1 Subgraph Isomorphism

A task that needs to be accomplished often in practice is to detect patterns in data. Graphs are one of the most common types of data representation and detecting patterns in a graph corresponds to finding a copy of some previously described smaller graph. For instance, one may want to detect a very dense subgraph on at about 20 nodes, or one may want to find a tour that goes through each node once. Each of these is an instance of the subgraph isomorphism problem that we will define below.

Definition 1.1. *Let $H = (V_H, E_H)$ and $G = (V, E)$ be graphs. A subgraph isomorphism from H to G is a function $f : V_H \to V$ such that if $(u, v) \in E_H$, then $(f(u), f(v)) \in E$. f is an induced subgraph isomorphism if in addition if $(u, v) \notin E_H$, then $(f(u), f(v)) \notin E$.*

The (Induced) Subgraph Isomorphism computational problem is, given H and G, determine whether there is a (induced) subgraph isomorphism from H to G. If such an f exists, then we call $f(H)$ a copy of H in G.

If H is part of the input, Subgraph Isomorphism is an NP-complete problem. It generalizes problems such as Clique, Independent Set, and Hamiltonian Path.

In this lecture we will consider the special case of Subgraph Isomorphism where H is a given fixed graph of constant size k.

First let us consider the trivial algorithm to determine whether G contains a copy of H: Let $V_H = \{h_1, \ldots, h_k\}$. Then try all n^k ordered tuples of distinct k vertices of G, $v_1, \ldots, v_k$, and check whether for all i and j, $(v_i, v_j) \in E$ if (and only if for induced copies) $(h_i, h_j) \in E_H$. If so, return that G contains an (induced) copy $\{v_1, \ldots, v_k\}$ of H. If none of the tuples work, then G does not contain a subgraph isomorphic to H. The running time of this algorithm is $O(n^k \cdot k^2)$. Since k is a constant, it is $O(n^k)$.

It is a natural question whether one can improve on this exhaustive search $O(n^k)$ running time. To address this question, let us consider small values of k.

For the case $k = 2$, there are only two graphs H, the independent set of two nodes and the graph consisting of a single edge. In the noninduced case, the first subgraph isomorphism problem can be solved in constant time: just check whether the graph has at least two nodes. In the induced case, the edge and the independent set problems are equivalent, since one can just consider the complement of G. Checking whether G contains an edge can be done in time linear in the number of vertices (in the adjacency list representation) by just checking whether each vertex has a neighbor.

The case $k = 3$ has four graphs H. They are the independent set on 3 nodes I_3, the triangle graph, the graph S consisting of an edge and an isolated node, and the complement graph $\bar{S}$ of S consisting of a node and two incident edges.

In the noninduced case, the subgraph isomorphism problem is easy for I_3, S and $\bar{S}$. An I_3 can be found in constant time by checking if the graph has at least 3 nodes. An S can be found in linear time by checking if the graph contains at least one edge and at least three vertices, and an $\bar{S}$ can be found by checking if there is a vertex of degree at least 2. The triangle detection problem is not known to have a linear time solution. As we will show later, the best known algorithms for triangle detection are much less efficient.

In the induced case, the subgraph isomorphism problem for I_3 is equivalent to that for triangle, and the problem for S is equivalent to that for $\bar{S}$. We will show in the homework, that even in the induced case, one can find an S or a $\bar{S}$ in linear time. Now we will give two algorithms for triangle finding.

Finding triangles in graphs.

Theorem 1.1. *Let $G = (V, E)$ be a graph on m edges and n nodes. There is an $O(mn)$ time algorithm that finds a triangle in G, or determines that G contains no triangles.*

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

```
foreach v ∈ V do
    foreach s, t ∈ N(v) do
        if (s, t) ∈ E then
            return (v, s, t) ;
return "No triangle";
```

Proof. Consider Algorithm 1. If G contains a triangle (a, b, c), then in some iteration of the algorithm, $v = a$ and $b, c \in N(v)$, and hence the triangle will be found. Since the algorithm only returns triangles, or determines that none exist, correctness follows.

The running time of the algorithm is as follows. For every vertex v, $deg(v)^2$ vertices s, t are looked at, and hence the runtime is

$$\sum_{v \in V} deg(v)^2 \leq n \sum_{v \in V} deg(v) \leq 2mn.$$

□

For dense graphs, a completely different approach is better suited.

Fact 1.1. *There is an $O(n^{2.38})$ time algorithm that can multiply any two $n \times n$ matrices with integer entries in $\{0, \ldots, n\}$.*

The running time in Fact 1.1 was first achieved in 1986 by Coppersmith and Winograd. Recent work (by Stothers, myself and Le Gall) has brought down the runtime of $n \times n$ matrix multiplication to $O(n^{2.373})$.

We will use Fact 1.1 to prove the following theorem.

Theorem 1.2. *There is an $O(n^{2.38})$ time algorithm that given an n node graph G, either finds a triangle in G, or determines that G contains no triangles.*

Proof. Given G, create its $n \times n$ adjacency matrix A that has $A[i, j] = 1$ if $(i, j) \in E$ and $A[i, j] = 0$ otherwise.

