6.890 Lecture 10 k-Cycle and Shortest Cycle/Girth
Scribe: Virginia Williams **Date:** October 11, 2021
Editor: Andrea Lincoln, Victor Kaiser-Pendergrast, Yinzhan Xu

1 Finding longer cycles

In lecture 2 we gave an $O(n^\omega)$ time algorithm for triangle detection. A natural question is, can one detect longer (not necessarily induced) cycles faster than triangles? Formally, the k-cycle problem asks that given a directed/undirected graph G, whether G contains a simple non-induced cycle of length k. Throughout this lecture, we will assume k is a constant. It turns out that for any $k = O(1)$, we can detect a k-cycle in the same time as detecting a triangle.

Theorem 1.1. *Let $k \geq 3$ be a constant integer. There is an $\tilde{O}(n^\omega)$ time algorithm (for $\omega < 2.373$) for k-cycle in n-node graphs.*

Let A be the adjacency matrix of the graph G. Recall that in the algorithm for triangle detection, we just compute the trace of A^3. If $A^3[i,i] \neq 0$, then there is a triangle containing node i. However, a similar algorithm does not work for k-cycle since $A^k[i,j] \neq 0$ if and only if there exists a length k walk from i to i, which is different from a length k *simple cycle*.

Proof of Theorem 1.1. The main technique we use is called color-coding; we saw it in the last lecture. For every $v \in V$, we pick a color $c(v) \in [k]$ independently at random. Note that for any fixed k cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$, with probability at least $\frac{1}{k^k}$, $c(v_i) = i$ for any $1 \leq i \leq k$.

For each $i \in [k-1]$, we define a matrix A_i. The rows of A_i are indexed by vertices of color i, and the columns of A_i are indexed by vertices of color $i+1$. The entry $A_i[u,v] = 1$ if $(u,v) \in E$ and $A_i[u,v] = 0$ otherwise. Now we compute the product $A_1 A_2 \cdots A_{k-1}$ in $O(n^\omega)$ time. For every $(u,v) \in E$ such that $c(u) = k$ and $c(v) = 1$, we check whether $(A_1 A_2 \cdots A_{k-1})[v,u] > 0$. If it is true for any u, v, then there exists a k-cycle. Otherwise, there is no k-cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$ where $c(v_i) = i$ for every i.

Note that if the algorithm detects a k-cycle, then G must have a k-cycle. Also, when G has a k-cycle, the algorithm detects a k-cycle with probability at least $\frac{1}{k^k}$. Thus, we can repeat the algorithm for $\Theta(\log n)$ rounds, and we will have the correct answer with high probability.

It is worthwhile to mention that it is possible to derandomize this algorithm using special hash functions, but we won't cover it in this lecture. ☐

Hence, a k-cycle for any constant k can be found in the same time as a triangle. Can a k-cycle be found faster than a triangle? We answer this question differently for two cases:

- For directed cycles or for undirected odd cycles, they are at least as hard as finding triangles.
- For undirected even cycles, they can be found more quickly than triangles.

In the rest of this section, we prove the first case, and we prove a special case of the second case for 4-cycles.

1.1 Odd Cycles and Directed Cycles are as Hard as Triangles

Theorem 1.2 (Directed cycles are at least as hard to find as triangles are)**.** *For any constant $k \geq 3$, if there is a $T(n)$ algorithm for k-cycle in directed graphs, then there is an $O(T(O(n)))$ time algorithm for triangle detection.*

Proof. Let $G = (V_1 \cup V_2 \cup V_3, E)$ be a graph on $3n$ nodes for which we want to find a triangle, where $|V_1| = |V_2| = |V_3| = n$. We create a new graph G' on kn nodes as follows.

The vertex set of G' is $V_1' \cup V_2' \cup V_{3,1}' \cup \cdots \cup V_{3,k-2}'$, which contains one copy of V_1, one copy of V_2 and $k-2$ copies of V_3. We add edges from V_1' to V_2' using edges between V_1 and V_2; add edges from V_1' to $V_{3,1}'$ using edges between V_1 and V_3; add edges from V_2' to $V_{3,k-2}'$ using edges between V_2 and V_3; finally, for every $1 \leq i \leq k-3$, and for every $u \in V_{3,i}'$, we add an edge from u to its corresponding copy in $V_{3,i+1}'$.

