Introduction to Exponential Time: Clever Enumeration.

Recall that in many cases the computational task at hand is *NP-complete*, meaning that we do not expect algorithms that solve any task-instance simultaneously (i) optimally and (ii) in time polynomial in the instance size. There are two main approaches to deal with this: *approximation algorithms* (which we already have seen) relax (i) to 'as good as possible' and *exponential algorithms* relax (ii) to 'as fast as possible'. The latter is useful for several reasons: first, some applications require optimal solutions; second, in some cases, the exponential behavior is only in a parameter that is sometimes small; third, it increases our understanding of NP-complete problems.

In this lecture, we'll start focusing on designing some exponential time algorithms. Specifically, today we'll see a number of tricks to cleverly enumerate all candidate solutions so we can check whether they include a real solution. All algorithms studied here are known to be NP-complete (and you probably have seen reductions proving this for some of these problems in previous courses).

6.1 CNF-Sat

Let's start with recalling a definition:

Definition 6.1. *Let $v_1, \ldots, v_n$ be Boolean variables. A* literal *is an expression of the type v_i or $\neg v_i$ for some i. A* clause *is a disjunction of several literals. A* CNF-formula *is a conjunction of clauses. A k-CNF-formula is a CNF-formula where all clauses are of length at most k.*

To get through this definition here are some examples:

Example 6.1. *The following logical formula's are CNF-formula's:*

1. $(v_1) \wedge (\neg v_2 \vee v_3 \vee \neg v_4)$,
2. $(v_3 \vee v_4) \wedge (\neg v_4 \vee v_2)$,
3. $(v_2 \vee v_1) \wedge (v_2 \vee \neg v_3) \wedge (\neg v_3 \vee v_5)$.

In 1., *the clauses are v_1 and $\neg v_2 \vee v_3 \vee \neg v_4$ and the literals are $v_1, \neg v_2, v_3, \neg v_4$. Only* 2. *and* 3. *are* 2*-CNF-formula's.*

As you may recall, the CNF-SAT problem of determining whether we can satisfy a given CNF-formula with some variable assignment is one of the, if not *the* canonical NP-complete problems. Our first exponential time algorithm is the following completely trivial algorithm:

Algorithm $\texttt{CNF}-\texttt{sat}(\varphi)$ — φ is a CNF-formula on variables $v_1, \ldots, v_n$.
Output: Whether there exists an assignment $(v_1, \ldots, v_n) \in \{true, false\}^n$ such that φ is true.
1: **for all** $(v_1, \ldots, v_n) \in \{true, false\}^n$ **do**
2: **if** φ is satisfied by v **then return true** — Check is easily done in time linear in formula size.
3: **return false**

Algorithm 1: $O(2^n(n+m))$ time algorithm for CNF-SAT.

Such a naïve algorithm as Algorithm 1 is often referred to as *brute-force*. By definition of NP, every problem in NP can be solved by such a brute-force algorithm (Exercise 6.8). However, the good news here is that in many cases we actually *can* significantly reduce the running time of these naive brute-force algorithms, even for many NP-complete problems!

Unfortunately, the unsatisfying[1] state of affairs for CNF-SAT is that currently no algorithm is known that solves this particular problem significantly faster[2] than Algorithm 1 for any instance.

6.2 3-coloring

Now let us move to a different problem that you also might have heard of, where we can do something more. We need the following definition:

Definition 6.2. *A* k-coloring *of a graph* $G = (V, E)$ *is a mapping* $c : V \to \{1, \ldots, k\}$ *such that for every edge* $(v, w) \in E$, $c(v) \neq c(w)$.

One question is: given a graph G, does it have a 3-coloring? As usual throughout these notes, we let n denote the number of vertices of G and m denote the number of edges of G. A first thought is to simply try all possibilities as for CNF-sat:

Algorithm $\texttt{3colv1}(G)$.
Output: Whether a 3-coloring exists.
1: **for all** $c \in \{1, 2, 3\}^V$ **do**
2: **if** c is a 3-coloring of G **then return true** — Check is easily done in $O(n+m)$ time.
3: **return false**

Algorithm 2: $O(3^n(n+m))$ time algorithm for detecting 3-colorings.

But we promised something more exciting. In the specific setting of 3-coloring, we exploit that checking whether a given graph is 2-colorable is easy using BFS. Given a linear time algorithm determining whether a graph is 2-colorable, we can just try all possible subsets of V that receive color 3 (and thus need to form and independent set), and for such a fixed set we can check whether the graph induced on the remaining vertices admits a 2-coloring.

[1] No pun intended.

[2] In case you're wondering, the current record is $2^{(1-\frac{1}{O(\log(m/n))})n}$ time, if m denotes the number of clauses.

