Algorithms

Dynamic Programming

LCS (Longest Common Subsequence)

abcddeb
bdabaed

$$Opt(n,m) = \begin{cases} A_n = B_m & 1 + Opt(n-1, m-1) \\ A_n \neq B_m & \max\begin{cases} Opt(n, m-1) \\ Opt(n-1, m) \end{cases} \end{cases}$$

d	0							
b	0							
a	0							
c	0	1	2					
b	0	1	1					
a	0	0	0					
ϕ	0	0	0	0	0	0	0	0
	ϕ	b	c	d	a	b	c	a

Ramsy Theory

Erdes and Szekeresh proved that given a permutation of $\{1, \dots, n\}$, at least one increasing or decreasing substring exists with length $> \sqrt{n}$.

For example, observe the following permutation of $\{1, \dots, 10\}$:

10 5 8 1 4 3 6 2 7 9

$(1,1)(1,2)(2,2)(1,3)$ …

The pairs present the number of the longest increasing substring and longest decreasing substring till that point (accordingly).
Since every number raises the length of one of the substrings by one (either the increasing or decreasing) each lengths pair is unique
Due to this fact, one of the numbers must be at least $\sqrt{n}$

Bandwidth Problem

The bandwidth problem is an NP hard problem.

Definition: Given a symmetrical matrix, is there a way to switch the columns s.t. the "bandwidth" of the matrix is smaller than a number received as input.
Another variant is to find the column switch that produces the smallest bandwidth.

The bandwidth of a matrix is defined as the largest difference between numbering of end points.

0	1		1			
1	0	1				
	1	0			1	
1			0			
				0		
		1			0	
						0

Special Cases

If the graph is a line, it's very easy finding the smallest bandwidth (it's 1 – need to align the vertices as a line). In case of "caterpillar" graphs however, the problem is already NP hard. ("Caterpillars" are graphs that consist of a line with occasional side lines)

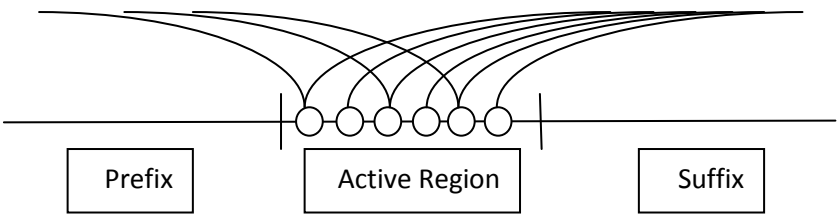
Parameterized Complexity

Given a parameter k, does the graph have an arrangement with bandwidth $\leq$ k?

One idea, is to remember the last 6 vertices (including their order!), and simply remember the vertices before and after (without order) so that we know not to repeat a vertex twice. A trivial solution would be to simply remember them all. But while it's feasible to remember the last 6 vertices, remembering the vertices we've passed takes up $\binom{n}{j-k}$ where j is the step index. In case of $j = \frac{n}{2}$ we would need time exponential in n.

A breakthrough came in the early 80s by Saxe. The main idea is to remember the vertices we've passed **implicitly**.
In order to do so, here are a few observations:

1) Wlog, G is connected (otherwise we can find the connected sub-graphs in linear time and execute the algorithm independently on each of them, concatenating the results at the end)
2) Maximal degree of the graph (if has bandwidth $\leq k$) is $\leq 2k$.



Due to these observations, a vertex is in the prefix, if it is connected to the active region without using dangling edges.
A vertex in suffix must use dangling edges to connect to active region.

--------end of lesson 1

Color Coding

$A(x)$

The measure of complexity would be the expected running time (over random coin tosses of the algorithm) for the worst possible input.

Monte Carlo Algorithms

There is no guarantee the solution is correct. It is probably correct.

Las Vegas Algorithms

You want the answer to be always correct. But the running time might differ.

Hamiltonian path

A simple path that visits all verties of a graph.

Logest Simple Path

TODO: Draw vertices

We are looking for the longest simple path from some vertex v to some vertex u.
Simple path means we are not allowed to repeat a vertex twice.
This problem is NP-Hard.

Given a parameter k and find a simple path of length k.
A trivial algorithm would work in time n^k.

Algorithm 1

- Take a random permutation over the vertices.
- Remove every "backward" edge
- Find the longest path in remaining graph using dynamic programming

TODO: Draw linear numbering

For some permutation, some of the edges go forward, and some go forward. After removing backward edges, we get a DAG.
For each vertex v from left to right, record length of longest path that ends at v.

Suppose the graph contains a path of length k.
What is the probability all edges of the path survived the first step?

The vertices have $k!$ possible permutations, and only one permutation is good for us. So with probability $\frac{1}{k!}$ we will not drop the edges of the path and the algorithm succeeds.

This is not too good! If $k = 10$ it is not too good!

