Résolution exacte de problèmes NP-difficiles
Lecture 4: More algorithmic techniques

25 January, 2021 Lecturer: Eunjung Kim

1 Randomized algorithm continues

1.1 Random separation

Another useful technique for designing a randomized algorithm is a *random separation* technique. Like color-coding, it is useful to design an algorithm to detect a small-sized substructure in a graph.

We exemplify this technique with the problem SUBGRAPH ISOMORPHISM: given an input graph G and a pattern graph H on k vertices, the task is to find a copy of H in G or correctly decide that G does not have H as a subgraph. We take the parameter $k + d$, where d is the maximum degree of G and present a randomized algorithm that runs in time $2^{dk+O(k \log k)} \cdot n^{O(1)}$ which detect H as a subgraph in G with high probability, if exists one[1].

The intuitive idea is to color the edges of G in blue or red so that the edges of H are 'isolate' in this coloring, and thus this isolated copy of H is easy to detect. Fix a subgraph $\tilde{H}$ of G which is isomorphic to H. A coloring $c : V(G) \to \{\text{red, blue}\}$ is *successful* if the next two conditions are satisfied:

1. all edges of $\tilde{H}$ is colored blue, and
2. all edges of $E(G) \setminus E(\tilde{H})$ incident with a vertex of $V(\tilde{H})$ is colored blue.

On how many edges does a successful coloring requests a specific color? All edges incident with a vertex of $\tilde{H}$ are requested to be either blue or red, depending on whether it belongs to $E(\tilde{H})$ or not. As there are at most $d \cdot V(\tilde{H}) = dk$ such edges, a random coloring c is successful with probability at least 2^{-dk}.

Now assume that the current coloring c is successful. Consider the subgraph G' of G consisting of the blue edges. Let us call a connected component in G' a blue component, and let $\mathcal{B}$ be the set of all blue components. Now we have narrow down which part of G we need to match H. To begin with, any blue component having more than k vertices has no chance of being $\tilde{H}$ (under a successful coloring).

[1] Mind that if you parameterize by k only, then we cannot expect to have an fpt-algorithm: when H is a clique on k vertices, it is known that deciding if G contains H as a subgraph or not is known to be W[1]-hard (you will learn this notion in the next class) parameterized by k. This means that under a widely-accepted complexity assumption that $W[1] \neq FPT$, SUBGRAPH ISOMORPHISM is unlikely to be fixed-parameter tractable with respect to k only.

Let $H_1, \ldots, H_p$ be the connected components of H (possibly $p = 1$). We make a bipartite graph W in which one part of the vertex bipartition is $\mathcal{H} = \{H_1, \ldots, H_p\}$ and the other part is $\mathcal{B}$, the set of all blue components. The bipartite graph W has an edge between $H \in \mathcal{H}$ and $B \in \mathcal{B}$ if H is isomorphic to B. As the isomorphism between H and B (on at most k vertices each) can be tested in time $k! \cdot k^{O(1)} = 2^{O(k \log k)}$, the construction of W can be done in time $2^{O(k \log k)} \cdot n$. It remains to observe that if $\tilde{H}$ exists and the current color is successful for $\tilde{H}$, this copy must be a disjoint union of p blue components each of which is isomorphic to $H_1, \ldots, H_p$ respectively. We can decide whether such p blue components exist by examining whether a maximum matching on W exists saturating all vertices in $\mathcal{H}$. The latter problem is polynomial-time solvable.

To summarize, if G contains a copy of H (say $\tilde{H}$), then with probability at least 2^{-dk} a random coloring is successful for $\tilde{H}$. Given a successful coloring, $\tilde{H}$ can be correctly retrieved from G in time $2^{O(k \log k)} \cdot n^{O(1)}$. Let us call this procedure $\mathcal{A}$. The probability[2] that no copy of H is detected after 2^{dk} repetitions of $\mathcal{A}$ while G contains a copy of H is at most

$$(1 - 2^{-dk})^{2^{dk}} = (1 - 2^{-dk})^{(-2^{dk}) \cdot (-1)} \approx e^{-1}$$

that is, with constant probability a copy of H is detected after 2^{dk} runs of $\mathcal{A}$. This constant success probability can be boosted to an arbitrarily high constant probability by repetitions.