Algorithm `3colv2`(G).
Output: Whether a 3-coloring exists.
1: **for all** $X \subseteq V$ **do**
2: **if** X is an independent set of G **then**
For $G = (V, E)$ and $X \subseteq V$, $G[X]$ denotes the graph $(X, \{e \in E : e \subseteq X\})$.
3: **if** `2colorable`$(G[V \setminus X])$ **then return true**
4: **return false**

Algorithm 3: $O(2^n(n+m))$ time algorithm for detecting 3-colorings.

6.3 Vertex Cover

Definition 6.3. *A* vertex cover *of a graph $G = (V, E)$ is a subset $X \subseteq V$ such that for every edge $(v, w) \in E$ it holds that $v \in X \vee w \in X$.*

In this section we study the decision variant of finding vertex covers: Given a graph G and an integer k, does G have a vertex cover of size at most k? This problem and the upcoming algorithm is the bread and butter of people working in the area of 'Parameterized Complexity' that is specialized in designing algorithms where 'the exponential behavior is only in a parameter that is sometimes small', as mentioned in the introduction. It is common to use the letter k for the parameter that is sometimes small, so we will also try to stick to this.

First Algorithm Note that a completely naïve algorithm would be to try all 2^n subsets and see which is a vertex cover of size at most k. Slightly less naïve would be to try all $\sum_{i=0}^{k} \binom{n}{i}$ subsets of V and see which is a vertex cover, which is already quite an improvement. But, since k typically is a lot smaller than $|V|$, the following is considerably better.

Algorithm `vc`$(G = (V, E), k)$
Output: Whether G has a vertex cover of size at most k.
1: **if** $E = \emptyset$ **and** $k \geq 0$ **then return true**
2: **if** $k \leq 0$ **then return false**
3: Let $(u, v) \in E$
4: **return** `vc`$(G[V \setminus u], k-1) \vee$ `vc`$(G[V \setminus v], k-1)$

Algorithm 4: $O(2^k k(n+m))$ time algorithm for detecting vertex covers of size at most k.

Now let us consider Algorithm 4 and let's first see whether it outputs what it promises. The main point is the following:

Claim 6.1. *If $G = (V, E)$ and $(u, v) \in E$ then G has a vertex cover of size at most k if and only if $G[V \setminus u]$ has a vertex cover of size $k-1$ or $G[V \setminus v]$ has a vertex cover of size $k-1$.*

Proof. If X is a vertex cover in G of size at most k, then $u \in X$ or $v \in X$ (or both) by definition. Without loss of generality, assume $u \in X$. Then $X \setminus u$ will be a vertex cover of $G[V \setminus u]$ since all edges not incident to u need to be covered by other elements of X, and clearly $|X \setminus u| \leq k-1$. For the other direction, if $X \setminus u$ is a vertex cover of $G[V \setminus u]$ of size at most $k-1$, then $X \cup \{v\}$ is a vertex cover of G of size at most k. □

Since we have a recursive algorithm, we'll use induction for proving its correctness. Since k is lowered in every recursive call, we'll prove the lemma by induction on k.

Lemma 6.1. $\texttt{vc}(G, k)$ *returns* **true** *if and only if* G *has a vertex cover of size at most* k*.*

Proof. For the base case $k = 0$ this clearly holds since $G = (V, E)$ has a vertex cover of size 0 if and only if $E = \emptyset$. Otherwise, Line 4 is reached so $\texttt{vc}(G, k) = \texttt{vc}(G[V \setminus u], k-1) \vee \texttt{vc}(G[V \setminus v], k-1)$ and we can apply Claim 6.1 in combination with the induction hypothesis. □

We claim that the running time of Algorithm 4 is at most $O(2^k k(n+m))$. To see this, first note that per recursive call we spend at most $O(n+m)$ time, and that the recursion depth is at most k since every time k is lowered by 1. Thus, denoting $T(k)$ for the maximum number of recursive calls of $\texttt{vc}(G, k)$ that do not recurse themselves, it is sufficient to show $T(k) \leq 2^k$. Note $T(k)$ satisfies $T(k) = 1$ for $k = 0$ and $T(k) \leq T(k-1) + T(k-1)$ for $k > 0$, so $T(k) \leq 2^k$ by induction.

An alternative way to upper bound $T(k)$ would be to imagine the 'branching tree' with a vertex for every recursive call and the leaves being the calls where the recursion stops. Clearly this tree has depth at most k. Moreover, since every recursive call invokes at most two recursive calls itself, we have an upper bound of 2^k on the number of leaves.

