A randomized algorithm in Python to detect small k-cycles in undirected Graphs

Alexander Olson

August 24, 2018

A **graph** (denoted G) in Computer Science is generally expressed in the form $G = (V, E)$, where V is a set of **vertices** (points in the graph) and E is a set of **edges**, pairs of connected vertices in the graph. By convention, we will let n denote the number of nodes in the graph, and m denote the number of edges.

A relatively small graph, for example, may have the vertices v_1, v_2 and v_3, comprising the set V: furthermore, we might give it the edges (v_1, v_3) and (v_1, v_2) comprising the set E, but not (v_2, v_3): in this situation, $n = 3$ and $m = 2$.

An **undirected** graph is one where all edges must go both ways: in other words, if (v_i, v_j) is an edge in graph G, then (v_j, v_i) must also be an edge in graph G.

A **k-cycle** is a sequence of vertices $x_1, x_2, ..., x_k$ in V such that all of the edges (x_1, x_2), (x_2, x_3), ... up to (x_{k-1}, x_k) are in E, as well as the closing edge, (x_k, x_1).

A **path** of length $k - 1$ is a sequence of vertices $x_1, x_2, ..., x_k$ such that all of the edges (x_1, x_2), (x_2, x_3), ... up to (x_{k-1}, x_k) are in E. A k-cycle can be considered to be such a path of length $k - 1$, with the closing edge.

Now, consider our n vertices $v_1, v_2, ..., v_n$. There exist as many as $n!$ permutations of these vertices: in other words, we can order them in $n!$ different ways. A permutation may be represented by the cycle π, and the function $\pi(v_i)$ represents the position of vertex i in that cycle. If $\pi(v_i) < \pi(v_j)$ for all $i < j$, then we may consider this to be a **π-increasing k-cycle.**

A graph $G = (V, E)$ is commonly represented as an **adjacency matrix**, A: the dimensions of the matrix will be $n \times n$, where V consists of n nodes. At any given entry a_{ij} in matrix A, we will take that:
$a_{ij} = 0$ if the edge (v_i, v_j) does not exist.
$a_{ij} = 1$ otherwise.

1 The Objective

Given an adjacency matrix A representing the graph $G = (V, E)$, and an input k, we want to detect if there is a k-cycle in the input graph such that $k \leq lg(n)$. If such a cycle exists, we would like to output "SUCCESS" with a probability of at least 90%: if it doesn't, we always want to output "FAILURE".

Our algorithm should be close to polynomial time.

2 The algorithm

2.1 Step 1: Choose a random ordering, and look for π-increasing k-cycles

For relatively small k, we will pick a random ordering π on the vertices, and then rearrange the matrix A so that the entry a_{ij} in A corresponds to the edge between $\pi(v_i)$ and $\pi(v_j)$: this new matrix will be saved, and denoted M_1. For every entry in m_{ij} in M_1 such that $i \not< j$, we will set it to zero: we only want the matrix M_1 to have 1s for π-increasing edges.

Next, we want to construct a list of matrices M_1, M_2, ..., $M_{2^{log(k)}}$: if there is a π-increasing path of length L between two vertices i and j, and l is a power of 2, then we want to the corresponding entry m_{ij} in M_L to be 1: otherwise, we want $m_{ij} = 0$.

Observe that a π-increasing k-cycle consists of a π-increasing path of length (k-1) between any two vertices v_i and v_j, and then the closing edge (j, i). If we could construct the matrix M_{k-1} as above, then we could easily spot this path, check for the closing edge and detect the k-cycle.

Now, suppose we have a π-increasing path (i, j) of length c in a given matrix M_c, and a π-increasing path (j, k) of length d in a given matrix M_d: if this is the case, then there exists a π-increasing path (i, k) of length $c + d$. Furthermore, since $m_{ij} = 1$ in M_c and $m_{jk} = 1$ in M_d, then $m_{ik} = 1$ in the product, $M_{c+d} = M_c M_d$. That means we can use matrix multiplication (and in fact, a relatively small number of matrix multiplications) to find a π-increasing path of length k.

We will construct the given matrices $M_1, M_2, ...M_{2^{log(k)}}$ and M_{k-1} as follows:

- Decompose $(k-1)$ into a sum of powers of 2: for example, $29 = 16 + 8 + 4 + 1$.
- For all M_L such that L is a multiple of 2 and $L < k$, calculate M_{2L} recursively: $M_{2L} = M_L M_L$.
- Refer to the powers of 2 that sum to $(k-1)$: initialize M_{k-1} to the identity matrix, and multiply it by M_p for each p which is a power of 2 occuring in the decomposition of $(k-1)$.

We then scan the resulting matrix M_{k-1}: if we find some entry in M_{k-1} where $m_{ij} = 1$ and some entry in A where $a_{ji} = 1$, then there is certainly a π-increasing k-cycle, and we return a **success**- otherwise, we return a **failure** on that run.