So what do we do? Keep running it until it succeeds.
The probability of a failure is $\left(1 - \left(\frac{1}{k!}\right)\right)$, but if we run it more than once it becomes:
$\left(1 - \left(\frac{1}{k!}\right)\right)^{ck!} \approx \left(\frac{1}{e}\right)^{c}$ expected running.

Expected running time $O(n^2 k!)$.
You can approximate $k! = \approx \frac{k^k}{e^k} \leq n, \quad k \cong \frac{\log n}{\log \log n}$

Algorithm 2

- Color the vertices of the graph at random with k colors.
- Use dynamic programming to find a colorful path.

	1	2	…	k
$v_1 S_1$				
$v_1 S_2$		$v_i + S_i$		
⋮				
$v_n S^{2^k}$				

S is the set of colors we used so far.

Probability that all vertices on path of length k have different colors: $\frac{k!}{k^k} \approx \frac{k^k}{e^k k^k} = \frac{1}{e^k}$

If $k \cong \log n$ then the success would be $\frac{1}{n^{1\ldots}}$.

Tournaments

A tournament is a directed graph in which for every pair of vertices there exists an edge (in one direction). An edge exists if the player (represented by the vertex of origin) won another player (represented by the second vertex).
We try to find the most consistent ranking of the players.

We want to find a linear order that minimizes the number of upsets.
An upset is an edge going in the other direction of another edge (tour?).

A Feedback Arc Set in a Tournament (FAST) is all the arcs that go backwards.

Another motivation

Suppose you want to open a Meta search engine. How can you aggregate the answers given from the different engines?

TODO: Draw the ranking…

The problem is NP-Hard.

k-FAST

Find a linear alignment in which there are at most k edges going backwards.

$$0 \leq k \leq \frac{\binom{n}{2}}{2}$$

We will describe an algorithm that is good for $n \leq k \leq a(n^2)$

Suppose there are t subparts that have the optimal solution and we want to merge them into a bigger solution.

If the graph G is partitioned into t parts, and we are given the "true" order within each part, merging the parts takes time $n^{O(t)}$ (O just for the bookkeeping).

Color Coding

Pick t $(a\ function\ of\ k)$ and color the vertices independently with t colors.
We want that: For the minimum feedback arc set F, every arc of F has endpoints of distinct colors. Denote it as "F is colorful".
Why is this desirable?
If F is colorful, then for every color class we do know the true order.
We use the fact that this is a tournament! The order is unique due to the fact that between every two vertices there is an arc (in some direction).

Lets look at two extreme cases:

1) $t > 2k$ - the probability the second vertex gets the first color then the probability is $\frac{k}{t} < 2$. However, the runtime would be $n^{O(k)}$
2) $t < \sqrt{k}$ - not too good! Intuition: You have a $\sqrt{k}$ vertices where the direction of the arcs is essentially random. So you can create a graph that has very bad chances (didn't provide a real proof).

If $t > 8\sqrt{k}$ then with good probability – F is colorful. The probability behaves something like: $\frac{1}{n^t}$. Then the expected running time is n^t, and with the previous running time the total running time is still $n^{O(t)}$.

$$\rightarrow \left(1-\frac{1}{t}\right)^k \approx \left(\frac{1}{e}\right)^{\frac{k}{t}} \rightarrow t \cong \sqrt{k} \quad \left(\frac{1}{e}\right)^{\sqrt{k}} \quad n^{O(\sqrt{k})}$$

Lemma: Every graph with k edges has an ordering of its vertices in which no vertex has more than $\sqrt{2k}$ forward edges.
Why is this true? Pick an arbitrary graph with k edges, and start arranging its edges with the lowest degree first. So each time we put the vertex of lower degree of the suffix.

$\deg i$ =degree of v_i
d_i =number of forward edges of v_i
$s = n - i$

$d \leq s$ - Because a vertex can have one forward edge to every vertex to its right
$d \leq \deg i \rightarrow d \cdot s \leq \deg i \cdot s \leq \sum_{j>i} \deg j \leq \sum_j \deg j = 2K$
Therefore, $d^2 \leq 2K \rightarrow d \leq \sqrt{2K}$.

The chance of failure for some vertex is bound by $\frac{d_i}{t}$ (each neighbor has a chance of having the wrong color). Therefore, the chance of success is at least $1 - \frac{d_i}{t}$.
However, since d_i is blocked by $\sqrt{2K}$, in order to have a valid expression, t must be larger than $\sqrt{2K}$.

$$\left(1-\frac{d_i}{t}\right) \cong e^{-\frac{d_i}{t}}$$

$$\prod_i \left(1-\frac{d_i}{t}\right) \leq \prod e^{-\frac{d_i}{t}} = e^{\frac{\sum d_i}{t}} = e^{-\frac{2_k}{t}} = e^{-\sqrt{k}}$$

-------end of lesson 2

Repeating last week's lesson:
In every graph with m edges, if we color its vertices at random with $\sqrt{8m}$ colors, then w.p. $\geq (2e)^{-\sqrt{8m}}$ the coloring is proper.

Assume we color the graph with t colors.
TODO: Draw the example that shows the coloring is dependent.

