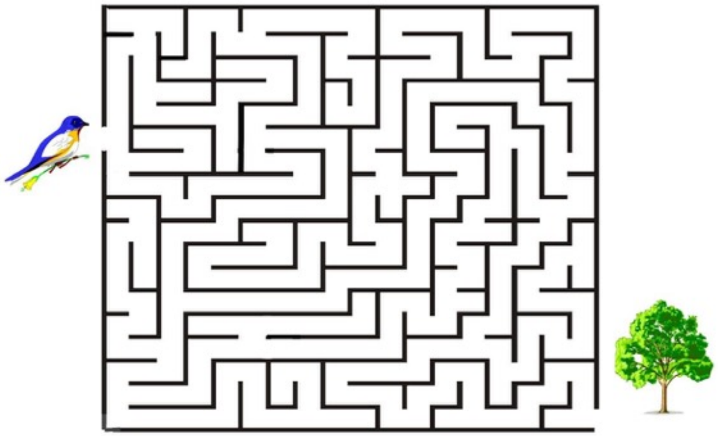
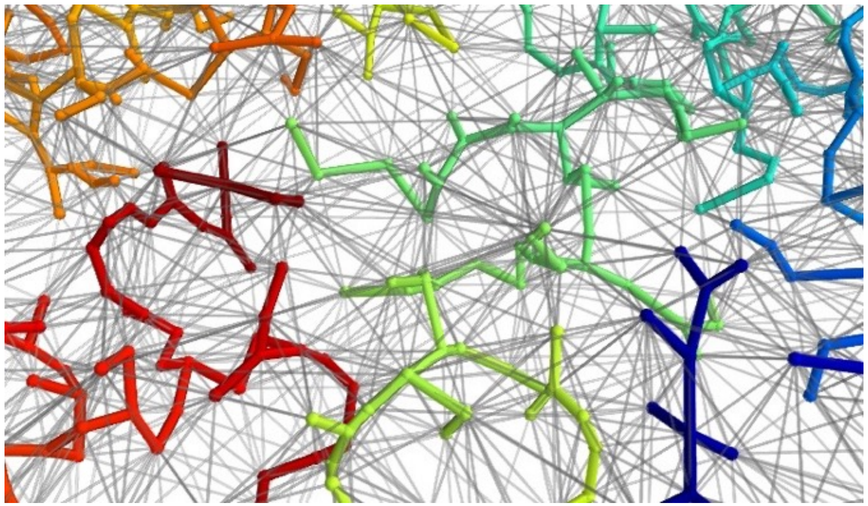
15-251
Great Theoretical Ideas in Computer Science

Lecture 10:
Graphs II: Graph Algorithms





September 29th, 2016

Today's Menu

- Graph search: DFS

- Minimum spanning tree

- Maximum matching

Graph Search

Motivating question

Given a map, and two locations x and y,
determine efficiently if it is possible to go from x to y.

How can we efficiently check if two vertices in a graph are connected or not?

I ❤️ Recursion

The basic idea:

To explore all the nodes you can reach from vertex x:

explore all the nodes you can reach from the neighbors of x.

Depth-First Search

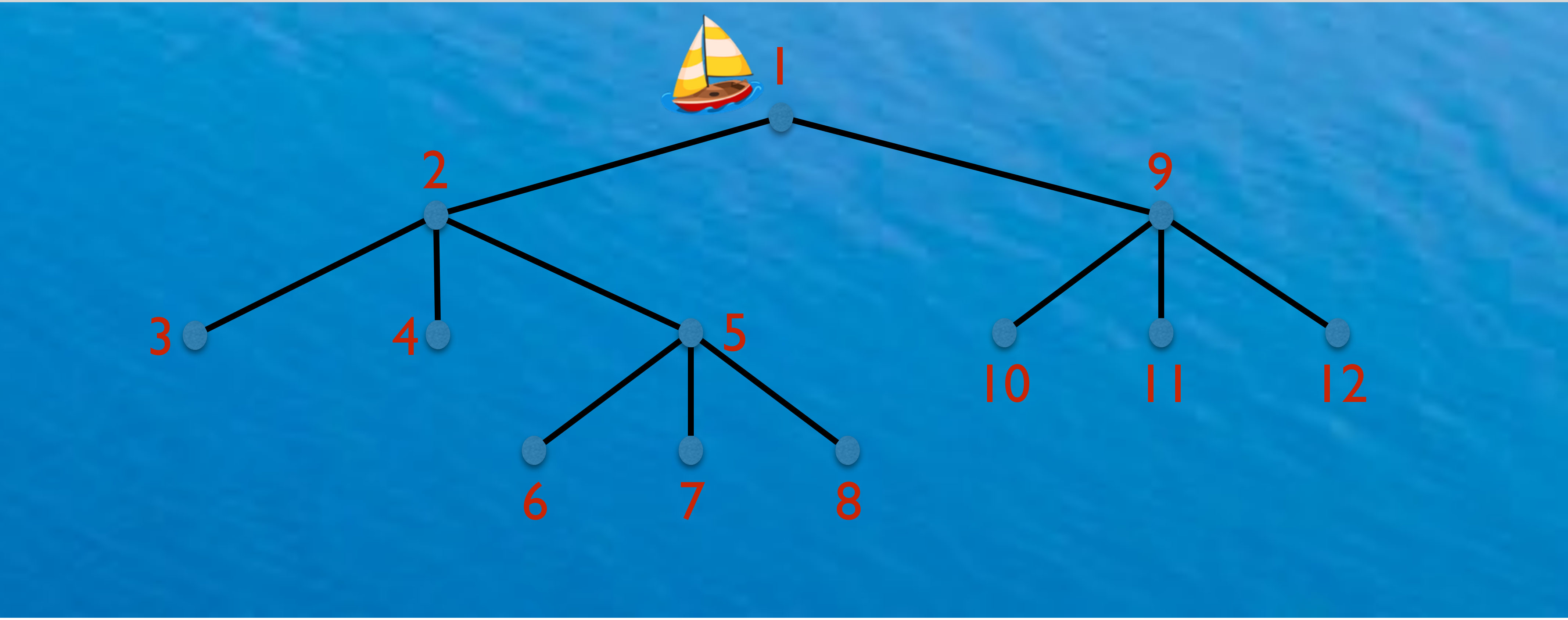
DFS: On input $G = (V, E)$, $x \in V$

Mark x as "visited".

For each $z \in N(x)$:

If z is not marked "visited", run DFS(G, z).

I ❤️ Recursion



Suppose x = 1

The order in which vertices marked "visited":

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

I ❤️ Recursion

DFS: On input $G = (V, E)$, $x \in V$

Mark x as "visited".

For each $z \in N(x)$:

If z is not marked "visited", run DFS(G, z).

The above visits every vertex *connected* to x.

To traverse every vertex in the graph:

DFS2: On input $G = (V, E)$

For each vertex v that is not marked "visited":

run DFS(G, v).

I ❤️ Recursion

DFS: On input G = (V, E), x ∈ V

Mark x as “visited”.

For each z ∈ N(x):

If z is not marked “visited”, run DFS(G, z).

Running time: $O(m)$ (exercise)

DFS2: On input G = (V, E)

For each vertex v that is not marked “visited”:

run DFS(G, v).

Running time: $O(n+m)$ (exercise)

I ❤️ Recursion

Can use DFS to solve:

- Check if there is a path between two given vertices.
- Decide if G is connected.
- Identify the connected components of G.
- (and other similar problems)

There are other graph traversing algorithms that you can use to solve above problems.

One famous one is Breadth-First Search (BFS).

Minimum Spanning Tree

Motivating question

Year: 1926

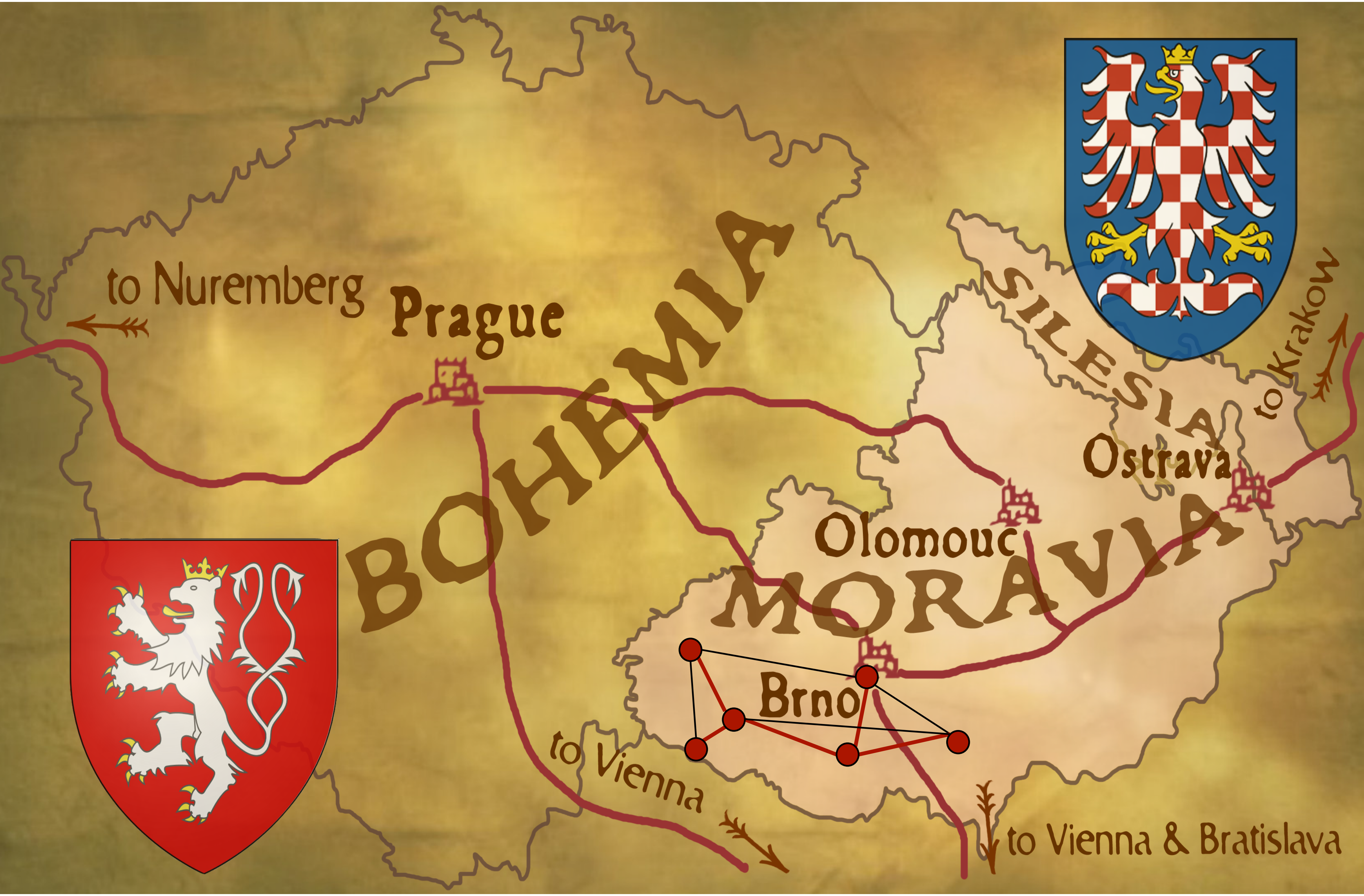
Place: Brno, Moravia

Our Hero: Otakar Boruvka

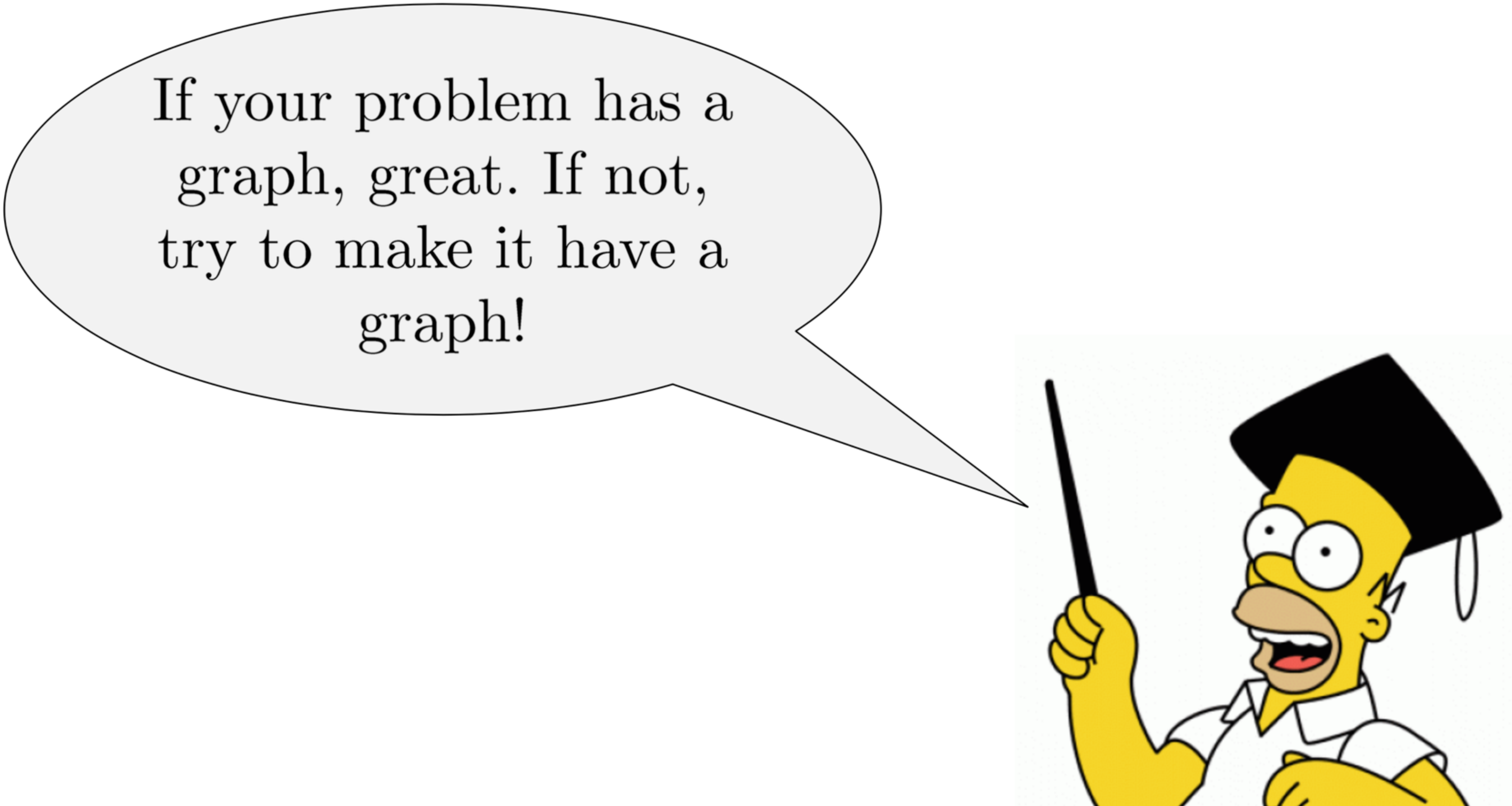
Boruvka's pal Jindrich Saxel was working for Zapadomoravske elektrarny (the West Moravian Power Plant company).

