2MMD30: Graphs and Algorithms Lecture 11 by Jesper Nederlof, 18/03/2016

Probabilistic Exponential Time Algorithms.

In this lecture we again look at some exponential time algorithms, but this time see how randomization is useful here. We are going to see algorithms in this lecture that never return false positives (i.e. the algorithm never outputs a **true** that should have been a **false**), but if the instance is a YES-instance, the algorithm may output **false** with constant probability.

11.1 Stirling's approximation

Stirling's approximation states that $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$. The $\approx$ symbol is strictly speaking not mathematical. A more mathematical way of putting it would be $n! = \theta(\sqrt{n}\left(\frac{n}{e}\right)^n)$, e.g. there are some constants c_1, c_2 such that $c_1\sqrt{n}\left(\frac{n}{e}\right)^n \leq n! \leq c_2\sqrt{n}\left(\frac{n}{e}\right)^n$.

11.1.1 Derivation of a variant of Stirling's approximation (will not be examined)

Since we'll use Stirling's approximation several times in this lecture, it's good to have seen a proof of some version of it. However, since this is mainly elementary calculus it will not be examined. Specifically we'll show $e\left(\frac{n}{e}\right)^n \leq n! \leq (n+1)e\left(\frac{n}{e}\right)^n$. Note that $\ln(n!) = \ln(1\cdot 2\cdot 3\cdots n) = \sum_{k=1}^n \ln(k)$, and $\sum_{k=1}^n \ln(k)$ can be sandwiched by two integrals as follows:

$$\int_1^n \ln(x)dx \leq \sum_{k=1}^n \ln(k) \leq \int_1^{n+1} \ln(x)dx,$$

to show that this is true, one can split the integral in n parts and argue that the i'th part is at most/least $\ln(i)$. A nice pictorial description of this argument is provided in Figure 11.1. Then we can compute these integrals by finding the anti-derivatives in the usual way:

$$\begin{aligned}
[x\ln x - x]_1^n &\leq \sum_{k=1}^n \ln(k) \leq [x\ln x - x]_1^{n+1} \\
n\ln n - n + 1 &\leq \sum_{k=1}^n \ln(k) \leq (n+1)\ln(n+1) - (n+1) + 1 = (n+1)\ln(n+1) - n,
\end{aligned}$$

and taking exponentials in the last expression we obtain

$$e\left(\frac{n}{e}\right)^n \leq n! \leq (n+1)\left(\frac{n+1}{e}\right)^n \leq (n+1)\left(1+\frac{1}{n}\right)^n\left(\frac{n}{e}\right)^n \leq (n+1)e\left(\frac{n}{e}\right)^n,$$

where we use $1 + x \le e^x$ in the last inequality.

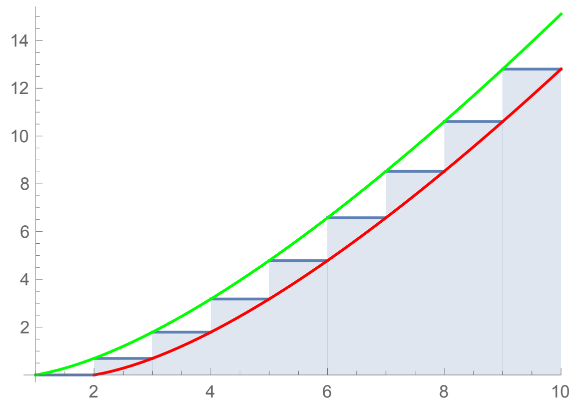

Figure 11.1: Sandwiching $\sum_{k=1}^{n} \ln(k)$ (the blue area) between $\int_1^n \ln(x)dx$ (the area below the red curve) and $\int_1^{n+1} \ln(x)dx$ (the area below the green curve). In the example, $n = 9$.

11.2 Color-coding

Our first example addresses the (directed) k-path problem: we are given a directed graph $G = (V, E)$ and integer k and need to determine whether G has a simple path on k vertices. We'll see two algorithms for this problem.