Definition of Fixed Parameter Tractability. As mentioned $O(2^k k(n+m))$ is considerably better than $O(2^n)$ or $O(\sum_{i=0}^{k} \binom{n}{i})$ for small k: for every fixed constant k this is linear time! To stress that such a running is highly desirable, let us tie a formal definition to it:

Definition 6.4 (Fixed Parameter Tractable)**.** *A parameterized problem is a language* $L \subseteq \{0,1\}^* \times \mathbb{N}$*. For an instance* $(x, k) \in \{0,1\}^n \times \mathbb{N}$*,* (x, k) *is called the* input *and* k *is called the* parameter*. A parameterized problem* L *is called* Fixed Parameter Tractable *if there exists and algorithm that given* $(x, k) \in \{0,1\}^n \times \mathbb{N}$*, correctly decides whether* $(x, k) \in L$ *in time bounded by* $f(k) \cdot |(x, k)|^c$ *for some constant* c *and some function* $f(\cdot)$*.*

Thus the above algorithm demonstrates that 'Vertex Cover parameterized by k (solution size) is FPT'.

Second Algorithm Now the machinery is set up, let's look at an improvement of Algorithm 4:

Algorithm $\texttt{vc2}(G = (V, E), k)$
Output: Whether G has a vertex cover of size at most k
1: **if** $E = \emptyset$ and $k \geq 0$ **then return true**
2: **if** $k \leq 0$ **then return false**
3: **if** $\exists u \in V : \deg(u) \geq 2$ **then**
4: **return** $\texttt{vc2}(G[V \setminus u], k-1) \vee \texttt{vc2}(G[V \setminus N[u]], k - \deg(u))$
5: **else**
6: **if** $k \geq |E|$ **then return true else return false**

Algorithm 5: $O(1.62^k k(n+m))$ time algorithm for detecting vertex covers of size at most k.

The proof of correctness has exactly the same structure as the proof of Lemma 6.2, except that we need to use the following alternative for Claim 6.1:

Claim 6.2. *If $G = (V, E)$ and $u \in V$ then G has a vertex cover of size at most k if and only if $G[V \setminus u]$ has a vertex cover of size $k-1$ or $G[V \setminus N[u]]$ has a vertex cover of size $k - \deg(u)$.*

Proof. If X is a vertex cover in G of size at most k, then $u \in X$ or $u \notin X$. If $u \in X$, then $X \setminus u$ will be a vertex cover of $G[V \setminus u]$ since all edges not incident to u need to be covered by other elements of X. If $u \notin X$, then $N(u) \subseteq X$, and $X \setminus N(u)$ will be a vertex cover of $G[V \setminus N[u]]$.

For the other direction, if $X \setminus u$ is a vertex cover of $G[V \setminus u]$ or if $X \setminus N(u)$ is a vertex cover of $G[V \setminus N[u]]$, then X is a vertex cover of G. □

Lemma 6.2. `vc2`(G, k) *returns* **true** *if and only if G has a vertex cover of size at most k.*

Proof. Suppose `vc2`(G, k) does not result in a recursive call. The statement clearly holds for $k \leq 0$. Otherwise, there is no vertex of degree at least 2 and thus no two edges of G overlap. Then the optimal vertex cover has size $|E|$ (we need one vertex for every edge, and for every edge one is sufficient).

Otherwise, Line 4 is executed and we can apply Claim 6.1 in combination with the induction hypothesis. □

But why do we get this odd-looking running time? To see this note that again we spend only $O(n + m)$ time per recursive call, and the recursion depth is at most k, so we only need to bound the number of leaves $T(k)$ in the recursion tree of `vc2`(G, k) by $O(1.62^k)$. We see that $T(k)$ satisfies

$$T(k) \leq \begin{cases} 1, & \text{if } k = 0 \\ \max_{d \geq 2} T(k-1) + T(k-d), & \text{if } k \geq 1. \end{cases}$$

Now we prove by induction on k that $T(k) \leq 1.62^k$ for every $k \geq 0$. That is easily seen to hold for $k = 0$, while for $k > 0$ we see that

$$T(k) \leq \max_{d \geq 2} T(k-1) + T(k-d) \leq \max_{d \geq 2} 1.62^{k-1} + 1.62^{k-d} \leq 1.62^{k-1} + 1.62^{k-2} \leq 1.62^k.$$

The second inequality uses the induction hypothesis. For the last inequality, note that $1.62^{-1} + 1.62^{-2} \leq 1.62$.

Thus, the smallest base of the exponential we can get via this analysis is exactly the unique positive solution to the equation $x^{-1} + x^{-2} = 1$ (which is known to be the *golden ratio*)[3].