Saxel asked:

What is the least cost way to electrify southwest Moravia?

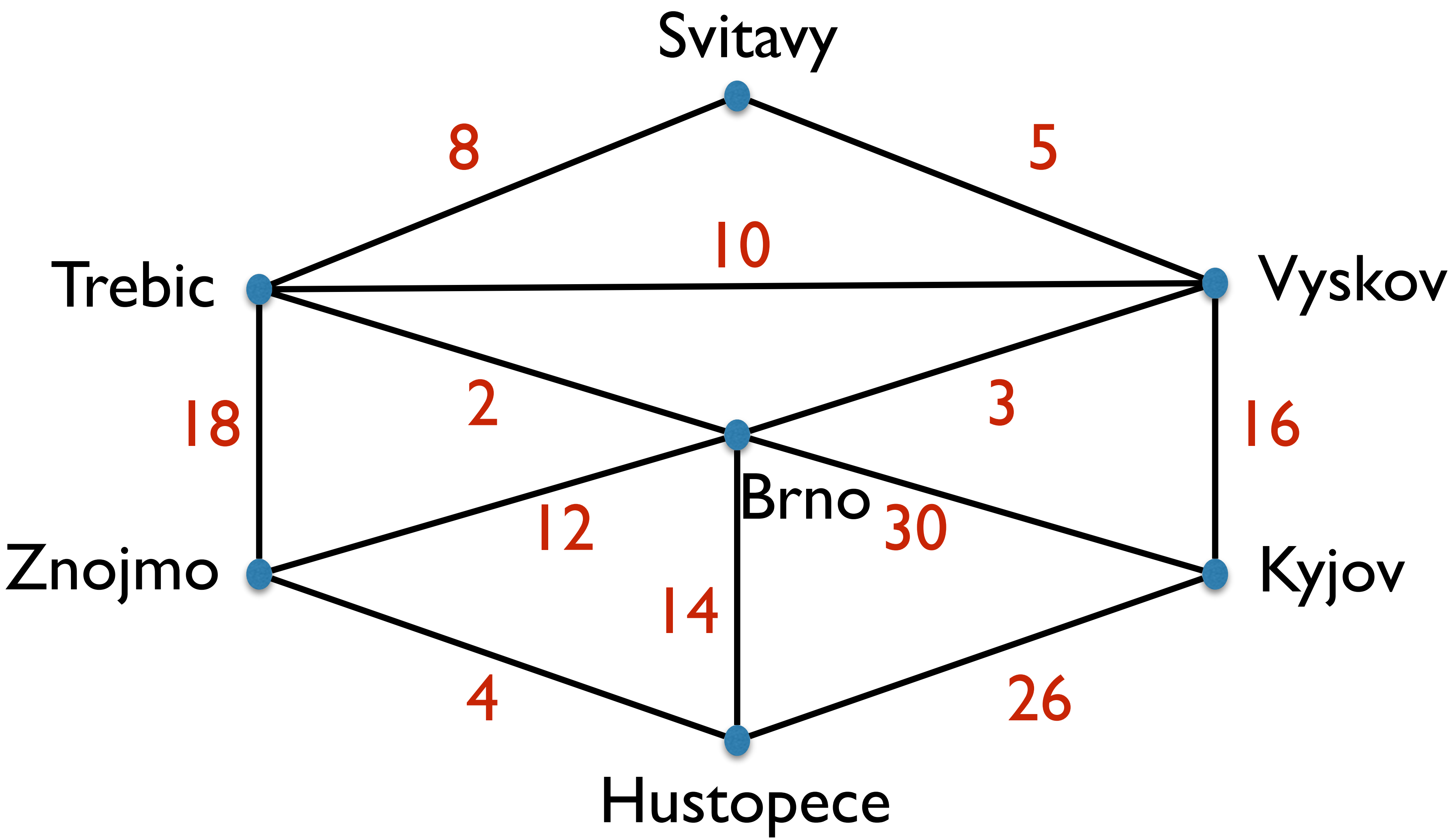

to Nuremberg
Prague
BOHEMIA
SILESIA
to Krakow
Ostrava
Olomouc
MORAVIA
Brno
to Vienna
to Vienna & Bratislava

Remember the CS life lesson



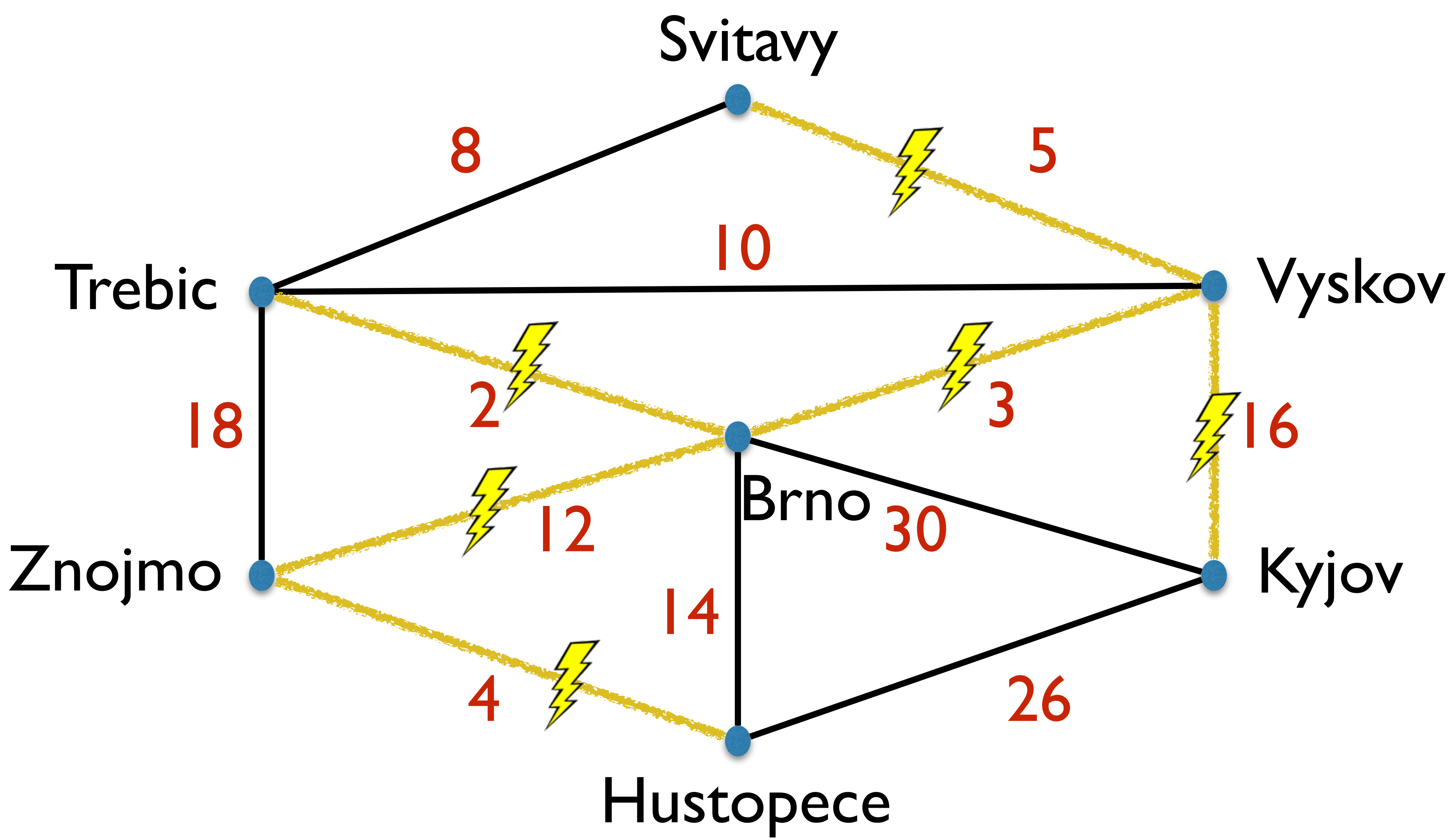
Graph representation

weighted graph



Graph representation

weighted graph



Total weight/cost: 42

Minimum spanning tree problem

Input: A connected graph $G = (V, E)$,
and a cost function $c : E \to \mathbb{R}^+$.

Output: Subset of edges with minimum total cost
such that all vertices are connected.

Observation:

The output must be a tree.

Recall
tree: connected, acyclic

If not (i.e. there is a cycle), you could delete an edge from the cycle to get a cheaper solution.

Minimum spanning tree problem

<u>Convenient Assumption:</u>

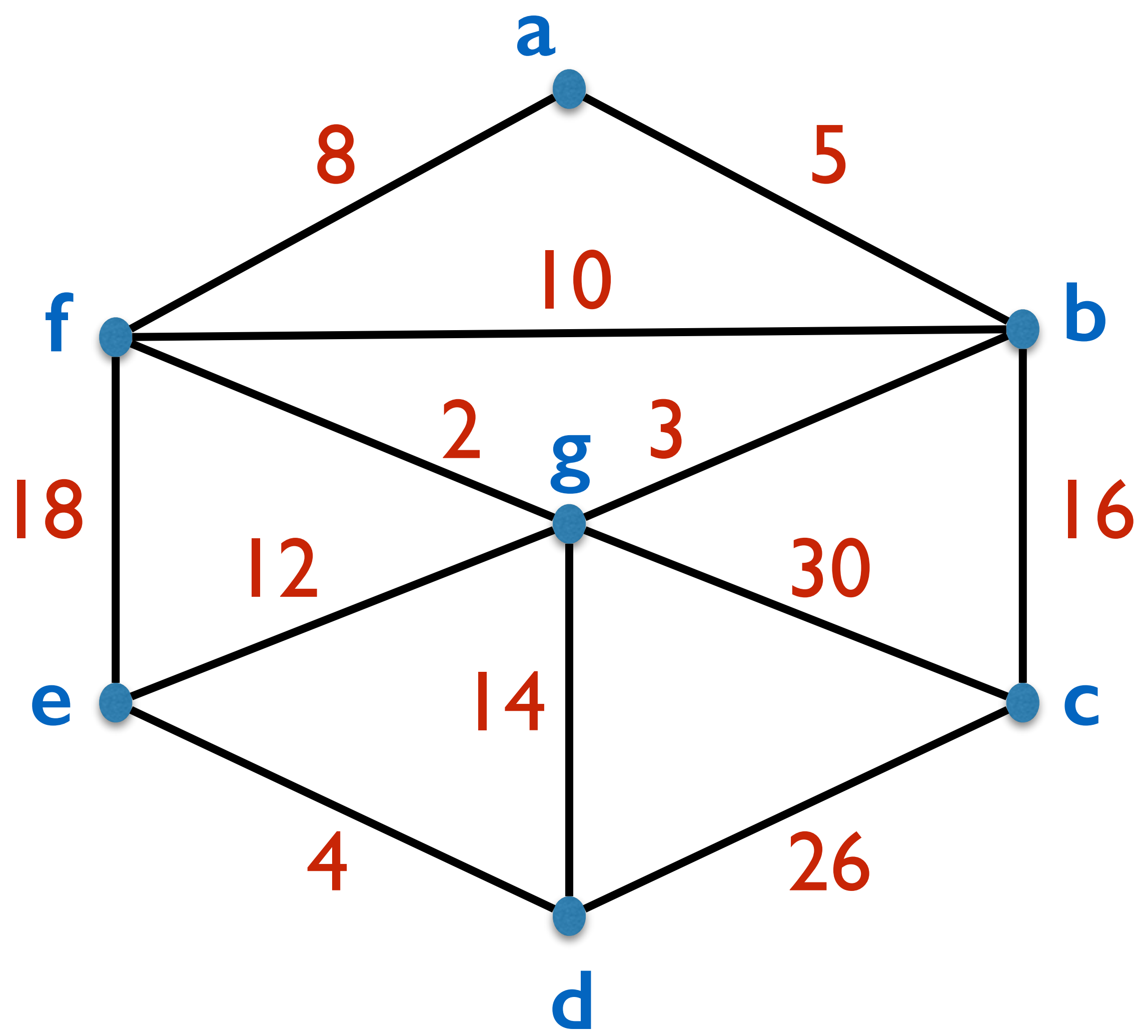
Edges have distinct costs.

<u>Exercise</u>: In this case *the* MST is unique.

A hint on why this is WLOG:

"Whether the distance from Brno to Breclav is 50km or 50km and 1cm is a matter of conjecture."

