Today’s Menu

- Graph search: DFS

- Minimum spanning tree

- Maximum matching
Graph Search
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?
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\((G, z)\).
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 $\text{DFS}(G, z)$.

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 $\text{DFS}(G, v)$.
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 $\text{DFS}(G, z)$.

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

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

run $\text{DFS}(G, v)$.

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

Running time: $O(n + m)$ (exercise)
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
Boruvka’s pal Jindrich Saxel was working for Zapadomoravské elektrárny (the West Moravian Power Plant company).

Saxel asked:
What is the least cost way to electrify southwest Moravia?
Remember the CS life lesson

If your problem has a graph, great. If not, try to make it have a 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.
Convenient Assumption:

Edges have distinct costs.

Exercise: 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

\[ V' = \text{vertices connected so far} \]

\[ E' = \text{edges in the solution so far} \]
V' = \{a\}  \quad \text{(start with an arbitrary node)}

E' = \{\}
\( V' = \{a, b\} \)

\( E' = \{\{a, b\}\} \)
\[ V' = \{a, b, g\} \]

\[ E' = \{\{a, b\}, \{b, g\}\} \]
$V' = \{a, b, g, f\}$

$E' = \{\{a, b\}, \{b, g\}, \{g, f\}\}$
Jarník-Prim Algorithm

\[ V' = \{a, b, g, f, e\} \]

\[ E' = \\{(a, b), (b, g), (g, f), (g, e)\} \]
\[ V' = \{a, b, g, f, e, d\} \]

\[ E' = \{\{a, b\}, \{b, g\}, \{g, f\}, \{g, e\}, \{e, d\}\} \]
V' = \{a, b, g, f, e, d, c\}

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

Total cost: 42
Jarník-Prim Algorithm

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

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

$E' = \emptyset$

While $V' \neq V$:

- 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 $E'$
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.
How do we know the algorithm is correct?

**Lemma**: (MST Cut Property)

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

Let $V' \subset V$ \hspace{1cm} ($V' \neq \emptyset$, $V' \neq V$).

Let $e = \{u, v\}$ be the cheapest edge with $u \in V'$, $v \not\in V'$.

Then the MST **must** contain this edge $e$. 
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. 
- clearly has smaller cost
- clearly has $n-1$ edges
- argue it must be connected

\( \text{Contradiction} \)
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?
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
1986: Gabow, Galil, T. Spencer, Tarjan improved the alg.

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

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

Incomprehensibly small!
2002: Pettie & Ramachandran gave a new algorithm. They proved its running 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.
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
If your problem has a graph, great. If not, try to make it have a graph!
A bipartite graph $G = (V, E)$ 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?
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:

\[ X = \text{machines}, \quad Y = \text{jobs} \]
An edge \( \{x, y\} \) means \( x \) is capable of doing \( y \).

\[ X = \text{professors}, \quad Y = \text{courses} \]
An edge \( \{x, y\} \) means \( x \) can teach \( y \).

\[ X = \text{students}, \quad Y = \text{internship jobs} \]
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.
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.
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

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: 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: a matching which cannot contain any more edges.
Matchings in bipartite graphs

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

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.
The problem we want to solve is:

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

**Output:** A maximum matching in \( G \).
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?

```
1 -- 2 -- 3 -- 4
    |     |
    5 -- 6 -- 7
    |     |
    8
```
Bipartite maximum matching problem

A good first attempt:

What if we picked edges greedily?

1
2
3
4
5
6
7
8
Bipartite maximum matching problem

A good first attempt:

What if we picked edges *greedily*?

![Graph showing a bipartite matching example](image-url)
A good first attempt:

What if we picked edges greedily?

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?

Consider the following path:
Bipartite maximum matching problem

A good first attempt:

What if we picked edges greedily?

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

Augmenting path: 4-8-2-5-1-7

matching = red edges

augmenting path  \implies \text{can obtain a bigger matching.}
Augmenting paths

Augmenting path: 2-5-1-7

matching = red edges

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

augmenting path ⇒ 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 \( \Rightarrow \) can obtain a bigger matching.
Augmenting paths and maximum matchings

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

In fact, it turns out:

no augmenting path $\implies$ 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.

\[ S = (M^* \cup M) - (M \cap M^*) \]
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.

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

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 $\exists$ 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