Remark 6.1. *Recursive algorithms as seen here often are called 'branching algorithms'. Intuitively both algorithm use Claims 6.1 and 6.2 to narrow down the search space to solutions with a particular property. We'll use this over and over again in branching algorithms.*

6.4 Cluster Editing and $O^*(\cdot)$ notation.

Definition 6.5. *A graph is a* cluster graph *if all its connected components are cliques. A* cluster editing *of a graph $G = (V, E)$ is a set $X \subseteq V \times V$ such that $(V, E \triangle X)$ is a cluster graph. Here $\triangle$ denotes the symmetric difference: $X \triangle Y = X \setminus Y \cup Y \setminus X$.*

[3]When analyzing branching algorithms of this type, one will be faced with a 'linear homogeneous recurrence relation'. It is known that the number of positive real roots of such a recurrence is exactly the number of 'sign changes' (see `https://en.wikipedia.org/wiki/Descartes%27_rule_of_signs`), which is always 1 in recurrences obtained from branching algorithms.

The following definition will give a useful perspective on cluster editing.

Definition 6.6. *An induced P_3 of $G = (V, E)$ is a triple (u, v, w) of vertices such that $uv, vw \in E$ and $uw \notin E$.*

The main use of this definition is that the following observation allows us to deal with induced P_3's instead of cluster editing.

Observation 6.1. *A graph is a cluster graph if and only if it does not have induced P_3's.*

Algorithm $\mathtt{ce}(G = (V, E), k)$
Output: Whether G has a cluster editing of size at most k.
1: **if** $k = 0$ **then return true** if G is cluster graph, **return false** otherwise
2: **if** $\exists$ induced P_3 (u, v, w) in G **then**
3: **return** $\mathtt{ce}((V, E \cup uw), k-1) \vee \mathtt{ce}((V, E \setminus uv), k-1) \vee \mathtt{ce}((V, E \setminus vw), k-1)$
4: **return true**

Algorithm 6: $O^*(3^k)$ time algorithm for cluster editing distance.

Again the correctness of the algorithm is proved via induction. In the base case, the answer is clearly correct if $k = 0$ or there exists no induced P_3. Otherwise, if (u, v, w) is an induced P_3, we can use the following claim:

Claim 6.3. *Let $G = (V, E)$ be a graph and let (u, v, w) be an induced P_3. Then G has a cluster editing of size at most k if and only if at least one of the graphs $(V, E \cup uw), (V, E \setminus uv), (V, E \setminus vw)$ has a cluster editing of size at most $k - 1$.*

Proof. For the forward direction, let $C \subseteq E$ be a cluster editing of G of size at most k. Then C contains either uv, vw or uw by Observation 6.1. Let $e \in C$ be one of these three edges. Then $C \setminus e$ is a cluster editing of $(V, E \triangle e)$ of size at most $k - 1$. For the reverse direction, suppose $(V, E \triangle e)$ has a cluster editing C' of size at most $k - 1$ with $e \notin C$, then $C' \cup e$ is a cluster editing of (V, E) of size at most k since $E \Delta (C' \cup e) = (E \triangle e) \Delta C'$. □

The running time analysis of this algorithm is also of a familiar type. It is easy to see that per recursive call we spend at most $O(n^3)$ time (due to Line 2). The recursion depth is at most k and the number of leaves $T(k)$ of the branching tree of $\mathtt{ce}(G = (V, E), k)$ satisfies $T(k) = 1$ for $k = 0$, and for $k \geq 0$ we have $T(k) \leq 3T(k-1)$, so $T(k) \leq 3^k$ is a valid upper bound.

When dealing with exponential time we are mostly mainly interested in the precise exponential dependence rather than the polynomial dependency in other parameters. For instance for Algorithm 6, one might be tempted to add all kinds of tricks to improve $O(n^3)$ bound in every recursive call, but to keep focus we will not concern ourselves with such issues here. To facilitate this, we use the following notation: if an algorithm runs in time $f((x, k))|x|^{O(1)}$, where k denotes some instance parameter and $|x|$ denotes the number of bits uses to encode the problem instance, then we say it runs in $O^*(f((x, k))$ time. So Algorithm 6 runs in $O^*(3^k)$ time.

6.5 Feedback Vertex Set

Now we introduce a general technique called *iterative compression*. We use this to give an algorithm that determines in $O^*(8^k)$ time whether a given graph has a *feedback vertex set* of size at most k. On top of this technique, we will again make use of the techniques from the previous section but in a bit less direct way.

Definition 6.7. *A* forest *is a graph without cycles. A* feedback vertex set (FVS) *of a graph $G = (V, E)$ is a subset $X \subseteq V$ such that $G[V \setminus X]$ is a forest.*

Alternatively, we could say that a FVS is a vertex set X such that for any cycle of G at least one of its vertices is in X.