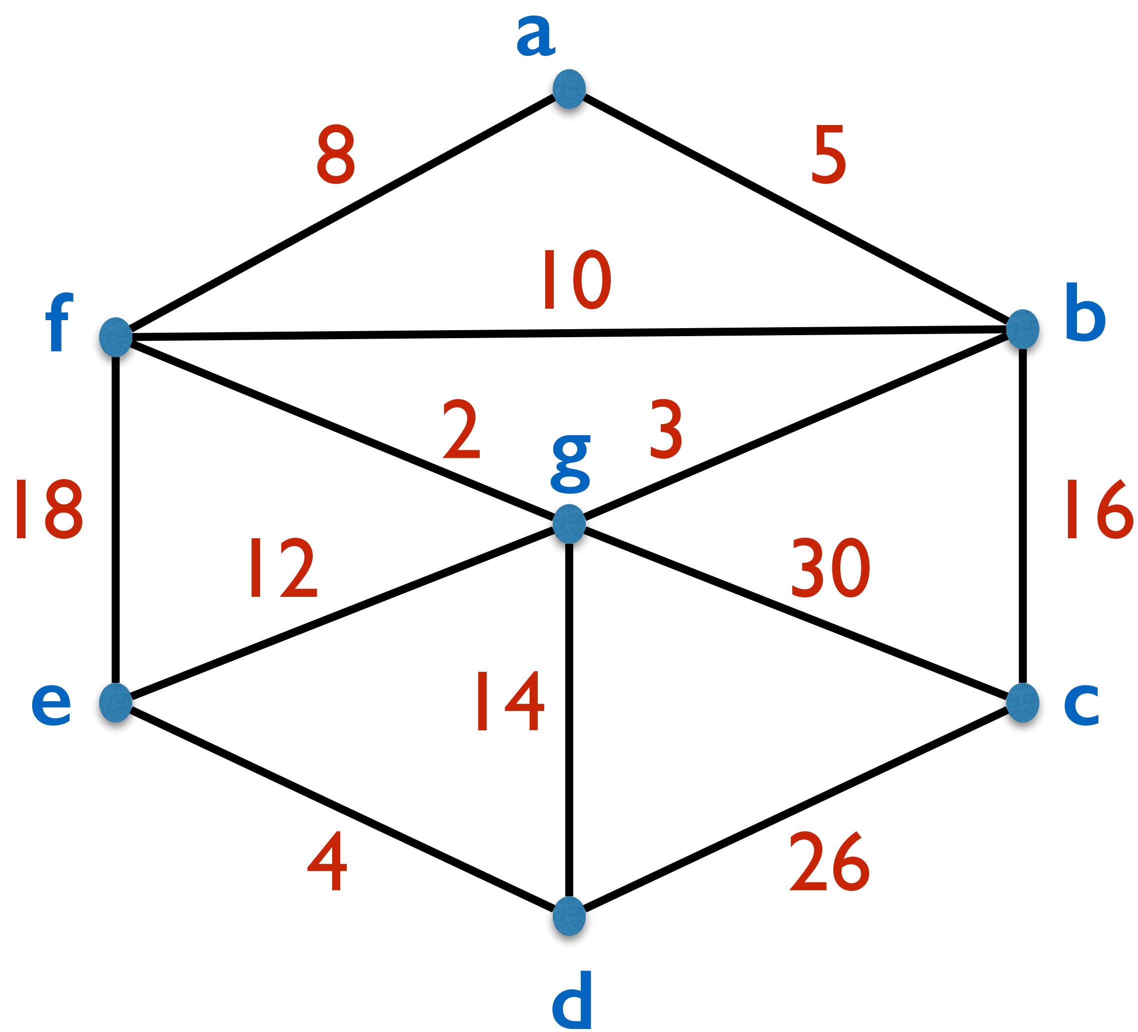
Jarník-Prim Algorithm



V' = vertices connected so far

E' = edges in the solution so far

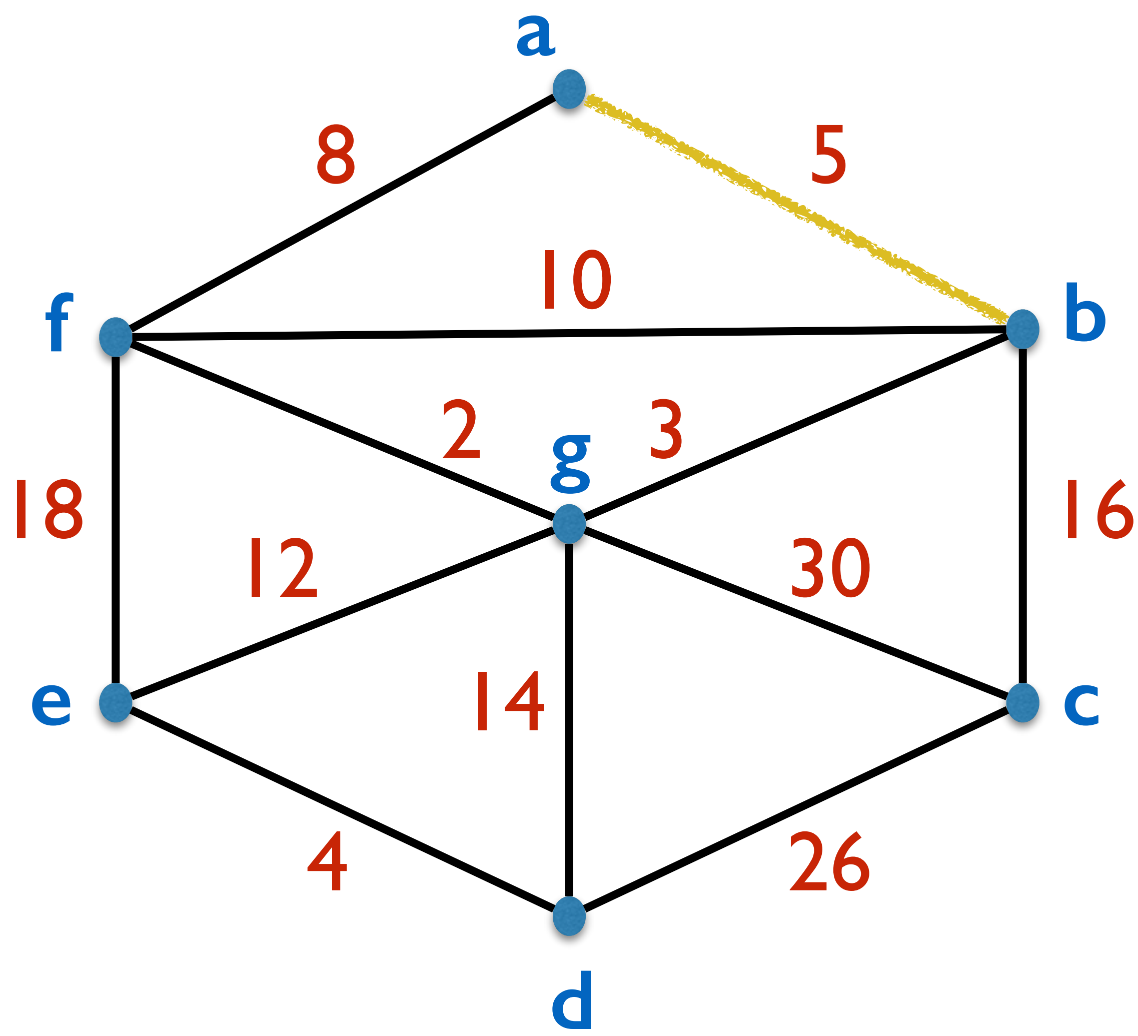
Jarník-Prim Algorithm



V' = {a}   (start with an arbitrary node)

