2MMD30: Graphs and Algorithms Lecture 6 by Jesper Nederlof, 26/02/2016

Dynamic Programming and Inclusion-Exclusion.

Dynamic programming is a very powerful techniques that you have probably seen already several times in previous courses. Together with clever enumeration, it is also one of the most basic techniques for designing exponential time algorithms for NP-hard problems. Recall that the idea of dynamic programming is the following: to solve a given problem, we define subproblems in a clever way, and gradually use solutions of easier subproblems to solve the harder subproblems until we have solved the hardest subproblem which is equal to the original problem. Specifically, dynamic programming algorithms work with a very large table of data stored in memory, and iteratively compute table entries out of previously computed table entries. The procedure that computes new table entries from old ones is often very easy and that is why it is often convenient to formalize a dynamic programming algorithm with a recurrence.

6.1 Subset Sum and Knapsack

Let us proceed with the probably cleanest application of dynamic programming, which happens to bring us to the Subset Sum problem again. Given an instance consisting of positive integers $w_1, \ldots, w_n$ and integer t we define table entries as follows: for $0 \leq i \leq n$ and $0 \leq j \leq t$, define $A[i,j]$ to be **true** if $\exists X \subseteq \{1, \ldots, i\}$ such that $\sum_{e \in X} w_e$, and **false** otherwise.

Lemma 6.1. *The following recurrence holds:*

$$A[i,j] = \begin{cases} \textbf{false} & \text{if } i = 0, j > 0, \quad (6.1) \\ \textbf{true} & \text{if } i = 0, j = 0, \quad (6.2) \\ A[i-1,j] & \text{if } i > 0,\ j < w_i, \quad (6.3) \\ A[i-1,j] \vee A[i-1, j-w_i] & \text{otherwise.} \quad (6.4) \end{cases}$$

Proof. Cases 6.1 and 6.2 are trivial. In Case 6.3 any subset $X \subseteq \{1, \ldots, i\}$ with $\sum_{e \in X} w_e = j$ cannot contain i since all integers are positive, and in Case 6.4 we continue searching for subsets X avoiding i (in which case $X \subseteq \{1, \ldots, i-1\}$ and $\sum_{e \in X} w_e = j$) and subsets X containing i (in which case $X \setminus i \subseteq \{1, \ldots, i-1\}$ satisfies $\sum_{e \in X \setminus i} w_e = j - w_i$). □

Now, we can use the recurrence to compute $A[n,t]$, and by definition the seeked set X exists if and only if $A[n,t] = \textbf{true}$. Algorithm 1 implements this, and solves Subset Sum in $O(nt)$ time. Note that this is *not* polynomial time since t could be exponential in the input size[1].

[1] Sometimes it is therefore referred to as *pseudo*-polynomial time.

Algorithm `sss`$(w_1, \ldots, w_n, t)$
Output: Whether there exists $X \subseteq \{1, \ldots, n\}$ with $\sum_{e \in X} w_e = t$.

```
1: A[0,0] ← true
2: for j = 1 to t do
3:    A[0,j] ← false
4: for i = 1 to n do
5:    for j = 1 to t do
6:       if  j < w_i then
7:          A[i,j] ← A[i−1,j]
8:       else
9:          A[i,j] ← A[i−1,j] ∨ A[i−1,j−w_i]
10: return A[n,t]
```

Algorithm 1: $O(n \cdot t)$ time algorithm for Subset Sum.

6.1.1 Approximation Scheme for Subset Sum

Consider the following optimization variant of Subset Sum: we are given integers $w_1, \ldots, w_n, t$ and now the goal is to find a subset X such that $\sum_{e \in X} w_e$ is maximized while $\sum_{e \in X} w_e \leq t$. Note we can still solve this problem using Algorithm `sss` by finding the largest $j \leq t$ such that $A[n, j] =$ **true**. Let us refer to the algorithm doing this as `optsss`.

This time, we are willing to settle for an approximation in this problem: we are additionally given a real number $\epsilon > 0$, and we only need to find a solution X' with $\sum_{e \in X'} w_e \leq t$ that is guaranteed to be at least $(1 - \epsilon)$ times the optimum. More formally, if there exists a subset $X \subseteq \{1, \ldots, n\}$ such that $\sum_{e \in X} w_e = OPT \leq t$ we need to find a subset X' such that $\sum_{e \in X'} w_e \geq (1 - \epsilon)OPT$. We'll give an algorithm that does this in time $O(n^3/\epsilon)$. To this end, first suppose that we have an instance $w_1, \ldots, w_n, t$ such that all integers are divisible by an other integer S. Then clearly $X \subseteq \{1, \ldots, n\}$ is an optimal solution of the instance $w_1, \ldots, w_n, t$ if and only if it is an optimal solution of the instance $\frac{w_1}{S}, \ldots, \frac{w_n}{S}, \frac{t}{S}$. If such a divisor S exists and it would be large, that would be very useful since then we could find the optimal X in time $O(nt/S)$ by dividing all integers by S and applying Algorithm 1. But of course, in our pessimistic worst-case scenario we cannot expect this to happen. However, because we are willing to settle for an approximation, this observation is actually highly useful when we just round our integers to the nearest multiple of S.