11.2.1 First algorithm

The first algorithm requires $O^*(k^k)$ time. It relies on the following observation: if G happens to be a DAG, i.e. it has no cycles we can actually determine whether a k-path exists in polynomial time. This it outlined in the following algorithm:

```
Algorithm kpathDAG($G = (V, E), k$)                Assumes $G$ is a directed acyclic graph
Output: Whether there exists a simple path on $k$ vertices in $G$.
1: Find topological ordering $v_1, \dots, v_n$ of $G$   i.e. ordering such that for every $(v_i, v_j) \in E$, $i < j$
2: for $i = 1, \dots, n$ do
3:     if $|N^-(v)| = 0$ then
4:         Set $A[v] = 1$
5:     else
6:         Set $A[v] = \max_{u \in N^-(v)} A[u] + 1$
7: return true iff $\max_{v \in V} A[v] \ge k$
```

The algorithm starts by finding a topological ordering $v_1, \dots, v_n$ of V which has the property that for all edges $(v_i, v_j) \in E$, $i < j$. Now we can use a very simple and fast dynamic programming algorithm: for $v \in V$ let $A[v]$ be the number of vertices of a longest path ending in v. If v has no in-neighbors this is clearly 1. If v has in-neighbors, $A[v] = \max_{u \in N^-(v)} A[u] + 1$ since any path ending at an in neighbor u and be extended with v since v could not have been in this path by the DAG property and any longest path ending in v has an in-neighbor of v as penultimate vertex.

But how does that help for solving k-path in arbitrary directed graphs? The point is we can make the graph acyclic by only keeping a subset of the edges picked in a probabilistic way such that if the original graph had a k-path, this k-path still exists in the new graph with reasonable probability. This is outlined in the following algorithm:

```
Algorithm kpath1($G, k$)                                          $G$ is directed
Output: Whether $G$ has a simple path on $k$-vertices, with constant one-sided error probability.
1: for $i = 1 \dots k^k$ do
2:     Pick for every $v \in V$ a number $c(v) \in \{1, \dots, k\}$ uniformly and independently at random
3:     Let $G' = (V, E')$ where $E' = \{(u, v) \in E : c(v) = c(u) + 1\}\}$
4:     if kpathDAG($G', k$) then return true
5: return false
```

Algorithm 1: $O^*(k^k)$ time randomized algorithm for detecting a k-path.

Let's first look at what happens inside the for-loop. We pick for every vertex a number at most k uniformly and independently at random (e.g. there are no correlations between the numbers in any way) and we only keep edges between vertices that are consecutively numbered. We right away see that G' is indeed acyclic since we cannot arrive back at a number by only increasing the number, and in fact any ordering of V that first puts all vertices with number 1, then vertices with number 2 and so on will be a topological ordering of V.

Now, if $p_1, \dots, p_k$ is a k-path in G, what is the probability it will still be a k-path in G'? Indeed this happens if $c(p_i) = i$ for every $1 \leq i \leq k$ and since the numbers $c(p_i)$ are picked uniformly at random we see

$$\Pr_c[\forall i \in \{1, \dots, k\} : c(p_i) = i] = \left(\frac{1}{k}\right)^k.$$

Thus we see that if G has a k-path, this k-path would be in G' and we would detect it in Line 4 with probability $\left(\frac{1}{k}\right)^k$. Thus, if G would have a k-path, the probability that we will not detect it in the loop at Line 1 is at most $(1 - \frac{1}{k^k})^{k^k} \leq e^{-1}$ using $1 + x \leq e^x$ where $x = \frac{-1}{k^k}$. Thus, if G would have a k-path the algorithm would find it with probability at least $1 - 1/e$.

11.2.2 Second algorithm

Now let's see how to improve this. It seems that the requirement that $\forall 1 \leq i \leq k : c(p_i) = i$ is a bit demanding. Suppose instead we require from our k-path $p_1, \dots, p_k$ that $\{c(p_i) : 1 \leq i \leq k\} = \{1, \dots, k\}$, i.e. all numbers $1, \dots, k$ occur exactly once in the k-path. Often these numbers $1, \dots, k$ are thought of as 'colors' and a k-path having this property is called *colorful.* Sticking to the color perspective the technique we'll going to see now is called *color-coding.*

