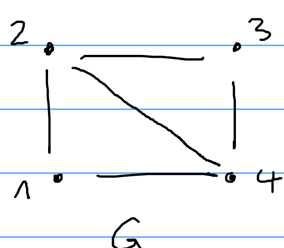
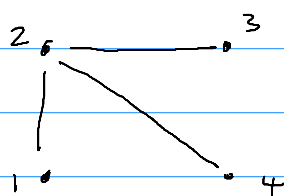
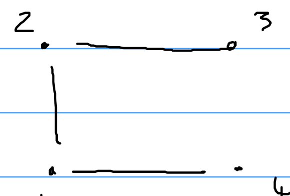
2.5 Spanning Trees

Definition Let G be a graph. Then a tree T is a spanning tree of G if T is a subgraph of G and $V(T) = V(G)$.

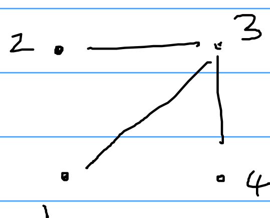


G

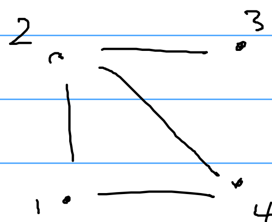




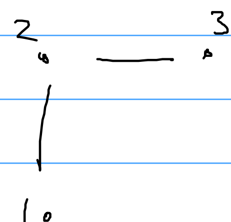
spanning trees of G



not a spanning tree of G



also not a spanning tree



also not a spanning tree

Theorem A graph is connected if and only if it has a spanning tree.

3. Complexity of Algorithms and Problems

3. Input and Running Time

Definition The running time $T(n)$ of an algorithm is the maximum, over all problem instances of size n, of the number of basic operations used by the algorithm in solving the instance.

We will use a notion of running time that ignores constant factors.

Intuition for problem size: space required on a piece of paper (or in a computer) to write down the problem instance.

- For problems on graphs G, a linear function of $|V(G)|$ and $|E(G)|$, $(|V(G)|+|E(G)|)$ times a constant
- For problems involving networks, a function that also depends linearly on $\log W$, where W is the largest weight

[1000]
1 — 2 $\{1,2\}$

3.2 Asymptotic Upper Bounds

$(f(n) \text{ "=" } O(g(n)))$

<u>Definition</u> We say that $f(n)$ is $O(g(n))$, or <u>asymptotically bounded from above</u> by $g(n)$, if there constants $c>0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

<u>Example</u> Let $f(n) = 4n^3 + 6n^2 + 3n + 4$

Goal: show that $f(n)$ is $O(n^3)$ Let $g(n) = n^3$

For $n \geq 1$, $f(n) \leq 4n^3 + 6n^3 + 3n^3 + 4n^3 = 17n^3$

Let $n_0 = 1$ and $c = 17$

Then, for all $n \geq n_0 = 1$, $f(n) \leq c \cdot g(n) = 17 \cdot n^3$, so $f(n)$ is $O(g(n)) = O(n^3)$

Example Consider a tree T with $V(T) = [n]$ given as a list of edges.

The Prüfer code algorithm runs for $n-2$ rounds. The first step in each round, finding the smallest leaf, can be done in $O(n)$ basic operations: we start by creating a vector of length n with all entries equal to zero; we then consider each of the $n-1$ edges in turn and increase the entries in the vector corresponding to the endpoints of the edge by 1 each; finally we go over the entries of the vector to find the index of the smallest entry equal to 1. The remaining steps in each round, writing down the unique neighbour of the smallest leaf, and deleting the smallest leaf along with its unique incident edge, can also be done in $O(n)$ steps.

The overall running time is

$$(n-2)\cdot(\underbrace{O(n)+O(n)}_{2\,O(n) \text{ is } O(2n) \text{ is } O(n)})$$

which is $(n-2)\,O(n)$ which is $n\,O(n)$

which is $O(n^2)$

Definition Consider an algorithm with running time $T(n)$. Then the algorithm is a <u>polynomial-time algorithm</u> if there exists a constant k such that $T(n)$ is $O(n^k)$.

This is the notion of an efficient algorithm we will use.

