1 Dynamic Graph Problems

So far we have only looked at "static" computational problems: given an input we need to compute some function of the input. Today we will look at "dynamic graph problems" which are essentially problems about data structures that store a graph, can make updates to the graph and can answer queries about the current state of the graph. Often in data structure problems people care about space usage. Today we will largely ignore the space usage. We will focus on the running times of operations.

In all dynamic graph problems, the running times of interest are: (1) the preprocessing time, (2) the update time, (3) the query time.

Let's look at two very similar dynamic graph problems: Dynamic Connectivity and Dynamic Reachability. Both problems are defined as follows:

Given a graph $G = (V, E)$, maintain a graph data structure to efficiently support the following operations:

- `query(u,v)` - Returns whether or not there is a path from u to v in the current graph
- `insert(e)` - Add edge e to E
- `delete(e)` - Delete edge e from E

In the **Dynamic Connectivity** problem, the graph G to be maintained is *undirected*, and in the **Dynamic Reachability** problem, G is directed.

It is known that for Dynamic Connectivity on n node graphs, all operations (updates or queries) can be supported in polylog(n) worst-case time (see [KKM13]) with near-linear preprocessing time. This is one of the huge successes of the area of dynamic algorithms. The best amortized update/query time is $\log n$, within polyloglog factors, and this is almost optimal since there is an $\Omega(\log n)$ lower bound in the cell probe model for the max of the update and the query time [PD06].

For Dynamic Reachability, however, the fastest algorithms algorithms are much worse. Meanwhile the best known cell probe lower bound for Dynamic Reachability was only recently proven to be (slightly) superlogarithmic [LY23]: the max of the update and query time needs to be $\Omega(\log^{1.5}(n))$. The algorithms for Dynamic Reachability are of two types:

1. *"Combinatorial"*: i.e. not using fast matrix multiplication. For graphs with at most m edges at all times an n nodes, one can achieve amortized update time $O(m\sqrt{n})$ and query time $O(\sqrt{n})$ [RZ08], or $O(m + n \log n)$ update time and $O(n)$ query time [RZ16]. When m is n^2, these algorithms always take $\Omega(n^2)$ time.

   Is there a combinatorial algorithm with $O(n^{2-\varepsilon})$ update and query time?

2. *Algebraic*: These are algorithms that use fast matrix multiplication; these are typically for dense graphs, where the input is only measured in terms of n. Here, one can achieve amortized update time $O(n^{1.53})$ and query time $O(n^{0.53})$ [San04] (and $O(n^\omega)$ time preprocessing) or both update and query time amortized $O(n^{1.407})$ [vNS19]. If $\omega = 2$, the latter algorithm would have amortized time $O(n^{5/4})$. This is the running time that minimizes the max of the update and query time. The $n^{5/4}$ running time is close to $O(n)$; in fact the previous known bound (for $\omega = 2$) was $n^{4/3}$. So it might happen that one could reach $O(n)$ update/query time. Can algebraic techniques do even better?

Is there a (potentially non-combinatorial algorithm) with $O(n^{1-\varepsilon})$ update and query time?

There is a huge discrepancy between the Dynamic Connectivity and Dynamic Reachability! We will try to explain the difficulty of Dynamic Reachability today.

In fact, we will consider an even simpler version of Dynamic Reachability, called $st-$Reach. Here one is to preprocess a graph G with two fixed vertices s, t so that edge insertions and deletions can be supported, and every query is about whether s can reach t in the given graph.

For problems such as st-Reach there are conditional lower bounds from 3SUM ([PD06] and later by others; these go through a certain Triangle Listing problem we will see later on) and also from Triangle Detection. The 3SUM-based lower bounds are typically not tight. We thus focus on the ones from Triangle Detection.

2 Encoding Triangle Detection in st-Reach, aka Dynamic Reachability

Let's make the assumption for a particular $c > 2$, that Triangle Detection in n node graphs requires $n^{c-o(1)}$ time. A popular assumption is that $c = \omega$; let's call this the

Triangle Hypothesis: In the word-RAM Model, Triangle Detection requires $n^{\omega-o(1)}$ time.

