1 Computing the Shortest Cycle

Last time we presented an algorithm BFS-Cycle(s) that always returns a cycle of length $\leq q+1$ if s lies on a cycle of length q. Recall also that if q is even, then the cycle returned by BFS-Cycle(s) is $\leq q$.

Now let G be a graph with unknown girth g. We have the following claims about BFS-Cycle.

Claim 1. *If g is even, then running* BFS-Cycle *on all nodes will compute the girth g exactly.*

Claim 2. *Suppose the estimate returned by running* BFS-Cycle *on all nodes is g'. If g' is odd, then $g' = g$.*

Proof of Claim 2. By contradiction. If $g' \neq g$, then $g = g' - 1$, which is even, so by Claim 1, $g' = g$, which is a contradiction. □

Thus, if running BFS-Cycle reports that the girth is odd, we know that it reported the true value. What remains is to show how to determine the true girth if the algorithm reports an even girth estimate. Let $g' = 2r$ be the reported estimate of the girth. We know that $g = 2r$ or $g = 2r - 1$, and we know r exactly.

Our job is now to determine whether the given graph has a cycle of length $2r - 1$. We know r, so if r happens to be a constant, then we can just run our $O(n^\omega)$ time algorithm that we mentioned last time and find a $2r - 1$-cycle if one exists. The trouble is that if r is not a constant, say $\log^2 n$, then the algorithm won't even run in polynomial time since the constant in the big-O depends exponentially on k. Here we present a different way to solve this problem by reducing it to triangle finding.

We will use the following very useful claim.

Claim 3. *If* BFS-Cycle*(s) returns a cycle of length $2r$, then all nodes of distance $r-1$ from s are visited by* BFS-Cycle*(s) and hence placed in L_{r-1}.*

Proof of Claim 3. [Sketch] In the case that a cycle of length $2r$ is returned by BFS-Cycle(s), an edge from level L_{r-1} to L_r was scanned to complete this cycle. However, in order for BFS-Cycle(s) to scan an edge out of L_{r-1}, no cycles were found until now and all nodes at distance $r-1$ must have been placed in L_{r-1}. This is because we only begin scanning edges out of L_{r-1} when all edges out of L_{r-2} have been scanned, and using induction we can show that L_{r-2} must contain all nodes at distance $r-2$ from s. □

Given this Claim we perform the following reduction to the problem of finding a triangle. Create a graph $G' = (V', E')$ which consists of three sets A, B, and C on n nodes each.[1]

For each node $v \in V$ there are three copies: $v_A \in A, v_B \in B$ and $v_C \in C$. We add edges (in E') to G' as follows. For each $(u, v) \in E$ of our original graph we will include an edge (u_B, v_C).

The nodes in A will represent the source nodes for each run of BFS-Cycle. For each $s \in V$, we add an edge between $s_A \in A$ and $v_B \in B$ if and only if $v \in L_{r-1}$ at the end of BFS-Cycle(s). Similarly $(s_A, u_C) \in E'$ iff $u \in L_{r-1}$ at the end of BFS-Cycle(s).

Claim 4. *If G has a $2r-1$ cycle and $g' = 2r$ then G' has a triangle.*

[1] In class we presented a different reduction where there only two sets of nodes, S and V. All edges of G are present within V. No edges are between nodes of S, and there's an edge between $s \in S$ and $v \in V$ if BFS-Cycle(s) returned a cycle of length $g' = 2r$ and v is in L_{r-1} in the run of BFS-Cycle(s). Then any triangle G' is either a triangle in G and hence a shortest cycle, or contains one node $s \in S$ and two nodes $u, v \in V$ with $(u, v) \in E$, and then taking the LCA of u and v in the BFS tree rooted at s we find a cycle from c to v to u and back to c of length $\leq 2(r-1) + 1 = 2r - 1$. Since the girth must be at least $2r - 1$, we get that the cycle is of length exactly $2r - 1$ (so $s = c$) and is a shortest cycle.

