6.S078 Fine-Grained Algorithms and Complexity **MIT**
Lecture 17: Algorithms for Finding Long Paths (Part 1) **November 2, 2020**

In this lecture and the next, we will introduce a number of algorithmic techniques used in exponential-time and FPT algorithms, through the lens of one parametric problem:

Definition 0.1 (k-Path) *Given a directed graph $G = (V, E)$ and parameter k, is there a simple path*[1] *in G of length $\geq k$?*

Already for this simple-to-state problem, there are quite a few radically different approaches to solving it faster; we will show you some of them. We'll see algorithms for the case of $k = n$ (Hamiltonian Path) and then we'll turn to "parameterizing" these algorithms so they work for all k.

A number of papers in bioinformatics have used quick algorithms for k-Path and related problems to analyze various networks that arise in biology (some references are [SIKS05, ADH$^+$08, YLRS$^+$09]).

In the following, we always denote the number of vertices $|V|$ in our given graph $G = (V, E)$ by n, and the number of edges $|E|$ by m. We often associate the set of vertices V with the set $[n] := \{1, \ldots, n\}$.

1 Hamiltonian Path

Before discussing k-Path, it will be useful to first discuss algorithms for the famous NP-complete Hamiltonian path problem, which is the special case where $k = n$. Essentially all algorithms we discuss here can be adapted to obtain algorithms for k-Path! The naive algorithm for Hamiltonian Path takes time about $n! = 2^{\Theta(n \log n)}$ to try all possible permutations of the nodes (which can also be adapted to get an $O^\star(k!)$-time algorithm for k-Path, as we'll see).

1.1 Dynamic Programming

Our first algorithm shows how to beat the $n!$ running time. You may have seen it in a prior algorithms class.

Theorem 1.1 (Bellman, Held-Karp'60s [HK65]) *Hamiltonian path can be solved in $O^\star(2^n)$ time.*

Proof. The basic idea of the algorithm is this: suppose you are walking along a path in the graph, and trying to construct a Hamiltonian path. After you have visited some of the vertices, you do not need to remember the actual order of vertices that you have visited in the past: you just need to remember the *set* of such vertices that you visited, in order to construct a Hamiltonian path.

More formally, we construct a table T, indexed by $2^{[n]} \times [n]$, such that $T(S, v) = 1$ if and only if there is a path that visits exactly the vertices in the set S and ends at $v \in S$. We can compute the table T using the following algorithm:

[1] A simple path does not go through a vertex more than once.

```
Set $T(\{v\}, v) = 1$ for all $v \in V$, and set all other entries of $T$ to 0.
For $t = 2, \dots, n$
    For all $S \subseteq [n]$ such that $|S| = t - 1$, and for all $u \in V$
        if $T(S, u) = 1$ then
            For all $v \notin S$ such that $(u, v) \in E$,
                Set $T(S \cup \{v\}, v) = 1$
        end if
end for
If $\exists v \in V$ such that $T([n], v) = 1$, then return **there's a ham path**
Otherwise, return **no ham path**
```

Exercise: Convince yourself that, whenever t is incremented to $t + 1$ in the algorithm, we have $T(S, u) = 1$ if and only if there is a simple path on t nodes through the subset S that ends in u.

Exercise: Prove that the algorithm runs in $O(n^2 2^n)$ time, assuming we can generate each new set S in constant time, and constant-time access to the table T (we can lookup and modify entries of T in constant time).

□

1.2 A More Space-Efficient Algorithm

Could we use the above algorithm to get a FPT algorithm for k-path? Not easily... If we simply restrict subsets S to be all sets of size at most k, the above algorithm will run in time $O^\star(\binom{n}{k})$, which is not FPT. Also, this algorithm has the issue that it uses at least $\Omega(2^n)$ space to store its table, while the naive algorithm of $O^\star(n!)$ time only costs polynomial space. Actually there is another algorithm that solves Hamiltonian path in both $O^\star(2^n)$ time and $O^\star(1) = \text{poly}(n)$ space, by the following theorem.