1.2 Derandomization

The randomization technique of color-coding and random separation can be derandomized. For derandomizing color-coding, we use a family of functions called an $(n.k)$-*perfect hash family*. A family $\mathcal{F}$ of functions $f : [n] \to [k]$ is an $(n.k)$-*perfect hash family* if for every subset $S \subseteq [n]$ of size k, there exists $f \in \mathcal{F}$ such that f assigns pairwise distinct values to the elements of S. Note that if S is the fixed object that we want to find, then such f will make S colorful. A repetition of random colorings in color-coding technique can be replaced by an (n,k)-perfect hash family with almost negligible computational overhead, due to the following theorem.

Theorem 1 (Naor, Schulman, Srinivasan 1995)**.** *For every $n, k \geq 1$, an (n,k)-perfect hash family of size $e^{k+o(k)} \cdot \log n$ can be constructed in time $e^{k+o(k)} \cdot n \log n$.*

For random separation, we use a different method. An (n,k)-*universal set* $\mathcal{U}$ is a family of subsets of $[n]$ such that for any $S \subseteq [n]$ of size k, all possible subsets of S appear in the projection of $\mathcal{U}$ on S, that is, $\{S \cap A : A \in \mathcal{U}\} = 2^S$. In our application to Subgraph Isomorphism on graphs with maximum degree at most d, S will correspond to the set of edges incident with a vertex of $V(\tilde{H})$, whose size is at most 2^{dk}, and with a successful coloring we are looking for a partition of these edges into edges in $\tilde{H}$ and the rest. Now repeated random colorings can be replaced by trying the colorings in $\mathcal{U}$, interpreting a set $A \in \mathcal{U}$ as blue edges. This strategy works with little overhead because of the following theorem

Theorem 2 (Naor, Schulman, Srinivasan 1995)**.** *For every $n, k \geq 1$, an (n,k)-universal set of size $2^{k+o(k)} \cdot \log n$ can be constructed in time $2^{k+o(k)} \cdot n \log n$.*

[2]We use the fact the natural log base e equals $\lim_{x \to 0}(1 + x)^{\frac{1}{x}}$.

2 Dynamic programming

If a problem can be optimally solved by combining the solutions to a smaller problem, then dynamic programming approach can be used. We give two dynamic programming algorithm, one for Hamiltonian Path and another for Steiner Tree. Both runs in time $2^n \cdot n^{O(1)}$ and requires exponential space.

Problem Hamiltonian Path

Input: a graph G with prescribed vertices s, t ($s \neq t$).

Task: decide if G has a Hamiltonian path from s to t.

For a graph $G = (V, E)$, and a vertex subset $K \subseteq V$, a *steiner subgraph* for K is a connected subgraph H of G which contains all vertices of K. Intuitively, a steiner subgraph for K is an essential structure in G that pairwise connect the vertices of K The vertices of K are called *terminals*. For a subgraph H of an edge-weighted graph G with weight function $\omega : E \to \mathbb{R}_{\geq 0}$, the *weight of* H is the sum $\sum_{e \in E(H)} \omega(e)$ over all H's edges and will be denoted by $\omega(H)$. In this vein, we are interested in finding a steiner subgraph with minimum number of edges, or of minimum weight. With non-negative weights, a steiner subgraph of minimum edge count/weight sum can be assumed to be a tree and we call a steiner subgraph which is a tree a *steiner tree*. This leads to the following fundamental problem.

Problem Steiner Tree

Input: an edge-weighted graph $G = (V, E)$ with weight function $\omega : E \to \mathbb{R}_{\geq 0}$, and a set of vertices $K \subseteq V$ (terminals)

Task: find a steiner tree for K of minimum weight, if one exists.