Proof of Claim 4. Assume there is a cycle through s of length $2r-1$ and that u, v are the nodes furthest from s on the cycle. Then u, v must be visited by BFS-Cycle(s) and in L_{r-1} since $d(s,u), d(s,v) \leq r-1$ and no $2r-1$-cycle (or shorter) was returned. This means that $(s_A, u_B), (s, v_C), (u_B, v_C) \in E'$, so a triangle exists in G'. □

Claim 5. *If G' contains a triangle, then there is a $2r-1$ cycle in G.*

Proof of Claim 5. Note the only way for a triangle to form in G' is if one node from each partition participates, as there are no edges within A or within B or within C. Thus, let there be a triangle between s_A, u_B, and v_C corresponding to nodes $s, u, v \in V$. Consider the paths from s to u and s to v in G which we will call p_u and p_v, respectively. While p_u and p_v can have some overlap, we know that $u \notin p_v$ and $v \notin p_u$. This is since p_u and p_v are the shortest paths from s to u and s to v, respectively, and $d(s,u), d(s,v) = r-1$.

We call the last node starting from s that p_u and p_v share s'. Note that s' exists since p_u and p_v share s; s' is exactly the least common ancestor of u and v in the BFS tree rooted at s.

Since $u \notin p_v$ and $v \notin p_u$, $s' \notin \{u, v\}$. Furthermore, the paths between s' and u and s' and v are internally disjoint. Thus, these paths form a simple path from u to v that is different from (u, v), and hence together with (u, v) it forms a simple cycle. The length of this cycle is

$$d(s', u) + d(s', v) + 1 \leq d(s, u) + d(s, v) + 1 \leq 2(r-1) + 1 = 2r - 1.$$

Since we know that $g \geq 2r-1$, the length of the cycle must be exactly $2r-1$ and it is thus the shortest cycle. □

2 Finding a cycle of length exactly k.

Now we will consider how long it takes to find a cycle of length exactly k in a graph on n nodes, with runtime depending on both k and n.

The main tool will be *Color-coding*.

The basic idea of Color-coding is to give nodes random colors, so that with high probability our object of interest (here, the k-cycle) is colored in a nice way that makes it easier to find. Color-coding was developed by Alon, Yuster and Zwick (1997) and has been widely used ever since. Usually color-coding ensures that some set on k items that we are interested in is colored by k distinct colors and that a colorful set of interest is easy to find.

While algorithms based on color-coding are naturally randomized, there is an efficient way to derandomize them The basic tool used for derandomization are k-perfect families of hash functions. Namely, there's a constant c s.t. for every n and k there exist $t \leq O(c^k \log n)$ functions (that are easy to construct) $h_1, \ldots, h_t$ so that each h_i is from $\{1, \ldots, n\}$ to $\{1, \ldots, k\}$ and for any set $S \subseteq \{1, \ldots, n\}$ of size k, there is some h_i such that the set $\{h_i(s) \mid s \in S\}$ contains all k colors $1, \ldots, k$, i.e. h_i makes S colorful. Then instead of picking a random coloring, one can iterate through the t hash functions h_i and interpret them as colorings.

We'll first give an $O(k^k n^\omega \log n)$ time algorithm for k-cycle. For sparse graphs we can also give an $O(k^k mn \log n)$ time algorithm that might run faster. We'll describe a randomized algorithm that is correct with high probability, but keep in mind that one can derandomize it by using the hash families as above.

Given a directed or undirected graph $G = (V, E)$ in which we want to find a k-cycle, we pick a random color $c(v)$ for each $v \in V$ independently uniformly at random from $\{1, \ldots, k\}$.

Now suppose that G contains some unknown k-cycle C, $x_1 \to x_2 \to \ldots, x_k \to x_1$. The probability that $c(x_i) = i$ is $1/k$ so the probability that for every i, $c(x_i) = i$ is $1/k^k$.