E' = { }

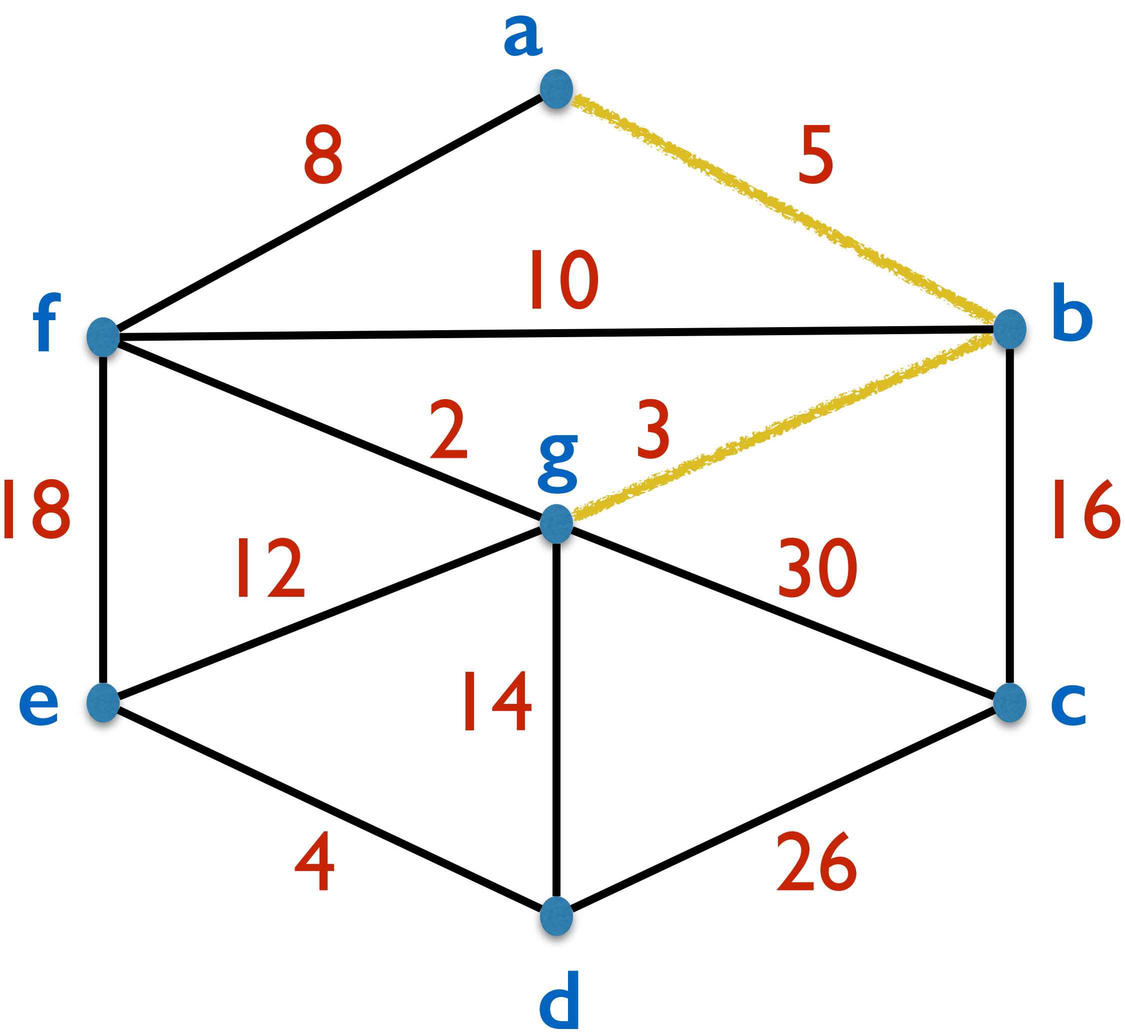
Jarník-Prim Algorithm



V’ = {a, b}

E’ = {{a, b}}

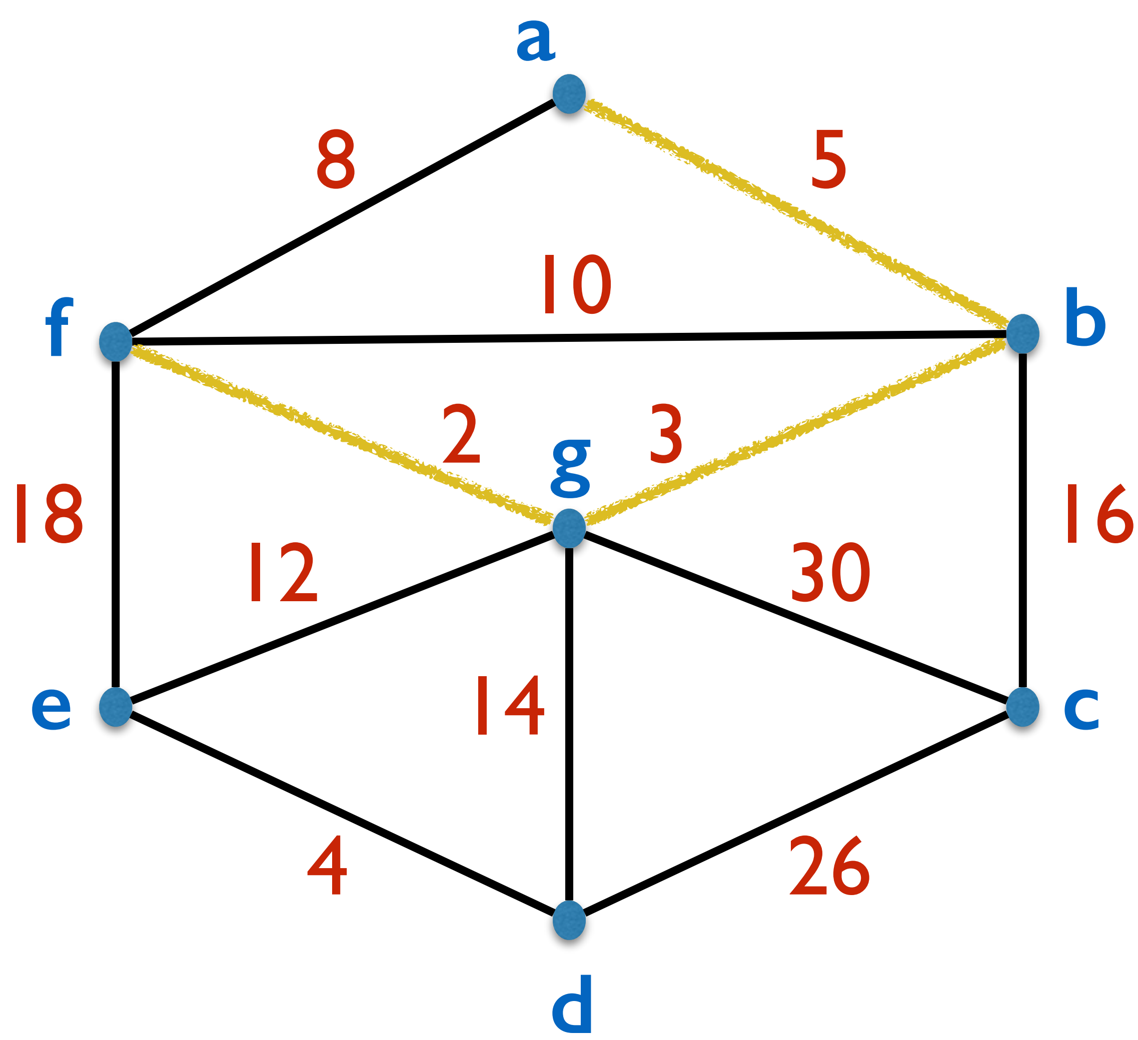
Jarník-Prim Algorithm



V' = {a, b, g}

E' = {{a, b}, {b, g}}

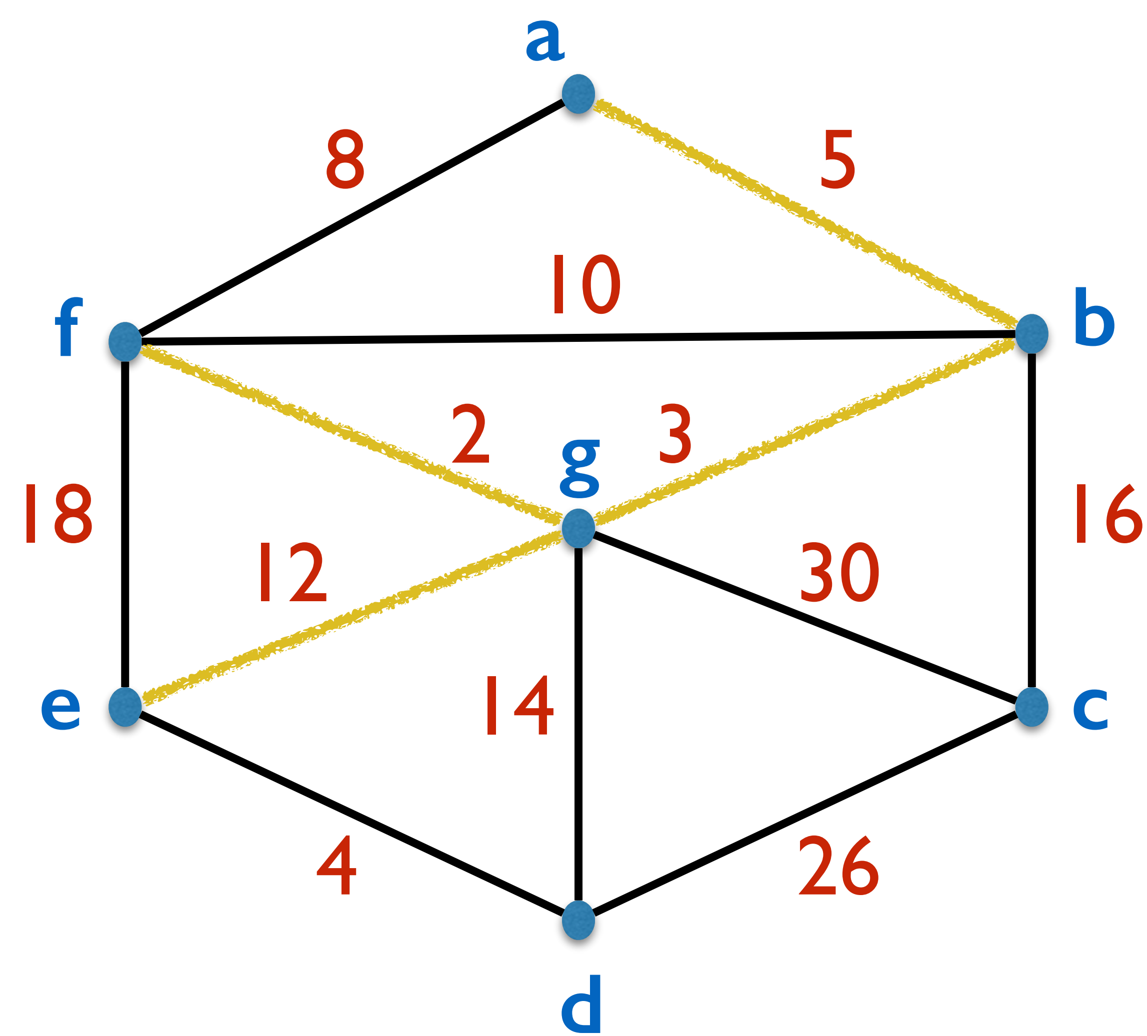
Jarník-Prim Algorithm



V’ = {a, b, g, f}

E’ = {{a, b}, {b, g}, {g, f}}

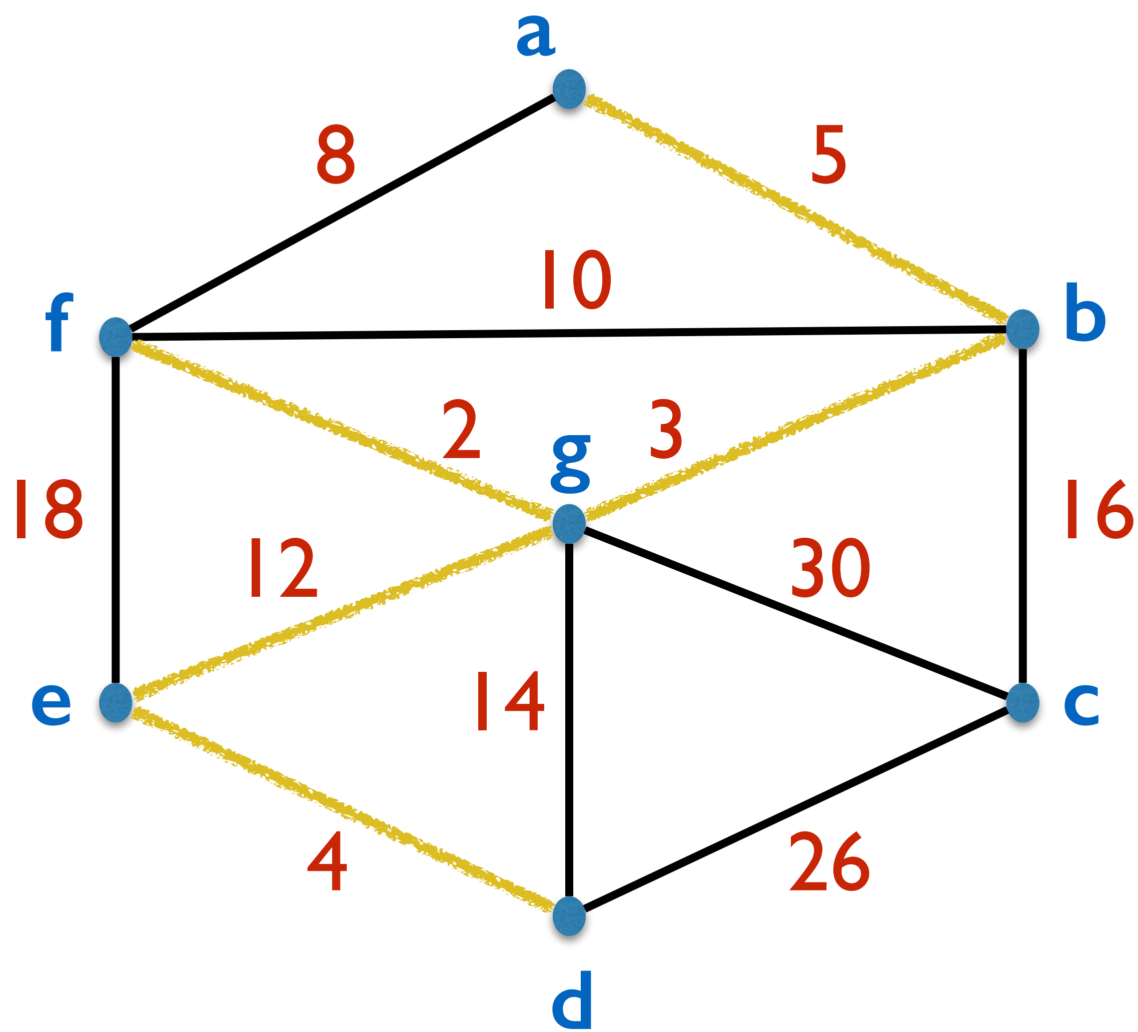
Jarník-Prim Algorithm



V’ = {a, b, g, f, e}

E’ = {{a, b}, {b, g}, {g, f}, {g, e}}

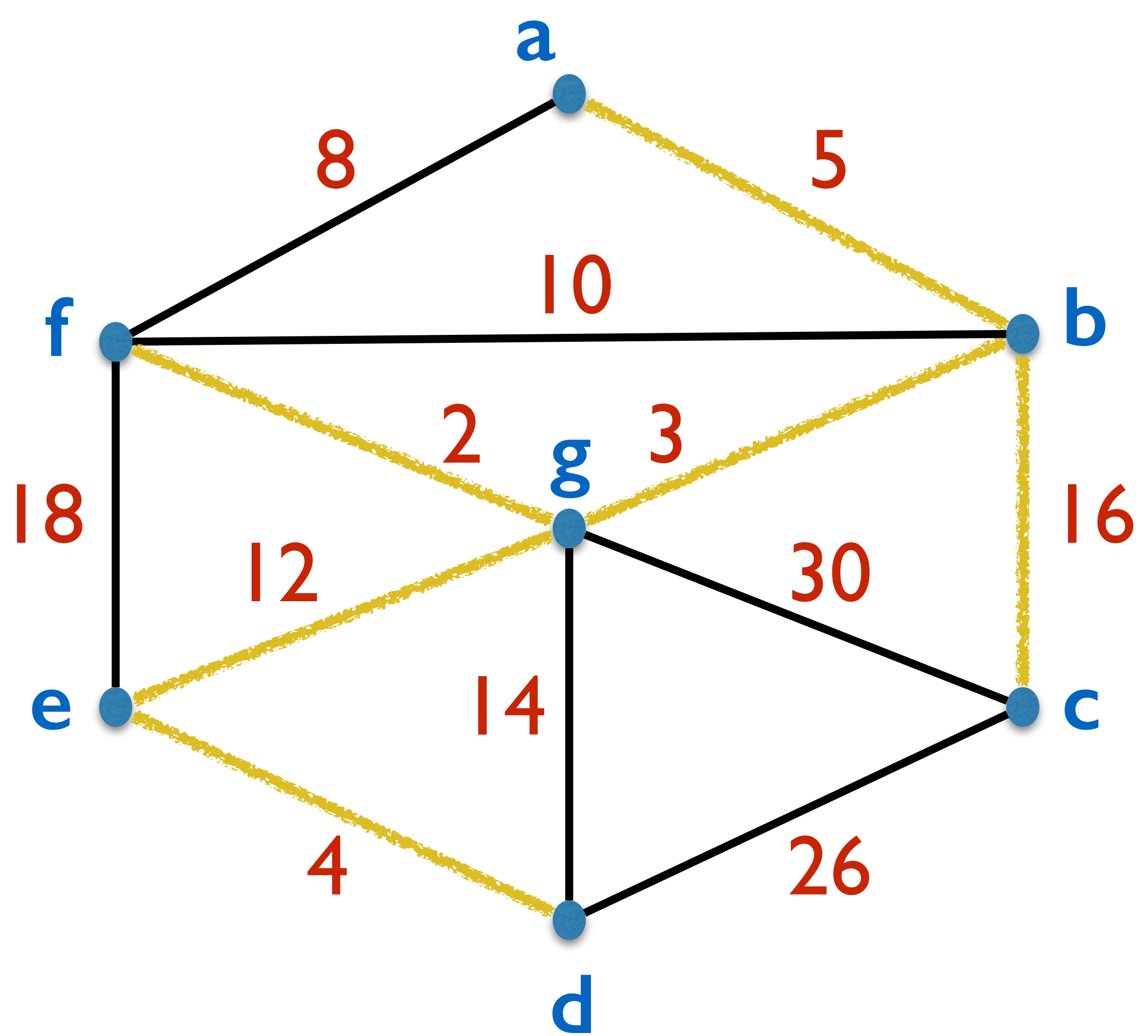
Jarník-Prim Algorithm



V' = {a, b, g, f, e, d}

E' = {{a, b}, {b, g}, {g, f}, {g, e}, {e, d}}

Jarník-Prim Algorithm



Total cost: 42

V' = {a, b, g, f, e, d, c}

E' = {{a, b}, {b, g}, {g, f}, {g, e}, {e, d}, {b, c}}

Jarník-Prim Algorithm

On input a weighted & connected graph **G** = (**V**, **E**):

V' = {**w**} (for an arbitrary **w** in **V**)

E' = ∅

While **V'** ≠ **V**:

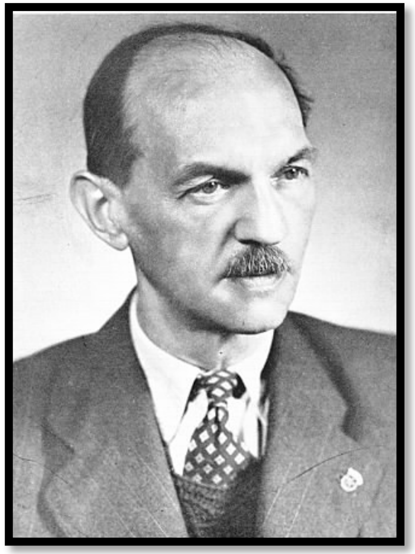
- Let {**u**,**v**} be the min cost edge such that **u** is in **V'**, **v** is not in **V'**.
- **E'** = **E'** + {**u**,**v**}
- **V'** = **V'** + **v**

Output **E'**

Jarník-Prim Algorithm

This is usually known as Prim's algorithm.
(due to a 1957 publication by Robert Prim)

Actually, first discovered by Vojtech Jarník, who described it in a letter to Boruvka, and later published it in 1930.



Boruvka himself had published a different algorithm in 1926.

Jarník-Prim Algorithm

How do we know the algorithm is correct?

Lemma: **(MST Cut Property)**

Let $G = (V, E)$ be a graph with distinct edge costs.

Let $V' \subset V$ ($V' \neq \emptyset$, $V' \neq V$).

Let $e = \{u, v\}$ be the cheapest edge with $u \in V', v \notin V'$.

Then the MST **must** contain this edge e .

MST Cut Property

Proof idea:

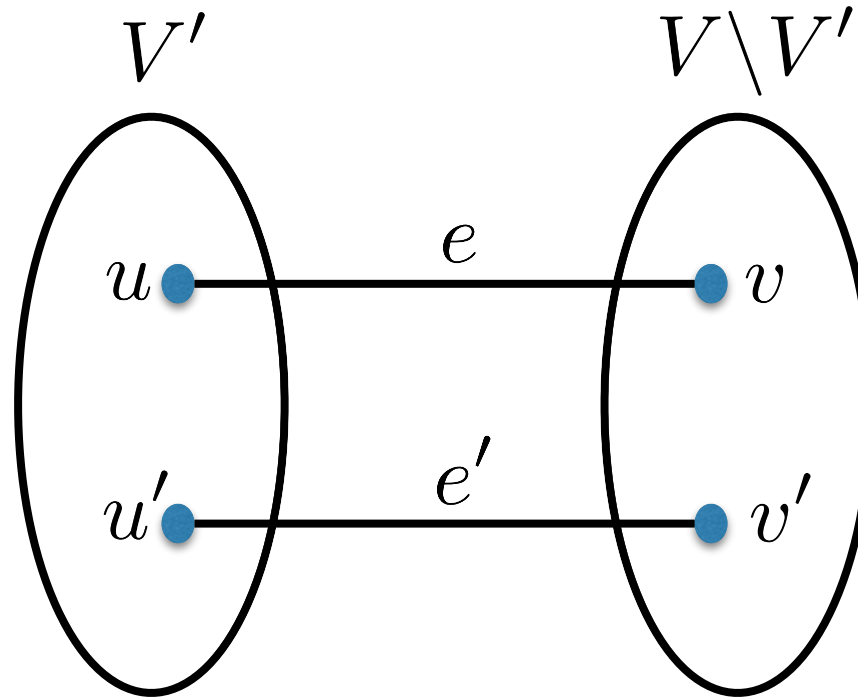
Proof by contradiction.