The probability that a k-path $p_1, \dots, p_k$ is colorful is computed as

$$\Pr_c[\{c(p_i) : 1 \leq i \leq k\} = \{1, \dots, k\}] = \frac{k!(n-k)^k}{k^k(n-k)^k} \geq \frac{\left(\frac{k}{e}\right)^k}{k^k} = \frac{1}{e^k},$$

here the first equality follows by dividing the number of functions c such that $p_1, \dots, p_k$ is colorful by the total number of possible functions, and the inequality uses Stirling's approximation. Thus analogous to `kpath1`, we can use the following algorithm:

```
Algorithm kpath2($G, k$)                                    $G$ is directed
Output: Whether there exists a simple path on $k$ vertices in $G$, with constant one-sided error
    probability.
1: for $i = 1 \dots e^k$ do
2:    Pick for every $v \in V$ a color $c(v) \in \{1, \dots, k\}$ uniformly and independently at random
3:    if colorfulkpath($G, c, k$) then return true
4: return false
```

Algorithm 2: $O^*((2e)^k)$ time randomized algorithm for detecting a k-path.

Assume `colorfulkpath` determines whether G has a colorful k-path with respect to c in $O^*(2^k)$ time. By the above discussion we see that if G has a k-path, then in Line 2 we choose c such that this k-path is colorful with probability $\frac{1}{e^k}$, so the probability that we will not detect it at all is at most $(1 - \frac{1}{e^k})^{e^k}$ which is at most e^{-1} so we will detect any k-path with probability at least $1 - 1/e$.

So we managed to improve this part, but how fast can we detect colorful paths? It is easy to see that when all vertices have distinct colors, this reduces to the NP-complete Hamiltonian path problem so we cannot expect to do this in polynomial time. It turns out however that familiar techniques can be used to solve this problem in $O^*(2^k)$ time.

Let's look at how to apply dynamic programming. Assume we are given a directed graph $G = (V, E)$ and coloring $c : V \to \{1, \dots, k\}$ and for a vertex v and $X \subseteq \{1, \dots, k\}$ define

$$A_v[X] = \begin{cases} \textbf{true} & \text{if } \exists \text{ path } p_1, \dots, p_{|X|} = v \text{ in } G \text{ s.t. } \{c(p_i) : 1 \le i \le |X|\} = X \qquad (11.1) \\ \textbf{false} & \text{otherwise.} \qquad (11.2) \end{cases}$$

Then we see that

$$A_v[X] = \begin{cases} \textbf{true} & \text{if } X = \{c(v)\} \qquad (11.3) \\ \textbf{false} & \text{if } X = \{i\},\ i \ne c(v) \qquad (11.4) \\ \bigvee_{u \in N^-(v) : c(u) \in X \setminus c(v)} A_u[X \setminus c(v)] & \text{otherwise.} \qquad (11.5) \end{cases}$$

We can transform this into a dynamic programming algorithm as we did in Lecture 6:

```
Algorithm colorfulkpath($G = (V, E), c, k$)
Output: Whether $G$ has a colorful simple path on $k$ vertices.
1: for $v \in V$ do
2:    Set $A_v[\{c(v)\}] =$ true, $A_v[\{i\}] =$ false for $i \in \{1, \dots, k\} \setminus c(v)$.
3: for $l = 2$ to $k$ do
4:    for $X \subseteq \{1, \dots, k\}$ such that $|X| = l$ do
5:       for $v \in V$ such that $c(v) \in X$ do
6:          Set $A_v[X] = \bigvee_{u \in N^-(v) : c(u) \in X \setminus c(v)} A_u[X \setminus c(v)]$
```

Since `colorfulkpath` runs in $O^*(2^k)$ time, `kpath2` runs in $O^*((2e)^k)$ time.

11.3 Schöning's algorithm for k-CNF-SAT

Let us get back to the CNF-SAT problem again: we are given a k-CNF-formula and need to determine whether it is satisfiable or not. In homework 2 you were asked to solve 100-CNF-SAT in $O^*((2-\epsilon)^n)$ for some $\epsilon > 0$, which typically led to an $O^*((2^{100}-1)^{n/100})$ time algorithm. Indeed $(2^{100}-1)^{1/100} < 2-\epsilon$, but only for $\epsilon < 10^{-30}$. Now we'll see how to do better using randomization for any (constant) k. In particular we'll see an algorithm solving k-CNF-SAT instances in time $O^*\left(\left(\frac{2k}{k+1}\right)^n\right)$, for any constant k. For $k = 100$ this reduces to about $O^*(1.9802^n)$ time.