Algorithm `apxsss`$(w_1, \ldots, w_n, t, \epsilon)$
Output: $OPT' \geq (1 - \epsilon)OPT$, where $OPT = \max\{\sum_{e \in X} w_e : X \subseteq \{1, \ldots, n\} \wedge \sum_{e \in X} w_e \leq t\}$

```
1: if Σ_{i=1}^n w_i < t then return Σ_{i=1}^n w_i
2: for i = 1,...,n do
3:    if w_i > t then remove w_i from the instance
4: let w_max = max_i w_i, let S = ε·w_max / n
5: for all i = 1,...,n do
6:    w'_i = S ⌈w_i / S⌉
7: t' = S ⌊t / S⌋
8: return S · optsss(w'_1/S, ..., w'_n/S, t'/S)
```

Algorithm 2: Approximation scheme for Subset Sum.

Let's first look at the running time: all steps before Line 8 take $O(n)$ time and the call to `optsss` takes $O(nt/S)$ time. By Line 1 we know that $t \leq nw_{\max}$ so $nt/S \leq n^2 w_{\max}/S \leq n^3\epsilon$, and thus the algorithm runs in time $O(n^3/\epsilon)$.

Now let's see whether it indeed does what it promises. Let X' be the subset reaching the maximum in the instance $w'_1/S, \ldots, w'_n/S, t'/S$ and let X be the subset reaching the maximum in the instance $(w_1, \ldots, w_n, t)$. Then since $w_i \leq w'_i$ we have that

$$OPT' = \sum_{i \in X'} w_i \leq \sum_{i \in X'} w'_i \leq t' \leq t.$$

So indeed `apxsss` returns something respecting the constraint. On the other hand

$$\begin{aligned}
OPT = \sum_{i \in X} w_i &\leq \sum_{i \in X} w'_i && && w_i \leq w'_i \\
&\leq \sum_{i \in X'} w'_i && && X' \text{ is optimal in the instance } w'_1, \ldots, w'_n, t' \\
&\leq \sum_{i \in X'} (w_i + S) && && w'_i \text{ is } w_i \text{ rounded to least higher multiple of } S \\
&\leq nS + OPT' && && |X'| \leq n \\
&\leq \epsilon \cdot w_{\max} + OPT' && && S = \tfrac{\epsilon \cdot w_{\max}}{n} \\
&\leq \epsilon OPT + OPT'. && && OPT \geq w_{\max},
\end{aligned}$$

so indeed $OPT' \geq (1-\epsilon)OPT$.

6.1.2 Knapsack

In the Knapsack problem we are given integers $w_1, \ldots, w_n, W$ and $v_1, \ldots, v_n$ and are asked to find a subset $X \subseteq \{1, \ldots, n\}$ that maximizes $\sum_{e \in X} v_e$ under the constraint $\sum_{e \in X} w_e \leq W$. For $0 \leq i \leq n$ and $0 \leq j \leq W$, define $A[i,j]$ to be $\max\{\sum_{e \in X} v_e : X \subseteq \{1, \ldots, i\} \wedge \sum_{e \in X} w_e \leq j\}$. Then, similarly to the previous section we see that

$$A[i,j] = \begin{cases} 0 & \text{if } i = 0, \quad (6.5) \\ A[i-1,j] & \text{if } i > 0,\ j < w_i, \quad (6.6) \\ \max\{A[i-1,j], v_i + A[i-1, j-w_i]\} & \text{otherwise.} \quad (6.7) \end{cases}$$

And similarly to the previous subsection this can be used to solve the Knapsack problem in $O(n \cdot W)$ time, since the answer can be read off from $A[n, W]$.

6.2 Coloring

Let us get back to graphs now and see how we can use dynamic programming here. In the Graph Coloring problem we are given a graph $G = (V, E)$ and need the determine the minimum k such that G admits a k-coloring. We'll solve this problem in $O^*(3^n)$ here.

It is easy to see that G has a k-coloring if and only if there exists a partition of V into at most k independent sets, e.g., there exists k independent sets $I_1, \ldots, I_k \subseteq V$ such that for every

vertex $v \in V$, $v \in I_i$ for exactly one i. This is of course because in any k-coloring all vertices of one particular color form an independent set, and an assignment in which all color classes are independent sets is always a coloring. Based on this we have the following natural definition of subproblems:

$$A_k[X] = \begin{cases} \textbf{true}, & \text{if } G[X] \text{ has a } k\text{-coloring} \\ \textbf{false}, & \text{otherwise.} \end{cases}$$

And the recurrence suggests itself:

$$A_k[X] = \begin{cases} \textbf{true}, & \text{if } k = 1,\ X \text{ is an independent set of } G \quad (6.8) \\ \textbf{false}, & \text{if } k = 1,\ X \text{ is not an independent set of } G \quad (6.9) \\ \bigvee_{Y \subseteq X} A_{k-1}[X \setminus Y] \wedge A_1[Y], & \text{otherwise.} \quad (6.10) \end{cases}$$

To see this, note that G is 1-colorable if and only if it is an independent set. For $k > 1$, we try all possible sets of vertices receiving color k in Case 6.10: if $Y \subseteq V$ receives color k then $G[Y]$ needs to be 1-colorable (or equivalently, an independent set), and $G[X \setminus Y]$ needs to be $k-1$ colorable. Moreover, if both conditions hold, we clearly have a k-coloring.