Inductive Coloring

Given an arbitrary graph G, you find the vertex with the least degree in the graph. Then remove that vertex. Then find the next one with the least degree and so on…
This determines an ordering of the vertices:
$v_1, v_2, \dots, v_n$
v_i - has the least degree in $G(v_i, \dots, v_n)$.
Then we start by coloring the last vertex. Each time we color the vertex according to the vertices to its right (so it will be proper).

If d is the maximum right degree of any vertex, then inductive coloring uses at most $d + 1$ colors.

In every planar graph, there is some vertex of degree at most 5.

Corollary: Planar graphs can be colored (easily) with 6 colors.

Every graph with m edges, has an ordering of the vertices in which all right degrees are at most $\sqrt{2m}$.

So we need at least $\sqrt{2m} + 1$ colors. But we don't want our chances of success to be too low, so we use twice that number of colors - $2\sqrt{2m}$.

Let the list of degrees to the right - $d_1, d_2, \dots, d_n$.

$$\sum_{i=1}^{n} d_i = m$$

What is the probability of violating the proper coloring for vertex i?
Number of colors left for vertex i - $\frac{t-d_i}{t}$

$$\prod_i \left(\frac{t-d_i}{t}\right) = \prod_i \left(1 - \frac{d_i}{t}\right)$$

But we know $\frac{d_j}{t} \leq \frac{\sqrt{2m}}{\sqrt{8m}} = \frac{1}{2}$
It's easier to evaluate sums than products.

Suppose $\frac{d_j}{t} = \frac{1}{k}$ (a very small number)

Why is this true:

$$1 - \frac{1}{k} \geq 2^{-\frac{2}{k}}$$

Let's raise both sides:

$$\left(1 - \frac{1}{k}\right)^{\frac{k}{2}} \geq \left(2^{-\frac{2}{k}}\right)^{\frac{k}{2}} = \frac{1}{2}$$

As long as $\frac{1}{k} < \frac{1}{2}$, each power only chops off less than half of what remains meaning the left side won't for below $\frac{1}{2}$. So the inequality is true.

So, back to the original formula:

$$\prod_i 2^{-\frac{2d_i}{t}} = 2^{-\Sigma\frac{d_i}{t}} = 2^{\frac{m}{t}}$$

Maximum weight independent set in a tree

Given a tree. In which, each vertex has a non-negative weight.
We need to select an independent set in the tree.
In graphs this problem is NP hard, but in trees we can do it in polynomial time.

TODO: Add a drawing of a tree

We pick an arbitrary vertex r as the root.
We think of the vertices as being directed "away" from the root.
Given a vertex v, denote by $T(v)$ as the set of vertices reachable from v (in the direction from the root).

So for each vertex, we will keep two variables:
$W^+(v)$ – is the maximum weight independent set in the tree $T(v)$, that contain v.
$W^-(v)$ – is the maximum weight independent set in the tree $T(v)$, that **do not** contain v.

Need to find $W^+(r), W^-(r)$ and the answer is the largest of the two.

The initialization is trivial. $W^+(v) = w(v)$ and $W^-(v) = 0$.

For every leaf l of T, determine $W^+(l) = w(l), W^-(l) = 0$ and remove it from the tree.
Pick a leaf of the remaining tree, with children $u_1, \dots, u_k$.

$$W^+(v) = w(v) + \sum_{i=1}^{k} W^-(u_i)$$

$$W^-(v) = 0 + \sum_{i=1}^{k} \max\{W^-(u_i), W^+(u_i)\}$$

This algorithm can also work on graphs that are slightly different than trees. (do private calculations for the non-compatible parts).

Can we have a theorem of when the graph is just a bit different than a tree and still the algorithm can run in polynomial time?

Tree Decomposition of a Graph

We have some graph G, and we want to represent it as a tree T.
Each node of the tree T would represent a **set** of vertices of graph G.

Every node of the tree is labeled by a set of vertices of the original graph G.
Denote such sets as **bags**.
We also have the following constraints:

1) Moreover, the union of all these sets is all vertices of G.
2) Every edge $\langle v_i, v_j \rangle$ in G is in some bag.
3) For every vertex $v \in G$, the bags containing v are connected in T. Meaning, that they are connected with vertices that contain v, and do not have to pass through vertices that do not contain v.

Given two bags - B_1 and B_2 $s.t.$ $B_1 \subseteq B_2$, they are connected through a single path (because it's a tree). This path must contain all vertices of B_1.

Tree Width of a Tree Decomposition

The Tree width of T is p if the maximal bag size is $p + 1$.
Tree width of G is the smallest p for which there is a tree decomposition of tree width p.
Intuitively – a graph is closer to a tree when its p is smaller.

Properties regarding Tree width of graphs

Lemma: If G has tree width p, then G has a vertex of degree at most p.