If $x, y \in \{0,1\}^n$, we let $H(x,y)$ denote the Hamming distance between x and y, i.e., the number of coordinates in which x and y differ.

11.3.1 Local Search

We start with solving a local search variant of the CNF-SAT problem: given φ and an assignment $x \in \{0,1\}^n$ of the variables of φ, can we change at most d coordinates to get an assignment satisfying the formula? Equivalently, is there an assignment $y \in \{0,1\}^n$ satisfying φ such that $H(x,y) \leq d$? A naïve approach would be to simply try all $\binom{n}{d}$ subsets where x and y could differ and see whether either works out, but we'll now see an algorithm that is quite a bit better for constant k based on clever enumeration of the type seen in Lecture 5:

```
Algorithm localSearch($\varphi, x, d$)     $\varphi$ is a $k$-CNF-formula on $n$ variables, $x \in \{0,1\}^n$, $d \in \mathbb{N}_{\geq 0}$.
Output: Whether there exists $y \in \{0,1\}^n$ that satisfies $\varphi$ and $H(x,y) \leq d$
1: if $d = 0$ and $\varphi(x) =$ false then return false
2: if $\exists$ clause $C_j$ not satisfied by $x$ then
3:     For $i = 1, \dots, \ell = |C_j|$ let $z^i \in \{0,1\}^n$ be the assignment obtained from $x$ by flipping the
4:     variable in the $i$'th literal of $C_j$
5:     return $\bigvee_{i=1}^{\ell}$ localSearch($\varphi, z^i, d-1$)
6: else
7:     return true
```

Algorithm 3: $O^*(k^d)$ time algorithm for detecting a solution of Hamming distance d from x.

For bounding the running time of `localSearch`, we can like usual analyse the recursion tree and let $T(d,k)$ be the number of leaves of this recursion tree. We see that $T(0,k) = 1$ and $T(d,k) = k \cdot T(k, d-1)$ since $\ell \leq k$, so we see $T(d,k) = k^d$. Thus, since we spend only polynomial time per recursive call, indeed this algorithm runs in time $O^*(k^d)$.

But why does it what it promises? If it returns **true** then it is correct: the algorithm only considers assignments of Hamming distance at most d since it decreases d every time it flips changes the assignment and x satisfies φ since there are no unsatisfied clauses. It remains to show that if an assignment y of Hamming distance at most d to x that satisfies φ exists, then the algorithm returns **true**. We'll prove this by induction on d. If $d = 0$, indeed **true** is returned since x satisfies φ (so the condition on Line 2 will not apply). Otherwise, if $d > 0$ and C_j is a clause that is not satisfied by x then let i be such that the i'th literal of C_j is satisfies by y. Since this literal is not satified by x, we know that x and y must assign a different value to the underlying variable. This implies that

$H(z^i, y) \leq d-1$ and by induction the recursive call `localSearch`$(\varphi, z^i, d-1)$ will return **yes**[1].

11.3.2 Using local search as a subroutine for faster k-CNF-SAT.

It's nice to know we can solve the local search version of CNF-SAT fast, but how does that help us to solve the normal CNF-SAT problem? The answer is in the following algorithm:

```
Algorithm kSAT($\varphi$)                    $\varphi$ is a $k$-CNF-formula on $n$ variables
Output: Whether $x \in \{0,1\}^n$ is satisfiable, with constant one-sided error probability.
1: $d = n/(k+1)$     Assume $d$ is integer (otherwise, add at most $k$ dummy variables)
2: for $i = 1 \dots \lceil 2^n/\binom{n}{d} \rceil$ do
3:     Pick $x \in \{0,1\}^n$ uniformly at random
4:     if localSearch($\varphi, x, d$) then return true
5: return false
```

Algorithm 4: Determining whether a k-CNF formula is satisfiable in $O^*\left(\left(\frac{2k}{k+1}\right)^n\right)$ time, for any constant k.

