Recall that in the k-Path problem one is given a graph $G = (V, E)$ with $m = |E|, n = |V|$, and one needs to either return a simple path of length k or return that no such path exists. Here k is the parameter. Last time we saw a randomized FPT algorithm for k-Path running in time $O^*(k!)$, and we also showed how to derandomize it with a slight overhead. Either way, the running time of the algorithms we have seen so far run in time $k^{O(k)}\text{poly}(n)$. Today we will present several more FPT algorithms whose running time is better, $2^{O(k)}\text{poly}(n)$.

1 Algorithm 2 - Color Coding (Alon, Yuster, Zwick '94)

In this section we prove the following theorem.

Theorem 1.1 *There is an $O^*((2e)^{k+o(k)})$ time deterministic k-path algorithm. (Note $2e < 5.5$.)*

The main idea is called "color-coding" and it has been used extensively to design fast FPT algorithms for problems, especially problems involving finding small subgraphs with certain structure.

Last time we chose a random *permutation* of the n nodes, and argued that for the k node subgraph we care about (the k-path), there's at least a $1/k!$ chance that the edges of that path are preserved.

This time, we'll instead choose a random *hash function*, not from $[n]$ to $[n]$, but from $[n]$ to $[k]$.

Think of the k numbers in our co-domain as "colors" $1, \ldots, k$.

Our algorithm will have two basic parts (which we will repeat a number of times):

1. Randomly color the nodes of the graph. For every vertex v in $[n]$, pick a color $c(v)$ independently and uniformly at random from $[k]$;
2. Instead of finding a path that visits distinct nodes, try to find a k-path that "visits distinct colors". We call this a *colorful* path: for all nodes i, j in the path, $c(i) \neq c(j)$.

We show that a colorful k-path can be found in $O^*(2^k)$ time, and that if one picks a random k-coloring as above, any fixed k-path is colorful with probability at least $1/e^k$.

Let's first show how one can find a colorful k-path. In the last lecture we discussed a dynamic programming algorithm for the Hamiltonian path problem. One important observation we made was that at any stage, the exact path one traverses through does not need to be stored; instead, one only needs to record the *set* of visited nodes, as well as the last visited node.

Here, we can make a similar observation: only the set of visited *colors* and the last node visited are necessary for us to extend a path.

For $S \subset [k]$ and $v \in V$, let $g(S, v)$ be 1 if there exists a path of length $|S|$ (here "length" is the number of vertices in the path) that ends at v and uses all the colors in S, and 0 otherwise. We initialize $g(\{(v)\}, v) = 1$ for all $v \in V$, and $g(c, v) = 0$ if $c \neq c(v)$.

For every size s from 1 to $k - 1$ and every vertex u, the algorithm processes all pairs (S, u) with $|S| = s$ using the following principle:

$$g(S, u) = 1 \text{ and } (u, v) \text{ is an edge and } c(v) \notin S \Rightarrow g(S \cup \{c(v)\}, v) = 1.$$

In other words, if there is an s-length path to u using all colors from S, u has an edge to v and v is colored using a color not in S, then we can reach v using an $s + 1$ length path using all colors in $S \cup \{c(v)\}$.

When all sets of size $k-1$ are processed, the algorithm can return 1 iff there is a set T of size k and a node u such that $g(T,u)=1$. The correctness of the algorithm follows by induction. The runtime of the algorithm is

$$\sum_{s=1}^{k-1}\binom{k}{s}m \leq m2^k.$$

This is since every edge (u,v) is processed once for each set S for which $g(S,u)=1$, and no more.

Now let us fix a particular k-path P and consider the probability that our random coloring $c : V \to [k]$ assigns the nodes of P distinct colors. This probability is $\frac{k!}{k^k}$ (there are k^k possibilities for coloring P, $k!$ of which are colorful). Since $k! > (\frac{k}{e})^k$ by Stirling's inequality, we know that

$$\Pr[k\text{-path } P \text{ is colorful}] > \left(\frac{1}{e}\right)^k.$$

Exercise: Show that if we choose $10e^k$ random colorings c and look for a colorful path using each of them, then the probability we successfully find a k-Path if one exists is constant. This gives a one-sided error: if a k-path is found, then the graph indeed has a k-path, and if no k-path is found, then the probability that the graph has a k-path is at most a constant.

Our final algorithm is thus as follows:
(1) Choose $10e^k$ random functions $c : [n] \to [k]$.
(2) For each of them, look for a colorful k-path.