Theorem 1.2 (Karp'80s [Kar82]) *Hamiltonian path can be solved in $O^\star(2^n)$ time and $O^\star(1)$ space.*

Proof. The key idea here is to shoot for solving a harder problem than just finding a Hamiltonian path: we *count the number* of Hamiltonian paths. To do this, we use the Inclusion-Exclusion Principle, which will actually give us a reduction from

counting *paths* in a graph (which is NP-hard)
to
counting *walks* in a graph (which is easy! polynomial time)

The catch is that the number of calls to counting walks in our reduction will be 2^n.

Recall that a *walk* in a graph is any sequence of vertices $(v_1, \dots, v_t)$ such that (v_i, v_{i+1}) is an edge. A walk on t vertices is called a *t-walk*. The difference between a walk and a path is that a walk can visit the same vertex several times, while a path cannot. Denote the number of n-node walks in G by W_G. W_G can be computed efficiently by the following lemma.

Lemma 1.1 *For any G, the number of n-node walks in G can be computed in time $O(poly(n))$.*

Proof of Lemma 1.1. Let A be the adjacency matrix of G. Consider the quantity

$$A^2(i, j) = \sum_k A(i, k) \cdot A(k, j).$$

Observe that this is equal to the number of 3-node walks from i to j (i.e., walks of the form (i,k,j)). Similarly, one can prove by induction that $A^{\ell-1}(i,j)$ is the number of ℓ-node walks from i to j, for all ℓ. Therefore

$$W_G = \sum_{i \neq j} A^{n-1}(i,j),$$

which is computable in polynomial time by repeated matrix multiplications. □

Note that W_G is a crude upper bound on the number of Hamiltonian paths: every Hamiltonian path is also an n-walk. But there can be a lot of walks which aren't Hamiltonian, of course. To help us "filter" these bad walks out, we observe:

Proposition 1.1 *An n-walk P is a Hamiltonian path if and only if P visits all vertices in the graph.*

Proof. Every Hamiltonian path must visit all vertices in the graph. In the opposite direction, if the walk P is n vertices long, and P visits all n vertices, then it must visit each vertex exactly once. □

To get "closer" to the true number of Hamiltonian paths, let's try to subtract the "bad" walks counted in W_G which don't visit every vertex. For any subset of vertices $S \subseteq V$, let $G - S = (V - S, E - (S \times V) - (V \times S))$. That is, $G - S$ is the subgraph of G with the vertex set S removed. Then, $W_{G-\{v\}}$ is the number of n-walks that do not go through $v \in V$. We want to subtract those kinds of walks from W_G. To use the Inclusion-Exclusion Principle, let S_i be the set of all n-walks that visit node i. Then

$$\left| \bigcap_{i=1}^{n} S_i \right|$$

is the number of Hamiltonian paths, by the above proposition. By the Inclusion-Exclusion Principle,

$$|\bigcap_{i=1}^{n} S_i| = W_G - \sum_i |\overline{S_i}| + \sum_{i<j} |\overline{S_i} \cap \overline{S_j}| - \cdots + (-1)^n \cdot |\overline{S_1} \cap \cdots \cap \overline{S_n}|.$$

There are 2^n terms on the RHS of the equation above, one for each subset of $[n]$. Observe that

$$|\overline{S_i}| = \text{the number of } n\text{-walks that do not contain } i = W_{G-\{i\}},$$

$$|\overline{S_i} \cap \overline{S_j}| = \text{number of } n\text{-walks containing neither } i \text{ nor } j = W_{G-\{i,j\}},$$

and in general, for $\{i_1, \ldots, i_k\} \subseteq [n] = V$,

$$|\overline{S_{i_1}} \cap \cdots \cap \overline{S_{i_k}}| = W_{G-\{i_1,\ldots,i_k\}}.$$