2.1 DP for Hamiltonian Path

For all subsets $s \in S \subseteq V$ and a vertex $v \in S$, we compute whether $G[S]$ contains a Hamiltonian path from s to v. Let $P[S, v]$ be 1 if such a Hamiltonian (s, v)-path in $G[S]$ exists and it takes value 0 otherwise. The dynamic programming will compute the values of P in a bottom-up manner in the sense that $P[S, v]$ will be computed using the tabulated values of P for smaller sets. Note that G has a Hamiltonian path from s to v if and only if $P[V, v] = 1$.

The base case is when $S = \{s\}$ and $v = s$, and we have $P[S, s] = 1$ trivially. For sets S containing s with $|S| \geq 2$, the next recursion for $P[S, v]$ is easy to see.

$$P[S, v] = \begin{cases} 0 & \text{if } v = s \\ \bigvee_{w \in N(v) \cap S} P[S \setminus v, w] & \text{if } v \neq s. \end{cases}$$

Each computation of $P[S, v]$ requires $O(|S|)$ lookups of the table P constructed already. As there are $2^{n-1} \cdot n$ entries in the table, the algorithm takes $O(2^n \cdot n^2)$-time.

2.2 DP for Steiner Tree

We may assume that every terminal has degree 1 in the input graph G: for $v \in K$, if v is not already of degree 1, then add a pendant vertex v' to v and replace v in K by v'. The weight on vv' is set to 0. If $|K| \leq 2$, then STEINER TREE has a trivial solution either a single vertex solution (of weight zero), or a steiner tree which is a shortest path between two terminals. Therefore, we assume $|K| \geq 3$. Also G can be assumed to be connected: if K resides in more than one connected components of G, there is no steiner tree for K and report so. If this is not the case, we can take as the input graph the unique connected component of G containing the entire set K.

The algorithm starts with an observation that under the above assumptions, any steiner tree T contains a non-terminal vertex u which has degree at least three in T. Consider two subtree T_1, T_2 of T, where T_1 takes u as a leaf and T_2 is the remaining part of T. The subtrees T_1 and T_2 splits the terminals into two parts, say K_1 and K_2, and the respective sizes have decreased by at least one. The idea is to find a steiner tree for $K_1 \cup u$ and $K_2 \cup u$. But we also want that adding a vertex like u as a terminal temporarily does not have an accumulating effect.

So, we view this non-terminal vertex u as an *interface* vertex for connecting K_1 and K_2. If K_1 contains a single vertex, then finding a steiner tree for K_1 becomes a shortest path problem. Otherwise, any steiner tree T_1 form $K_1 \cup u$ again contains a non-terminal vertex w which has 'branches out' with K_1: Consider T_1 as a tree rooted at u and choose w of shortest distance to u in T_1 with at least two children. The crucial point here is that by choosing w closest to u, we ensured that the subtree of T_1 containing u and taking w as a leaf is a *path*. Note that w can be possibly identical to u. Now, w will take the role of u for the partition of K_1. Mind that we are blind to which non-terminal vertex will actually take the role of u or w, also blind to which partition the hypothetical w will induce on K_1. Therefore, we compute optimal partial solution for all possible choices of w and possible partitions.

With the above observation, we end up with the next recursion. For a terminal set $D \subseteq K$ and a non-terminal vertex $u \in V \setminus K$, let $P[D, u]$ is the minimum possible weight of a steiner tree for $D \cup u$ in G.

$$P[D, u] = \min_{w \in V \setminus K, \emptyset \subsetneq D' \subsetneq D} \mathsf{dist}_G(u, w) + P[D', w] + P[D \setminus D', w]$$

3 Inclusion-Exclusion based algorithms

3.1 Inclusion-Exclusion formula

Theorem 3 (Inclusion-Exclusion, union version)**.** *Let A_i for $i = 1, \ldots, n$ be finite sets. Then,*

$$|\bigcup_{i\in[n]} A_i| = \sum_{\emptyset\neq X\subseteq[n]} (-1)^{|X|+1} |\bigcap_{i\in X} A_i|.$$