Algorithm `kcolor`$(G = (V, E), k)$
Output: Whether G is k-colorable.

```
1: for X ⊆ V do
2:    if X is an independent set of G then set A_1[X] = true else set A_1[X] = false
3: for l = 2 to k do
4:    for X ⊆ V do
5:       set A_l[X] = false
6:       for Y ⊆ X do
7:          if A_{l-1}[X \ Y] ∧ A_1[Y] then set A_l[X] = true
8: return A_k[V]
```

Algorithm 3: An $O^*(3^n)$-time algorithm for k-coloring

6.3 Traveling Salesman Problem

Some notation It will be convenient to introduce some notation here. This may look like overkill now, but it will be useful later in this lecture and also in later lectures. Let $G = (V, E)$ be a graph (that may or may not be directed) and let $\omega : E \to \mathbb{N}$ be weights of the edges. For a subset $X \subseteq E$ as parameter ω is naturally extended to $\omega(X) = \sum_{e \in X} \omega(e)$. A *walk of* $G = (V, E)$ is a sequence $P = (p_1, \ldots, p_\ell) \in V^\ell$ such that $(p_i, p_{i+1}) \in E$ for $i = 1, \ldots, \ell - 1$. A *path* is a walk $P = (p_1, \ldots, p_\ell) \in V^\ell$ such that $p_i \neq p_j$ for every i, j. We say P is *from* s *to* t if $p_1 = s$ and $p_\ell = t$.

A *cyclic walk of* $G = (V, E)$ is a walk $P = (p_1, \ldots, p_\ell, p_1) \in V^\ell$. A *cycle* is a cyclic walk $P = (p_1, \ldots, p_\ell, p_1) \in V^\ell$ such that $p_i \neq p_j$ for every $i, j \leq \ell$.

The set of vertices visited by a (cyclic) walk, path or cycle P (i.e., the set obtained by ignoring the order of P and removing possible copies) is denoted by $V[P]$, and P *avoids* X if $V[P] \cap X = \emptyset$. The length of a (cyclic) walk, path or cycle is the number of vertices in the sequence minus 1 (e.g., the size of $E[P]$).

The algorithm With this terminology, note that the traveling salesman problem asks to find a cycle C of G such that $V[C] = V$ and $\omega(E[C])$ is minimized. We now see how this problem can be solved in $O^*(2^n)$ time with dynamic programming in directed graphs (which generalizes the problem for undirected graphs by adding directed edges in both directions). To this end, pick a vertex $s \in V$ arbitrarily and define for $X \subseteq V \setminus s$ and $t \in X$:

$$A_t[X] = \min\{\omega(E[P]) : P \text{ is a path from } s \text{ to } t, \text{ and } V[P] \setminus s = X\}.$$

Letting $N^-(t)$ denote the set of in-neighbors of t and using the standard convention that the minimum of an empty set is ∞, we have that for all $X \subseteq V \setminus s$ and $t \in X$:

$$A_t[X] = \begin{cases} \infty, & \text{if } |X| = 1 \text{ and } (s,t) \notin E, \quad (6.11) \\ \omega(s,t), & \text{if } |X| = 1 \text{ and } (s,t) \in E, \quad (6.12) \\ \min\limits_{t' \in N^-(t) \cap X} A_{t'}[X \setminus t] + \omega(t',t), & \text{otherwise.} \quad (6.13) \end{cases}$$

Let us now motivate this recurrence. For Case 6.11 and Case 6.12, note the path (s,t) is the only relevant path so $A_t[X]$ equals the weight of the edge (which is ∞ if it is absent). For Case 6.13, we prove the equality by splitting it in two inequalities:

LHS $\geq$ RHS: If $P = p_1, \ldots, p_\ell$ is a path from s to t and $V[P] \neq s$, then $P' = (p_1, \ldots, p_{\ell-1})$ is a path from s to $p_{\ell-1} = t' \in N^-(t)$ and the $\omega(E[P]) = \omega(E[P']) + \omega(t',t)$, so the expression in Case 6.13 lower bounds $A_t[X]$.

LHS $\leq$ RHS: $P' = (p_1, \ldots, p_{\ell-1})$ is a path from s to $p_{\ell-1} = t' \in N^-(t)$ visiting $X \setminus t$, then $P = (p_1, \ldots, p_{\ell-1}, t)$ is a path visiting X.