It is not too hard to see that there is a directed k-cycle in G' if and only if there is a triangle in G. Thus, we can use the $T(n)$ time algorithm for k-cycle on G', which has $O(n)$ vertices. Thus, there is an $O(T(O(n)))$ time algorithm for triangle detection.

□

Theorem 1.2 implies that if we assume triangle detection requires $n^{\omega-o(1)}$ time, then directed k cycle also requires $n^{\omega-o(1)}$ time. The following theorem, combined with Theorem 1.2, shows that odd cycles in undirected graphs also require triangle detection time.

Theorem 1.3 (Odd cycles are at least as hard to find as triangles)**.** *If k is an odd constant, then directed k-cycle can be reduced to undirected k-cycle.*

Proof. Let G be a graph for which we want to detect k-cycle. First, via color-coding, we can assume G is a k-partite layered graph. We make a new graph G' just by removing the edge directions of G. We claim that G has a k-cycle if and only if G' has a k-cycle.

The forward direction is easier. If G has a k-cycle C, then the same cycle is a k-cycle in G'. For the backward direction, assume G' has a k-cycle, then this k-cycle must have a node in each of the k parts of G'. Suppose not, then this k-cycle is contained in $G' \setminus V_i$, where V_i is one of the k parts of G'. However, $G' \setminus V_i$ is bipartite, so it cannot contain an odd cycle, which leads to a contradiction. Thus, this k-cycle must have a node in each of the k parts of G'. Such a k-cycle is also a k-cycle in G. □

1.2 Even Cycles in Undirected Graphs

We can find even cycles in undirected graph faster than the time needed for triangle detection. The following theorem in by Yuster and Zwick.

Theorem 1.4 (Even cycles can be found more quickly than triangles)**.** *For any constant $k \geq 2$, there is an $O(n^2)$ time algorithm that finds a $2k$-cycle in a given undirected n-vertex graph, or determines that no such cycle exists.*

In this lecture, we will only show a special case for Theorem 1.4 when $k = 2$. We will show how to find a 4-cycle or detect that there is no 4-cycle in $O(n^2)$ time.

Theorem 1.5. *There is an $O(n^2)$ time algorithm that finds a 4-cycle in any given n-node graph G, or determines that G does not contain any 4-cycles.*

Proof. Consider Algorithm 1:

In this algorithm, T is an n by n matrix whose entries are either 0 or vertices of G, with the meaning that $T[s,t]$ is a vertex that is a neighbor of both s and t.

If u, s, v, t is returned by the algorithm, then s, t are neighbors of v, and $u = T[s,t]$ which also means that u is a neighbor of both s and t. Since $u \neq v$, the algorithm has returned a 4-cycle.

Suppose that G contains a 4-cycle a, b, c, d, where a is the latest vertex visited in the outer loop. Then, when the outer loop has $v = a$, and the inner loop has $s = b$, $t = d$, either the algorithm has already found a 4-cycle, or the algorithm will find that $T[s,t]$ is already set. Observe that the loop for c has already run, and that b and d have been found to be neighbors of c. Hence, a 4-cycle will be returned. Thus, the algorithm is correct.

Let's consider the runtime: in each iteration, either the algorithm halts, or a new pair (s,t) is set in T. Thus, the runtime of the algorithm is upper-bounded by the number of entries of T, which is $O(n^2)$. □

Algorithm 1: FourCycle($G = (V, E)$)

```
$T \leftarrow 0$;
foreach $v \in V$ do
    foreach $s, t \in N(v)$ do
        if $T[s, t] = 0$ then
            $T[s, t] \leftarrow \{v\}$;
        else
            $u \leftarrow T[s, t]$;
            return $u, s, v, t$;
return no 4-cycles;
```

2 Shortest Cycle / Girth

In this section, we will discuss how to find the shortest cycle of a given graph. Formally, we want to find a simple cycle of minimum length. We define the *Shortest Cycle* problem as follows: given an undirected graph $G = (V, E)$, compute the length of the shortest simple cycle, called the girth, (or report that no cycles exist in G).