Observe a tree decomposition of G.
It has some leaf v. The bag of this leaf has $p + 1$ vertices at most. It has only one neighbor (since it's a leaf).
Since no bag contains another bag, there is some vertex that exists in its neighbor that is not in v. TODO: Copy the rest

Fact: $TW(G \backslash v) \leq TW(G)$. Since we can always take the original tree decomposition and remove the vertex.

Corrolery: If G has tree width p then G can be properly colored $p + 1$ colors.
Indicates that if G is a tree, its tree width is 1 (since a tree is a bi-part graph and therefore can be colored by 2 colors).

A graph with no edges has tree width 0, since you can have each bag as a singleton of a vertex.

A complete graph on n vertices has $TW = n - 1$ (one bad holding all vertices)

A compete graph missing an edge – $\langle u, v \rangle$ has $TW = n - 1$:
We can construct two bags – $G - u$ and $G - v$ and connect them.

Theorem: G has $TW = 1$ iff G is a tree.

Assume G has $TW = 1$. Has a vertex v of degree 1. Remove v. The graph is still a tree! So we can continue…
We assume the graph is connected. But it doesn't have a cycled! If it had a cycle, we would have a contradiction. A connected graph with no cycles is a tree.

Assume G is a tree. Lets construct the decomposition as follows:
Lets define each vertex as a bag with two vertices. An edge is connected to all edges that are other edges of the contained vertices.

Series-Parallel graphs

TODO: Draw resistors…

Series-Parallel graphs are exactly all graphs with $TW = 2$.

Start from isolated vertices.

1) Add a vertex in series.
2) Add a self loop
3) Add an edge in parallel to an existing edge
4) Subdivide an edge

Series-Parallel⇒ $TW(2)$.

TODO: Draw

------ end of lesson 3

Graph Minor

A graph H is a minor of graph G if H can be obtained from G by:

(1) Removing vertices
(2) Removing edges
(3) Contracting edges

TODO: Draw graph

Definition: A sub-graph is a graph generated by removing edges and vetices
Definition: An **induced** sub-graph is a graph with a subset of the vertices that includes all remaining edges.

Contracting an edge is joining the two vertices of the edge together, such that the new vertex has edges to all the vertices the original vertices had.

A graph is planar if and only if it does not contain neither K_5 nor $K_{3,3}$ as a minor.

TODO: Draw the forbidden graphs

Definition: A graph is a tree or a forest if it doesn't contain a cycle⇔A clique of 3 vertices as a minor.

A graph is Series parallel, if it does not contain a k_4 as a minor.

Theorem: There are planar graphs on n vertices with tree width $\Omega(\sqrt{n})$

Let's look at an $\sqrt{n}$ by $\sqrt{n}$ grid

```
·  —  ·  —  ·  —  ·
|     |     |     |
·  —  ·  —  ·  —  ·
|     |     |     |
·  —  ·  —  ·  —  ·
|     |     |     |
·  —  ·  —  ·  —  ·
```

We will construct $\sqrt{n} - 1$ bags.

A bag i contains columns i and $i + 1$

Bags:

```
·  —  ·  —  ·  …
```

This is a tree decomposition by all the properties.

Vertex Separators

Vertex Separators: A set of S of vertices in a graph G is a vertex separator if removing S from G, the graph G decomposes into connected components of size at most $\frac{2n}{3}$

TODO: Draw schematic picture of a graph

It means we can partition the connected components into two groups, none of them with more than $\frac{2n}{3}$ vertices.

Every tree has a separator of size 1

Let T be a tree. Pick an arbitrary root r.
$T(v)$ =the size of the sub-tree of v (according to r).
$T(r) = n$.
All leaves have size 1. So $\exists v$ with $T(v) > \frac{2}{3}n$ and $T(u) \leq \frac{2}{3}n$ for all children u of v.
That v is the separator.

If a graph G has tree width p, then it has a separator of size $p + 1$.

My summary:
Let D be some tree decomposition.
Each bag has at most $p + 1$ vertices. We can now find the separator of D and consider its $p + 1$ vertices as the separator of the graph G.
Note that when we calculate $T(v)$ for some $v \in D$, we should count the number of vertices **inside** the bags below it (not the number of bags).

His summary:
Consider a tree decomposition T of width p.
Let r serve as its root. And orient edges of T away from r.
Pick S to be the lowest bag whose sub-tree contains more than $\frac{2}{3}n$ vertices.

Every Separator in the $\sqrt{n}$ by $\sqrt{n}$ grid is of size at least $\frac{\sqrt{n}}{6}$

Why is this so?
Let's assume otherwise. So there is such a separator.
Let S be a set of $\frac{\sqrt{n}}{6}$ vertices.
$\frac{5\sqrt{n}}{6}$ rows that do not contain a vertex from S.
Same for columns.
Let's ignore all vertices in the same row or column with a vertex from S.
So we ignore at most $\frac{\sqrt{n}}{6} \cdot \sqrt{n} + \frac{\sqrt{n}}{6} \cdot \sqrt{n} = \frac{n}{3}$ vertices.

Claim: All other vertices are in the same connected component. Since we can walk on the row and column freely (since no members of S share the same row and column). Therefore we have $\frac{2}{3}n$ connected components – A contradiction.