Now suppose that we perform $100k^k \ln n$ iterations of coloring the vertices. What is the probability that in every single trial, C is never colorful (i.e. in each trial, for some i, $c(x_i) \neq i$)? It's

$$\left(1 - \frac{1}{k^k}\right)^{100k^k \ln n} \leq \left(\frac{1}{e}\right)^{100 \ln n} = \frac{1}{n^{100}}.$$

Thus with probability at least $1 - 1/n^100$, one of the colorings will color C nicely. Now fix that coloring c. We will show how to find a nicely colored k-cycle in G.

Consider the directed graph G' that has the same vertices as G, but (u, v) is a (directed) edge if and only if $(u, v) \in E$ and $c(v) = c(u) + 1$.

We claim that if u is a node of color 1 and v is a node of color k, then v is reachable from u in G' if and only if there is a k-path in G from u to v, $u = x_1 \to x_2 \to \ldots x_k = v$ where $c(x_i) = i$ for every $i \in \{1, \ldots, k\}$.

The first direction is immediate. If there is a k-path in G from u to v, $u = x_1 \to x_2 \to \ldots x_k = v$ where $c(x_i) = i$ for every $i \in \{1, \ldots, k\}$, then every edge (x_i, x_{i+1}) is in G' and hence v is reachable from u. For the other direction, if there is a path in G from u to v, $u = x_1 \to x_2 \to \ldots x_L = v$ for some L, then since (x_i, x_{i+1}) is an edge in G' only if $c(x_i) + 1 = c(x_{i+1})$ and (x_i, x_{i+1}) is an edge of G, we get that the colors on the path are increasing by 1 with each edge, and since $c(v) = k$ and $c(u) = 1$ we get that $L = k$ and we have a nicely colored k-path from u to v in G.

Similarly, suppose that C is a k-cycle in G, $C = x_1 \to x_2 \to \ldots, x_k \to x_1$ and that for each $i \in \{1, \ldots, k\}$, $c(x_i) = i$. Then in G', x_1 can reach x_k. Conversely, if there is an edge $(u, v) \in E$ such that $c(u) = 1$ and $c(v) = k$ and u can reach v in G', then there is a (nicely colored) k-cycle in G.

Hence, in order to find a nicely colored k-cycle in G (if one exists) it suffices to compute all pairs of nodes u, v such that u can reach v in G', i.e. compute the transitive closure of G'. Then one can check whether there is an edge $(u, v) \in E$ such that $c(u) = 1$ and $c(v) = k$ and u can reach v in G' in only $O(m)$ extra time (where $|E| = m \leq n^2$).

One way to compute the transitive closure of G' is to run BFS from every node v in G' - this computes for every v the nodes reachable from v and runs in $O(mn)$ time since G' has n nodes and $\leq m$ edges. Another way to compute the transitive closure is to use matrix multiplication.

Define A to be the $n \times n$ matrix where $A[i, j] = 1$ if $(i, j) \in E$ and $c(j) = c(i) + 1$, and $A[i, j] = 0$ otherwise. Consider the square A^2 of A. $A^2[i, j] = \sum_k A[i, k] \cdot A[k, j]$ which is nonzero whenever there is a k such that $(i, k), (k, j) \in E$ and $c(j) = c(k) + 1 = c(i) + 2$. Similarly, we can show by induction that for every ℓ, and every i, j,

$A^\ell[i, j] \neq 0$ if and only if there are distinct nodes $i = x_1, x_2, \ldots, x_k$ s.t. $\forall t, (x_t, x_{t+1}) \in E$ and $c(x_{t+1}) = c(x_t)+1$.

Thus, if we compute A^{k-1}, we would have what we want. We can easily compute A^{k-1} using $k - 2$ matrix multiplications, $A^i \leftarrow A \cdot A^{i-1}$. However, one can also compute it using $O(\log k)$ matrix products: imagine k is a power of 2, then we perform the products $A^j \leftarrow A^{j/2} \cdot A^{j/2}$ for j from 2 until k. This is called successive squaring.