Definition 2.1 (Girth)**.** *The* girth *of a graph G is the length of the shortest cycle in G, or ∞ if G contains no cycles.*

The girth g is a natural graph parameter, and its properties are well studied in graph theory. Let us look at how large g can be. By the definition of a simple cycle, $3 \leq g \leq n$; moreover, for any number of nodes n, there exist graphs which have girth g for any choice of g in $\{3, \ldots, n\}$. (To see this, consider constructing the minimum cycle first and then adding additional nodes as necessary.)

We know that the problem of finding the longest cycle of a graph is NP-Complete, but as it turns out the problem of finding the shortest cycle is contained in P. The question we aim to answer is how quickly we can actually find such a shortest cycle.

We first observe that any algorithm for finding a shortest cycle must take at least the amount of time it takes to detect whether or not a triangle exists in the graph: a triangle does not exist if and only if the girth is greater than 3. Intuitively though, the problem at first seems even harder, as we may need to consider cycles of arbitrary size, not just triangles. But in fact, a theorem from Itai and Rodeh [1] shows otherwise:

Theorem 2.1 (Itai, Rodeh'78)**.** *If there is an algorithm A that finds a triangle in an n-node graph in time $T(n)$, then one can also compute the girth of an n-node graph in $O(n^2 + T(2n))$ time.*

Recall that finding a triangle deterministically takes $\Omega(n^2)$ time as we at the very least need to read the input. Also, for all "nice" functions (e.g. polynomials, polylogarithms etc) $T(2n) = O(T(n))$, so the times are essentially asymptotically equivalent.

The triangle problem and the shortest cycle problem on n node graphs are equivalent.

Itai and Rodeh prove another theorem, claiming the existence of an additive approximation algorithm for computing the girth.

Definition 2.2. *An additive c-approximation (or $+c$-approximation) for a quantity g is a quantity g' such that $g \leq g' \leq g + c$.*

Theorem 2.2 (Additive 1-approximation, Itai, Rodeh'78)**.** *There exists an $O(n^2)$ time algorithm that finds a cycle of length $\leq g + 1$ in any n-node graph of girth g, or determines that G contains no cycles. If g is even, then the algorithm finds the shortest cycle of G.*

We will prove these theorems as follows: first, we will exhibit the $O(n^2)$ algorithm described by Theorem 2.2. Then, we will use this algorithm to show a proof for Theorem 2.1.

2.1 Computing an Additive Approximation using BFS

We start our discussion of how to approximate the girth of an arbitrary graph with an algorithm for finding cycles given a starting node.

Lemma 2.1. *There exists an algorithm* BFS-CYCLE*(s) that given $G = (V, E)$ and $s \in V$ that lies on a cycle C of length q runs in $O(n)$ time and returns a cycle of length $\leq q + 1$. If q is even, it returns a cycle of length $\leq q$.*

Algorithm 2: BFS-CYCLE(s)

```
$L_0 \leftarrow \{s\}$;
$visited[s] \leftarrow$ true;
foreach $u \neq s \in V$ do
    $visited[u] \leftarrow$ false;
foreach $i \geq 0$ do
    if $|L_i| = 0$ then
        return "no cycle";
    $L_{i+1} \leftarrow \{\}$;
    foreach $u \in L_i$ do
        foreach $(u, x) \in E$ do
            remove $(u, x)$ from $E$;
            if $visited[x]$ = $false$ then
                $visited[x] \leftarrow$ true;
                $L_{i+1} \leftarrow L_{i+1} \cup \{x\}$;
            else
                find least common ancestor $c$ of $u$ and $x$;
                return $(u, x) \cup$ path $u \cdots c \cdots x$ in BFS-tree;
```

To be able find the least common ancestor of u and x, it suffices to store, for every node v, a pointer to the parent $p(v)$ in the BFS tree, i.e. the node that was scanned to first visit v. Then, starting at u and x, one follows parent pointers up the tree to find the first common ancestor. This only increases the runtime by a constant factor.