If G has tree width p, then G is colorable by $p+1$ colors.

Every planar graph is 5-colorable.
Proof: A planar graph has a vertex of degree at most 5.
We will use a variation of inductive coloring.
Pick the vertex with the smallest degree.
Assume we have a vertex with a degree 5. These 5 neighbors cannot form a clique! (since the graph is planar)
So one edge is missing. Contract the two vertices of that missing edge with the center vertex (the one with degree 5).

Fact: Contraction of edges maintains planarity. This is immediately seen when thinking of bringing the edges closer in the plain.

Now we color the new graph (after contraction). Then we give all "contracted" vertices the same color in the original graph (before contraction).
After the contraction we have degree 4, so we can color it with 5 colors.

Hadwiger: Every graph that does not contain K_t as a minor, can be colored with $t-1$ colors. (A generalization of the 4 colors theorem).

If a graph has tree width p, then we can find its chromatic number in time $O(f(p)\cdot n)$

Note: The number of colors is definitely between 1 and $p+1$ (we know it can be colored by $p+1$ colors… The question is whether it can be colored with less).

For a value of $t \leq p$, is G t-colorable?

Theorem (without proof): Given a graph of tree width p, a tree decomposition with tree width p can be found in time linear in $n \cdot f(p)$

TODO: Draw the tree decomposition

For each bag, we will keep a table of size t^{p+1} of all possible colorings of its vertices. For each entry in the table, we need to keep one bit that determines whether that coloring is legal with the bags below that bag.
We can easily do it for every leaf (0/1 depending on: "is this coloring is legal with sub-graph below bag).
A coloring is legal if the sub-tree can be colored such that there is no collision of assignment of colors.

A family F of graphs is minor-closed (or closed under taking minors) if whenever $G \in F$ and H is a minor of G, then $H \in F$.

Characterizing F:

1) By some other property
2) List all graphs in the family (only if the family is finite)
3) List all graphs not in F (if the complementary of the family is finite)
4) List all forbidden minors
5) List a minimal set of forbidden minors
   For planar graphs, this minimal set was K_5 and $K_{3,3}$
   A list of minors is a list of graphs: G_1, G_2 … such that no graph is a minor of the other.
   A conjecture by Wagner: Minor relation induces a "well quasi order"⇔No infinite antichain ⇔ A chain of graphs such that no graph is a minor of another graph.
   So this is always a finite property!!!
   The conjecture was proved by Robertson and Seymour.

----- End of lesson 4

Greedy Algorithms

When we try to solve a combinatorical problem, we have many options:

- Exhaustive Search
- Dynamic Programming

Now we introduce greedy algorithms as a new approach

Scheduling theory

Matroids

Interval Scheduling

There are jobs that need to be performed at certain times that take a certain time

$j_1 \quad s_1 - t_1$

$j_2 \quad s_2 - t_2$

$\vdots$

We can't schedule two jobs at the same time!

TODO: Draw intervals

We can represent the problem as a graph in which two intervals will share an edge if they intersect. Then we will look for the maximal independent set.
Such a graph is called interval graph.

There are classes of graphs in which we can solve Independent Set in polynomial time. One such family of graphs is Perfect graphs.

Algorithm: Sort intervals by earliest ending times and use a greedy algorithm to select the interval that ends sooner. Remove all its intersecting intervals and repeat until no more intervals remain.

Why does it work?
Proof:
Suppose the greedy algorithm picked some intervals $g_1, g_2, \dots$
Consider the optimal solution that has the largest common prefix with the greedy one:
$o_1, o_2, \dots$

Consider first index i such that $o_i \neq g_i$
Since g_i ends at the same time (or sooner) than o_i we can generate a new optimal solution that has g_i instead of o_i. Such a solution would still be optimal (same number of intervals) and is legal ⇒ there exists a solution with a larger common prefix, a contradiction!

Interval Scheduling 2

$J_1 \ : \ w_1 > 0, l_i > 0$

$J_1 \ : \ w_1 > 0, l_i > 0$
w determines how important the job is. l determines how much time would it take for a CPU to perform the job.

Penalty for job i given a particular schedule is the ending time of $i \times w_i$
Total penalty is $\sum_i p_s(i)$
Find schedule s than minimizes the total penalty.

$J_1 \dots J_i J_j \dots$
Let's flip some job:
$J_1, \dots J_j J_i \dots$
And suppose that by flipping the order grew.
The penalty for the prefix and the suffix stays the same!
So we should only consider what happens to the penalty from J_i, J_j that resulted the switch
$w_i(\cancel{t} + \cancel{l_i}) + w_j(\cancel{t} + l_i + \cancel{l_j})$
$w_j(\cancel{t} + \cancel{l_j}) + w_i(\cancel{t} + l_j + \cancel{l_i})$
$w_j l_i < w_i l_j \Rightarrow \frac{w_j}{l_j} < \frac{w_i}{l_i}$
So this suggests the following algorithm:
Schedule jobs in decreasing order of $\frac{w_i}{l_i}$
This is optimal.
Proof: Consider any other schedule which does not respect this order.
Then there must be some i in which $\frac{w_j}{l_j} < \frac{w_i}{l_i}$. Then we can reverse the order, and get a better scheduling – a contradiction!

