ORCO – Graphs and Discrete Structures
November 16, 2022 – Lecture 7

1 Finding a triangle

What is the complexity of deciding whether some graph G with n vertices and m edges has a triangle? Given the adjacency matrix A_G of G as input, we can check for all triples of vertices x, y, z if there are all possible edges between these three vertices in G. This costs $O(n^3)$ time. A slightly more clever way is by first listing all m edges of G (in time $O(n^2)$), and then, for all edges uv in the list and all vertices w in G, testing if w is adjacent to both u and v. This costs $O(n^2 + mn)$ time, and is more efficient than the previous algorithm whenever $m = o(n^2)$. However in general $O(n^2 + mn) = O(n^3)$ so this is not better.
A third way of finding a triangle is to use *matrix multiplication*. It is known that two n by n matrices can be multiplied in time $O(n^\omega)$ with $\omega \approx 2.373$ (beware that the corresponding algorithm is not practical because of the huge hidden multiplicative constant in the complexity, so in practice other algorithms are used). This can be used to find if G has a triangle in time $O(n^\omega)$ as follows: we compute A_G^2 in time $O(n^\omega)$ and then check for any edge uv in G (or in other words for any u, v such that $(A_G)_{u,v} = 1$), whether $(A_G^2)_{u,v} > 0$. Such a pair u, v exists if and only if there is a triangle in G, as $(A_G^2)_{u,v}$ counts the number of common neighbors of u and v.

2 Finding a path

We now consider the problem of finding whether G contains P_k (the path on k vertices) as a subgraph. The algorithm detailed in this section, called *Color coding*, is due to Alon, Yuster, and Zwick (1994).

Theorem 1. *There is a one-sided randomized algorithm running in time $2^{O(k)} \cdot n$, that decides whether n-vertex graphs contain P_k as a subgraph (here one-sided means that if there is no P_k the algorithm will always be correct while if there is a P_k, the algorithm will fail with probability at most $1/2$).*

We will prove the result in 4 steps. Given a graph G whose vertices are colored with k colors, we say that a path P is *colorful* if all vertices of P have distinct colors (note that a colorful path contains at most k vertices).

2.1 Step 1

There is a deterministic algorithm running in time $2^{O(k)} \cdot m$, that given an m-edge graph G with k colors and a vertex s, decides whether G contains a colorful P_k starting at s as a subgraph.

The algorithm uses dynamic programming. For any $1 \le i \le k$ and any vertex v, let $\mathcal{C}(v,i)$ be the collection of all sets of colors appearing on colorful paths of i vertices starting at s and ending at v. Note that all sets in $\mathcal{C}(v,i)$ contain i elements, so each collection $\mathcal{C}(v,i)$ has size at most $\binom{k}{i}$.
Let $c(v)$ denote the color of each vertex v in G. We start with $\mathcal{C}(s,1) = \{\{c(s)\}\}$ and $\mathcal{C}(v,1) = \emptyset$ for each $v \neq s$.
For each $i \ge 1$, we determine the collections $\mathcal{C}(v,i+1)$ from the collections $\mathcal{C}(u,i)$ as follows. For each pair (u,v) of adjacent vertices, and for each set $S \in \mathcal{C}(u,i)$, we check whether $c(v)$ appears in S and if so we add $S \cup \{c(v)\}$ to $\mathcal{C}(v,i+1)$ (if it is not there already).

The main loop runs $2m$ times, and the inner loops require checking whether $c(v) \in S$ for at most $\binom{k}{i}$ sets S of size i, so it takes time at most $i\binom{k}{i}$. It follows that the process from going from step i to step $i+1$ takes time at most $2m \cdot i\binom{k}{i} \le 2mk\binom{k}{i}$. Summing this for all $1 \le i \le k$, the overall time is at most $2m \cdot k \cdot 2^k = 2^{O(k)} \cdot m$, as desired.

2.2 Step 2

There is a deterministic algorithm running in time $2^{O(k)} \cdot m$, that given an m-edge graph G with k colors, decides whether G contains a colorful P_k as a subgraph.

Here we simply add a new vertex s to G, join it to all vertices of G, and color it with color $k+1$. Note that this graph admits a colorful P_{k+1} starting at s if and only if G has a colorful P_k as a subgraph.

2.3 Step 3

There is a one-sided randomized algorithm running in time $2^{O(k)} \cdot m$, that decides whether n-vertex graphs contain P_k as a subgraph (here one-sided means that if there is no P_k the algorithm will always be correct while if there is a P_k, the algorithm will fail with probability at most $1/2$).

We assign colors from $1, \ldots, k$ uniformy at random to the vertices of G, and then run Step 2 above. Note that if G has no P_k, it has no colorful P_k for any coloring, so Step 2 will always answer (correctly) no in this case. So assume G has a P_k and consider such a path P on k vertices in G. As there are k^k possible colorings of P and precisely $k!$ colorings in which P is colorful, the probability that P is colorful is $k!/k^k \geq e^{-k}$ (where we have used Stirling formula or any other variant).

This probability is too small so we need to boost it: we repeat $t = e^k$ times the experiment above (with new random colorings that are independent each time). Now the probability that P failed to be colorful in each of the t random colorings of G is at most $(1 - e^{-k})^t \leq \exp(-e^{-k} \cdot t) = 1/e \leq 1/2$ (where we have used $1 - x \leq \exp(-x)$ for $x \geq 0$).

2.4 Step 4

There is a one-sided randomized algorithm running in time $2^{O(k)} \cdot n$, that decides whether n-vertex graphs contain P_k as a subgraph (here one-sided means that if there is no P_k the algorithm will always be correct while if there is a P_k, the algorithm will fail with probability at most $1/2$).

To prove the desired result, it remains to see how to change the time complexity from $2^{O(k)} \cdot m$ in Step 3 to $2^{O(k)} \cdot n$.

We proceed as follows: before running Step 3, we run a DFS in G, stopping as soon as the depth of the DFS tree reaches k (in this case we have a root-to-leaf path on k vertices, so we are happy, we do not even need to run Step 3). If at the end of the DFS, the DFS tree has height at most k, then we claim that $m \leq kn$ and thus indeed $2^{O(k)} \cdot m = 2^{O(k)} \cdot n$. This is because in a DFS tree all neighbors of a vertex v in G are either in the subtree rooted in v, or are ancestors of v in the tree. By orienting all edges of G from bottom to top (from vertices to their ancestors), we see that each vertex has out-degree at most k, the height of the tree, so $m \leq kn$. This also explains why, if we stop the DFS whenever we reach depth k, the number of steps in the algorithm is $O(kn)$ (instead of $O(m)$ for a general DFS).

2.5 Final remarks

The randomized algorithm presented in the previous section can be adapted to decide if G contains a cycle on k vertices in time $2^{O(k)} \cdot n^{\omega}$ (where $\omega \approx 2.373$

is the exponent of matrix multiplication, see Section 1), generalizing the result on triangles presented in Section 1. More generally, for any k-vertex graph H of treewidth at most 2, we can decide in time $2^{O(k)} \cdot n^{\omega}$ if an n-vertex graph G contains H as a subgraph.

For graphs H of larger treewidth, this a bit more costly: for any k-vertex graph H of treewidth at most t, we can decide in time $2^{O(k)} \cdot n^{t+1}$ if an n-vertex graph G contains H as a subgraph.

All the algorithms presented above or mentioned here can be derandomized with a small loss in the complexity (a multiplicative factor of $\log n$) using so-called *k-perfect families of hash functions*.

Finally, for even k, we can find a cycle of length k deterministically in time $O(k! \cdot n^2)$ using different techniques.