From the definition of $A_t[X]$, we see that the asked minimum is $\min_{t \in N^-(V)} A_t(X) + \omega(t,s)$, so indeed we can use the above recurrence to get the claimed running time. For completeness, let us now outline the algorithm in pseudo-code:

Algorithm `tsp`$(G = (V,E), \omega)$
Output: The minimum $\omega(E[C])$ over all cycles of G (e.g., cycles C satisfying $V[C] = V$).
1: Pick an arbitrary vertex $s \in V$
2: Initiate a table $A_t[X]$ for every $X \subseteq V \setminus s$ and $t \in X$.
3: Set $A_t[\{t\}] = \omega(s,t)$ for every $t \in V$, where $\omega(s,t) = \infty$ if $(s,t) \notin E$.
4: **for** $i = 2$ to n **do**
5: **for** $X \subseteq V$ such that $|X| = i$ **do**
6: Set $A_t[X] = \min_{t' \in N^-(t) \cap X} A_{t'}[X \setminus t] + \omega(t',t)$.
7: **return** $\min_{t \in N^-(V)} A_t[X] + \omega(t,s)$.

Algorithm 4: An $O^*(2^n)$-time algorithm for TSP

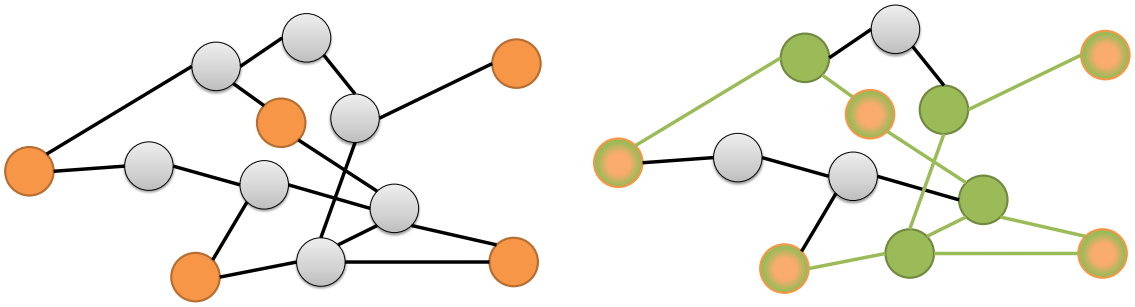
Note that indeed this gives us the minimum weight of an Hamiltonian cycle since s will be on this Hamiltonian cycle and we go over all possibilities of the vertex before s on this Hamiltonian cycle in Line 7.

6.4 Steiner Tree

In the Steiner Tree problem we are given a graph $G = (V,E)$ and set of terminals T and we need to find a set $S \subseteq V$ of minimum size with $T \subseteq S$ such that $G[S]$ is connected. This problem is

called Steiner tree since it is equivalent to finding a tree (S, E') where $E' \subseteq E$ and $T \subseteq S \subseteq V$.

Example 6.1. *In Figure 6.1 we see an example of an instance of Steiner Tree along with a solution.*



(a) The graph G, the terminals T are marked orange. (b) A solution S (the green vertices) of size 9.

Figure 6.1: An example of an instance of Steiner tree with a solution

Now let us define the subproblems. Similarly to the previous example, the subproblems are again closely related to a smaller instance of the original problem:

$$A_{r,j}[X] = \begin{cases} \textbf{true} & \text{if } \exists S \subseteq V : |S| \leq j \wedge X \cup r \subseteq S \wedge G[S] \text{ is connected.} \quad (6.14) \\ \textbf{false} & \text{otherwise.} \quad (6.15) \end{cases}$$

Indeed, the minimum j such that $A_{r,j}[X] = \textbf{true}$ is just the solution to the instance of Steiner Tree with terminal set $X \cup r$. The recurrence this time looks as follows:

$$A_{r,j}[X] = \begin{cases} \textbf{true} & \text{if } j \geq 1,\ X \subseteq \{r\} \quad (6.16) \\ \textbf{false} & \text{if } j = 1,\ X \not\subseteq \{r\} \quad (6.17) \\ \bigvee_{Y \subseteq X} \bigvee_{r' \in N(r)} \bigvee_{j'=1}^{j} A_{r',j'}[Y] \wedge A_{r,j-j'}[X \setminus Y] & \text{otherwise.} \quad (6.18) \end{cases}$$

For Case 6.16 and Case 6.17, the only set of interest is $S = \{j\}$ which is trivially connected. For Case 6.18, if $j > 1$ we prove the equivalency by two implications

LHS $\rightarrow$ **RHS** If S as in (6.14) exists, we know it contains r and, since $X \not\subseteq \{r\}$, a neighbor r' of r. Then, S can be partitioned into S_1 and S_2 such that $G[S_1]$ and $G[S_2]$ are connected. Picking $Y = S_1 \cap X$ and $j' = |S_1|$ we see that the right hand side evaluates to **true**.