Each call to colorful k-path in step (2) takes $O^*(2^k)$ time. Thus we get $O^*((2e)^k) \leq O^*(5.437^k)$ time in total, and a constant probability of success.

A k-perfect hash family of functions can be used to *derandomize* the algorithm, similar to last time. If one uses the Naor-Schulman-Srinivasan family, the runtime of the deterministic algorithm is within a $k^{O(\log k)}$ factor of the randomized one.

2 Algorithm 3: Using a larger palette

The above algorithm runs in about $O^*(5.44^k)$ time. We can in fact get a better running time, by choosing a slightly "larger" color palette than k.

Theorem 2.1 (Hueffner et al. 07) *k-Path is in $O^*(4.32^k)$ randomized time.*

Consider the following modification of our previous color-coding algorithm. For a parameter $a \geq 1$:

(1) Randomly map the n nodes of the graph to $a \cdot k$ colors (instead of k).
(2) Find a colorful k-path in a graph with $a \cdot k$ colors.

For part (1) we need to consider the probability that a fixed k-path P is colorful:

$$p := \Pr_{c:[n]\mapsto[ak]}[k\text{-path } P \text{ is colorful}] = \frac{|\{c : [k] \mapsto [ak] \mid P \text{ is colorful}\}|}{|\{c : [k] \mapsto [ak]\}|} = \frac{\binom{ak}{k}\cdot k!}{(ak)^k}.$$

To see the above, note that to specify a function c in the *numerator*, we can pick the set of k distinct colors from $[a \cdot k]$ that go in the k-path, then we can pick a permutation on those colors. The denominator is just the total number of such mappings.

This success probability p is slightly better than before (although it may be hard to see in its current form).

For part (2), the running time is

$$O^*\left(\sum_{i=1}^{k}\binom{a\cdot k}{i}\right),$$

as we can proceed with the same algorithm as before, except that the sets of colors come from $[a\cdot k]$ instead of $[k]$. This is slightly worse than before, but not much worse if a is close to 1.

Define $\binom{a\cdot k}{\leq k} := \sum_{i=1}^{k}\binom{a\cdot k}{i}$.

> **Exercise:** Convince yourselves that one can indeed find a colorful k-path in a graph with ak colors in $O^*(\binom{a\cdot k}{\leq k})$ time.

Repeating for $1/p$ times, to achieve constant success probability, our running time is then

$$O^*\left(\frac{(ak)^k\cdot\binom{a\cdot k}{\leq k}}{\binom{ak}{k}\cdot k!}\right).$$

We want to pick a to minimize this expression.

It turns out that the best choice of a is close to 1, and so we have to use the crude bound $\binom{ak}{\leq k}\leq 2^{ak}$, and we need to use the binary entropy function to estimate $\binom{ak}{k}$.

(Note for $a\leq 2$, we already have $\binom{ak}{\leq k}\geq\Omega(2^{ak}/\sqrt{ak})$).

In particular, setting $a:=1.3$, we get $p\geq 1/1.752^k$, and a running time of $O^*(2^{1.3k}\cdot 1.752^k)\leq O^*(4.32^k)$.

> **Exercise:** Check that the above calculations make sense.

3 Algorithm 4 - Divide and Conquer

One shared weakness of the previous algorithms is their excessive space usage (exponential). To solve this problem, we introduce a divide-and-conquer algorithm which we call ALGO, similar to the one for Hamiltonian path.

The idea is to randomly assign each node to either a set L or another set R with equal probability, thus partitioning the nodes. Once we have divided all the nodes into the two sets, we recurse on the two sets, and require the first $\lceil\frac{k}{2}\rceil$ nodes of the path to be in L and the last $\lfloor\frac{k}{2}\rfloor$ to be in R; we will show that this happens with good probability. In order to be able to patch up two subpaths, we solve a more general problem: instead of looking for one k-path, we compute an $n\times n$ matrix, the (u,v) entry of which is nonzero if and only if there is a k-path between u and v.