6.5.1 Iterative Compression

The main idea of iterative compression is to design an algorithm for a variant of the problem at hand where we are already given a slightly large solution. Then 'iterative compression' can be used to convert an algorithm for this variant into one that determines whether a solution of size at most k exists.

We describe the 'iterative compression' part here and in the next subsection we'll provide an algorithm for the mentioned variant. Specifically, we assume there is an algorithm $\texttt{disjFVS}(G, X, k)$ that given graph G, FVS X of G and integer k, determines whether there exists a FVS of G disjoint from X of size at most k. We describe $\texttt{disjFVS}(G, X, k)$ in the next subsection but now show how to use it. We do this as follows:

Algorithm $\texttt{FVS}(G = (V, E), k)$
Output: Whether G has a feedback vertex set of size at most k

1: Let $V = \{v_1, \ldots, v_n\}$
2: Let $X = \{v_1, \ldots, v_k\}$
3: **for** $i = k + 1, \ldots, n$ **do**
4: $X \leftarrow X \cup v_i$ — X is a FVS of $G[\{v_1, \ldots, v_i\}]$ of size at most $k + 1$
start compression
5: **for** $Y \subseteq X$ **do**
6: **if** $\texttt{disjFVS}(G[V \setminus Y], X \setminus Y, k - |Y|)$ **then**
7: Construct the FVS X' of $G[V \setminus Y]$ of size $k - |Y|$ using self-reduction
8: $Z \leftarrow X' \cup Y; X \leftarrow Z$ — in order to be able to refer to Z in the analysis
end compression
9: **if** $|X| = k + 1$ **then return false** — check whether the compression was successful
10: **return true**

Algorithm 7: Algorithm for Feedback Vertex Set.

Let us now motivate why this algorithm is correct. The algorithm constructs a FVS of the graph $G[\{v_1, \ldots, v_i\}]$ using a FVS X of $G[\{v_1, \ldots, v_{i-1}\}]$ as follows: first it adds v_i to X (then it is easy to see that X is a FVS of $G[\{v_1, \ldots, v_i\}]$ of size at most $k + 1$), and then it attempts in Line 5-8 to *compress* X to obtain a FVS Z of $G[\{v_1, \ldots, v_i\}]$ of size at most k.

To find this FVS Z, we try all $2^{|X|} \leq 2^{k+1}$ possibilities of $Y = X \cap Z$. if Line 8 is reached, Z is easily seen to be a FVS of $G[\{v_1, \ldots, v_i\}]$ since all cycles incident to Y are hit and all other cycles are hit by X'. On the other hand, if Z exists, surely we try $Y = X \cap Z$ at some iteration, and then $Z \setminus Y$ will be a FVS of size at most $k - |Y|$ of $G[V \setminus Y]$ that is disjoint from $X \setminus Y$. Thus, it will be detected by `disjFVS`. If it is detected that it exists, it can be constructed by a small modification of `disjFVS` or by standard self-reduction arguments within a similar time bound. Thus Line 5-8 will always find Z if it exists, so if $|X| = k + 1$ on Line 9, we know for sure X could not have been compressed so we can safely conclude G has no FVS of size at most k.

6.5.2 Algorithm for Disjoint Feedback Vertex Set

Now let us focus on the procedure `disjFVS`(G, W, k). It will be convenient to work with *multigraphs* rather than graphs. In a multigraph E is a multiset rather than a set, so there might be several parallel edges between two vertices of G. We say that $\deg(\cdot)$ also counts these copies of edges and that two edges (u, v) already form a cycle on their own.

Algorithm `disjFVS`$(G = (V, E), W, k)$ W is a FVS of G
Output: Whether G has a feedback vertex set of size at most k disjoint from W
1: **if** $k < 0$ **then return false**
2: **if** $k = 0$ **then return true** if G is a forest, **return false** otherwise
3: **if** $\exists v \in V$ such that $\deg(v) \leq 1$ **then**
4: **return** `disjFVS`$(G \setminus v, W, k)$ $G \setminus v$ shorthands $(V \setminus v, E)$
5: **if** $\exists v \in V \setminus W$ such that $G[W \cup v]$ contains a cycle **then**
6: **return** `disjFVS`$(G \setminus v, W, k - 1)$
7: **if** $\exists v \in V \setminus W$ such that $\deg(v) = 2$ and at least one neighbor from v in G is in $V \setminus W$ **then**
8: Let G' be obtained from G by adding the edge between both neighbors of v and removing v
Note: if there was such an edge already, there are multiple edges now!
9: **return** `disjFVS`(G', W, k)
10: Let $x \in V$ such that x has at most one neighbor in $V \setminus W$ and at least two neighbors in W.
11: **return** `disjFVS`$(G \setminus x, W, k - 1) \vee$ `disjFVS`$(G, W \cup x, k)$

Algorithm 8: $O^*(2^{|W|+k})$ time algorithm for Disjoint Feedback Vertex Set.