Let's check whether this algorithm does what it promises: since `localSearch` only returns **true** when it found a satisfying solution, the same holds for this algorithm. Now suppose φ is satisfied by some variable assignment $y \in \{0,1\}^n$. At Line 3 we pick $x \in \{0,1\}^n$ uniformly at random, and we have seen that `localSearch`$(\varphi, x, d) =$ **true** if $H(x,y) \leq d$. So let's look at the probability that this happens:

$$\Pr_x[H(x,y) \leq d] = \frac{\sum_{i=0}^{d} \binom{n}{i}}{2^n} \geq \frac{\binom{n}{d}}{2^n}.$$

So in one single iteration we detect y if it exists with probability at least $\binom{n}{d}/2^n$, so the probability that we will not detect such a y in the whole loop of Line 2 is at most $(1-\binom{n}{d}/2^n)^{2^n/\binom{n}{d}} < 1/e$.

Now let's look at the running time. Note that we add at most k variables in Line 1 but since k is a constant, $O^*\left(\left(\frac{2k}{k+1}\right)^{n+k}\right) = O^*\left(\left(\frac{2k}{k+1}\right)^{n}\right)$, so there's no need to worry about the increase of the number of variables. The running time then is easily seen to be $O^*(k^d 2^n/\binom{n}{d})$, so let's try to simplify that expression. Stirling tells us that $(\frac{n}{e})^n \leq n! \leq 3n(\frac{n}{e})^n$, and using $\binom{n}{d} = \frac{n!}{d!(n-d)!}$ we see that

$$\binom{n}{d} = \frac{n!}{d!(n-d)!} \geq \frac{(n/e)^n}{3n(d/e)^d 3n((n-d)/e)^{n-d}} = \frac{n^n}{d^d(n-d)^{n-d}9n^2}.$$

Plugging this in the running time and expanding $d = n/(k+1)$ we see

$$k^d \frac{2^n}{\binom{n}{d}} = \frac{2^n (kd)^d (n-d)^{n-d} 9n^2}{n^n} = \frac{2^n \left(n\frac{k}{k+1}\right)^d \left(n\frac{k}{k+1}\right)^{n-d} 9n^2}{n^n} = O^*\left(\left(\frac{2k}{k+1}\right)^n\right),$$

so indeed we found a $O^*\left(\left(\frac{2k}{k+1}\right)^n\right)$ time algorithm for k-CNF-SAT, for any constant k.

[1]Let us remark here that one can also give and analyse an algorithm that is more close to Papadimitriou's 2SAT algorithm, but it's analysis is slightly more involved (`https://www.ics.uci.edu/~eppstein/280e/001121.pdf`)

11.4 Feedback Vertex Set

Let's look at yet another problem we have seen before: the Feedback Vertex Set problem. Recall:

Definition 11.1. *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.

We'll look at the problem on multigraphs, as we also did before. Recall that we said that in a multi-graph the edge set may be a multiset so two edges vertices might share parallel edges (two of which already form a cycle). Moreover, this time we'll also encounter loops: these are edges of the type (v, v) for some vertex v. And loops also form a cycle on their own already. As before we also use degree of a vertex v for the number of edges incident to v (which counts parallel edges and loops as well).

Before we get into the algorithm, we need the following graph-theoretic observation:

Lemma 11.1. *Let G be a multigraph on n vertices with minimum degree at least 3. Then, for every feedback vertex set X of G, more than half of the edges of G have at least one endpoint in X.*

Proof. Let $F = V \setminus X$ (note we call it F because $G[F]$ needs to be a forest since X is a FVS). Let $E_F = E \cap (F \times F)$ be the edges inside the forest and let $I = \{(u, v) \in E : u \in X \vee v \in X\}$. We see that E_F and I partition E (i.e. $E_F \cup I = E$ and $E_F \cap I = \emptyset$) so $|E_F| + |I| = |E|$. Thus, since the lemma claims that $|I| \geq |E|/2$, it is sufficient to show that $|I| \geq |E_F|$.