The Triangle Hypothesis only makes sense if $\omega > 2$. If $\omega = 2$, as one needs to read the input, Triangle Detection does need $\Omega(n^2)$ time in dense graphs.

The "combinatorial" Triangle Detection Hypothesis says that $c = 3$ for "combinatorial" algorithms.

By our reduction in the previous lecture from All-Pairs Triangle Problems to Triangle Detection problems, we get that this hypothesis is equivalent to the

BMM Hypothesis: In the word-RAM Model, no "combinatorial" algorithm can multiply two $n \times n$ Boolean matrices in $O(n^{3-\varepsilon})$ time for $\varepsilon > 0$.

We will use these hypotheses as a basis for hardness for st-Reach.

Let's assume that we have a data structure for st-Reach on n-node graphs that has preprocessing time $p(n)$, update time $u(n)$ and query time $q(n)$.

Suppose that we are given an instance of Triangle Detection. Wlog this is a tripartite graph $G = (V, E)$ with nodes partitioned into three independent sets I, J, K with n nodes each, so that the edges are in $(I \times J) \cup (J \times K) \cup (I \times K)$. We want to know whether there exist $i \in I, j \in J, k \in K$ that form a triangle.

We will reduce Triangle Detection to st-Reach as follows.

Take G and create a new graph G' by "unfolding" I similar to what we did in the previous lecture. I.e. the node set of G' contains the node sets I, J, K and a copy I' of I. For each $i \in I$, denote by i' its copy in I'. For every edge (i, j) of G with $i \in I, j \in J$ add a directed edge (i, j) in G'. For every edge (j, k) for $j \in J, k \in K$ in G, add a directed edge (j, k) in G'. Finally, for every edge (i, k) for $i \in I, k \in K$ in G we add a directed edge (k, i') in G' from $k \in K$ to the copy i' of i in I'.

Finally we add two new nodes s, t to G' with no edges incident to them. G' has $3n + 2$ nodes.

We feed G' to the preprocessing algorithm of the st-Reach data structure, which preprocesses G' in $p(4n + 2)$ time.

Now we begin a phase for every vertex $i \in I$ (See Figure 1):

1. Add the directed edge (s, i) to G'
2. Add the directed edge (i', t) to G' (for the copy $i' \in I'$ of i)
3. Query whether s can reach t. If so, return "Triangle" and exit.

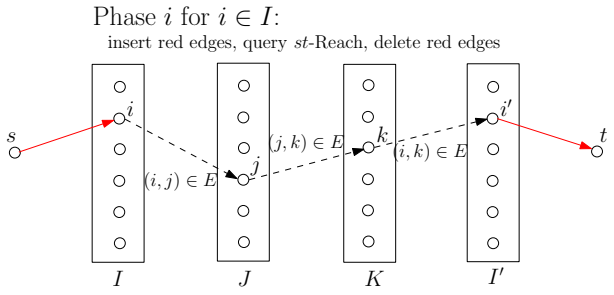

Figure 1: In this figure we present the first simple reduction from Triangle Detection to st-Reach.

4. Remove the two added edges in steps 1 and 2.

In total the number of updates is $4n$ and the number of queries is n. Thus we can solve Triangle detection in time

$$p(4n+2) + 4n \cdot u(4n+2) + n \cdot q(4n+2).$$

If $u(N), q(N) \leq O(N^{2-\varepsilon})$ and also $p(N) \leq O(N^{3-\varepsilon})$ for some $\varepsilon > 0$ then the above running time becomes $O(n^{3-\varepsilon})$.

Hence, under the BMM Hypothesis, there can be no combinatorial dynamic algorithm for st-Reach with truly subquadratic update and query times and truly subcubic preprocessing time.

As one can always recompute the reachability from scratch using BFS/DFS in $O(n^2)$ time during each update or query, with no preprocessing, and this is combinatorial, the above statement says that this trivial recomputation is essentially optimal for combinatorial algorithms.