General Definition of Greedy-Algorithm solvable problems – Matroids

$S = \{e_1, \dots, e_n\}$ (sometimes we have matroids in which the items represent edges)
F = a collection of subsets of S. (Short for "Feasible". Often called "independent sets").
We want F to be hereditary.
Hereditary - If $X \in F, Y \subset X \Rightarrow Y \in F$
And we also need a cost function:
$c: S \to R^+$

Find a set $X \in F$ with maximum cost. $\sum_{e_j \in X} c(e_j)$

Example: Given a graph G, the items are the vertices of G. F is the independent sets of G, $\forall x. c(x) = 1$.

$MIS(G)$ is NP hard.

Definition: A hereditary family F is a matroid, if and only if $\forall X, Y \in F$ if $|X| > |Y|$ then $\exists e \in X, e \notin Y$ such that $Y \cup \{e\} \in F$.

Proposition: All maximal sets in a matroid have the same cardinality.

Suppose $|X| > |Y|$, then there should be some item in X that we can add to Y to make it feasible, and therefore Y does not have the maximal cardinality.

All maximal sets is called "bases", and the cardinality of the maximal sets is called the "rank of the matroid".

We have all sorts of terms: Matroids, Independent Sets, Basis, Rank. How are they related?
Consider a matrix. The items are the columns of the matrix.
The independent sets are columns that are linearly independent columns.
Note that it is hereditary. Because if a set of columns is linearly independent, then any subsets is also linearly independent.
The basis of the matrix is the largest set of columns that spans the space. And the rank is also the same as in linear algebra.

Theorem: For every hereditary family F, the greedy algorithm finds the maximum solution for **every** cost function $c \Leftrightarrow F$ is a matroid

Greedy Algorithm: Iteratively, add to the solution the item e_j with largest cost that still keeps the solution feasible.

Proof: Assume that F is not a matroid.
So (by definition) there are some sets X, Y such that $|X| > |Y|,\ X, Y \in F$
and $\forall e \in X \backslash Y. Y \cup \{e\} \notin F.$
$\forall e \in Y.\ c(e) = 1 + \epsilon \quad \epsilon < \frac{1}{n}$ if $n = |F|$ (or something similar)
$\forall e \in X \backslash Y.\ \ c(e) = 1$
$\forall e \notin X \cup Y.\ \ c(e) = \epsilon^2$

Since the elements of Y have the highest cost, they will be selected until the set Y is chosen and cannot increase any further. But the optimal solution would be to select the elements of X – so the greedy algorithm does not solve the problem!

Suppose F is a matroid.
Since $c: S \to R^+$ - all weights are positive, the optimal solution is a basis.
But likewise, the greedy algorithm is also a basis
Sort items in solution in order of decreasing cost.
Suppose the items in the greedy solutions are $g_1, \dots, g_r$ where r is the rank of the matroid.
And the items in the optimal solution are $o_1, \dots, o_r$ where r is the rank of the matroid.
Suppose the maximum prefix is not the entire list of items. Suppose index j is different. So $o_j \neq g_j$ but $o_{j-1} = g_{j-1}$.
But we know that:
$c(g_j) \geq c(o_j)$
So let's build another optimal solution.

Observe the set:
$o_1, \dots o_{j-1}, g_j$
Because this is a matroid, we definitely have some item in $o_1, \dots, o_{j-1}, o_j, \dots, o_r$ that can be added and still have an element of F. We can continue doing so until the group is just as large as $o_1, \dots, o_{j-1}, o_j, \dots, o_r$. But all added items have to be of the set $\{o_j, \dots, o_r\}$.
But since all items are ordered all of them must have a cost $\geq c(o_j)$ and $c(o_j) \leq c(g_j)$

Greedy works also for general c, if the requirement is to find a basis.

Graphical Matroid

Graph G. items are edges of G.F - forests of G. Sets of edges that do not close a cycle. G is connected – bases⇔spanning trees.
Greedy algorithm - Finds maximum weight spanning tree.
We can also find minimal spanning tree – Kruskal's algorithm.

----- End of lesson 5

Matroids – a hereditary family of sets $X \in F, Y \subset X \Rightarrow Y \in F$
$X, Y \in F,\ |X| > |Y| \Rightarrow e \in X\ s.t.\ Y \cup \{e\} \in F$

The natural greedy algorithm, given any cost function: $c: S \to R^{+}$, finds the independent set (members of S) of maximum total weight/cost.

The maximal sets in f all have the same cardinality, r(rank). The maximal sets are called "basis".

$S^{'} \subset S$
And for every $X \subset S, X \in F. X^{'} = X \cap S^{'}$

This gives rise to a family $F^{'}$.