Claim 1. BFS-CYCLE *runs in $O(n)$ time.*

Proof. The algorithm returns once a node is visited more than once, so the runtime is bounded by the number of nodes n. □

We'll call a node, t, "scanned" if the loop "**foreach** $u \in L_i$ **do**" from Algorithm 2 is completed for $u = t$.

Claim 2. *If (u, x) completes a cycle and $u \in L_i$ then $x \in L_i$ or $x \in L_{i+1}$.*

Proof. Whenever an edge from L_{i-1} to L_i is scanned, it is removed. Hence once the last node of L_{i-1} is scanned, there are no more edges from L_{i-1} to L_i. Thus if (u, x) completes a cycle and $u \in L_i$, we cannot have that $x \in L_{i-1}$. By the properties of BFS, x must be in $L_i \cup L_{i+1}$. □

Now we prove a crucial lemma:

Lemma 2.2. *If s is part of a cycle of length q, then if (u, x) closes the cycle returned by* BFS-CYCLE*(s) and $u \in L_i$ then $i \leq \lceil q/2 \rceil - 1$.*

Proof. Let C be a cycle of length q through s. Let (u,x) close the cycle in BFS-CYCLE(s). Recall that a node t is "scanned" if the loop "**foreach** $u \in L_i$ **do**" from Algorithm 2 has been completed for $u = t$.

Assume (for contradiction) that *all* nodes of L_j for $j \leq \lceil q/2 \rceil - 1$ have been scanned and no cycle was found.

Now consider C. Since $|C| = q$, if C is odd, then for every node $x \in C$, $d(s,x) \leq \lceil q/2 \rceil - 1$, and so all nodes of C have been scanned. If C is even, then let v_0 be the furthest node from s on C. Then all nodes x in $C \setminus \{v_0\}$ have $d(s,x) \leq \lceil q/2 \rceil - 1$, and so all nodes of C (except for possibly v_0) have been scanned.

Suppose we have a node $y \in C$ such that its neighbors on C, which we call x, x', were both scanned before y. Then when x and x' were scanned, (x,y) and (x',y) were both present, and y was visited twice. We will show that such a node exists and its neighbors were scanned in level $\lceil q/2 \rceil - 1$ at the latest, thus contradicting our original assumption that all nodes of L_j for $j \leq \lceil q/2 \rceil - 1$ were been scanned and no cycle was found

Suppose first that C is even and v_0 was not scanned among the first $\lceil q/2 \rceil - 1$ levels. Then, since all nodes at distance $\leq \lceil q/2 \rceil - 1$ from s were scanned (among the first $\lceil q/2 \rceil - 1$ levels), the neighbors v_1 and v_2 of v_0 were scanned before v_0. Thus v_0 is visited twice, and since v_1 and v_2 are at distance $\leq \lceil q/2 \rceil - 1$ from s, we see the contradiction to our assumption.

Now suppose that either C is odd or C is even and v_0 was scanned among the first $\lceil q/2 \rceil - 1$ levels. Then by our assumption, all nodes of C have been scanned among the first $\lceil q/2 \rceil - 1$ levels. Let y be the last node on C to be scanned. But then, its neighbors on C, x and x' were scanned before y, thus again giving a contradiction to our assumption.

Hence, in all cases, a cycle is closed by some (u,x) with $u \in L_i$ for $i \leq \lceil q/2 \rceil - 1$. □

We see that Lemma 2.2 allows us to show that if a node s is on a cycle C, of length q, then BFS-Cycle returns a value at most $2\lceil q/2 \rceil$, which equals q if q is even, or $q+1$ if q is odd, thus proving the theorem. To see this, let c be the least common ancestor of u and x in the BFS tree out of s. We can bound the distances from c to u and x as $d(c,u) \leq d(s,u) \leq i$ and $d(c,x) \leq d(s,x) \leq i+1$ because we know that $u \in L_i$ and $x \in L_i \cup L_{i+1}$ by Claim2. This means that the cycle returned has length at most $\leq 1 + i + (i+1) = 2(i+1)$. Thus if Lemma 2.2 holds, then $i \leq \lceil q/2 \rceil - 1$ and we get a cycle of length at most $\leq 2 \cdot \lceil q/2 \rceil$.