LHS $\leftarrow$ **RHS** If S_1 establishes $A_{r',j'}[Y]$ and S_2 establishes $A_{r,j-j'}[X \setminus Y]$, $S_1 \cup S_2$ establishes $A_{r,j}[X]$.

Thus the recurrence is correct and similar as in the previous sections we can use it to compute $A_{l,t}[T]$ where t is an arbitrary terminal by the definition of $A_{l,r}[T]$ we see that the minimum l such that $A_{l,t}[T] = \textbf{true}$ is the optimal value.

6.5 Inclusion / Exclusion

In a nutshell, inclusion-exclusion is a way of computing a union through expressing it as a number of intersections. This can be useful since in many settings it is easier to compute or reason about

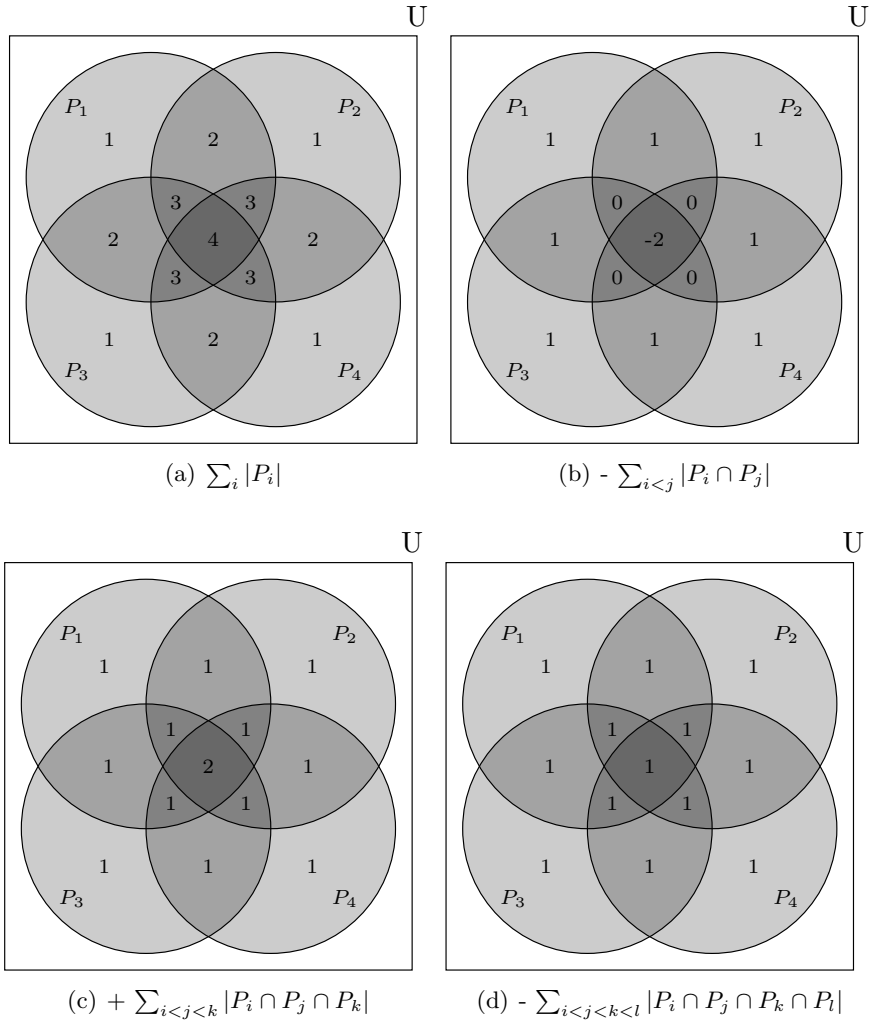
these intersections instead. Probably familiar examples are

$$(a)\ |P_1 \cup P_2| = |P_1| + |P_2| - |P_1 \cap P_2| \qquad (b)|P_1 \cap P_2| = |P_1| + |P_2| - |P_1 \cup P_2| \tag{6.19}$$

Let's look at the case with 4 sets:

Example 6.2. *Suppose we are given 4 subsets $P_1, \ldots, P_4$ of a set U which are illustrated in Venn-diagram Figure 6.2. For notational ease, assume $A_i = \emptyset$ for $i < 1$ and $i > 4$. We want to express $|\bigcup_i P_i|$, without using any union. First, in (a) we sum over all sets, counting elements of frequency i exactly i times. To compensate, we subtract the size of the intersection between each pair of sets in (b), subtracting elements of frequency i exactly $\frac{i*(i-1)}{2}$ times. After adding all intersections of 3 sets and subtracting elements in 4 sets, every element in one of the sets is exactly counted once:*

$$|\bigcup_i P_i| = \sum_i |P_i| - \sum_{i<j} |P_i \cap P_j| + \sum_{i<j<k} |P_i \cap P_j \cap P_k| - \sum_{i<j<k<l} |P_i \cap P_j \cap P_k \cap P_l|.$$