Proof: Notice that an element not in $\bigcup_{i\in[n]} A_i$ contributes neither to any term of the right-hand side, nor to the left-hand side. For an element $x \in \bigcup_{i\in[n]} A_i$, its contribution to the left-hand side is 1. It remains to show that the sum of contribution of x to the right-hand side is precisely 1. Let $Y \subseteq [n]$ be the set of indices i such that $x \in A_i$. Then for every $\emptyset \neq X \subseteq Y$, $\bigcap_{i\in X} A_i$ contains x. Conversely, for every $\emptyset \neq X \nsubseteq Y$ we have $x \notin \bigcap_{i\in X} A_i$. Therefore, x creates the following terms of the right-hand side:

$$\begin{aligned}
\sum_{\emptyset\neq X\subseteq Y} (-1)^{|X|+1} \cdot 1 &= (-1) \sum_{\emptyset\neq X\subseteq Y} (-1)^{|X|} \\
&= -\sum_{i=1}^{|Y|} \sum_{X\subseteq Y, |X|=i} (-1)^i \\
&= -\sum_{i=1}^{|Y|} \binom{|Y|}{i} (-1)^i 1^{|Y|-i} \\
&= -\Big(\sum_{i=0}^{|Y|} \binom{|Y|}{i} (-1)^i 1^{|Y|-i} - 1\Big) \\
&= 1 - (-1+1)^{|Y|} = 1.
\end{aligned}$$

□

Theorem 4 (Inclusion-Exclusion, intersection version)**.** *Let A_i for $i = 1, \ldots, n$ be sets of a finite universe U. Then,*

$$|\bigcap_{i\in[n]} A_i| = \sum_{X\subseteq[n]} (-1)^{|X|+1} |\bigcap_{i\in X} (U \setminus A_i)|.$$

Proof: First, we note that for finite sets B_i, $i \in [n]$,

$$U \setminus \bigcup_{i\in[n]} B_i = \bigcap_{i\in[n]} (U \setminus B_i). \tag{1}$$

Therefore, by Theorem 3 it holds that

$$\begin{aligned}
|U \setminus \bigcup_{i\in[n]} B_i| &= |U| + \sum_{\emptyset\neq X\subseteq[n]} (-1)^{|X|} |\bigcap_{i\in X} B_i| \\
&= \sum_{X\subseteq[n]} (-1)^{|X|} |\bigcap_{i\in X} B_i|. \qquad (2)
\end{aligned}$$

Set $A_i = U \setminus B_i$ and combine the equations (1)-(2). Now,

$$\begin{aligned}|\bigcap_{i\in[n]} A_i| = |\bigcap_{i\in[n]}(U \setminus B_i)| &= |U \setminus \bigcup_{i\in[n]} B_i| \\ &= |U| - \sum_{\emptyset\subsetneq X\subseteq[n]} (-1)^{|X|+1}|\bigcap_{i\in X} B_i| \\ &= \sum_{X\subseteq[n]} (-1)^{|X|}|\bigcap_{i\in X}(U \setminus A_i)|,\end{aligned}$$

where the last equation follows from the convention of writing $U = \bigcap_{i\in\emptyset} B_i$. $\square$

3.2 IE-based algorithm for Hamiltonian Cycle

Using the Inclusion-exclusion formula we can compute Hamiltonian Cycle in $2^n \cdot n^{O(1)}$-time. In fact we can count the number of Hamiltonian cycles in the same running time.