Let T be the MST.

Suppose e={u,v} is not in T.

e'={u',v'} is in T. (e' chosen carefully)

c(e') > c(e)



T - e' + e is a spanning tree with smaller cost. CONTRADICTION

- clearly has smaller cost
- clearly has n-1 edges
- argue it must be connected

} it is a spanning tree

Runtime race for MST: An amusing story

A naïve implementation of Jarník-Prim runs in time $O(m^2)$.

A better implementation runs in time $O(m \log m)$.

In practice, this is pretty good!

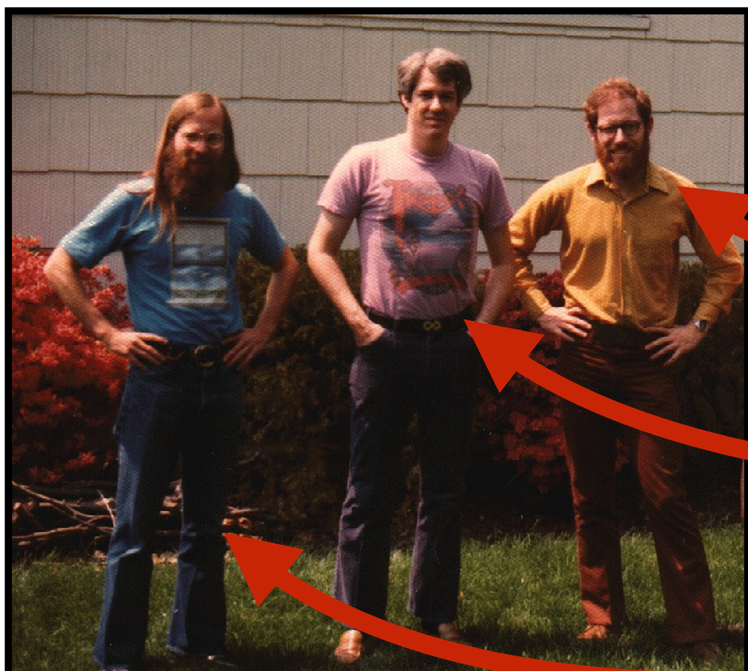
But a good algorithm designer always thinks:

Can we do better?

Runtime race for MST: An amusing story

1984: Fredman & Tarjan invent the "Fibonacci heap" data structure.

Running time improved from $O(m \log m)$ to $O(m \log^* m)$



also not Fredman

not Fredman

Tarjan

Runtime race for MST: An amusing story

1986: Gabow, Galil, T. Spencer, Tarjan improved the alg.

Running time improved from $O(m \log^* m)$ to

$$O(m \log(\log^* m))$$

Gabow

Galil

Tarjan & Not-Spencer

Runtime race for MST: An amusing story

1997: Chazelle invents "soft heap" data structure.

Running time improved from $O(m \log(\log^* m))$ to

$O(m\ \alpha(m) \log \alpha(m))$

What is $\alpha(m)$?

Bernard Chazelle   Damien Chazelle (writer & director)

Runtime race for MST: An amusing story

What is $\alpha(m)$?

It is known as the Inverse-Ackermann function.

$\log^*(m)$ # times you do $\log$ to go down to 2.

$\log^{**}(m)$ # times you do $\log^*$ to go down to 2.

$\log^{***}(m)$ # times you do $\log^{**}$ to go down to 2.

$\alpha(m)$ # $*$'s you need so that $\log^{***\cdots***}(m) \leq 2$

Incomprehensibly small!

Runtime race for MST: An amusing story

2002: Pettie & Ramachandran gave a new algorithm.

They proved its runnin time is $O(\text{optimal})$.

Would you like to know its running time?

So would we! It is **unknown**.

All we know is: whatever it is, it's optimal.

Pettie

Ramachandran

Maximum matching problem
(in bipartite graphs)

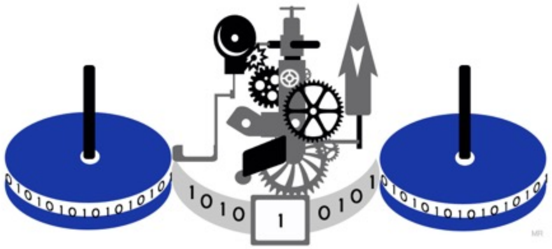
Some motivating real-world examples

matching machines and jobs

Job 1

Job 2

⋮ ⋮



Job n

Some motivating real-world examples

matching professors and courses

⋮

15-110

15-112

15-122

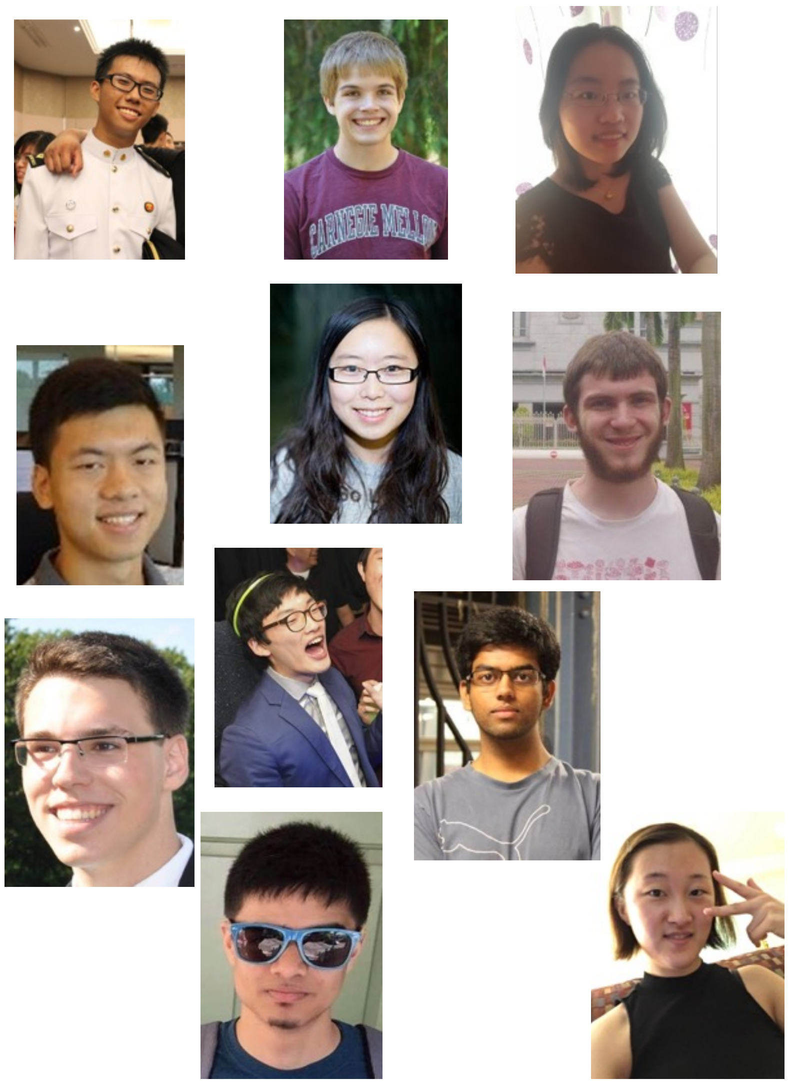
15-150

15-251

⋮

Some motivating real-world examples

matching students and internships



Macrosoft

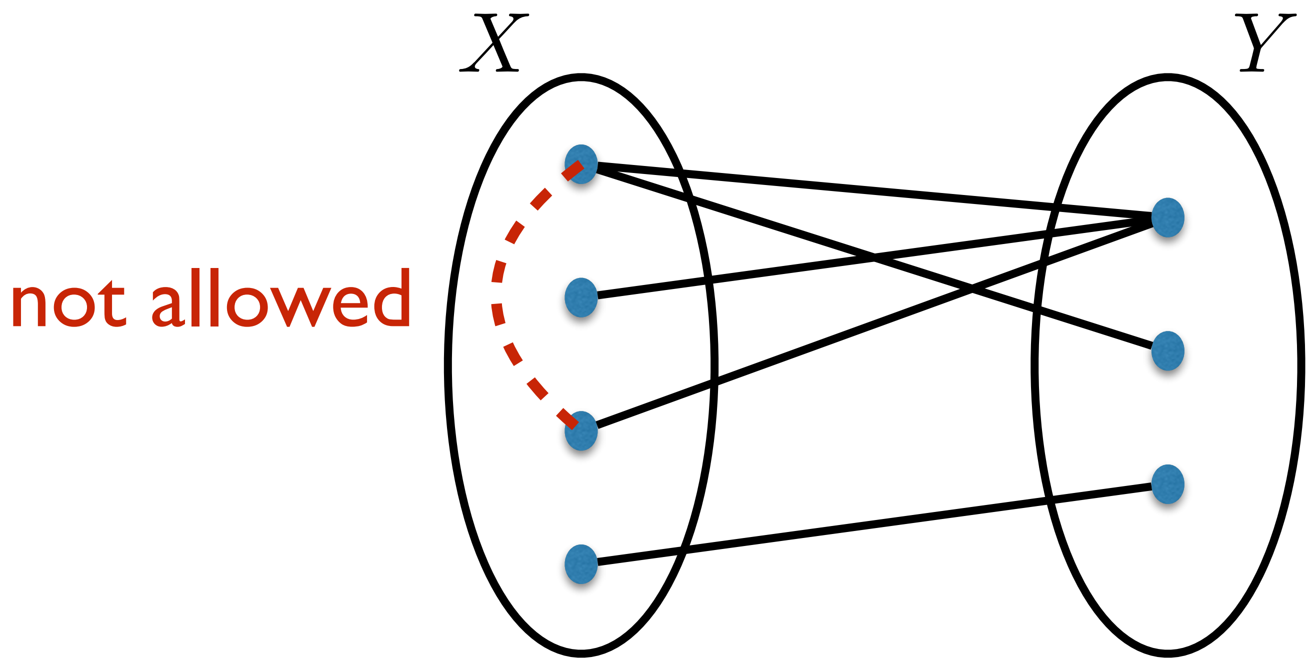
Moogle
Google

UMBRELLA
CORPORATION
"Our business is life itself..."

Remember the CS life lesson

If your problem has a graph, great. If not, try to make it have a graph!

Bipartite Graphs



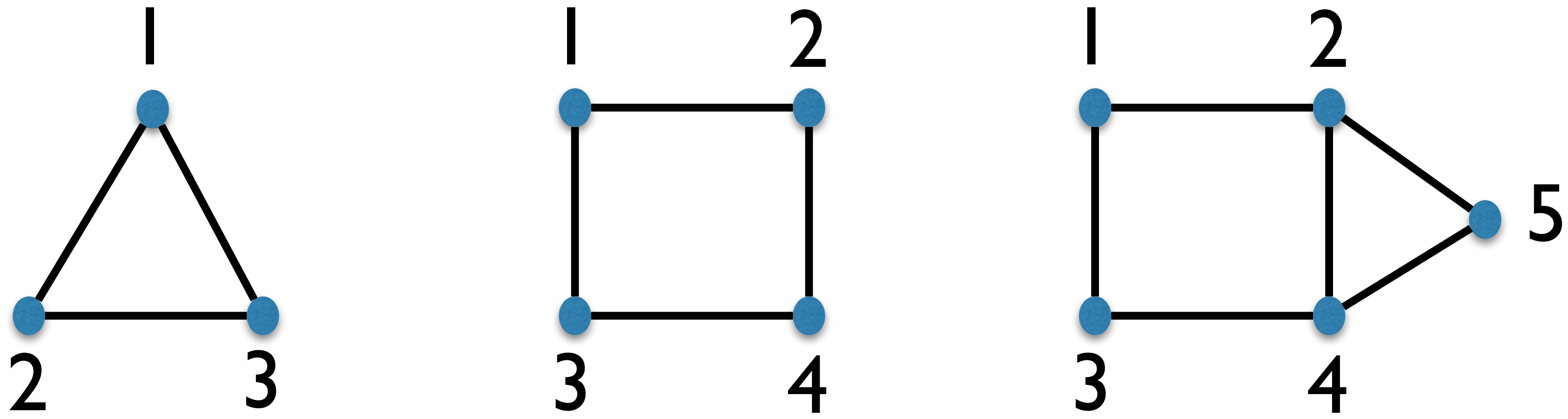
$G = (V, E)$ is **bipartite** if:

- there exists a bipartition of V into X and Y
- each edge connects a vertex in X to a vertex in Y

Given a graph $G = (V, E)$, we could ask, is it bipartite?

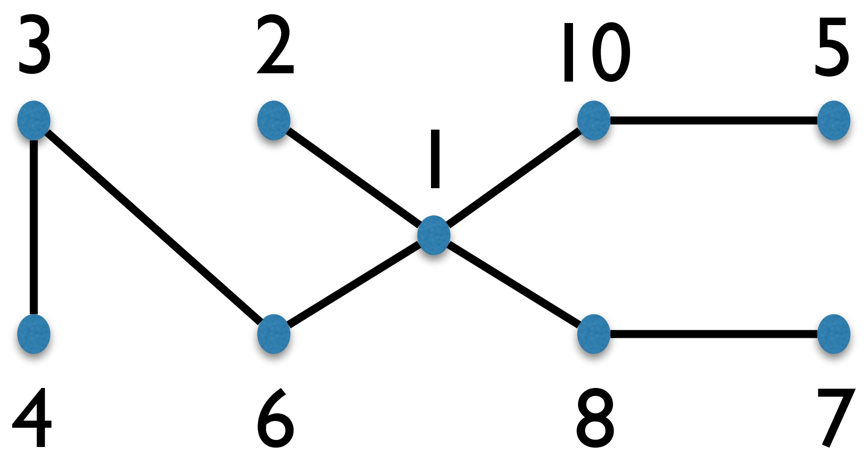
Bipartite Graphs

Given a graph $G = (V, E)$, we could ask, is it bipartite?