Specifically, let B_V be a $n\times n$ matrix where

$$B_V(k,u,v)=\begin{cases}1 & \text{if there exists a } k\text{-path in } V \text{ from } u \text{ to } v,\\ 0 & \text{otherwise.}\end{cases}$$

Then we can compute B_V from the matrices B_L and B_R computed recursively on L and R (i.e. by ALGO(L, $\lceil\frac{k}{2}\rceil$) and ALGO(R, $\lfloor\frac{k}{2}\rfloor$)) by noticing that

$$B_V(u,v)=1 \text{ if there an edge } (x,y) \text{ such that } B_L(\lceil\frac{k}{2}\rceil,u,x)=1 \text{ and } B_R(\lfloor\frac{k}{2}\rfloor,y,v)=1.$$

At the end of all recursive calls, we return a k-path, if one is found between any one of the pairs of vertices $u, v \in V$.

Let K be the original value of the parameter, and let k be its value in the current call of ALGO. Suppose that instead of just one random partition into L and R, we try $2^k \ln(2K)$ random partitions, and return 'yes' iff one of them succeeds (and also store a corresponding witness path for each 'yes'). Then the running time function T satisfies

$$T(n,k) \leq 2^k \cdot \ln(2K)(T(n, \lceil \frac{k}{2} \rceil) + T(n, \lfloor \frac{k}{2} \rfloor) + n^3).$$

Note that in the above recurrence K is separate from k.

Solving the recurrence gives

$$T(n,k) = O^*(4^{k+o(k)}(\log K)^{2\log k}).$$

To see this, suppose (for instance) that $T(n,k') \leq 4^{k'+\sqrt{k'}} n^c (\ln(2K))^{2\log k'}$ for all $k' < k$ and $c > 3$. Then

$$T(n,k) \leq 2^k \ln(2K)[n^3 + n^c(\ln(2K))^{2\log(\lceil k/2 \rceil)} \cdot (4^{\lceil k/2 \rceil + \sqrt{\lceil k/2 \rceil}} + 4^{\lfloor k/2 \rfloor + \sqrt{\lfloor k/2 \rfloor}})] \leq$$

$$\leq 2^k \cdot 4^{k/2+O(1)+\sqrt{k/2}} (\ln(2K))^{1+2(\log k)-1} \cdot n^c$$

and for k larger than a fixed constant, this is at most

$$2^k (4^{k/2+\sqrt{k}} n^c)(\ln(2K))^{2\log k} = 4^{k+\sqrt{k}} n^c (\ln(2K))^{2\log k}.$$

(Above we could have taken a smaller function besides square root but square root suffices.)

Thus, $T(n,K) \leq O^*(4^{K+o(K)})$.

Now consider the probability that the algorithm will find a fixed K-path. There are a total of $(K-1)$ partitions of the path vertices that the algorithm needs to find correctly: the partition that splits the path into the left $\lceil K/2 \rceil$ nodes and the right $\lfloor K/2 \rfloor$ nodes, and the $\lceil K/2 \rceil - 1$ partitions of the left nodes, and the $\lfloor K/2 \rfloor - 1$ partitions of the right nodes (until one node is left in each set).

Consider a fixed partition of k nodes into L and R during a recursive call of the algorithm. The probability that one random trial works is $1/2^k$, and the probability that none of the $2^k \ln(2K)$ random partitions work is at most $(1 - 1/2^k)^{2^k \ln(2K)} \leq 1/e^{\ln(2K)} = 1/(2K)$. By a union bound, the probability that at least one of the $K-1$ partitions of the K-path fails is at most $(K-1)/2K < 1/2$. Thus with probability at least $1/2$ for some branching path of the algorithm all $(K-1)$ splits of the path are found, and the K-path is computed.

Finally, let's see how to derandomize the algorithm. For an n-bit string s and a set $U \subset [n]$, let s_U be the $|U|$-bit substring of s obtained by deleting $s[j]$ for $j \notin U$. For a set $S = \{s^1, \ldots, s^m\}$, let $S_U = \{s^1_U, \ldots, s^m_U\}$.

Definition 3.1 *A set X of n-length binary strings is (n,k)-universal if and only if for any subset $U \subset \{1, 2, \ldots, n\}$ such that $|U| = k$, there exists a set $S \subset X$ with $|S| = 2^k$ such that S_U is exactly the possible 2^k binary strings of length k.*

If we have a universal set of strings, then instead of the $2^k \ln(2K)$ random partitions, we can go through all strings s in the universal set, and assign each node v to L if $s[v] = 0$ and to R otherwise. By definition, some s will give the correct partition of the k-path into L and R. There exists a (n,k)-universal set of binary strings of size $2^k k^{O(\log k)}$ (Naor, Schulman, Srinivasan '95), and hence there is a nearly optimal derandomization.