CS267 Lecture 2 Cycles and Girth
Scribe: Virginia Williams **Date:** September 30, 2016
Editor: Andrea Lincoln, Victor Kaiser-Pendergrast

1 Finding longer cycles.

Last lecture we gave some algorithms for triangle finding. A natural question is, can one find longer cycles faster than triangles?

We will first state a theorem that we won't prove for now.

Theorem 1.1. *Let $k \geq 3$ be an integer. There is an $O(n^\omega)$ time algorithm (for $\omega < 2.373$) that given an n node graph can find a k-cycle in the graph if one exists.*

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:

Theorem 1.2 (Odd cycles are at least as hard to find as triangles are)**.** *Let $\ell > 1$ be a constant. Suppose that there is an $O(N^t)$ time algorithm (for some constant t) that finds a simple $2\ell+1$-cycle in any given N-node graph, or determines that no such cycle exists. Then, there is also an $O(n^t)$ time algorithm that finds a triangle in any given n-node graph, or determines that no triangle exists.*

Theorem 1.3 (Even cycles can be found more quickly than triangles)**.** *Let $\ell > 1$ be a constant. There is an $O(n^2)$ time algorithm that finds a 2ℓ-cycle in a given n-node graph, or determines that no such cycle exists.*

In the rest of the lecture, we prove Theorem 1.2, and we prove the special case of Theorem 1.3 for 4-cycles.

Proof of Theorem 1.2. Let $G = (V, E)$ be a graph on n nodes for which we want to find a triangle. We create a new graph G' on $(2\ell+1)n$ nodes as follows.

For each $i = 1, \ldots, 2\ell+1$, create a set V_i on n nodes where every node $v \in V$ has a copy $v_i \in V_i$. For every edge $(u, v) \in E$, add edges $(u_1, v_2), (u_2, v_3)$ and (u_3, v_4). These are all edges in $V_1 \times V_2$, $V_2 \times V_3$ and $V_3 \times V_4$.

The rest of the edges of G' are as follows: for every $v \in V$ and every i with $4 \leq i \leq 2\ell+1$, add an edge (v_i, v_{i+1}), where $v_{2\ell+2}$ is the same as v_1. Notice that this makes edges in $V_i \times V_{i+1}$ between the copies of the same vertex in adjacent V_i, V_{i+1} sets.

This completes the description of G'. An example of a simple G and the associated G' is shown in Figure 1.

Suppose first that G contains a triangle u, v, w. Then the following is a $2\ell+1$ cycle in G': $u_1 \to v_2 \to w_3 \to u_4 \to u_5 \to \ldots \to u_{2\ell+1} \to u_1$. Thus, if G contains a triangle, then G' contains a $2\ell+1$ cycle.

Now suppose that G' contains a $2\ell+1$ cycle C. Suppose that there is some V_i such that C does not contain any node of V_i. Consider $G' \setminus V_i$. This graph is now bipartite, and hence cannot contain any odd cycles: this is a contradiction since C is odd and is supposedly contained in $G' \setminus V_i$. Thus C contains at least one node from each V_i. More precisely, C contains exactly one node from each V_i since there are $2\ell+1$ sets V_i. Let u_1 be the node in C that is in V_1. Then u_4 is the node of C that is in V_4 since the edges between V_i and V_{i+1} for $i \geq 4$ are matchings. Suppose x_2 and y_3 are the nodes of C in V_2 and V_3. By the construction of G' we have that $(u, x), (x, y), (y, u) \in E$, and hence u, x, y is a triangle in G.

Hence G' contains a $2\ell+1$ cycle if and only if G contains a triangle. G' has $O(n)$ nodes, and hence an $O(n^t)$ algorithm for $2\ell+1$ cycle would find a triangle in G in $O(n^t)$ time. □

Now we show how to find a 4-cycle in $O(n^2)$ time:

Theorem 1.4. *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.*

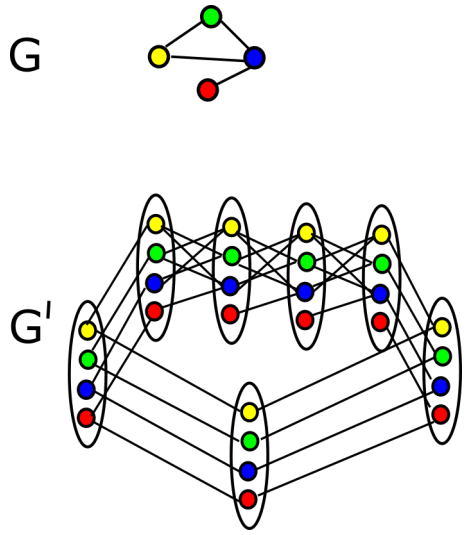

Figure 1: An example of a simple graph G and the associated G' where $2\ell + 1 = 7$.

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

```
T ← ∅;
foreach v ∈ V do
    foreach s, t ∈ N(v) do
        if (s, t) ∉ T then
            T((s, t)) ← {v};
        else
            u ← T((s, t));
            return u, s, v, t;
```

Proof. Consider Algorithm 1:

In it, T is a hash table with keys that are pairs of vertices of G and values that are 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 node with the largest label. Then, either the algorithm has already found a 4-cycle, or when the first loop has $v = a$, and the inner loop has $s = b$, $t = d$, 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. The correctness of the algorithm follows this reasoning.

Let's consider the runtime: in each iteration, either the algorithm halts, or a new pair (s,t) is added to T. Thus, the runtime of the algorithm is upper-bounded by the size of T. The size of T however is at worst n^2 since this is the maximum number of distinct pairs (s,t). Hence the algorithm runs in $O(n^2)$ time. □

2 Shortest Cycle

Setup

Now 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 (1978) 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 $O(n^2 + T(3n))$ 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) $T(3n) = O(T(n))$, 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.

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 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 $\leq q$.*

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.*

Algorithm 2: BFS-Cycle(s)

```
L_0 ← {s};
visited[s] ← true;
foreach u ≠ s ∈ V do
    visited[u] ← false;
foreach i do
    L_{i+1} ← {};
    foreach u ∈ L_i do
        foreach (u, x) ∈ E do
            remove (u, x) from E;
            if visited[x] = false then
                visited[x] ← true;
                L_{i+1} ← L_{i+1} ∪ {x};
            else
                find least common ancestor c of u and x;
                return (u, x)∪ path u ··· c ··· x in BFS-tree;
```

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 $i \leq \lceil q/2 \rceil - 1$.*

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 $2\lceil q/2 \rceil$, 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 $|C| \leq 1 + i + (i + 1) = 2(i + 1)$. So if Lemma 2.2 holds, then $i \leq \lceil q/2 \rceil - 1$ and $|C| \leq 2 \cdot \lceil q/2 \rceil$.

Next, we prove Lemma 2.2: *Proof of Lemma 2.2.* 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 $i \leq \lceil q/2 \rceil - 1$. □