(a) $\sum_i |P_i|$ (b) - $\sum_{i<j} |P_i \cap P_j|$

(c) + $\sum_{i<j<k} |P_i \cap P_j \cap P_k|$ (d) - $\sum_{i<j<k<l} |P_i \cap P_j \cap P_k \cap P_l|$

Figure 6.2: An example of an inclusion-exclusion formula

In this section, it is convenient to think of sets P_i as *properties* of elements in U, e.g., all elements of U in P_i have property P_i. Traditionally inclusion exclusion is formulated as above, but when we actually want to use it we are often looking for objects that have many properties simultaneously, so it is often convenient to express the intersection of the sets as the intersection of the *complement* of these sets. Note that this is just a notational change since $|\bigcap_i P_i| = |U \setminus \bigcup_i \overline{P_i}|$.

The *inclusion-exclusion formula* then reads as follows:

Theorem 6.1. *Let U, let $P_1, \ldots, P_n \subseteq U$. Using $\overline{P_i}$ to denote $U \setminus P_i$, and with the convention*[2] *$\cap_{i\in\emptyset}\overline{P_i} = U$, we have:*

$$|\bigcap_{i=1}^{n} P_i| = \sum_{F\subseteq\{1,\ldots,n\}} (-1)^{|F|} |\bigcap_{i\in F} \overline{P_i}|. \tag{6.20}$$

Proof. Let $e \in U$ and let $p(e) \subseteq \{1, \ldots, n\}$ be the indices of the properties e has, e.g., $p(e) = \{i \in \{1, \ldots, n\} : e \in P_i\}$. We see that we can rewrite the RHS of (6.20) as follows

$$\begin{aligned}\sum_{F\subseteq\{1,\ldots,n\}} (-1)^{|F|} |\bigcap_{i\in F} \overline{P_i}| &= \sum_{F\subseteq\{1,\ldots,n\}} (-1)^{|F|} |\{e \in U : p(e) \cap F = \emptyset\}| \\ &= \sum_{e\in U} \sum_{F\subseteq\{1,\ldots,n\}\setminus p(e)} (-1)^{|F|}.\end{aligned}$$

Now note that for any set X, $\sum_{Y\subseteq X}(-1)^{|Y|}$ is 1 if $X = \emptyset$ and 0 otherwise. For $X = \emptyset$ this follows directly, while for $|X| > 0$ this follows from the binomial theorem or the fact that every non-empty set has equally many even as odd subsets.

Thus, any summand in the last turn of the derivation if 1 is $p(e) = \{1, \ldots, n\}$, so the whole term equals $|\bigcap_{i=1}^{n} P_i|$. □

6.6 Hamiltonian cycle in $O^*(2^n)$ time and polynomial space

To illustrate the use of Theorem 6.1, we will look at Hamiltonian cycle problem now[3]. Given a directed graph $G = (V, E)$, an *Hamiltonian cycle* is a cycle of length n. Note that a cyclic walk C is an Hamiltonian cycle if and only if it is of length n and visits all vertices in the sense that $V[C] = V$. Thus we may also focus on finding a cyclic walk of length n visiting all vertices. In fact we are going to count the number of such cyclic walks using inclusion-exclusion.

Let $V = \{1, \ldots, n\}$. Apply Theorem 6.1, where U is the set of all cyclic walks of length n of G; for every $1 \leq i \leq n$ let P_i be all cyclic walks of length n of G that visit i. In this specific case, Equation (6.20) boils down to:

$$\#\text{ cyclic walks of } G \text{ of length } n \text{ visiting all vertices} = |\bigcap_{i=1}^{n} P_i| = \sum_{F\subseteq\{1,\ldots,n\}} (-1)^{|F|} |\bigcap_{i\in F} \overline{P_i}|. \tag{6.21}$$

[2] Just as $\sum_{i\in\emptyset}, \prod_{i\in\emptyset}, \min_{i\in\emptyset}$ commonly denote $0, 1$ and ∞, respectively

[3] We'll see here an old algorithm for Karp from 1962 that runs in $O^*(2^n)$ time. Six years ago, in a breakthrough result by Björklund, a randomized $O^*(1.69^n)$ time algorithm was given (see `https://www.youtube.com/watch?list=PLn0nrSd4xjjbIHhktZoVlZuj2MbrBBC_f&v=httHBboc6wY` for a nice popular science video on this result). We'll cover this in a later lecture.

Since we have $O^*(2^n)$ time, we can afford to go over all summands in the right hand side, but how do we compute $|\bigcap_{i\in F}\overline{P_i}|$? We see $|\bigcap_{i\in F}\overline{P_i}|$ equals the number of cyclic walks of length n avoiding F which equals the number of cyclic walks of length n in $G[V\setminus F]$, and this number can be computed in polynomial time as follows.