First, since F is a forest, we know that $|E_F| \leq |F|$ (i.e. a forest cannot have more edges than vertices since we can reduce it to a graph with no edges by removing leaves along with their incident edge). Let us partition F into sets $F_{\leq 1}, F_2, F_{\geq 3}$ which are all vertices of F with respectively degree at most 1, exactly 2 and at least 3 in G. We have that $|F_{\geq 3}| < |F_{\leq 1}|$. To see this, note we can root the tree and replace subtrees rooted at degree ≥ 3 vertices with only degree ≤ 2 below it with leafs until we are no degree ≥ 3 vertices left; at every step we remove 1 degree ≥ 3 vertex and reduce the number of leaves by at least 1.

Now we can lower bound the number of edges with one endpoint in F and one end point in X (and all of these edges are in I) using the assumption that every vertex has degree at least 3; we have that

$$|I| \geq 2|F_{\leq 1}| + |F_2| \geq |F_{\leq 1}| + |F_2| + |F_{\geq 3}| = |F| \geq |E_F|,$$

where we use $|F_{\geq 3}| < |F_{\leq 1}|$ in the second inequality. □

Now let's think about how to use this lemma for an algorithm. Using reduction rules similar to what we did in our previous algorithm for FVS, we'll see that indeed we can make sure that our graph has minimum degree 3 by using some reduction rules. Lemma 11.1 tells us that if this is the case and X is a FVS of G, we can pick an edge uniformly at random, then pick an endpoint of that edge uniformly at random and we end up picking a vertex that is in X with probability at least 1/4. This motivates some kind of bounded search tree algorithm, using randomization:

```
Algorithm FVSpoly($G, k$)                                                  $G$ may be a multigraph
Output: false if $G$ has no FVS of size at most $k$, true with probability at least $4^{-k}$ otherwise
 1: if $k = 0$ then return true if $G$ is a forest, return false otherwise
 2: if $\exists (v, v) \in E$ then
 3:     return FVSpoly($G \setminus v, k-1$)            the only way to hit a loop $(v, v)$ is to include $v$
 4: if $\exists v \in V$ such that $\deg(v) \leq 1$ then
 5:     return FVSpoly($G \setminus v, k$)            seen in L5 as well, $v$ is not on any cycle so irrelevant
 6: if $\exists v \in V$ such that $d(v) = 2$ with neighbors $u, w$ then
 7:     Let $G'$ be the graph obtained by removing $v$ and adding an edge $(u, w)$
                     variant seen in L5, $u = w$ if parallel edges, every cycle containing $v$ includes $u, w$
 8:     return FVSpoly($G', k$)            so in every FVS $v$ can be replaced with $u$ or $w$
 9: Pick an edge $e$ of $G$ uniformly at random
10: Pick an endpoint $v$ from $e$ uniformly at random
11: return FVSpoly($G \setminus v, k-1$)
```

The reduction rules are familiar to what we have seen in L5: for the rule on Line 6 we note that any FVS containing v can be altered by replacing v with either u or w so we can safely make the decision to not include v. Note that if $u = w$, the algorithm adds a loop (u, u)

If the algorithm returns **true**, it can be easily verified that it has found a FVS (we will not go through all steps here again). On the other hand, if there exists a FVS X of size at most k, we claim it returns **true** with probability at least 4^{-k}. We can prove this using induction on k. If $k = 0$ this is trivial, and if any of the reduction rules apply we can directly use the induction hypothesis since we already have seen the reduction rules are correct. Otherwise, we pick a random edge and then a random endpoint v of this edge. By Lemma 11.1 and the argumentation below it, we see that $\Pr_v[v \in X] \geq 1/4$, and conditioned on this event $\Pr[\texttt{FVSpoly}(G \setminus v, k-1) = \textbf{true}] \geq 4^{k-1}$ so indeed we return true with probability at least 4^{-k}.

Now we can transform this algorithm into an exponential time algorithm with constant one-sided error probability by the standard boosting technique:

```
Algorithm FVS2($G, k$)
Output: Whether there exists a simple path on $k$-vertices in $G$, with one-sided error probability.
 1: for $i = 1 \dots 4^k$ do
 2:     if FVSpoly($G, k$) then return true
 3: return false
```

Algorithm 5: $O^*(4^k)$ time randomized algorithm for detecting a k-path.