In Line 1 to 9, Algorithm `disjFVS` checks for several *reduction rules*: if there is some special structure that we know how to deal with, then we do this and recurse to some simplified instance or return the answer immediately. The reduction rules are motivated as follows:

- Line 3: a vertex of degree at most 1 will never be on a cycle so it is not relevant.
- Line 5: since we are looking for a FVS disjoint from W, the only way to hit this cycle is to include v.
- Line 7: if the neighbors of v are u, w, then u, w are in all cycles that include v, so if v is used to hit a cycle in the FVS we can as well remove it from the FVS and add u or v instead. Effectively, we cannot simply remove v in the graphs since it may very well be used to connect v and w in a cycle, but to compensate this we can simply add the edge (u, w).

If all of this does not apply, there must be a leaf x of the forest $G[V \setminus W]$ such that x has at least two neighbors that are in W (if it had 0, Line 3 would apply; if it had 1, Line 7 would apply). Now we make a decision on x: will it be in the feedback vertex set or not? Based on this we recurse on the appropriate sub-instances (if x is picked remove it, otheriwse add it to W). This ends to motivation of the correctness of `disjFVS`.

What about its running time? Clearly we spend polynomial time per recursive call again, but bounding the number of leaves of the recursion tree seems harder since we do not lower k always. But if the algorithm makes sense, the recursive call $\texttt{disjFVS}(G', W, k)$ should somehow solve a 'simpler' instance. How do we quantify this?

After a moment's thought we may realize that, since the reduction rule on Line 5 did not apply, the neighbors of x that are in W are not connected in $G[W]$, but in $G[W \cup x]$ they are. So we notice that $\#cc(G[W \cup x]) \leq \#cc(G[W]) - 1$, where $\#cc(\cdot)$ denotes the number of connected components. Thus, on Line 11 we recurse on two instances: in one k decreases and in the other $\#cc(G[W])$ decreases. Therefore, if we define a *measure* $\mu(G, W, k) = k + \#cc(G[W])$ of the 'hardness' of an instance and let $T(\mu)$ be the maximum number of leaves of the recursion tree of $\texttt{disjFVS}(G, W, k)$ where $\mu(G, W, k) = \mu$ then we decrease μ by at least one in every recursion step.

Summarizing, we claim that $T(\mu) \leq 2^\mu$. To see this, first note that in the reduction rules μ never increases, and if $\mu = 0$ then $k \leq 0$ so $\texttt{disjFVS}(G, W, k)$ does not recurse. So we get

$$T(\mu) \leq \begin{cases} 1, & \text{if } \mu \leq 0 \\ \max_{d \geq 1} T(\mu - 1) + T(\mu - d), & \text{if } \mu > 1, \end{cases}$$

and similarly to before we can bound this with $2^\mu = 2^{\#cc(G[W])+k}$ for all $\mu \geq 0$. Therefore, `disjFVS` runs in time at most $O^*(2^{k+\#cc(G[W])})$ which is at most $O^*(2^{k+|W|})$, and the running time of `FVS` is bounded by $O^*(8^k)$.

6.6 Subset Sum

In the Subset Sum problem we are given integers $w_1, \ldots, w_n$ and an additional integer t and are interested in determining whether there exists a subset $X \subseteq \{1, \ldots, n\}$ such that $\sum_{e \in X} w_e = t$. Note that the trivial algorithm would solve this in time $O^*(2^n)$. In this section we use a technique that is quite different from the prevous sections that is sometimes called *meet in the middle*.

Again, we solve the problem by first designing an algorithm for different problems: in the 2-SUM problem we are given integers $a_1, \ldots, a_m$ and $b_1, \ldots, b_m$ and want to determine whether there exist i, j such that $a_i + b_j = t$. We'll see soon that this problem can be solved in $O(m \lg m)$ time. Now we can use this algorithm to solve Subset Sum as follows: given integers $w_1, \ldots, w_n$ (possibly adding a 0, we may assume n is even), for every subset $X \subseteq \{1, \ldots, n/2\}$ construct an integer $a_i = \sum_{e \in X} w_i$ and similarly for every $X \subseteq \{n/2 + 1, \ldots, n\}$ construct an integer $b_i = \sum_{e \in X} w_i$. It is easy to see that pairs $a_i + b_j = t$ are in 1 to 1 correspondence with $Z \subseteq \{1, \ldots, n\}$ such that $\sum_{e \in Z} w_e = t$. In the created 2-SUM instance $m = 2^{n/2}$ so using the $O(m \lg m)$ time 2-SUM algorithm solves the Subset Sum problem in $O^*(2^{n/2})$ time.