3.3 Complexity of Problems: P and NP

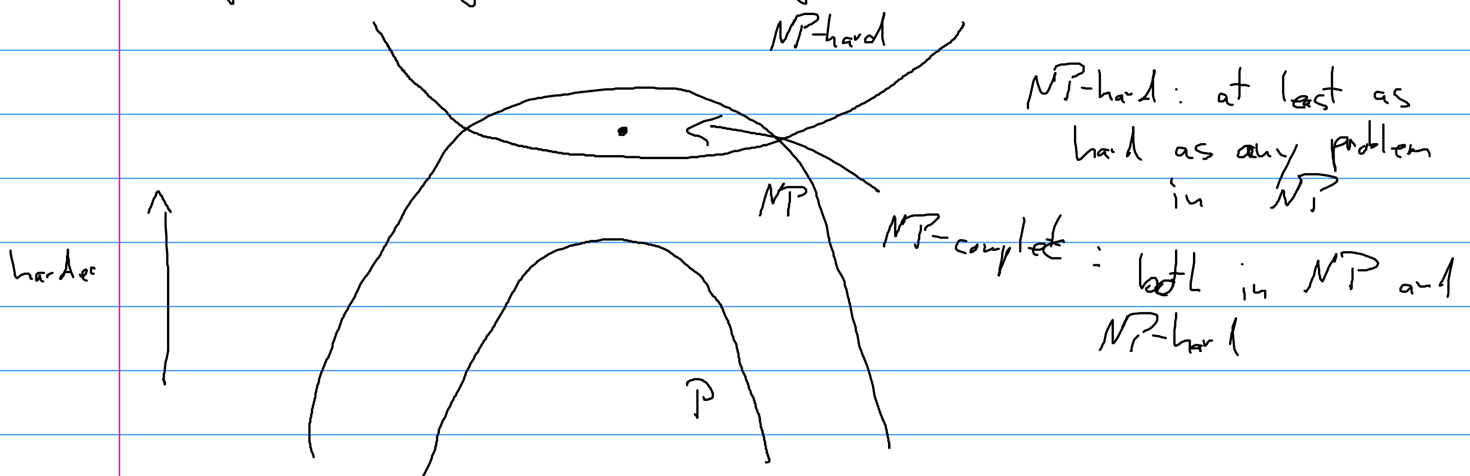
The obvious way to define the complexity of a problem is as the running time of the fastest algorithm that solves the problem.

Definition The complexity class P is the set of all problems for which there exists a polynomial-time algorithm.

We do not currently have a way to show that a problem is not in P. We do have such a way subject to the "famous" conjecture that $P \neq NP$.

NP, which stands for non-deterministic polynomial time, is the class of problems for which a given solution can be verified in polynomial time.

Example: determining whether two graphs are isomorphic is not obviously in P, but it is in NP: given a bijection that preserves edges and non-edges it is easy to verify (in polynomial time) that it actually preserves edges and non-edges.



4. Graph Traversal

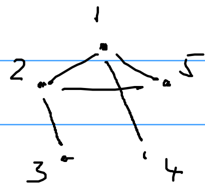
4.1 Breadth-First and Depth-First Search

Algorithm Consider a graph G and $v \in V(G)$. **Tree search** from v starts from T with $V(T) = \{v\}$ and $E(T) = \emptyset$ and then repeats the following steps:

1. Let $u \in V(T)$ and $w \in V(G) \setminus V(T)$ such that $uw \in E(G)$. If no such vertices exist, stop.
2. Add w to $V(T)$ and uw to $E(T)$.

Algorithm In Step 1, let U be the sequence of the vertices in $V(T)$ in the order in which they were added to $V(T)$. **Breadth-first search** selects u to be the first vertex in U such that there exists $w \in V(G) \setminus V(T)$ with $uw \in E(G)$. **Depth-first search** selects u to be the last vertex in U such that there exists $w \in V(G) \setminus V(T)$ with $uw \in E(G)$.

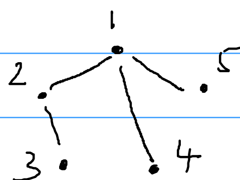
Example



$v = 1$

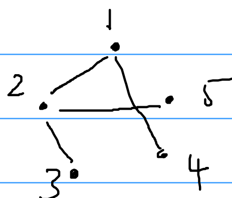
breadth-first search

$U = 1, 2, 4, 5, 4$



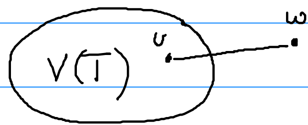
depth-first search

$U = 1, 2, 3, 5, 4$



Theorem For any graph G and $v \in V(G)$, tree search stops after at most $|V(G)|$ many iterations (rounds). When it stops, T is a spanning tree of the connected component of G that contains v.

Proof. In a given iteration, the algorithm either stops or adds a vertex to $V(T)$. We thus stop after at most $|V(G)|$ iterations. We now show by induction that throughout the algorithm T is always a tree. At the beginning this is obvious. Assume now that T is a tree at the beginning of a particular iteration. We want to show that at the end of that iteration T still is a tree. During the iteration we add a new vertex to T, and a new edge between this vertex and a vertex in the tree.



The new edge guarantees the existence of a path from any vertex in $V(T)$ to w, and it cannot form a cycle with any of the edges in $E(T)$.
When the algorithm stops, it does so because there are no edges that have exactly one endpoint in $V(T)$. Thus T is a maximal graph that contains v, is connected, acyclic, and a subgraph of G, i.e., a spanning tree of the connected component of G that contains v. □

Tree search runs for at most $|V(G)|$ iterations. Step 1 in each iteration can be completed in $O(|E(G)|)$ basic operations. Step 2 can be done in a constant number of basic operations.

So the overall running time is $O(|V(G)| \cdot |E(G)|)$.

We will refer to the vertex v the algorithm starts from as the root of the tree it constructs.

4.2 Connected Components

A graph is connected if and only if tree search started from any vertex finds all vertices. If it is not connected all connected components can be found by running the algorithm again and again from a vertex it has not yet seen.