Let $G = (V, E)$ be on n vertices $v_1, \ldots, v_n$, and let $v_0 = v_n$. A *closed walk* is a sequence of vertices of G whose start and end vertices are identical, and any two consecutive vertices are adjacent in G. Notice that a vertex or an edge might appear in a walk multiple times. The length of a closed walk is the length of vertex sequence minus one. By v_0-walk, we mean a closed walk that begins and ends with v_0. To apply the (intersection version) of inclusion-exclusion formula, we define the ground set U as follows:

$$U = \{\text{all } v_0\text{-walks of length } n\}.$$

Now we can view a Hamiltonian cycle (with an orientation) as a v_0-walk of length n which visits every $v \in V$. Notice that each Hamiltonian cycle yields two v_0-walks of length n visiting every vertex v. Therefore with A_i defined as

$$A_i = \{\text{all } v_0\text{-walks of length } n \text{ visiting } v_i\},$$

the Hamiltonian cycles, the v_0-walks of length n visiting all $v \in V$ to be precise, are captured by $\bigcap_{i\in[n]} A_i$. Its cardinality can be computed by computing $|\bigcap_{i\in X}(U \setminus A_i)|$ for every $X \subseteq [n]$ instead thanks to Theorem 4.

So, what kind objects constitute $\bigcap_{i\in X}(U \setminus A_i)$? Observe that $I \setminus A_i$ are precisely the v_0-walks of length n which *avoid* v_i, and thus $\bigcap_{i\in X}(U \setminus A_i)$ are v_0-walks of length n which avoid all vertices corresponding to X. In other words, $\bigcap_{i\in X}(U \setminus A_i)$ are the set of all v_0-walks of length n in $G - X$ (formally $G - \{v_i : i \in X\}$).

Finally, the number of (v_i, v_j)-walks of length ℓ in a graph H can be computed in polynomial time by computing ℓ-th power of the adjacency matrix of H and reading off the (i, j)-entry of the resulting matrix. This completes the algorithm and it is straightforward to see that after 2^n steps all the terms of $\sum_{X\subseteq[n]}(-1)^{|X|}|\bigcap_{i\in X}(U \setminus A_i)|$ have summed up. We remark that this algorithm works both for directed and undirected graphs.

3.3 IE-based algorithm for k-Coloring

To apply the intersection version of inclusion-exclusion formula, we view a k-coloring as a k-tuple of independent sets of G. Namely, we define

$$U = \{(I_1, \ldots, I_k) : I_i \text{ is an independent set of } G\}.$$

Notice that two independent sets in a tuple may intersect and even coincide. Observe that there is a (proper) k-coloring if and only if there is k-tuple of independent sets covering all vertices of G. Therefore let

$$A_i = \{(I_1, \ldots, I_k) \in U : v_i \in I_1 \cup \cdots \cup I_k\},$$

and G admits a proper k-coloring if and only if $\bigcap_{i\in[n]} A_i \neq \emptyset$. Due to Theorem 4, we can decide this via computing the value $\sum_{\emptyset\neq X\subset[n]}(-1)^{|X|+1}|\bigcap_{i\in X}(U \setminus A_i)|$.

Again, $\bigcap_{i\in X}(U \setminus A_i)$ is the set of all k-tuples of independent sets avoiding the vertices in X altogether. In other words, it is the set of all k-tuples of independent sets of $G - X$. Let $i(G)$ be the number of independent sets of G and observe

$$|\bigcap_{i\in X}(U \setminus A_i)| = i(G - X)^k.$$

Now $i(G)$ can be computed with dynamic programming. Choose an arbitrary vertex $v \in G$ and note that

$$i(G) = i(G - v) + i(G - N[v])$$

where the first term in r.h.s counts the independent sets of G *not* containing v and the second term counts the independent sets of G containing v, thus excluding $N(v)$. This recursion indicates that $i(G[Z])$ over all subsets Z of V can be tabulated, and this can be done in time $2^n \cdot n^{O(1)}$.

With the above table containing values for $i(G - X)$ for all $X \subseteq [n]$, we can compute $\sum_{X\subseteq[n]}(-1)^{|X|+1}|\bigcap_{i\in X}(U \setminus A_i)|$ in time $2^n \cdot n^{O(1)}$.