For potentially non-combinatorial algorithms, under the Triangle Hypothesis, the reduction implies that one cannot have $u(N), q(N) \leq O(N^{\omega-1-\varepsilon})$ time and $p(N) \leq O(N^{\omega-\varepsilon})$ time for $\varepsilon > 0$. For the current value of ω, for instance we get that we can't have $u(N), q(N) \leq O(N^{1.37})$ and $p(N) \leq O(N^{2.37})$.

This reduction is nice and simple but it is not satisfactory, as the conditional lower bound doesn't say anything about algorithms that take more than cubic preprocessing time (for combinatorial algorithms) or more than n^ω preprocessing (otherwise).

Towards removing the restriction on the preprocessing time, we begin with another very simple reduction from Triangle Detection to st-Reach.

A different simple reduction from Triangle to st-Reach. In the previous reduction we essentially gave the full instance of Triangle to the preprocessing algorithm. Here we will give it only part of the input.

Recall that we have a tripartite graph $G = (V, E)$ with nodes partitioned into three independent sets I, J, K with n nodes each, so that the edges are in $(I \times J) \cup (J \times K) \cup (I \times K)$. We want to know whether there exist $i \in I, j \in J, k \in K$ that form a triangle.

We take the subgraph $G[J \cup K]$ of G consisting of J, K and the edges between them and add two vertices s and t, forming a graph G'. We feed G' to the $p(n)$ time preprocessing algorithm.

Now we begin a phase for every $i \in I$, see Figure 2.

1. For every $j \in J$, if $(i, j) \in E$, add a directed edge (s, j) to G'.
2. For every $k \in K$, if $(k, i) \in E$, add a directed edge (k, t) to G'.
3. Query whether s can reach t. If so, return "Triangle" and exit.
4. Remove all edges out of s and into t (i.e. those inserted in steps 1 and 2).

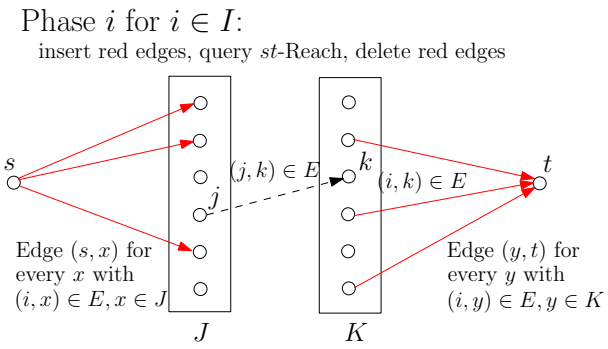

Figure 2: In this figure we present the second simple reduction from Triangle Detection to st-Reach.

In this reduction, in the phase for $i \in I$, we insert the edges corresponding to edges incident to i and the query again checks whether there is a triangle involving vertex i. The number of insertions and deletions is $O(n^2)$ overall and the number of queries is $O(n)$.

On the face of it, this reduction looks worse than our first reduction since it only says that there is no dynamic algorithm with $p(n) \leq O(n^{3-\varepsilon}), q(n) \leq O(n^{2-\varepsilon}), u(n) \leq O(n^{1-\varepsilon})$ which is worse than the earlier reduction since the update time dependence is worse. However, it turns out that this reduction can be changed to get a lower bound for arbitrary polynomial preprocessing time!

A reduction that works for arbitrary polynomial preprocessing time. Suppose that $p(N) = O(N^c)$ for some potentially large constant $c > 2$. The idea is to run the reduction above on multiple smaller instances.

Let's first partition the n-node parts J and K of the Triangle Instance into n/t parts on t nodes each: $J = J_1 \cup \ldots \cup J_{n/t}$ and $K = K_1 \cup \ldots \cup K_{n/t}$.

Go over all pairs $a, b \in [n/t]$ and run the reduction from before on the subgraph induced by I, J_a, K_b:

Take the subgraph $G[J_a \cup K_b]$ of G induced by $J_a \cup K_b$, direct all its edges from J_a to K_b and add to it two isolated vertices s, t. This instance $G_{a,b}$ is fed to the preprocessing algorithm of the st-Reach data structure. Since the graph has $O(t)$ nodes, the preprocessing time is $O(t^c)$.