Before we move to the algorithm for 2-SUM let us remark that since this algorithm uses $2^{n/2}$ space, a natural question is whether this space usage can be lowered. To this end, we also study the 4-SUM problem where we are given integers $a_1, \ldots, a_m, b_1, \ldots, b_m, c_1, \ldots, c_m, d_1, \ldots, d_m, t$ and look for i, j, k, l such that $a_i + b_j + c_k + d_l = t$. In turns out that this problem can be solved in

$O(m^2 \lg m)$ time and $O(m \lg m)$ space, and similarly to our previous algorithm, this can be used to solve Subset Sum in time $O^*(2^{n/2})$ time and $O^*(2^{n/4})$ space (Exercise 6.5).

6.6.1 $O(m \lg m)$ time algorithm for 2-SUM

The algorithm for 2SUM is as follows:

Algorithm `2SUM`(L, R, t) L and R are lists of length m of integers, $t \in \mathbb{Z}$
Output: Whether there exist $l \in L$, $r \in R$ such that $l + r = t$
1: Sort L, R to obtain lists $l_1, \ldots, l_m$ and $r_1, \ldots, r_m$
2: $i \leftarrow 1$, $j \leftarrow m$.
3: **while** $i \leq n \wedge j \geq 1$ **do**
4: **if** $l_i + r_j = t$ **then return true**
5: **if** $l_i + r_j < t$ **then** $i \leftarrow i + 1$ **else** $j \leftarrow j - 1$
6: **return false**

Algorithm 9: $O(m \lg m)$ time, $O(m)$ space algorithm for 2-SUM.

It is easy to see that this runs in $O(m \lg m)$ time. If the algorithm returns **true** it clearly is correct. On the other hand, suppose there exist x, y such that $l_x + r_y = t$ and suppose i reaches x before j reaches y. Then $l_i + r_j > t$ and the while loop will decrease j until Line 4 applies. Otherwise, j reaches y before x reaches i and the while loop will increase i until Line 4 applies.

6.6.2 $O(m^2 \lg m)$ time and $O(m \lg m)$ space algorithm for 4-SUM

Algorithm `4SUM`(A, B, C, D, t) A, B, C, D are lists of length m of integers, $t \in \mathbb{Z}$
Output: Whether there exist $a \in A$, $b \in B$, $c \in C$ and $d \in D$ such that $a + b + c + d = t$
1: Sort A, B, C, D to obtain lists $a_1, \ldots, a_m, b_1, \ldots, b_m, c_1, \ldots, c_m, d_1, \ldots, d_m$
2: Initiate a priority queue Q_L, where the lowest key has the highest priority
3: Initiate a priority queue Q_R, where the highest key has the highest priority
4: **for** $i = 1, \ldots, m$ **do**
5: `insert` (a_1, b_i) with key $a_1 + b_i$ to Q_L
6: `insert` (c_m, d_i) with key $c_m + d_i$ to Q_R
7: **while** Q_L and Q_R are non-empty **do**
8: $(a_i, b_j) \leftarrow$ `pull`(Q_L)
9: $(c_k, d_l) \leftarrow$ `pull`(Q_R)
10: **if** $a_i + b_j + c_k + d_l = t$ **then return true**
11: **if** $a_i + b_j + c_k + d_l < t$ **then**
12: **if** $i \leq m$ **then** `insert` (a_{i+1}, b_j) with key $a_{i+1} + b_j$ to Q_L
13: `insert` (c_k, d_l) with key $c_k + d_l$ to Q_R
14: **else**
15: `insert` (a_i, b_j) with key $a_i + b_j$ to Q_L
16: **if** $k \geq 1$ **then** `insert` (c_{k-1}, d_l) with key $c_{k-1} + d_l$ to Q_R
17: **return false**

Algorithm 10: $O(m^2 \lg m)$ time, $O(m \lg m)$ space algorithm for 4-SUM.

Now we study an algorithm that mimicks Algorithm 9 in a more space-efficient way. It makes use of a priority queue data-structure with `insert` and `pull` operations (recall the definition from `http://en.wikipedia.org/wiki/Priority_queue` if needed).