2.2 Girth is in Triangle Detection Time

We will prove Theorem 2.1 in this section.

Proof of Theorem 2.1. First, we run BFS-Cycle from every $s \in V$ and get a cycle of length q at most $g+1$, where g is the length of girth. If q is odd, then by Theorem 2.2 $q = g$ and we are done. Thus, we assume $q = 2\ell$ is even in the rest of the algorithm. By Theorem 2.2, g could be q or $q-1$.

Now we argue that running BFS-Cycle from every $s \in V$ has computed all pairwise distances up to $\ell - 1$. Suppose there exists s such that $L_{\ell-1}$ is not fully computed. In this case, BFS-Cycle must have returned a cycle for some $i < \ell - 1$. Thus, the cycle returned by BFS-Cycle has length at most $2i+2 \leq q-2$, a contradiction.

We create a triangle detection instance G'. First, we put a copy of G in G'. Then we create n vertices $v'_1, v'_2, \ldots, v'_n$, which are copies of vertices in G. We add an edge between v'_i and v_j if $d_G(v_i, v_j) = \ell - 1$. Since the distance up to $\ell - 1$ are already computed, it is possible to tell whether the distance between two vertices are $\ell - 1$ or not.

Claim 3. *The graph G' has a triangle if and only if the girth of G is $q-1$.*

Proof. We first prove the forward direction. There could be two types of triangles in G'. For the first type, the triangle completely lies in G, which is a cycle of length less than q. Thus the girth must be $q-1$. For the second type, the triangle has one vertex v'_i connected with v_j and v_k. In this case, the cycle $v'_i \rightsquigarrow v_j \to v_k \rightsquigarrow v'_i$ has length $2(\ell-1)+1 = q-1$. Even if this cycle is not simple, there must be an even shorter simple cycle. Thus, the shortest cycle in G has length less than q, so it must have length $q-1$ since q is a 1-additive approximation.

For the backward direction, suppose G has a girth of length $q-1$. Let s be an arbitrary vertex on girth, and let v_0, v_1 be the two furthest points from s on this cycle. Clearly, the distance from s to v_0 and v_1 are both $\ell - 1$, since otherwise there are shorter cycles. Thus, s', v_0, v_1 must be a triangle in G'. $\square$

By Claim 3, it is sufficient to call triangle detection on G', which has $2n$ vertices. If there is a triangle, we return $q-1$ as the girth; otherwise, we return q as the girth. $\square$

There are extensions of Theorem 2.1. For instance, Roditty and Vassilevska W. [2] provide an $\tilde{O}(n^2)$ time reduction from the girth/shortest cycle problem in an n node undirected graph with edge weights in $\{1, \ldots, M\}$ for $M = \text{poly}(n)$ to the minimum weight triangle problem in an $O(n)$ node graph with weights in $\{1, \ldots, O(M)\}$. They also show that one can reduce the girth/shortest cycle problem in an n node *directed* graph with edge weights in $\{-M, \ldots, M\}$ in $\tilde{O}(Mn^{\omega})$ time to minimum weight triangle in an $O(n)$ node graph with weights in $\{1, \ldots, O(M)\}$. We won't cover the details of these extensions in the lectures.

References

[1] Alon Itai, Michael Rodeh: *Finding a Minimum Circuit in a Graph.* SIAM J. Comput. 7(4): 413-423 (1978).

[2] Liam Roditty, Virginia Vassilevska Williams: *Minimum Weight Cycles and Triangles: Equivalences and Algorithms.* FOCS 2011: 180–189.