If (S, F) is a matroid, then so is $(S^{'}, F^{'})$ $|X^{'}| > |Y^{'}|$
The rank of the new matroid is $r^{'} \Rightarrow r^{'} \leq r$

Dual of a Matroid

Given a matroid (S, F), it's dual (S, F^{*}) is the collection of all sets where each X satisfies $S \backslash X$ still contains a basis for (S, F)

In the graphical matroid:
S edges of graph G.
F - forests of graph G
The bases are the spanning trees of the graph.

F^{*} - Any collection of edges that by removing it the graph is still connected.

Theorem: The dual F^{*} of matroid F is a matroid by itself.
Moreover, $(F^{*})^{*} = F$.

Proof:
We need to show that the dual is hereditary – but this is easy to see. If we remove an item from x, it still doesn't interfere with the bases of S.

$X, Y \in F^{*},\ |X| > |Y|\ \exists e \in X. Y \cup \{x\} \in F^{*}$

TODO: Draw sets X and Y.

Let's look at $S^{'} = S - (X \cup Y)$

We know that $(S^{'}, F^{'})$ is a matroid, and it has rank $r^{'}$.
If $r^{'} = r$ we can move any item from X to Y and we will still be in F^{*}.
So the only case we have a problem is when $r^{'} < r$ (it can't be larger).

B' be a basis for (S', F'). $|B| = r'$
$r - r' \leq |Y \backslash X|$

Complete B' to a basis of F using B_y.
The number of elements of B_y that we use is exactly $r - r' \leq Y \backslash X < X \backslash Y$.
So some item wasn't used in $X \backslash Y$!
We can take that item, and add it to Y and therefore still get a basis when Y plus that item is removed from S.

$(F^*)^* = F$?
Because the bases of the dual is just the complement of the bases of the original matroid.

$|S| = n$ Also all the bases of the dual are of size $n - r$. $r^* = n - r$.

How did we find the minimal spanning tree?
We sorted all weights by their weight, and added an edge in the spanning tree as long as it doesn't close a cycle.
Minimum weight basis for F=complement of maximum weight basis for F^*.

Graphical Matroids on Planar Graphs

TODO: Draw planar graphs

Every interior (a shape closed by edges) is denoted as a vertex. Then every two vertices are connected if they share a common "side" (or edge). The exterior is a single vertex.

A minimal cut set in the primal is a cycle in the dual.
The dual of a spanning tree is a spanning tree.
The complement of a spanning tree in the primal is a spanning tree in the dual.

Assume we have a planar graph with v vertices, f faces, and e edges.

$$v - 1 + f - 1 = e$$

We can always triangulate a planar graph, thus increasing the number of vertices but keeping it planar.
In such graphs $3f = 2e$.

$$v - 1 + \frac{2e}{3} - 1 = e$$
$$v - 2 = \frac{e}{3}$$
$$e = 3v - 6$$

$2\left(\frac{e}{v}\right) = 6 - \frac{12}{v} = 6$ minus something – at least one with 5 or less!

$(S, F_1), (S, F_2)$
We can look at their intersection such that $X \in F_1$ and $X \in F_2$

Partition Matroid

Partition S into:
$S_1, S_2, \dots, S_r$

And every set that contains at most one item from each partition.

TODO: Draw stuff

A matching is a collection of edges where no two of them touch the same vertex.
The intersection of two partition matroids is the set of matchings in bipartite graph.

In a bipartite graph the set of matchings are not a matroid.

TODO: Draw example graph.

Theorem: For every cost function $c: S \to R^+$, one can optimize over the intersection of two matroids in polynomial time.

Intersections of 3 matroids.

TODO: Draw partitions for the matroid.

Given a collection of triples – find a set of maximum cardinality of disjoint triples. This problem is NP-Hard.

Things we will see:

1) Algorithm for maximum cardinality matching in bipartite graphs.
2) Algorithm for maximum cardinality matching in arbitrary graphs.
3) Algorithm for maximum weight matching in bipartite graphs.

There is also an algorithm for maximum weight matching in arbitrary graphs, but we will not show it in class.

Vertex cover – a set of vertices that cover (touch) all edges.
Min vertex cover $\geq$ maximum matching
Min vertex cover $\leq 2 \cdot$maximal matching.

For bipartite graphs minV.C.=max matching.

Alternating Paths

TODO: Draw graphs

Vertices not in our current matching will be called “exposed”.

An alternating path connects two exposed vertices and edges alternates with respect to being in the matching.

Given an arbitrary matching M and an alternating path P, we can get a larger matching by switching the matching along P.

Alternating forest: A collection of rooted trees. The roots are the exposed vertices, and the trees are alternating.

TODO: Draw alternating forest

In a single step:

- connect exposed vertices
- Add two edges to a tree (some tree)

The procedure to extend a tree:
Pick an outer vertex, and connect it to

- exposed vertex – alternating path – extend matching
- outer vertex – different tree. So we can get from a root of some tree to the root of another tree.
- Extened the tree (alternating forest) by two edges.