2.2 Step 2: Run it until it works

The procedure above is perfectly accurate if the k-cycle is increasing, for the given permutation π: unfortunately, of the $k!$ different ways that the vertices may be arranged in π, only

2k of them (clockwise and counterclockwise, and starting with any one of the k different vertices) will result in the cycle being k-increasing.

There are a few ways to tackle this problem. Since we deliberately chose a small (almost constant) k, we'll rely on brute force- specifically, we run the algorithm $(2)(k-1)!$ times, and return "SUCCESS" if we spot the k-cycle on any individual one of these runs. To make sure our method is correct, we need to prove this gives us an accuracy guarantee of at least 90%. In the worst case, there will be only one cycle that we might detect- let's consider the chance of finding it in only $k!$ trials:

$P(\text{find k-cycle}) = 1 - P(\text{miss k-cycle}) = 1 - (\frac{k!-2k}{k!})^{(k-1)!}$

$= 1 - (\frac{(k-1)!-2}{(k-1)!})^{(k-1)!} = 1 - (1 - \frac{2}{(k-1)!})^{(k-1)!}$

We can use the common inequality $1 + x < e^x$, substituting x for $-\frac{2}{(k-1)!}$:

$P(\text{find k-cycle}) > 1 - (e^{-\frac{2}{(k-1)!}})^{(k-1)!} = 1 - e^{-2} > 0.86$
And so:
$P(\text{miss k-cycle}) < 0.14$

Since we run the algorithm for another $(k-1)!$ trials, it follows that:

$P(\text{miss k-cycle, twice}) < (0.14)^2 < 0.02$

So we have at least a 98% chance of finding the k-cycle, which meets our threshold.

2.3 Complexity Analysis

First, we want to analyze the procedure in Step 1.

1) Each random permutation is generated by allocating an array of integers from 1 to n, and then permuting them uniformly at random with a technique commonly known as the "Knuth shuffle"- here, both allocating the array and shuffling it may be done in linear time. (A quick google search will explain how the Knuth shuffle works!)

2) We then allocate the $n \times n$ matrix M_1- originally, an empty matrix of 0s. Iterating through all entries a_{ij} in the adjacency matrix A, we copy their values into the corresponding spot $m_{\pi(v_i)\pi(v_j)}$ in the new matrix M_1: then, we check all entries m_{ij} in the matrix M, and set $m_{ij} = 0$ if $i \nless j$. These operations can be done in quadratic time.

For the next few steps, the complexity of matrix multiplication depends on the specific implementation used (of which there are many), but is typically polynomial: we will refer to it as $O(n^\omega)$ where ω is about 3, or slightly better. Our code uses the method *np.dot*() from Python's *numpy* package.

3) We can easily break down $(k-1)$ into a sum of powers of 2 with a sequence of mod and division operations: numbers are stored in binary, so this is a series of only $O(lg(k))$ steps that take $O(1)$ time: we will assume an efficient implementation of subtraction that takes $O(lg(k))$ time as well. We allocate an array of size $O(lg(k))$ to remember which powers of 2 are involved in the sum.

4) We need to store and calculate $O(lg(k))$ different matrices tracking paths of size 1, 2, 4, 8, and so on- this operation takes $O(lg(k)n^2)$ space, and $O(lg(k)n^\omega)$ time for all the steps of multiplication.

5) We allocate the identity matrix (of size $n \times n$), and then multiply it by each power-of-2 matrix involved in the decomposition of $(k-1)$. This involves $O(n^2)$ space, and $O(lg(k)n^\omega)$ time for at most $lg(k)$ steps of multiplication.

Now, we can sum the time complexity for each of the required steps:
$O(n) + O(n^2) + O(lg(k)) + O(lg(k)n^\omega) + O(lg(k)n^\omega) = O(lg(k)n^\omega)$.

Step 2 repeats the procedure in Step 1 as many as $2(k-1)!$ times, so the time complexity of this procedure is $O((k-1)!lg(k)n^\omega)$.
We can break this down further with the use of **Stirling's approximation**:
$n! \in O(n^n)$

Since we mandate that $k \leq lg(n)$, it follows that
$(k-1)! \in O((lg(n)-1)^{lg(n)-1}) = O(lg(n/2)^{lg(n/2)}) = O(2^{lglg(n/2)})^{lg(n/2)} = O(2^{lg(n/2)})^{lglg(n/2)}$
$= O((n/2)^{lglg(n/2)}) = O(n^{lglg(n/2)}/lg(n/2)) = O(n^{lg(lg(n)-1)}/(lg(n)-1) = O(n^{lg(lg(n)-1)}/(lg(n))$

Thus, the final time complexity of our algorithm is
$O((k-1)!lg(k)n^\omega) = O(n^{\omega+lg(lg(n)-1)}\frac{lglg(n)}{(lg(n))})$

Which we will simplify to simply being in $O(n^{\omega+lg(lg(n)-1)})$, for all reasonably large input sizes n. The algorithm is unfortunately superpolynomial, but in practice it works relatively well if the input size is small or if the cycle size k is a particularly small constant.