour, cost ca. 19.1




Optimal tour, cost ca. 14.7

**Theorem.** APPROX-TSP-TOUR is a polynomial time 2-approximation algorithm for the TSP problem with triangle inequality.

**Proof.** Polynomial running time obvious, simple MST-PRIM takes $\Theta(V^2)$, computing preorder walk takes no longer.

Correctness obvious, preorder walk is always a tour.

Let $H^*$ denote an optimal tour for given set of vertices.

Deleting any edge from $H^*$ gives a spanning tree.

Thus, weight of **minimum** spanning tree is lower bound on cost of optimal tour:

$$c(T) \leq c(H^*)$$

A **full walk** of $T$ lists vertices when they are **first visited**, and also when they are **returned to**, after visiting a subtree.

*Example:* a,b,c,b,h,b,a,d,e,f,e,g,e,d,a

Full walk $W$ traverses every edge **exactly twice**, thus

$$c(W) = 2c(T)$$

Together with $c(T) \leq c(H^*)$, this gives

$$c(W) = 2c(T) \leq 2c(H^*)$$

We want to find connection between cost of $W$ and cost of "our" tour.

**Problem:** $W$ is in general **not** a proper tour, since vertices may be visited more than once...

**But**: using the **triangle inequality**, we can **delete** a visit to any vertex from $W$ and cost does **not increase**.

**Deleting** a vertex $v$ from walk $W$ between visits to $u$ and $w$ means going from $u$ **directly** to $w$, without visiting $v$.

This way, we can consecutively remove all multiple visits to any vertex.

*Example:*

full walk a,b,c,b,h,b,a,d,e,f,e,g,e,d,a becomes a,b,c,h,d,e,f,g.

This ordering (with multiple visits deleted) is **identical** to that obtained by preorder walk of $T$ (with each vertex visited only once).

It certainly is a Hamiltonian cycle. Let's call it $H$.

$H$ is just what is computed by APPROX-TSP-TOUR.

$H$ is obtained by deleting vertices from $W$, thus $c(H) \leq c(W)$

Conclusion:

$$c(H) \leq c(W) \leq 2c(H^*)$$

Done.

Although factor 2 looks nice, there are better algorithms.

There's a $3/2$ approximation algorithm by Christofedes (**with** triangle inequality).

## The general TSP

Now $c$ does no longer satisfy triangle inequality.

**Theorem.** If P $\neq$ NP, then for any constant $\rho \geq 1$, there is no polynomial time $\rho$-approximation algorithm for the general TSP.