For all $S \subset V$, we can compute each of the W_{G-S} in polynomial time and space. Depending on $|S|$, this W_{G-S} term is either added or subtracted from the total sum on the RHS of the equation. Once we've computed all W_{G-S} and added/subtracted them, we have the number of Hamiltonian paths. The running time is $O^\star(2^n)$ and the space used is only $O^\star(1)$ because from one sum W_{G-S} to another, we only have to store the current subset S. □

Exercise: Given an algorithm that counts the number of Ham paths, how would we get an algorithm to find a Ham path? Suppose the counting algorithm runs in time $T(n)$; how fast can you make the finding algorithm?

1.3 Dynamic Programming Vs Inclusion-Exclusion

So far, we have seen two ways to solve Ham Path:

- DP: $O^\star(2^n)$ time and space

- IE: $O^\star(2^n)$ time and $O^\star(1)$ space.

The IE algorithm can be improved to $O^\star(1.66^n)$ randomized time for Hamiltonian path in undirected graphs [Bjö14]. It is not known how to improve the running time of 2^n for directed graphs!

A nice aspect of the DP algorithm is that it generalizes to the Traveling Salesman Problem (TSP), where in an edge-weighted graph, we want to find a *minimum weight* Hamiltonian path.[2] Therefore TSP can also be solved in time $O^\star(2^n)$, and this is the fastest known worst-case algorithm for TSP.[3] IE apparently does not generalize similarly! (There are ways to do it, but they run in *pseudopolynomial* time in the weights of the edges: the running time is exponential in the bit complexity of the weights.) It is an open problem if TSP can be solved in both time $O^\star(2^n)$ and space $O^\star(1)$.

However, there is *some* interesting polynomial-space algorithm known for TSP:

Theorem 1.3 ([GS87]) *TSP can be solved in $O^\star(4^n)$ time and $O^\star(1)$ space.*

Here we will just give the basic idea. Consider the sequence of nodes in an optimal TSP solution. Conceptually think of breaking this sequence into two subsequences of about $n/2$ nodes each; call the set of nodes in the first half L and the nodes in the second half R. We will try each possible choice of L, and recurse on L and $V \backslash L$. We'll have both of these recursive calls return $n/2$ by $n/2$ matrices A and B, storing the minimum weight path from i to j for $i, j \in L$ and for $i, j \in V \backslash L$, respectively. Using all of these pairs of matrices A and B that are returned over all possible choices for L, we can construct an $n \times n$ matrix M which stores the minimum weight path from i to j for $i, j \in V$. (Think about how you would do this! If $w(k, j)$ denotes the weight of the edge from k to j, note that

$$A[i, k] + w(k, j) + B[j, \ell]$$

gives the minimum weight path that starts at $i \in L$, passes through all the vertices of L ending at $k \in L$, takes the edge from k to $j \in V - L$, then passes through all vertices in $V - L$, ending in $\ell \in V - L$. By trying all L, and all edges that pass between L and $V - L$, we can compute the (i, ℓ) entry of M.)

The recurrence for the running time is

$$T(n) \leq \binom{n}{n/2} \cdot 2 \cdot T(n/2) + O^\star(1) \leq O^\star(2^{n+n/2+n/4+\cdots}) = O^\star(4^n)$$

and it needs only poly space to hold its current matrices, and the recursion stack.

2 Onward to k-Path

Our first k-Path algorithm will show how to solve the problem in $O^\star(k!)$ time for every k, generalizing the brute-force algorithm for Hamiltonian Path. We will give a randomized reduction from

k-Path on arbitrary graphs (which is NP-hard for $k = n$)
to
k-Path on directed acyclic graphs (which is easy even when $k = n$)

The catch is that our randomized reduction will only succeed with probability $1/k!$, so we'll have to repeat it for $O(k!)$ times. Then we will get rid of the randomization.