Claim: When I'm stuck I certainly have a maximum matching.

Proof by picture.
Select the inner vertices as the vertex cover.

--- end of lesson 6

Matchings

TODO: Draw an alternating forest

1) If you find an edge between exposed vertices then you can just add it to the matching
2) An edge between two outer vertices (in different trees) – alternating path
3) Edge from outer vertex (the exposed vertices are considered as outer vertices) to an unused matching edge

In cases 1 and 2 we restart building a forest

Gallai-Edmond's Decomposition

Outer vertices ar denoted as C (components)
Inner vertices are denoted as S (separator)
The rest are denoted as R (Rest)

We don't have edges between C and itself otherwise the algorithm would not stop
We also don't have edges between C and R otherwise it wouldn't have stopped

In short, we may have edges between C and S, S can have edges between it and itself, S can have edges between it and R and R can have edges between it and itself.

The only choice of matching we used are internal edges of R and edges between C and S.
All vertices of R are matched, all vertices of S are matched, and the number of vertices of C participate in the matching is $|C| - |S|$

Another definition of C, S and R:
C is the set of vertices that are not matched some maximum matching.
S is all neighbors of C
R is the rest

The number of vertices not matched in a maximum matching is exactly $|C| - |S|$.

General Graph

In general graphs we might have odd cycles. So case 2 doesn't work anymore (because an edge can exist in the same tree)

We must have a new case:
2a) An edge between two outer vertices of the same tree.
In this case we get an odd cycle.
Contract the odd cycle! The contracted vertex is an outer vertex.

Suppose we had a graph G, and now we have G' as the contracted graph.
First note the following: Size of matching in G' is equal to size of matching in $G - k$ if the odd cycle had length $2k + 1$.

Lift matching from G' to G and restart building a forest (instead of just restart building a forest).

Now we can see why we call the outer vertices C - since they might represent components (contracted odd cycles)

Can be see that each component always represents an odd number of vertices.

Now, we might have components that might have edges to themselves, but not between two components. So the separator really separates the different components from each other and S.

Vertices of S are all matched. Components in C are either connected to some vertex in S or none (at most connected to one vertex in S)
But then this is the optimal matching! Which means that the algorithm is correct.

So the algorithm finds a matching that misses $|C| - |S|$. But this is the minimal number of edges we miss by any covering – so our solution is the optimal one.

Theorem: If in a graph there is a set S of vertices where removal leaves in the graph t components of odd size (and any number of components of even size, then the size of the maximum matching is at most $\frac{n-(t-|S|)}{2}$

Minimum Weight Perfect Matching in Bipartite Graphs

Weights are non-negative, and we try to find the perfect matching with the maximal weight.

Let's observe the variant where we search for a maximal weight.
If an edge is missing, we can add zeroes instead. Then a perfect matching always exist. And then we can apply a transformation on the maximal variant to make it a minimal variant. So the problem is well defined for non-perfect graphs.

We will use:

- Weight Shifting
- Primal-Dual method

In our case the primal problem – minimum weight covering of vertices by edges. But a perfect matching is a covering problem with minimal weight.

The dual problem is the packing problem: Assign non-negative weights to the vertices such that for every edge, sum of weights of its endpoints is at most weight the edge. Maximize sum of weights of vertices.

TODO: Draw a bipartite graph

The primal is a minimization problem, so it's at least as large as the dual.
For optimal solution there is equality! We will show that…
This is a theorem of Evergary of 1931

Let's try to reduce the problem to an easier problem.
Let w_m the smallest weight of an edge
Subtract w_m from the weight of every edge.
Every perfect matching contains exactly $n \cdot w_m$ from its weight.

As for the dual problem, we can start with the weight of every vertex as 0, and increase the weight of every vertex by $\frac{w_m}{2}$. The dual we got is a feasible dual, and we decreased the weight of the edges.

Let v be some vertex. We can subtract w_v from all of its edges. Since one of them has to be in the final perfect matching, the perfect matching lost exactly w_v
And in the dual, we can increase the weight of v by w_v

We will keep the following claims true:

- For every edge, the amount of weight it loses is exactly equal to the number of weight gained by its endpoints.
- At any state, edges have non-negative weights.

Consider only edges of zero weight and find maximum matching.
There are two possibilities now:

- It is a perfect matching. In this case it is optimal.
- It's not a perfect matching. We can observe the Galai-Edmond decomposition of the graph.

Let ϵ be the minimum weight of an edge between C and R or an edge of weight 2ϵ between C and C.
For every vertex in C, we will add to its weight $+\epsilon$
For every vertex is S we will reduce its weight by ϵ

With respect to the matching, we did not change the weight. But we created one more zero weight edge!
(Note that no edges became negative)

Either we increased the matching (for an edge between C and C)

Or we increased S (for an edge between C and R, since the new connected component of R now belongs to S since a neighbor of some vertex in C)

Every time we make progress. So in $O(n^2)$ weight shifting steps, get a perfect matching of weight zero.