If $4SUM$ returns true, it trivially is correct. Suppose there exists w, x, y, z such $a_w + b_x + c_y + d_z = t$. Note that the algorithm will run through either all pairs a_i, b_j or all pairs c_k, d_l. Suppose it runs through all pairs a_i, b_j and it arrives at a_w, b_x before c_y, d_z is reached. Then the algorithm will iterate though the pairs k, l until $k = y, l = z$ is reached. A similar reasoning holds for the other cases (and indeed the algorithm completely mimicks $2SUM(L, R, t)$ where $L = \{a_i + b_j : 1 \leq i, j \leq n\}$ and $R = \{c_k + d_l : 1 \leq k, l \leq n\}$).

For the running time, there are basic datastructures for priority queues such that `pull` and `insert` run in $\lg(m)$ time and the loop on Line 7 runs for at most $2m^2$ iterations since in every iteration we move in L or R.

6.7 Exercises

Exercise 6.1. Recall that in the Traveling Salesman problem, we are given a graph $G = (V, E)$ with an integer weight w_e and the question is to find a Hamiltonian cycle $C \subseteq E$ minimizing $\sum_{e \in C} w_e$. Can you solve it in $O^*(n!)$ time?

Exercise 6.2. In the k-coloring problem we are given a graph G and integer k and need to determine whether G has a k-coloring. Do you expect this problem parameterized by k to be FPT?

Exercise 6.3. Find an algorithm detecting cliques of size at least k in $O(n^k k^2)$ time, why is this running time not sufficient to prove the problem to be FPT?

Exercise 6.4. Show that if G has a FVS of size at most k, it has a $k + 2$-coloring. Can you give an example of a graph with a FVS of size at most k but no $k + 1$ coloring?

Exercise 6.5. Give an $O^*(2^{n/2})$ time, $O^*(2^{n/4})$ space algorithm for Subset Sum using the 4SUM algorithm.

Exercise 6.6. Can you solve 4-coloring in $O^*(2^n)$ time? What about 3-coloring in $O^*((2 - \epsilon)^n)$ time, for some $\epsilon > 0$ (Hint: use that $\binom{n}{k} \leq 2^{0.92n}$ for $k \leq n/3$)?

Exercise 6.7. Solve Vertex Cover in $O^*(1.4656^k)$ time.

Exercise 6.8. Recall the definition of NP. Why can any problem instance $x \in \{0, 1\}^n$ of a language in NP be solved in $2^{\mathtt{poly}(|x|)}$ time?

Exercise 6.9. An algorithm running in time $n^{\lg(n)^c}$ for some constant c is called *quasi-polynomial*. Recently, in a big breakthrough[4] László Babai showed that the 'Graph Isomorphism problem' can be solved in quasi-polynomial time. Graph Isomorphism is not known to be NP-complete. Can

[4](see e.g., `http://www.quantamagazine.org/20151214-graph-isomorphism-algorithm/`)

you explain why a quasi-polynomial time algorithm for an NP-complete problem would be a *huge* result (Hint: recall the definition of NP-completeness)?

Exercise 6.10. Show that Feedback Vertex Set is NP-hard. In particular, show that given an instance (G, k) of vertex cover, we can compute in polynomial time an equivalent instance (G', k) of feedback vertex set.

Exercise 6.11. The n'th Fibonacci number f_n is defined as follows: $f_1 = 1$,$f_2 = 1$ and for $n > 2$, $f_n = f_{n-1} + f_{n-2}$. What is the running time of the following algorithm to compute f_n?

Algorithm `FIB1`(n)
Output: f_n
1: **if** $n = 1$ or $n = 2$ **then return** 1
2: **return** `FIB1`$(n-1)$+`FIB1`$(n-2)$.

Exercise 6.12. In the Set Partition problem we are given $F_1, \ldots, F_m \subseteq U$ and need to find a subset of the sets that partition U. Can you do this in $O^*(2^{m/2})$ time?

Exercise 6.13. In this exercise we'll look at the d-Hitting Set problem: given sets $F_1, \ldots, F_m \subseteq U$ of size d each, where $|U| = n$, we need to find a subset $X \subseteq U$ with $|X| = k$ that 'hits' every set in the sense that $F_i \cap X \neq \emptyset$ for every i.

1. By which other name do you know 2-Hitting Set? Why is it equivalent?
2. Can you solve 3-Hitting Set in time $O^*(3^k)$?
3. Can you solve 3-Hitting Set in time $O^*(2.4656^k)$
   - Hint: Use iterative compression. Suppose you are also given a hitting set of size $k+1$, can you solve the problem in time $O^*(\sum_{i=1}^{k+1} \binom{k+1}{i} 1.4656^i)$. This equals $O^*(2.4656^k)$ by the binomial theorem.

Exercise 6.14. Give an algorithm that determines whether a given 3-CNF-Sat formula is satisfiable in time $O^*((2-\epsilon)^n)$, for some $\epsilon > 0$.