[2] Instead of storing a 0-1 value indicating if there is a path, we store the value for the minimum sum weighted path, over all paths that pass through the subset S and end at v.

[3] In STOC 2020, Nederlof shows how to solve TSP in bipartite directed graphs in $O(1.9999^n)$ time, assuming matrix multiplication of $n \times n$ matrices can be done in $n^{2+o(1)}$ time [Ned20].

Theorem 2.1 *k-Path is solvable in randomized $O^\star(k!)$ time. In particular, there is a randomized algorithm which always reports "no path" when there's no k-path, and reports a k-path when one exists with probability at least* 99%.

Proof. Given G, let $\pi : [n] \to [n]$ be a random permutation on n elements. If G is undirected, replace all edges $\{i, j\}$ by the directed edge (i, j) if $\pi(i) < \pi(j)$, and replace $\{i, j\}$ by (j, i) otherwise. If G is a directed graph, we remove all directed edges that do not "respect" the permutation, all edges (j, i) where $\pi(j) > \pi(i)$. In either case, this process results in a DAG, G_π. Then, we compute the longest path in this DAG G_π.

Exercise: Show that finding the longest path in directed acyclic graphs (DAGs) can be done in polynomial time. For simplicity, assume you already know the permutation π, as above. (You could try dynamic programming on the nodes.)

If there is a k-path in G, then we claim that there is a k-path in G_π with probability at least $1/k!$. Let the k-path be the sequence $i_1, \ldots, i_k \in [n]$. Since every permutation of the k nodes in the path is equally likely, the probability that the random permutation π satisfies $\pi(i_1) < \cdots < \pi(i_k)$ is $1/k!$. In that case, G_π will contain the path $i_1, \ldots, i_k$.

If there is no k-path in G, then there will certainly be no k-path in G_π: the set of k-paths in G_π is a subset of the set of k-paths in G.

Repeating the above randomized reduction (from G to G_π) for $10 \cdot k!$ times, we can therefore determine whether there is a k-path in G with high probability. The running time will be $O(k! \cdot \text{poly}(n))$. $\square$

One can think of this algorithm as some analogue of the $n!$ time algorithm for Ham Path. It's randomly picking permutations, and (whp) will find at least one "good" permutation for the k-path out of the $10k!$ that it tries.

2.1 Derandomization (Optional)

How would we "derandomize" this algorithm, and solve k-Path in deterministic $O^\star(k!)$ time? Here is a common theme in "derandomization":

- Show the analysis of the randomized algorithm $\mathcal{A}$ only relied on certain properties of its b uniform random bits.
- Construct a small collection $\mathcal{C} \subsetneq \{0, 1\}^b$ of random strings, where $|C| \ll 2^b$, but the distribution of strings chosen from $\mathcal{C}$ still satisfies these certain properties.
- To get a deterministic algorithm, run $\mathcal{A}$ deterministically on all possible strings from $\mathcal{C}$, instead of all possible 2^b choices for the b random bits. When $|\mathcal{C}|$ is small, this leads to a good deterministic running time.

You can think of the collection $\mathcal{C}$ as being a "pseudorandom generator" that "fools" the algorithm $\mathcal{A}$ into behaving the same as if it were getting uniform random bits.

In the above algorithm, we want to replace the choice of random permutation π with a set $\mathcal{C}$ of $k!\cdot\text{poly}(n)$ permutations, which achieves the same guarantee: for every k-path in G, there is some DAG G_π with $\pi \in \mathcal{C}$ which is a subgraph of G and which "preserves" the k-path. Intuitively, a small collection should be possible, because a k-path is only a set of k nodes; a single permutation on all n nodes should actually cover many of the possible ways to have a k-path.