Use Fact 1.1 to multiply A by itself, forming A^2. Notice that $A^2[i, j] = \sum_k A[i, k] \cdot A[k, j] > 0$ if and only if there exists some k such that $(i, k), (k, j) \in E$. If in addition $(i, j) \in E$, then i and j appear in some triangle with some k. Thus, after computing A^2, we can check in $O(n^2)$ time for each (i, j) whether $(i, j) \in E$ and $A^2[i, j] > 0$. If we don't find such an edge (i, j), then G cannot contain any triangles. Otherwise, if we find such an (i, j), we can go through all possible n candidates k for the third node in the triangle, and hence we can find some triangle in $O(n^2)$ time, after the initial multiplication of A by itself.

We obtain an $O(n^{2.38})$ time algorithm for triangle finding. □

An even more interesting fact is that if we multiply A by itself three times, forming A^3, then the trace of A^3 is exactly six times the number of triangles in G. Thus, we can also determine if G contains a triangle by computing $Tr(A^3)$. The best known algorithm for computing this trace however also uses matrix multiplication. (In fact, in a certain model of computation, any algorithm for computing $Tr(A^3)$ can be used to design an algorithm for matrix multiplication.) Hence we do not currently gain an asymptotic improvement if we compute $Tr(A^3)$ instead of A^2. Nevertheless, it could be that some day someone will design an extremely clever faster algorithm for the trace of A^3.

Proposition 1. *$Tr(A^3) = 6\times$ number of triangles in G.*

Proof.

$$Tr[A^3] = \sum_i \sum_j \sum_k A[i,j]A[j,k]A[k,i].$$

Since A is symmetric (as G is undirected), for $i < j < k$, each nonzero term $A[i,j]A[j,k]A[k,i]$ appears 6 times, once for each permutation of i, j and k. Since $A[i,i] = 0$ for all i, these are the only nonzero terms. Hence,

$$Tr[A^3] = 6 \sum_{i<j<k} A[i,j]A[j,k]A[k,i].$$

Now, $A[i,j]A[j,k]A[k,i]$ is nonzero only if $(i,j), (j,k), (i,k) \in E$, and so exactly when i, j, k is a triangle. Also, every triangle i, j, k appears in the sum exactly once. Thus we get that $Tr(A^3) = 6\times$ number of triangles in G. □

The above algorithms for triangle detection are interesting when the graph is relatively dense. However, when the number of edges m is less than $n^{1.38}$, the "trivial" $O(mn)$ time algorithm is actually faster than the algorithms based on matrix multiplication. A natural question arises: what is the best algorithm for triangle detection for sparse graphs? Clearly, any deterministic algorithm at least needs to read the input, so that $\Omega(m)$ time is necessary. Is there a linear time algorithm for triangle detection?

We do not currently have the techniques to answer such questions. However, it turns out that one can design an algorithm that is faster than $O(mn)$ for all values of m. The runtime in the theorem below is the best known for triangle detection in sparse graphs.

The algorithm design follows a widely used "high degree – low degree"' technique. In this technique one picks a parameter t and handles parts of the input with different algorithms depending on t. In the case of the proof t is a degree threshold and the nodes of degree $< t$ are handled via the $O(mn)$ algorithm, whereas the rest of the nodes are handled via the $O(n^{2.38})$ time algorithm.

Theorem 1.3. *There is an $O(m^{1.41})$ time algorithm for triangle detection in m edge graphs.*

Proof. Let t be a threshold parameter to be set later. We will call all vertices of degree $< t$ "low degree" and all others "high degree".

For every low degree node v and every edge (u, v) incident to it, go through all other neighbors u' of v and check if $\{u, v, u'\}$ is a triangle.

If this step finds a triangle, then the algorithm is done. Otherwise, there are no triangles going through low degree nodes, and the algorithm can now focus on the subgraph induced by the high degree nodes. The runtime of the low degree step is $O(mt)$ since there are at most m edges (u, v) and at most t choices for the second neighbor u' for each v.

Now the algorithm removes all low degree nodes in $O(m)$ time and runs the matrix multiplication algorithm for detecting a triangle on the graph induced by the high degree nodes. Since there are at most $2m/t$ high degree nodes, the runtime of this step is $O((m/t)^{2.38})$.

The overall runtime is asymptotically

$$(m/t)^{2.38} + mt.$$

It is minimized when we set the two summands equal to pick the parameter t. $mt = (m/t)^{2.38}$ implies $t^{3.38} = m^{1.38}$ so that $t = m^{0.41}$ gives a runtime of $O(m^{1.41})$. □

Reducing Subgraph Isomorphism to triangle detection. The triangle subgraph isomorphism problem seems to be the hardest out of all of the subgraph isomorphism problems whenever $k = 3$. There is another sense in which triangle finding is "hard". Nešetril and Poljak considered the subgraph isomorphism problem for arbitrary constant size subgraphs H and showed that if one has a really good algorithm for finding triangles, one can also obtain good (though slightly slower) algorithms for finding a copy of H in a given graph.