For $s,t\in V\setminus F$ and an integer k, let $w_F(s,t,k)$ be the number of walks from s to t of length k avoiding F, then

$$w_F(s,t,k)=\begin{cases}0 & \text{if } k=0 \text{ and } s\neq t \quad (6.22)\\ 1 & \text{if } k=0 \text{ and } s=t, \quad (6.23)\\ \sum_{t'\in N^-(t)\setminus F} w_F(s,t,k-1) & \text{otherwise.} \quad (6.24)\end{cases}$$

To see this, note that if $k=0$ the only walk of length 0 from s is (s) and otherwise, every walk of length $\ell>0$ from s to t avoiding F is built in a unique way from a walk from s to t' where $t'\in N^-t\setminus F$ by concatenating t in the end.

Thus by evaluating (6.21) and using the polynomial time dynamic programming algorithm to compute $|\bigcap_{i\in F}\overline{P_i}|=\sum_{s\in V\setminus F} w_F(s,s,n)$ we indeed get an $O^*(2^n)$ time algorithm for in fact computing the number of Hamiltonian cycles. Note we already solved the TSP problem in Section 6.3 which is a generalization of Hamiltonian cycle, but this new algoritm uses only space polynomial in n, which is a significant advantage. In the future, we'll just say that an algorithm using space polynomial in the input size uses *polynomial space*.

6.7 k-coloring in $O^*(2^n)$ time

Let's have a look again at the k-coloring problem. Suppose we are given $G=(V,E)$, an integer k and would like to know whether G has a k-coloring. We now see an algorithm that decides this in $O^*(2^n)$ time. We use that G has a k-coloring if there exist k independent sets $I_1,\ldots,I_k$ of G such that $\cup_{i=1}^k I_i=V$.

We again use inclusion-exclusion. Let U be the set of all k-tuples of independent sets, e.g., $U=\{(I_1,\ldots,I_k): I_i \text{ is an independent set of } G\}$. Let $V=\{1,\ldots,n\}$ and for every $1\leq i\leq n$ let P_i be the set of k-tuples of independent sets $(I_1,\ldots,I_k)\in U$ such that $v_i\in\cup_{j=1}^k I_j$. Equation 6.1 tells us that

$$\#(k\text{-tuples of independent sets } (I_1,\ldots,I_k) \text{ with } V=\cup_{i=1}^k I_i)=\sum_{F\subseteq\{1,\ldots,n\}}(-1)^{|F|}|\bigcap_{i\in F}\overline{P_i}| \quad (6.25)$$

Again we need to compute $|\bigcap_{i\in F}\overline{P_i}|$. Note that here this equals the number of k-tuples of independent sets of G that do not contain any vertex of F, which just is the number of independent sets of $G[V\setminus X]$ raised to the power k. For $X\subseteq V$, let $i[X]$ be the number of independent sets of $G[X]$, it is easy to see that $i[X]$ can be computed in time $O^*(2^{|X|})$, so this gives us a $O^*(\sum_{x=1}^n\binom{n}{x}2^x)$ time algorithm which is at most $O^*(3^n)$ time by the binomial theorem $\sum_{i=0}^x\binom{n}{x}a^xb^{n-x}=(a+b)^n$.

But, we promised $O^*(2^n)$ time in the title. To obtain this, note that before we start we can compute a table with $i[X]$ for every $X\subseteq V$ in $O^*(2^n)$ time with the following recurrence:

$$i[X]=\begin{cases}1 & X=\emptyset \quad (6.26)\\ i[X\setminus v]+i[X\setminus N[v]] & \text{if } v\in X. \quad (6.27)\end{cases}$$

Note that here $N[v]$ denotes the inclusive neighborhood of v, e.g., $N(v) \cup v$. To see this, note that the empty graph has 1 independent set (the empty set), and if $v \in X$ then every independent set of $G[X]$ either contains v, in which case it does not contain any other vertex set from $N[v]$, or it does not, in which case it also is an independent set of $G[V \setminus v]$.

Summarizing, we can solve k-coloring for every k in time $O^*(2^n)$ by first computing $i[X]$ for every $X \subseteq V$ and subsequently evaluating the formula $\sum_{F \subseteq \{1,\ldots,n\}} (-1)^{|F|} i[V \setminus F]^k$.

6.8 Weighted Independent Set on trees

Now we move back do a quite basic polynomial time algorithm, as an introduction to a subject that we will see next time called 'treewidth'. Suppose we are given a rooted tree $T = (V, E)$ and a weight function $\omega : V \to \mathbb{N}$. The goal is to find an independent set $X \subseteq V$ maximizing $\sum_{e \in X} \omega(e)$. Denote $ch(v)$ to be the set of children of v (which is the empty set if v is a leaf of T). Define $A[v]$ to be the maximum weight of an independent set of $T[v]$, then we have that

$$A[v] = \begin{cases} \omega(v), & \text{if } T[v] \text{ is a single node,} \quad (6.28) \\ \max \left\{ \sum_{c \in ch(v)} T[c], \omega(v) + \sum_{c_1 \in ch(v)} \sum_{c_2 \in ch(c_1)} T[c_2] \right\}, & \text{otherwise.} \quad (6.29) \end{cases}$$

To see this, note that if $T[v]$ is a single node, the maximum independent set is to include v. Otherwise, an independent set may not include v, in which case it will induce an independent set in $T[c]$ for every child c of v, or it will include v, in which case it may not include any child of v so it will induce a maximum independent set in $T[c_2]$ for all 'grand-children' of v (e.g., children of children).