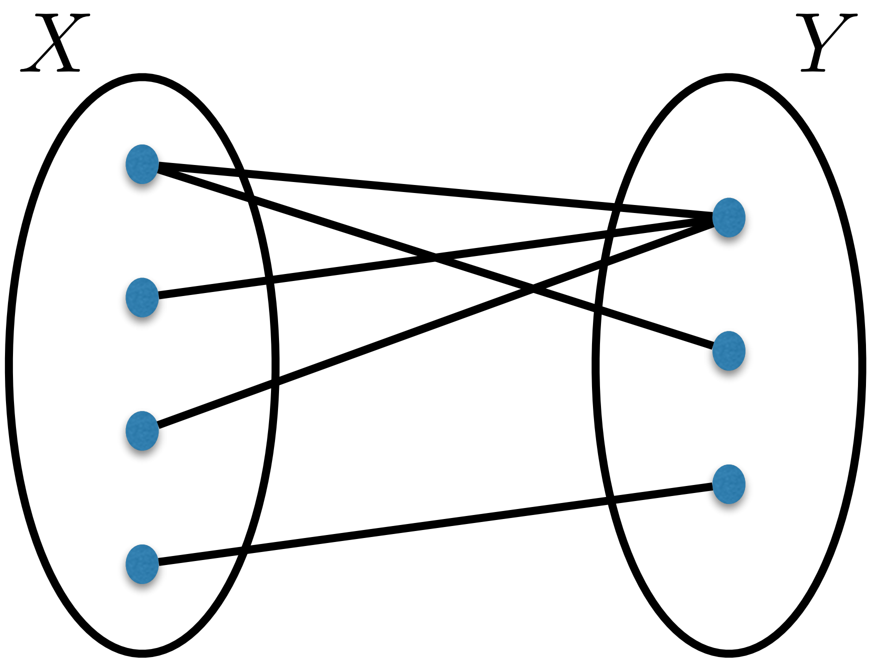
Poll

Is this graph bipartite?



- Yes
- No
- Beats me

Bipartite Graphs



Often we write the bipartition explicitly:

$$G = (X, Y, E)$$

Bipartite Graphs

Great for modeling relations between two classes of objects.

Examples:

X = machines, Y = jobs

An edge $\{x, y\}$ means x is capable of doing y.

X = professors, Y = courses

An edge $\{x, y\}$ means x can teach y.

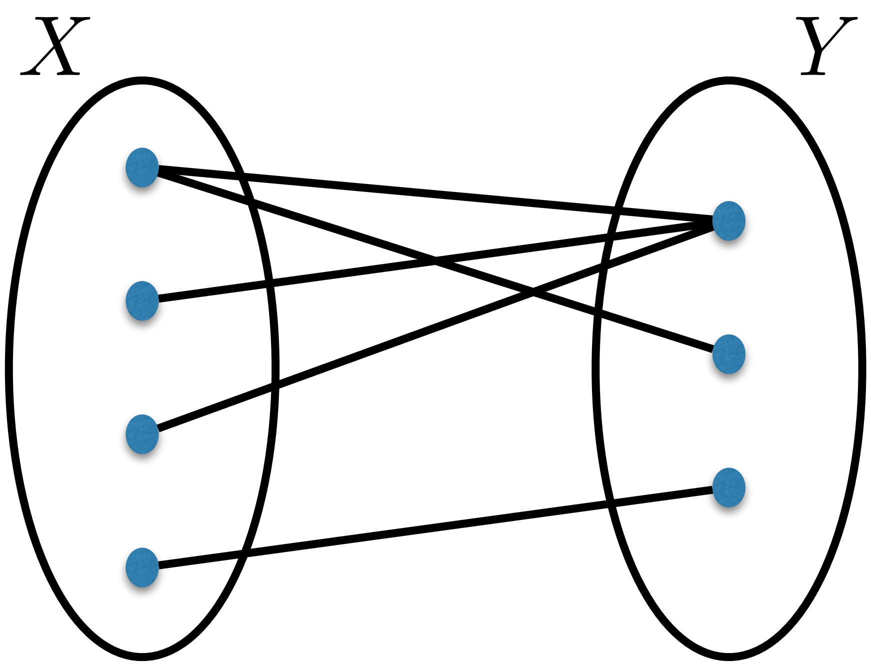
X = students, Y = internship jobs

An edge $\{x, y\}$ means x and y are interested in each other.

...

Matchings in bipartite graphs

Often, we are interested in finding a **matching** in a bipartite graph



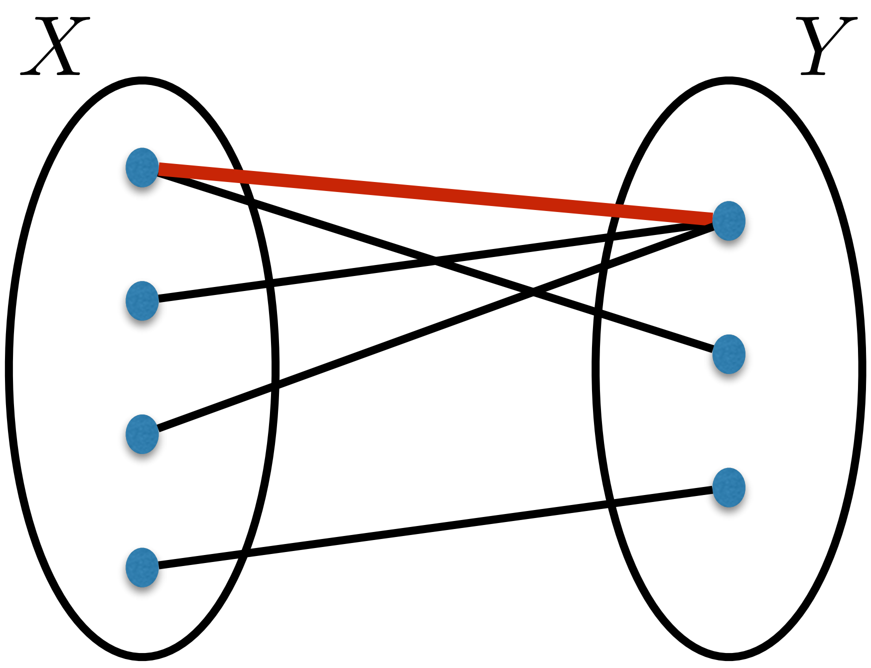
A **matching** :

A subset of the edges that do not share an endpoint.

Matchings in bipartite graphs

Often, we are interested in finding a **matching** in a bipartite graph

matching



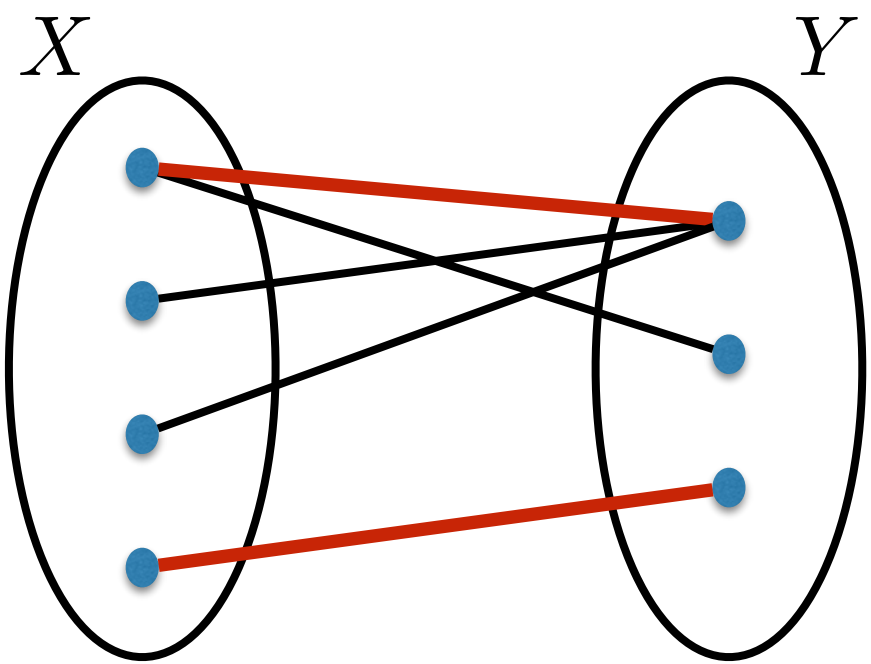
A **matching** :

A subset of the edges that do not share an endpoint.

Matchings in bipartite graphs

Often, we are interested in finding a **matching** in a bipartite graph

matching



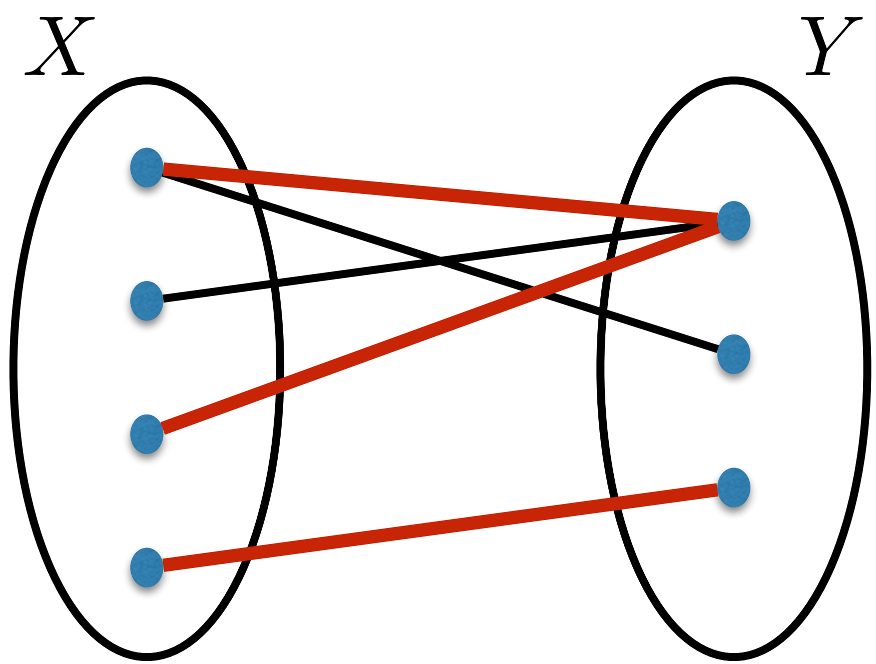
A **matching** :

A subset of the edges that do not share an endpoint.

Matchings in bipartite graphs

Often, we are interested in finding a **matching** in a bipartite graph

not a
matching



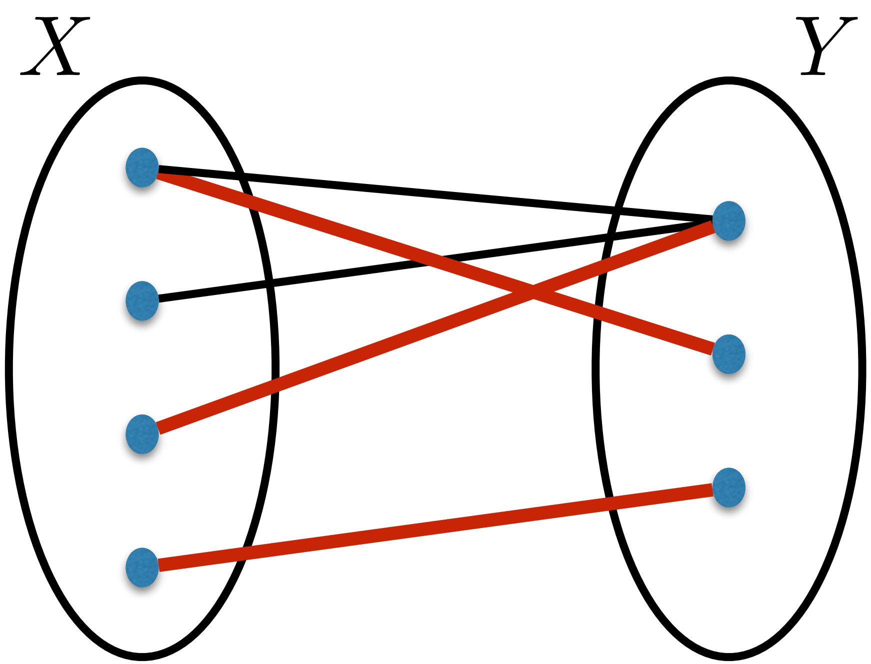
A **matching** :

A subset of the edges that do not share an endpoint.

Matchings in bipartite graphs

Often, we are interested in finding a **matching** in a bipartite graph

maximum matching

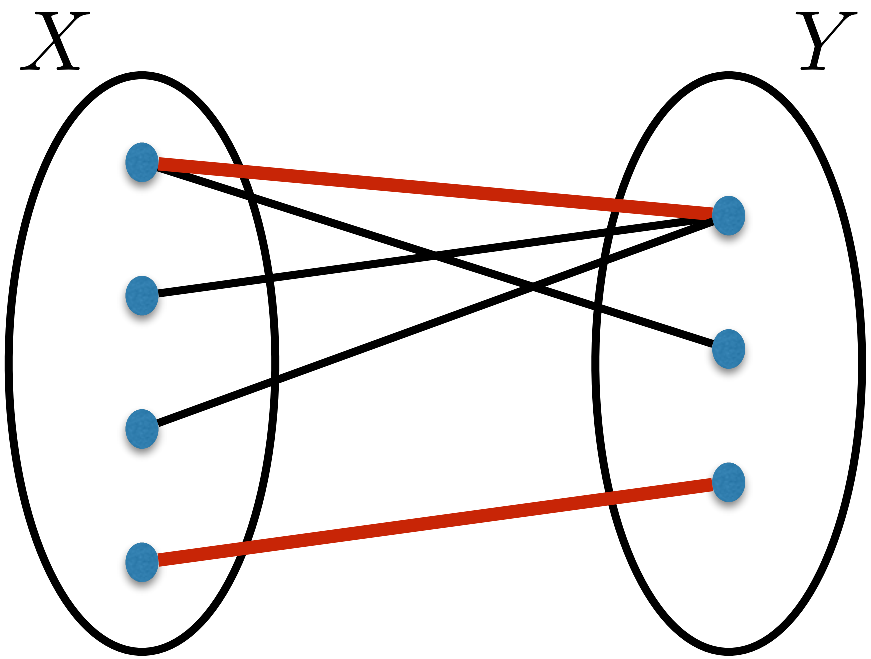


Maximum matching: a matching with largest number of edges (among all possible matchings).

Matchings in bipartite graphs

Often, we are interested in finding a **matching** in a bipartite graph

maximal matching



Cannot add more edges.