We will give their proof in the next theorem.

Theorem 1.4 (Nešetril and Poljak)**.** *Let k be a constant. Suppose that there is an $O(N^t)$ time algorithm (for some constant $2 \leq t < 3$) for finding a triangle (if one exists) in an N node graph. Let H be any graph on $3k$ nodes. There is an $O(n^{tk})$ time algorithm for finding a (induced or noninduced) copy of H (if one exists) in a given n-node graph G.*

We note that the requirement that $t \geq 2$ in the theorem is actually necessary. This is because any deterministic algorithm for triangle detection must be able to distinguish between graphs with at least one triangle and triangle-free graphs. There exist graphs with $\Theta(n^2)$ edges that do not contain any triangles- for instance the complete bipartite graph on n nodes in each partition. Any deterministic algorithm needs to at least read the input in order to be able to determine whether the graph is triangle-free. Hence, it must take $\Omega(n^2)$ time.

We mentioned earlier that the brute force algorithm for subgraph isomorphism for arbitrary H runs in $O(n^k)$ time. Nešetril and Poljak's result says that any algorithm for triangle finding that is faster than brute-force can be converted into a faster than brute-force algorithm for the Subgraph Isomorphism problem for arbitrary H.

Proof. The proof proceeds as follows. First, take H and arbitrarily split it into three subgraphs H_1, H_2 and H_3 on k nodes each. Let the vertices of H_i be $\{h_1^i, \ldots h_k^i\}$, for $i = 1, 2, 3$.

Now, we will create a huge graph G' on $O(n^k)$ nodes and $O(n^{2k})$ edges.

Fix $i \in \{1, 2, 3\}$. For each of the n^k ordered k-tuples $\{v_1, \ldots, v_k\}$ of the vertices of G, check whether $\{v_1, \ldots, v_k\}$ is a copy of H_i under the isomorphism $h_j^i \to v_j$ for $j \in \{1, \ldots, k\}$. That is, check whether for all $s, t \in \{1, \ldots, k\}$, $(v_s, v_t) \in E$ if (and only if for induced) $(h_s^i, h_t^i) \in E_H$. If $\{v_1, \ldots, v_k\}$ is a copy of H_i, then add a vertex $U^i(v_1, \ldots, v_k)$ to G'.

Now, G' contains $O(n^k)$ nodes, each corresponding to a copy of H_1, H_2 or H_3.

We add edges to G' as follows.

Fix $i \neq j$ for $i, j \in \{1, 2, 3\}$. For all pairs of nodes $U^i(v_1, \ldots, v_k)$ and $U^j(u_1, \ldots, u_k)$ of G', check whether $\{v_1, \ldots, v_k, u_1, \ldots, u_k\}$ is isomorphic to $H_i \cup H_j$ under the isomorphism $h_s^i \to v_s$ and $h_s^j \to u_s$ for $s \in \{1, \ldots, k\}$. That is, check for every $s, t \in \{1, \ldots, k\}$ whether $(v_s, u_t) \in E$ if (and only if for induced) $(h_s^i, h_t^j) \in E_H$, and whether $\{v_1, \ldots, v_k\}$ and $\{u_1, \ldots, u_k\}$ are disjoint. If this happens, then add an edge between $U^i(v_1, \ldots, v_k)$ and $U^j(u_1, \ldots, u_k)$.

Finally, notice that any copy of H in G has three nodes $U^1(\cdot), U^2(\cdot), U^3(\cdot)$ corresponding to the copies of H_1, H_2, H_3, and moreover, these nodes form a triangle since all three edges must be present. Furthermore, any triangle in G' corresponds to three disjoint sets of nodes, corresponding to copies of H_1, H_2 and H_3, and the edges between any pair of the copies are consistent with H. Thus all triangles in G' correspond to copies of H. We have that G' contains a triangle if and only if G contains a copy of H.

Creating G' takes $O(n^{2k}k^2)$ time since we have $O(n^{2k})$ pairs of nodes in G' and the number of edges that we need to check between two copies is k^2. Since k is a constant, this is $O(n^{2k})$.

Finally, if there is an $O(N^t)$ time algorithm for finding a triangle in an N-node graph, then we can use this algorithm on G' to find a copy of H, if one exists, in $O((n^k)^t) = O(n^{tk})$ time. □

Because triangle finding is in $O(n^{2.38})$ time, we immediately obtain the following.

Corollary 1.1. *Let k be a constant. Let H be any $3k$-node graph. Then there is an $O(n^{2.38k})$ time algorithm that can find a copy of H in any given n-node graph, or determine that no such copy exists.*

If H is a graph on $3k+q$ nodes where $q \in \{1, 2\}$, that is if $|V_H|$ is not divisible by 3, then the above result can be modified slightly to show that there is an $O(n^{2.38k+q})$ time algorithm for finding a copy of H in G. Since $2.38k + q < 3k + q$ for all $k > 0$, we see that for $K \geq 3$ one can beat the trivial runtime of $O(n^K)$ for all K-node subgraphs H.