Again the one-sided error probability addresses that the algorithm return **true** if a solution exists and it returns **false** in this case with probability at most $(1 - 4^{-k})^{4^k} < 1/e$ so it return **true** with probability at least $1 - 1/e$.

11.5 Exercises

Note: In all the exercises below your algorithms may have one-sided constant error probability.

Excercise 11.1. A triangle in an undirected graph $G = (V, E)$ is a triple of vertices (u, v, w) with a $(u, v), (v, w), (u, w) \in E$. A k-triangle-packing is a collection of triangles $T_1, \ldots, T_k$ that are mutually disjoint.

- Give an $O^*(k^{O(k)})$ time algorithm that given (G, k) determines whether G has a k-triangle packing. Hint: assign numbers $c(v) \in \{1, \ldots, 3k\}$ uniformly and independently at random to every vertex v and look for triangle packings where all triangles (u, v, w) satisfy $c(w) = c(v) + 1 = c(u) + 2 = 3i$ for different integers i.
- Give an $O^*(2^{O(k)})$ time algorithm that given (G, k) determines whether G has a k-triangle packing. Hint: assign numbers $c(v) \in \{1, \ldots, 3k\}$ uniformly and independently at random and look for colorful triangle packings using dynamic programming or Inclusion-Exclusion.

Excercise 11.2. Find an algorithm that, given a directed acyclic graph, finds a topological ordering in polynomial time.

Excercise 11.3. Change Line 1-3 in Algorithm `kpath1` to get an algorithm running in time $O^*(k!)$.

Excercise 11.4. Graphs $G = (V, E), P = (W, F)$ are isomorphic if there exists a bijection $\pi : V \leftrightarrow W$ such that $(u, v) \in E$ if and only if $(\pi(u), \pi(v)) \in F$. In this exercise we assume P is connected.

- Suppose that $|P| = k$. Show that we can determine in $O^*(k!)$ time whether P is isomorphic to one of the connected components of G.

In the Subgraph Isomorphism problem, we are given graphs $G = (V, E), P = (W, F)$ and are asked whether P (the 'pattern') is isomorphic to subgraph of G (recall that a subgraph is obtained by removing vertices and edges).

- Show that the Subgraph Isomorphism problem is NP-complete by picking a specific n-vertex graph P for every n.

Suppose G has maximum degree d.

- Suppose that P is isomorphic to a subgraph of G and G' is obtained from G by removing every edge with probability $1/2$. Lower bound the probability that P is isomorphic to a connected component of G'.
- Show how to solve the Subgraph Isomorphism problem in $O^*(2^{kd}k!)$ time, if $|P| = k$ and G has maximum degree d.

Excercise 11.5. Suppose we would like to solve an instance $(G = (V, E), k)$ of FVS where $|V| = n$ and $k = n/2$. We can directly run the $O^*(4^k)$ time algorithm but that is not too impressive. In this question we try to use the $O^*(4^k)$ time algorithm in this setting in a more clever way.

- Why is running the $O^*(4^k)$ time algorithm not too impressive?
- Consider the following problem: given an instance $(G = (V, E), k)$ of FVS and a set W, does G have a FVS X such that $|X| \leq k$ and $W \subseteq X$. Use the $O^*(4^k)$ time algorithm to solve this problem in $O^*(4^{k-|W|})$ time.
- Suppose W is picked uniformly at random from the set of all $(n/4)$-sized subsets of V [2]. If X is a FVS of size $n/2$, lower bound the probability that $W \subseteq X$. Use that $\binom{x}{x/2} \geq 2^x/x$
- Give an algorithm that determines whether a FVS of size $k = n/2$ exists in time $O^*(\binom{n}{n/4}) \leq O^*(2^{.82n})$.

Excercise 11.6. Solve the k-path problem in $O^*((2e)^k)$ time and polynomial space using Inclusion-Exclusion. Hint: To look for a colorful k-path, use as universe the set of all walks with $k - 1$ edges (e.g. on k vertices).

[2]this can be easily done in polynomial time, but we will not concern ourselves with this technical issue here