It is easy to see that a naïve evaluation of 6.28 takes only $O(n)$ time since there are at most $O(n)$ 'child' and 'grandchild' relations.

6.9 Excercises

Exercise 6.1. Download the excel sheet subsetsum.xls from `http://www.win.tue.nl/~jnederlo/2MMD30/`. It was used to solve the instance

$$w = \{3, 20, 58, 90, 267, 493, 869, 961, 1000, 1153, 1246, 1598, 1766, 1922\}, t = 5842$$

of Subset Sum to find the solution $20, 58, 90, 869, 961, 1000, 1246, 1598$. Are there more solutions? If so, can you find one?

Exercise 6.2. How many integers in $\{1, \ldots, 100\}$ are not divisible by 2, 3 or 7?

Exercise 6.3. At the 5th of December it is common in the Netherlands to buy presents for each other. To do this when there are n persons $p_1, \ldots, p_n$ celebrating together, there are various processes to pick a random permutation $f : \{1, \ldots, n\} \leftrightarrow \{1, \ldots, n\}$. We call a permutation good if $f(i) \neq i$ for every i. Suppose $n = 5$, how many good permutations are there?

Exercise 6.4. 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 FIB2(n)
Output: f_n
1: Initiate a table F with $F[i] = -1$ for $i = 1, \ldots, n$
2: **return** FIBREC(n).

Algorithm FIBREC(n)
Output: f_n
1: **if** $n = 1$ or $n = 2$ **then return** 1
2: **if** $F[n] = -1$ **then**
3: $x \leftarrow$ FIBREC($n-1$)+FIBREC($n-2$)
4: $F[n] \leftarrow x$
5: **return** x.
6: **else**
7: **return** $F[n]$.

Exercise 6.5. Let G be bipartite graph with parts A, B, $|A| = |B| = n$. Use inclusion exclusion to count the number of perfect matchings of G in $O^*(2^n)$ time. Can you do with polynomial space?

Exercise 6.6. In the Weighted Steiner Tree problem we are given a graph $G = (V, E)$, a weight function $\omega : E \to \mathbb{N}$ and a set of terminals $T \subseteq V$. We need to find a connected tree (S, E') minimizing $\sum_{e \in E'} \omega(e)$ such that $T \subseteq S \subseteq V$. Solve this problem in time $O^*(2^{n-k})$, where $|V| = n$ and $|T| = k$.

Exercise 6.7. In the Set Partition and Set Cover problems we are given sets $A_1, \ldots, A_m \subseteq U$ where $|U| = n$. In the Set Cover problem we need to find $X \subseteq \{1, \ldots, m\}$ such that $\bigcup_{i \in X} A_i = U$. In the Set Partition problem we need to find $X \subseteq \{1, \ldots, m\}$ such that $\bigcup_{i \in X} A_i = U$ and $A_i \cap A_j = \emptyset$ for every $i, j \in X$ with $i \neq j$. Solve both problems in $O^*(2^n)$ time. Can you also solve both in $O^*(2^n)$ time and polynomial space?[4] Note: $O^*(\cdot)$ suppresses factors polynomial in the *input size*, so $O^*(2^n m^{100})$ is also $O^*(2^n)$.

Exercise 6.8.[Floyd Warshall] Suppose we are given a graph $G = (V, E)$ with for every edge $(u, v) \in E$ a distance $d(u, v)$. Let $V = \{1, \ldots, n\}$. The goal of this exercise if to compute for every pair $i, j \in V$ the shortest path from i to j in a total of $O(n^3)$ time. Show how to do this with dynamic programming. Specifically, for let $d_{i,j}^{(k)}$ be the length of the shortest path from i to j for which all intermediate vertices are in the set $\{1, \ldots, k\}$ (see also p630 of the book 'Introduction to Algorithms' by Cormen et al.).

Exercise 6.9. First solve Knapsack in time $O(n \lg(v_{\max} n) v_{\max})$ where $v_{\max} = \max_i v_i$. How can you use is to solve Knapsack in time $O(n \lg(v_{\max} n) v_{\max}/S)$, if all values are divisible by S? Argue one could construct an approximation scheme for knapsack similar as in Subsection 6.1.1.[5]

[4] I do not expect you to reproduce the $O^*(2^n)$ time polynomial space algorithm for Set Partition.
[5] I do not expect you to reproduce the approximation scheme for Knapsack.