We start with the useful notion of a "perfect hash family". This is a collection of functions mapping n elements to k elements, such that for every k-set S of n elements, there's a function f_i in the family that maps every element of S to a unique, distinct element in $\{1, \ldots, k\}$. Formally:

Definition 2.1 *A family of functions $\mathcal{F} = \{f_i \mid [n] \to [k]\}$ is a k-**perfect hash family** if for all subsets $S \subseteq [n]$ with $|S| = k$, there is an $f_i \in \mathcal{F}$ such that $f_i(S) = [k]$.*

(Note that $|f_i(S)| \le k$, so if $f_i(S) = [k]$ then it must be that every element of S got mapped by f_i to a distinct element in $\{1, \ldots, k\}$.) We will need a deterministic construction of such functions (which we will use as a black-box).

Theorem 2.2 (Schmidt-Siegel'90 [SS90], Naor-Schulman-Srinivasan'95 [NSS95]) *For all n, k, there are k-perfect hash families $\mathcal{F}_{n,k}$ with at most $F(n,k) = e^k \cdot k^{O(\log k)} \cdot \text{poly}(\log n)$ functions, such that all functions in the family can be constructed in $O(F(n,k))$ time.*

Now in the above randomized algorithm for k-Path, we make the following change. Instead of choosing a completely random π, we try all $f_i \in \mathcal{F}_{n,k}$, and try all permutations $\pi' : [k] \to [k]$. For each f_i and π', we make a subgraph G' that only contains edges (u, v) such that $\pi'(f_i(u)) < \pi'(f_j(v))$, and find the longest path in each G'. (Think about what this does: Since π' is only a permutation on k elements, this G' is a k-*partite* directed acyclic graph.)

Clearly, if G does not have a k-path, then none of these G' will also have a k-path.

Exercise: Show that if G has a k-path, then some G' will also have a k-path.

There are $k! \cdot |\mathcal{F}_{n,k}|$ such G' to consider. Hence we have proved:

Corollary 2.1 *k-Path is in deterministic $O^\star(k! \cdot e^k \cdot k^{O(\log k)})$ time.*

Can we reduce the running time dependence on k further? Considering what we know for the case of $k = n$, we could expect to possibly get the running time down to $O^\star(2^k)$...

References

[ADH$^+$08] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):i241–i249, 2008.

[Bjö14] Andreas Björklund. Determinant sums for undirected hamiltonicity. *SIAM J. Comput.*, 43(1):280–299, 2014.

[GS87] Yuri Gurevich and Saharon Shelah. Expected computation time for hamiltonian path problem. *SIAM J. Comput.*, 16(3):486–502, 1987.

[HK65] Michael Held and Richard M. Karp. The construction of discrete dynamic programming algorithms. *IBM Syst. J.*, 4(2):136–147, 1965.

[Kar82] Richard M. Karp. Dynamic programming meets the principle of inclusion and exclusion. *Oper. Res. Lett.*, 1(2):49–51, 1982.

[Ned20] Jesper Nederlof. Bipartite TSP in $o(1.9999^n)$ time, assuming quadratic time matrix multiplication. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 40–53. ACM, 2020.

[NSS95] Moni Naor, Leonard J. Schulman, and Aravind Srinivasan. Splitters and near-optimal derandomization. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*, pages 182–191. IEEE Computer Society, 1995.

[SIKS05] Jacob Scott, Trey Ideker, Richard M. Karp, and Roded Sharan. Efficient algorithms for detecting signaling pathways in protein interaction networks. In *RECOMB*, pages 1–13, 2005.

[SS90] Jeanette P. Schmidt and Alan Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM J. Comput.*, 19(5):775–786, 1990.

[YLRS$^+$09] Esti Yeger-Lotem, Laura Riva, Linhui Julie Su, Aaron D Gitler, Anil G Cashikar, Oliver D King, Pavan K Auluck, Melissa L Geddie, Julie S Valastyan, David R Karger, et al. Bridging high-throughput genetic and transcriptional data reveals cellular responses to alpha-synuclein toxicity. *Nature genetics*, 41(3):316–323, 2009.