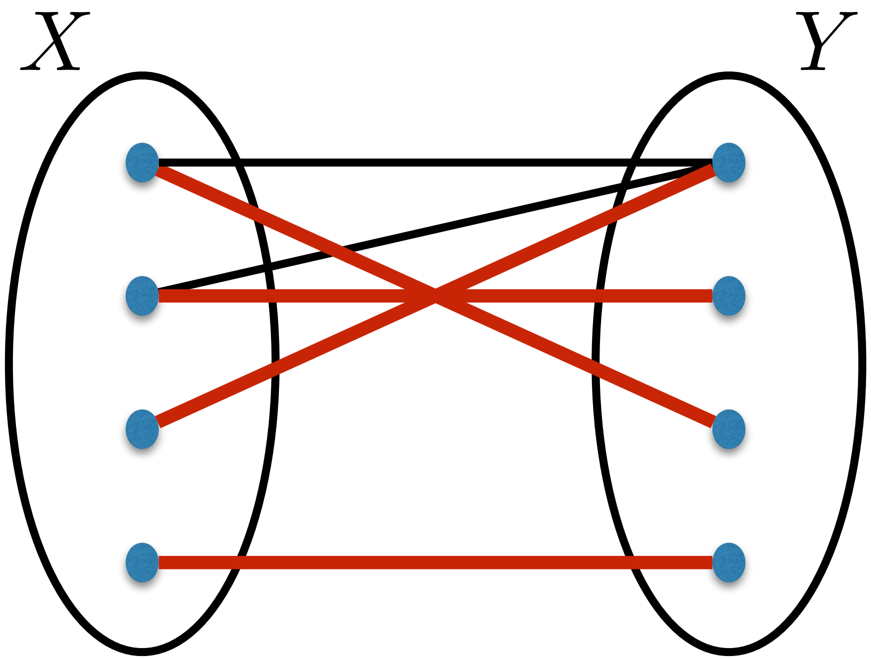
"*local optimum*"

Maximal matching: a matching which cannot contain any more edges.

Matchings in bipartite graphs

Often, we are interested in finding a **matching** in a bipartite graph

perfect matching

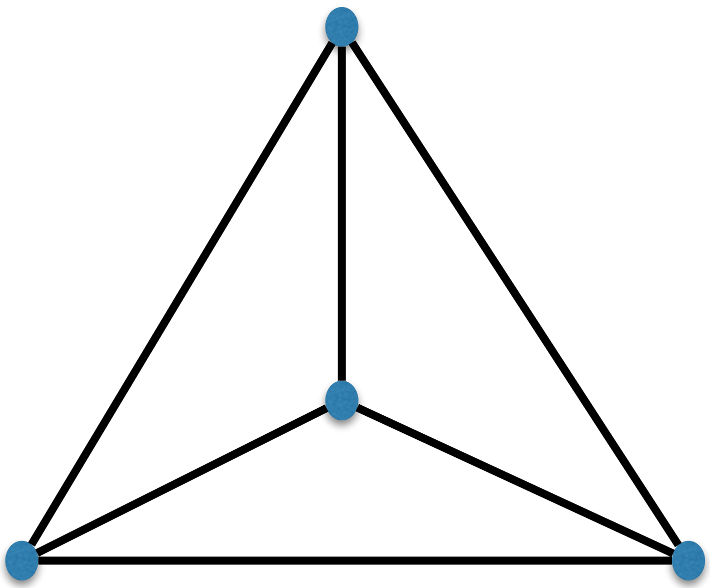


a necessary condition for perfect matching: $|X| = |Y|$

Perfect matching: a matching that covers all vertices.

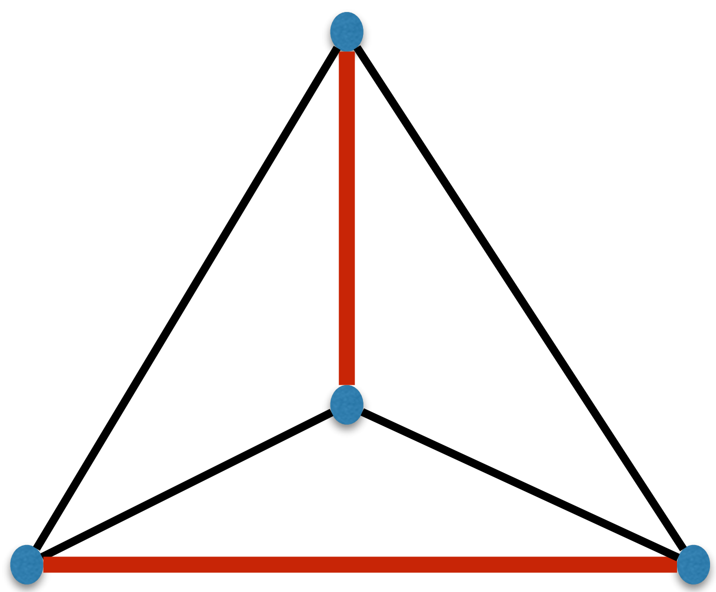
Important Note

We can define matchings for non-bipartite graphs as well.



Important Note

We can define matchings for non-bipartite graphs as well.



Maximum matching problem

The problem we want to solve is:

Maximum matching problem

Input: A graph $G = (V, E)$.

Output: A maximum matching in G.

Bipartite maximum matching problem

Actually, we want to solve the following restriction:

Bipartite maximum matching problem

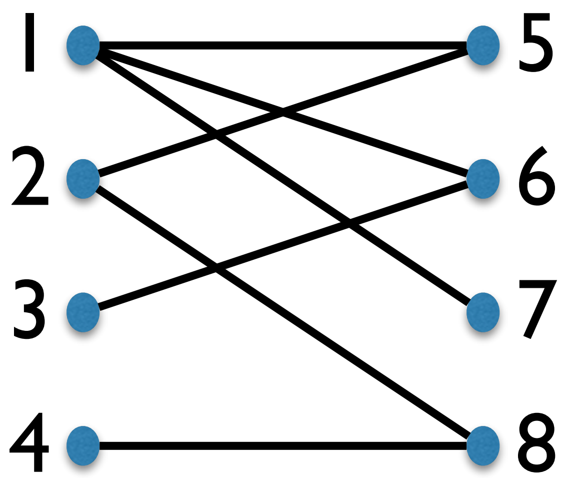
Input: A *bipartite* graph $G = (X, Y, E)$.

Output: A maximum matching in G.

Bipartite maximum matching problem

A good first attempt:

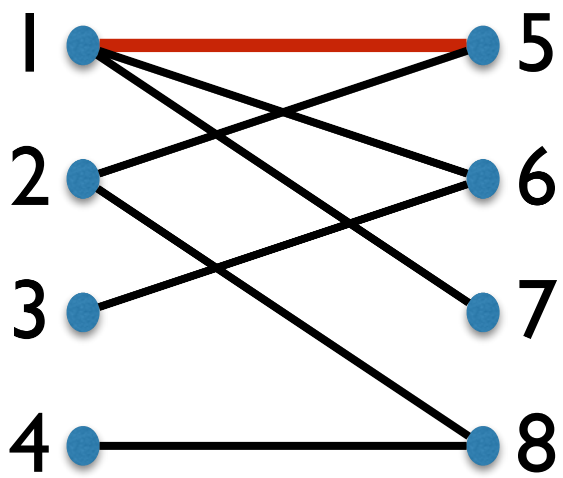
What if we picked edges greedily?



Bipartite maximum matching problem

A good first attempt:

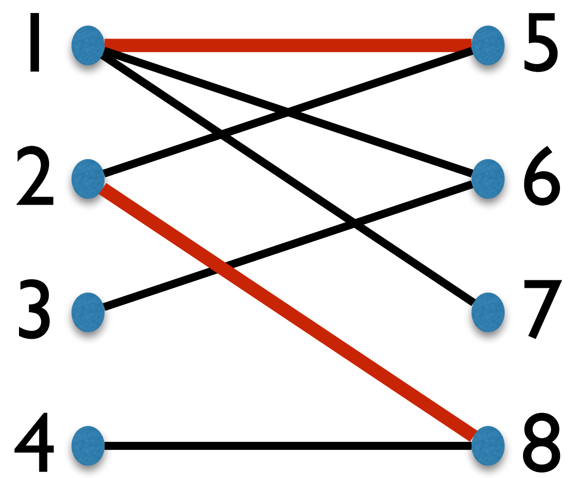
What if we picked edges greedily?



Bipartite maximum matching problem

A good first attempt:

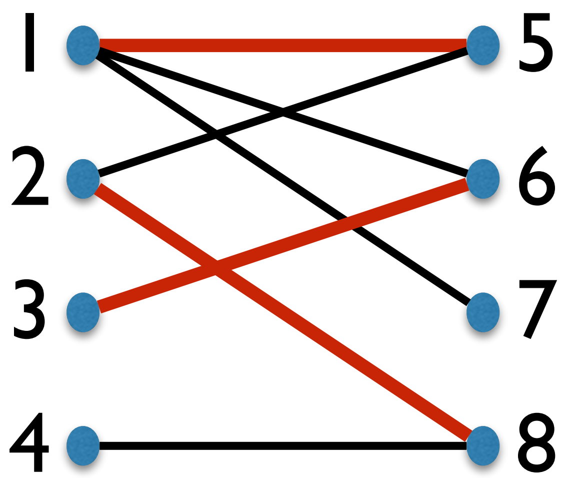
What if we picked edges greedily?



Bipartite maximum matching problem

A good first attempt:

What if we picked edges greedily?



maximal matching

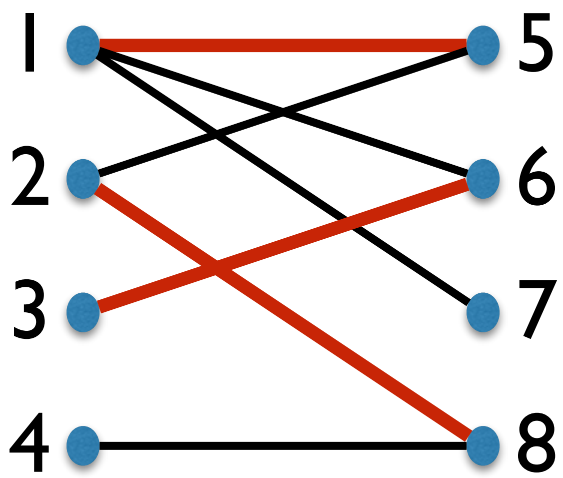
but not maximum

Is there a way to get out of this *local optimum*?

Bipartite maximum matching problem

A good first attempt:

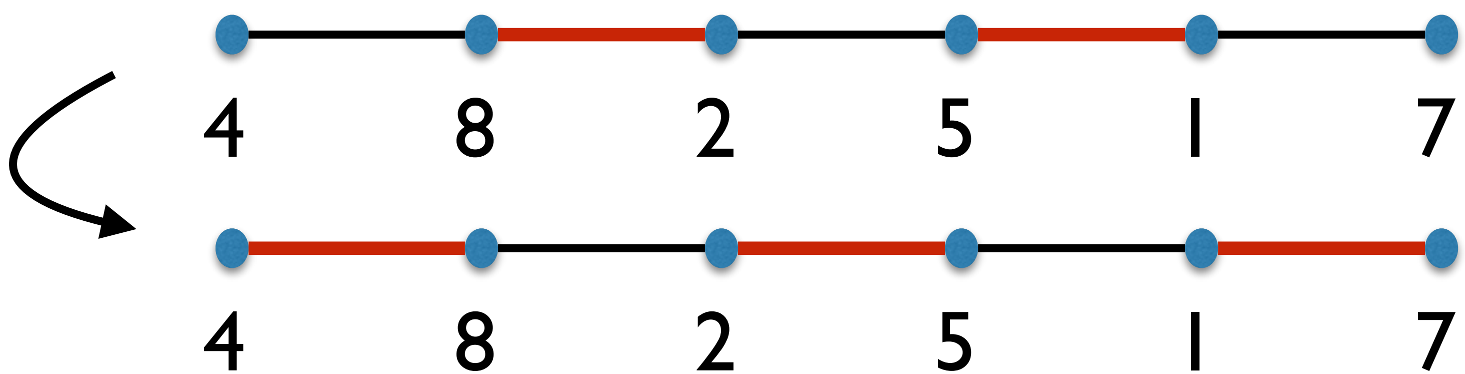
What if we picked edges greedily?



maximal matching

but not maximum

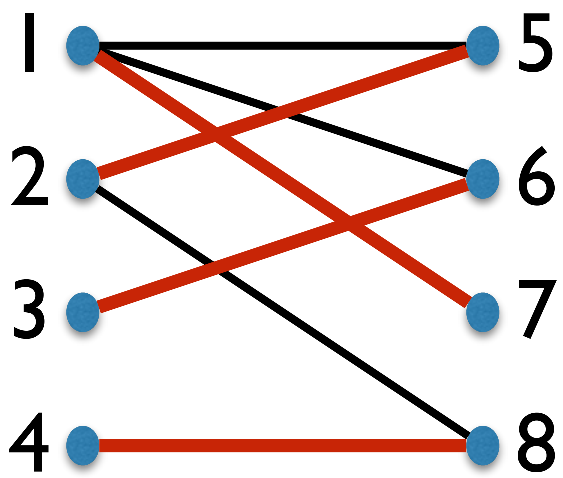
Consider the following path:



Bipartite maximum matching problem

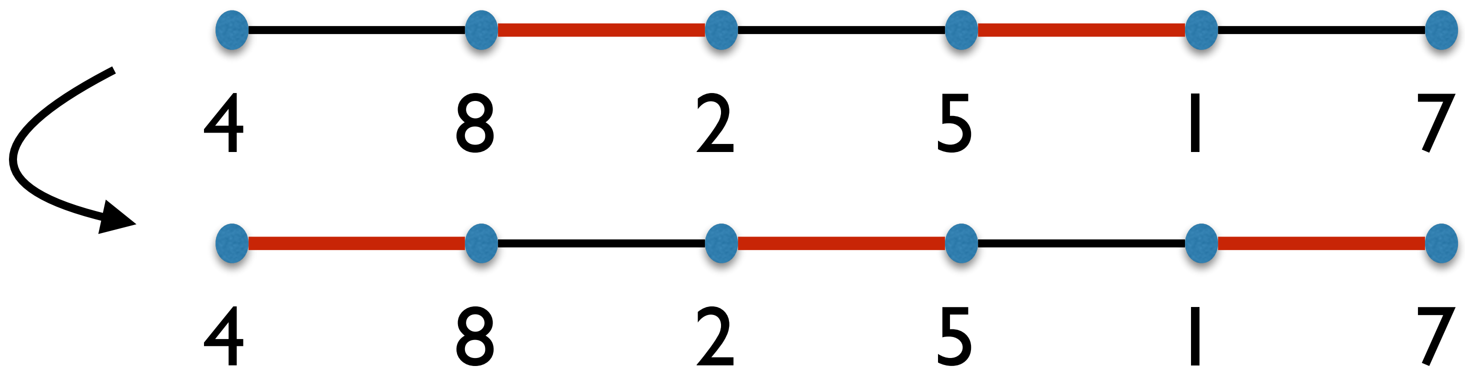
A good first attempt:

What if we picked edges greedily?



now maximum

Consider the following path:

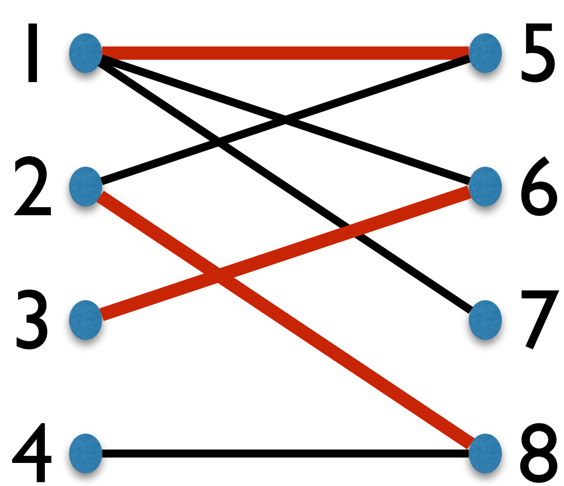


Augmenting paths

Let **M** be some matching.