However, there's an even better result. Fischer and Meyer (1971) showed that the transitive closure of an n node directed or undirected graph can be computed in $O(n^\omega)$ time, i.e. no extra logs are necessary. Thus, using their algorithm, we can find a k-cycle in $O(k^k n^\omega \log n)$ time with high probability.

Improving k^k to c^k. The k^k dependence in the runtime of the algorithm is not great. It means that we can find a cycle of length $O(\log n / \log \log n)$ in polynomial time, but we might want to find a longer cycle. We don't believe that we can get a subexponential dependence on k since the case $k = n$ is the Hamiltonian cycle problem which is widely believed to require exponential time. However, a c^k dependence for some constant c might be possible. Here we give some intuition on how to get this.

We draw random colors for the nodes of G as before. Let C be a k-cycle in G again, $C = x_1 \to x_2 \to \ldots, x_k \to x_1$.

Suppose that instead of requiring that for each $i \in \{1, \ldots, k\}$, $c(x_i) = i$, we only require that C is colorful in the sense that the set of colors $\{c(x_1), \ldots, c(x_k)\}$ contains all k colors $\{1, \ldots, k\}$. Then actually, the probability that C is colorful is $k!/k^k > 1/e^k$ which is much better than $1/k^k$. Then, we can just repeat our coloring $O(e^k \log n)$ times and we will know that with probability at least $1 - 1/n^{100}$ (say), in one of the colorings C is colorful.

The question is, how fast can one find a colorful k-cycle in the graph. We no longer know what color the ith cycle node uses; we only know that all the cycle nodes have distinct colors.

Here we can use dynamic programming. The dynamic programming table T will contain an entry for every subset $U \subseteq \{1, \ldots, k\}$ and $T(U)$ will be an $n \times n$ matrix such that

$T(U)[u, v] \neq 0$ if and only if there are $|U|$ nodes $u = x_1, x_2, \ldots, x_{|U|} = v$ such that $\{c(x_i) \mid i = 1, \ldots, |U|\} = \{1, \ldots, k\}$.

In other words, $T(U)$ contains all pairs of nodes u, v connected by a colorful path of length $|U|$ that uses only and all nodes of U. Then, G contains a colorful k-cycle if and only if there is an edge $(v, u) \in E$ such that $T(\{1, \ldots, k\})[u, v] \neq 0$.

To compute $T(U)$, for every $U' \subseteq U$ with $|U'| = \lfloor |U|/2 \rfloor$, compute the matrix $M_{U,U'} = T(U') \cdot T(U \setminus U')$ and then set $T(U)[u, v]$ to be 1 if there is some U' such that $M_{U,U'}[u, v] \neq 0$ and otherwise set $T(U)[u, v] = 0$.

Intuitively we are picking the colors U' of the first $\lfloor |U|/2 \rfloor$ nodes on the colorful $|U|$-path from u to v, and so the colors of the last $\lceil |U|/2 \rceil$ nodes on the path must have colors from $U \setminus U'$. The product of $T(U')$ and $T(U \setminus U')$ exactly computes those pairs of vertices u, v that have a colorful U-path between them where the first half of the nodes are colored using U' and the second half using $U \setminus U'$.

The runtime for computing $T(U)$, given the $T(U')$ and $T(U \setminus U')$ matrices for all $|U'| = \lfloor |U|/2 \rfloor$ takes $O(2^{|U|} n^\omega)$ time since there are $O(2^{|U|})$ choices for U'.

The final runtime of the algorithm is thus no more than (asymptotically)

$$n^\omega \sum_{j=1}^{k} \binom{k}{j} 2^j \leq 3^k n^\omega.$$

With some extra observations one can replace the 3^k with a 2^k. The overall runtime is obtained by multiplying by the $O(e^k \log n)$ number of trials, giving a $O(c^k n^\omega \log n)$ time algorithm for $c \leq 2e$.