An ***augmenting path*** with respect to **M** is a path in **G** such that:

- the edges in the path alternate between being in **M** and not being in **M**
- the first and last vertices are **not** matched by **M**

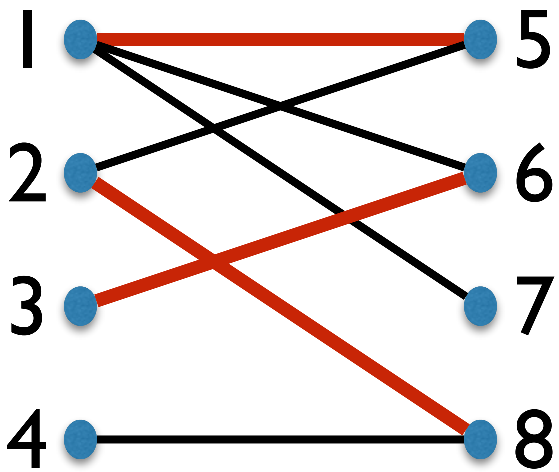


matching = red edges

Augmenting path:

4-8-2-5-1-7

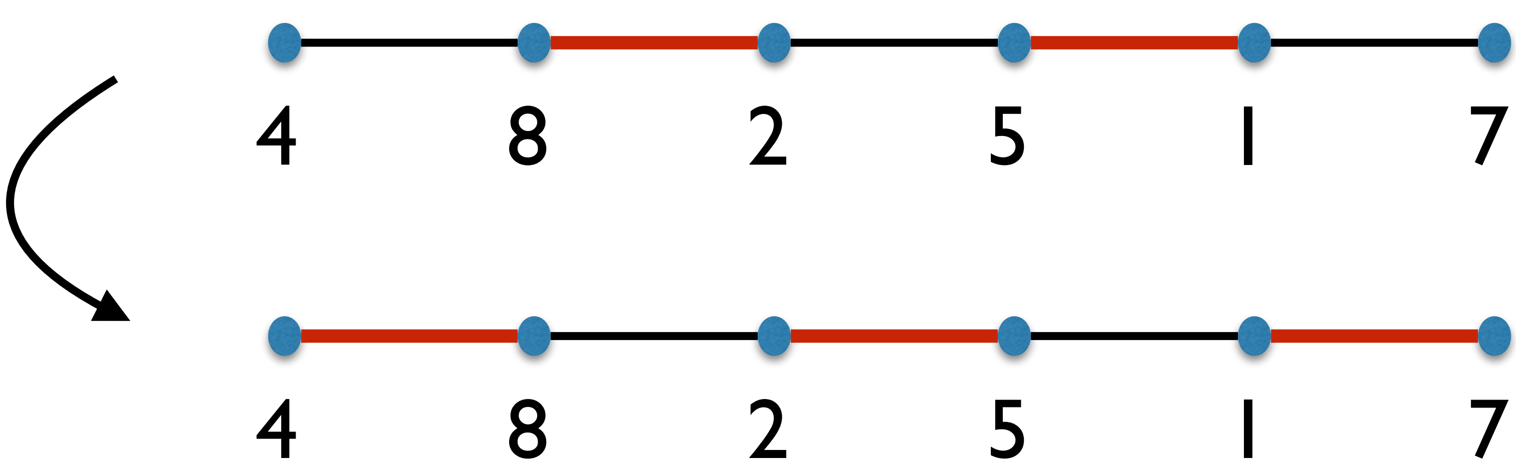
Augmenting paths



matching = red edges

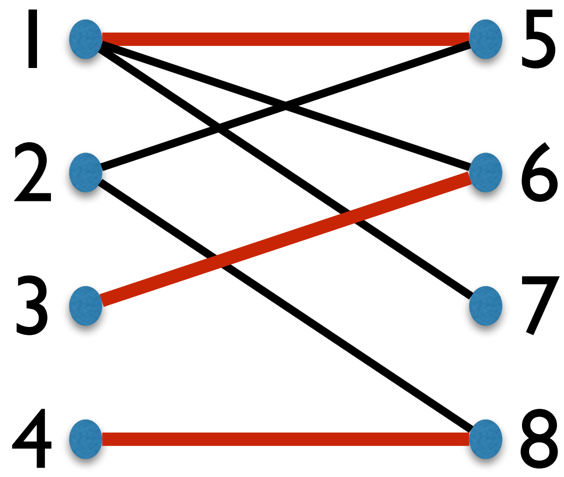
Augmenting path:

4-8-2-5-1-7



augmenting path $\Longrightarrow$ can obtain a bigger matching.

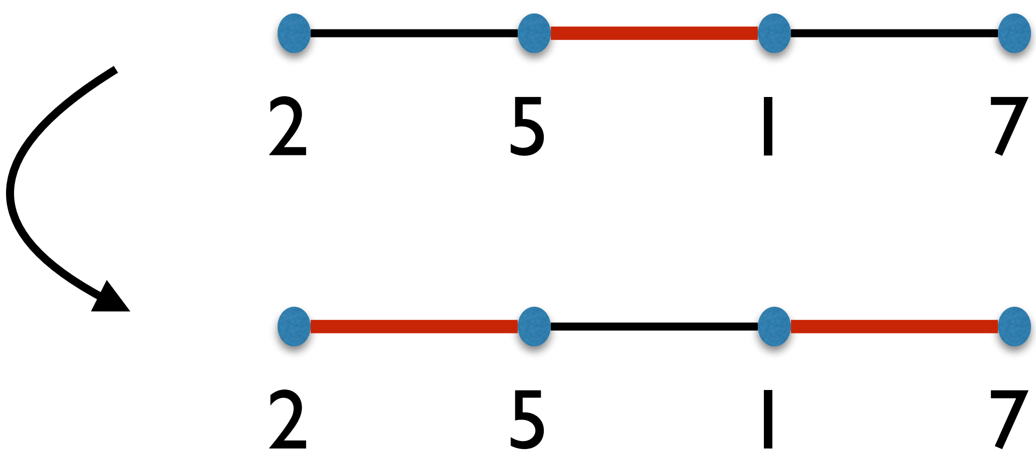
Augmenting paths



matching = red edges

Augmenting path:

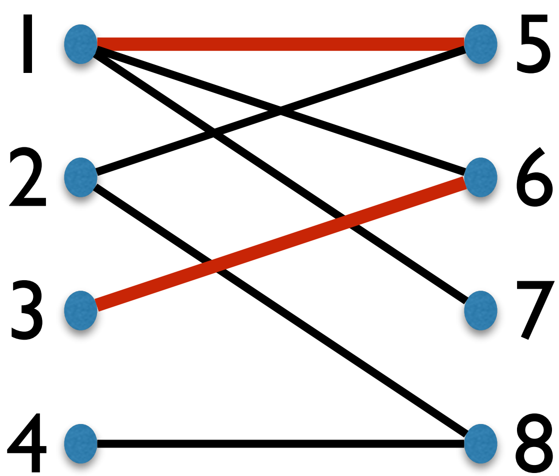
2-5-1-7



An augmenting path
need **not** contain
all the edges of the matching.

augmenting path $\Longrightarrow$ can obtain a bigger matching.

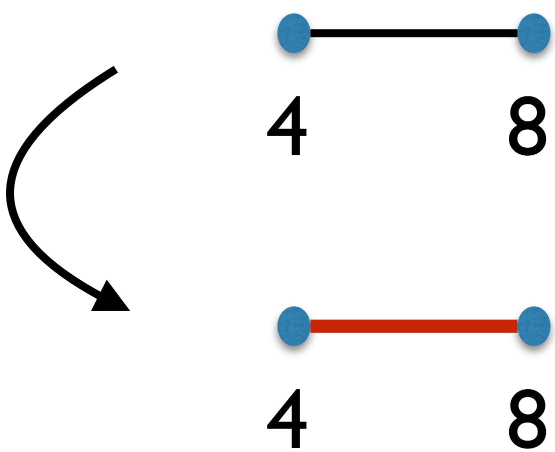
Augmenting paths



matching = red edges

Augmenting path:

4-8



An augmenting path
need **not** contain
any of the edges of the matching.

augmenting path $\Longrightarrow$ can obtain a bigger matching.

Augmenting paths and maximum matchings

augmenting path $\Longrightarrow$ can obtain a bigger matching.

In fact, it turns out:

no augmenting path $\Longrightarrow$ maximum matching.

Theorem:

A matching **M** is maximum **if and only if** there is **no** augmenting path with respect to **M**.

Augmenting paths and maximum matchings

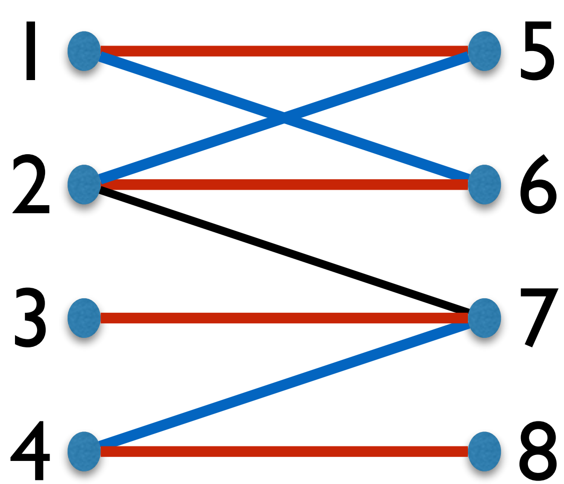
Proof:

If there is an augmenting path with respect to **M**, we saw that **M** is not maximum.

Want to show:

If **M** is not maximum, then there is an augmenting path.

Let **M*** be a maximum matching. |**M***| > |**M**|.

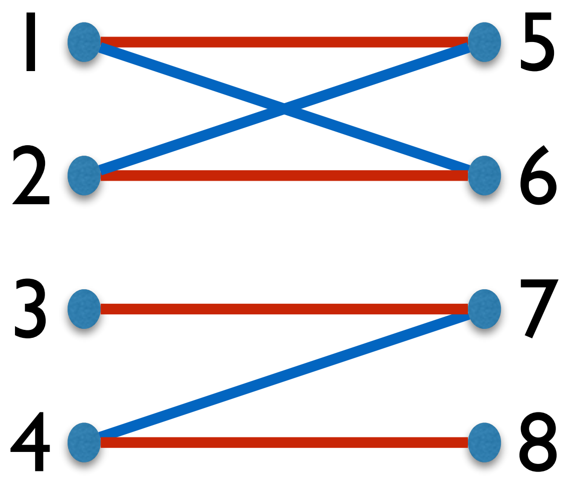


Let **S** be the set of edges contained in **M*** or **M** but not both.

S = (**M*** ∪ **M**) - (**M** ∩ **M***)

Augmenting paths and maximum matchings

Proof:



Let **S** be the set of edges contained in **M*** or **M** but not both.

$$S = (M^* \cup M) - (M \cap M^*)$$

(will find an augmenting path in **S**)

What does **S** look like?

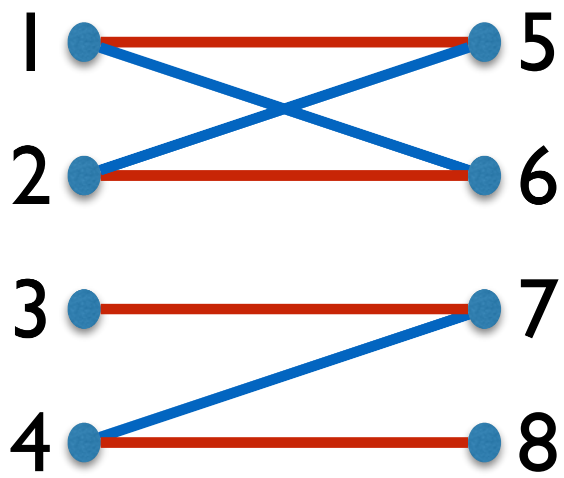
Each vertex has degree at most 2. (why?)

So **S** is a collection of **cycles** and **paths**. (exercise)

The edges alternate **red** and **blue**.

Augmenting paths and maximum matchings

Proof:



Let **S** be the set of edges contained in **M*** or **M** but not both.

S = (**M*** ∪ **M**) - (**M** ∩ **M***)

So **S** is a collection of **cycles** and **paths**. (exercise)

The edges alternate **red** and **blue**.

\# red > # blue in **S**

\# red = # blue in **cycles**

So ∃ a **path** with # red > # blue.

This is an ***augmenting path*** with respect to **M**. □

Algorithm to find maximum matching

Theorem:

A matching **M** is maximum **if and only if** there is **no** augmenting path with respect to **M**.

Algorithm:

- Start with a single edge as your matching **M**.
- Repeat until there is no augmenting path w.r.t. **M**:
  - Find an augmenting path with respect to **M**.
  - Update **M** according to the augmenting path.

OK, but how do you find an augmenting path?

Not too bad for bipartite graphs (attend recitation).

Today's Menu

- Graph search: DFS

- Minimum spanning tree

- Maximum matching