Now we begin a phase for every $i \in I$ (n phases overall):

1. For every $j \in J_a$, if $(i,j) \in E$, add a directed edge (s,j) to $G_{a,b}$.
2. For every $k \in K_b$, if $(k,i) \in E$, add a directed edge (k,t) to $G_{a,b}$.
3. Query whether s can reach t. If so, return "Triangle" and exit.
4. Remove all edges out of s and into t (i.e. those inserted in steps 1 and 2).

If the algorithm doesn't return "Triangle" for any of the choices of a, b, return "No Triangle".

It is clear that in the phase for i for $G_{a,b}$, s can reach t in step 3 iff there is a triangle in G containing i and some node $j \in J_a, k \in K_b$. As we try all choices of a, b, the algorithm is correct.

The total preprocessing time over all a, b is $O((n/t)^2 \cdot t^c) \leq O(n^2 t^{c-2})$. If we set

$$t = n^{\delta/((c-2))},$$

for arbitrarily small constant $\delta > 0$, the preprocessing time would be $O(n^{2+\delta})$, as close to n^2 as we want.

For a fixed choice of a, b, the number of edge insertions and deletions is $O(nt)$ in total over all phases, and the number of queries is $O(n)$. Thus over all $(n/t)^2$ choices of a, b, the total number of updates is

$$O(n^3/t)$$

and the total number of queries is

$$O(n^3/t^2).$$

Suppose that for some $\varepsilon > 0$, $u(N) \leq O(N^{1-\varepsilon})$ and $q(N) \leq O(N^{2-\varepsilon})$ (for some combinatorial algorithm with $p(N) = O(N^c)$). Then, we can detect a triangle (combinatorially) in time asymptotically

$$\frac{n^3}{t} \cdot t^{1-\varepsilon} + \frac{n^3}{t^2} \cdot t^{2-\varepsilon} + n^2 t^{c-2} = n^{3-\varepsilon\delta/(c-2)} + n^{2+\delta}$$

which is truly subcubic for any $\varepsilon, \delta > 0$. We hence get that:

> Under the BMM Hypothesis, there is no **polynomial preprocessing** time combinatorial dynamic algorithm for *st*-Reach with truly subquadratic query time and truly sublinear update time.

We can view the reduction above as taking an algorithm that uses **arbitrary polynomial preprocessing** and converting it to one that uses $\mathbf{O(n^{2+\delta})}$ time preprocessing, while keeping the property that the update time is truly subquadratic.

3 Removing the word "combinatorial".

Under the BMM Hypothesis we only get the highly unsatisfactory combinatorial conditional lower bounds. We would really like to have statements about arbitrary algorithms.

Now, if we stare at our reduction above, we notice that it solves a potentially harder problem:

> **Online Triangle Detection** (OTD): Preprocess a tripartite graph G on vertex parts I, J, K (where I starts off empty) and support updates of the form: insert a new vertex v into I with all its incident edges to vertices in $J \cup K$. After each insertion, output whether v participates in a triangle.

Let's call the following assumption the **Online Triangle Detection Hypothesis**: There is no algorithm for OTD with $n = |J| = |K|$ that performs n^a vertex insertions into I (and the respective query) in total time $O(n^{2+a-\varepsilon})$ time for $\varepsilon > 0$.

Consider the reduction from OTD to *st*-Reach we just performed to remove the assumption on the preprocessing time. Each call to *st*-Reach can be viewed as accessing an OTD data structure for a graph on t nodes, and there are $(n/t)^2$ such data structures. This view of the reduction gives us a reduction from OTD to OTD with arbitrary polynomial preprocessing time.

We get that:

> The Online Triangle Detection Hypothesis implies that there is no algorithm that preprocesses an instance of OTD in any polynomial time in $n = |J| = |K|$ and then performs vertex insertions into I in $O(n^{2-\varepsilon})$ amortized time for $\varepsilon > 0$.

(The first place where this was proven was in Theorem 4.2 of [VW18], the full version of the FOCS'2010 paper. The proof appears again in [HKNS15].)

Our simple reduction to *st*-Reach then obtains that under the Online Triangle Detection Hypothesis *st*-Reach doesn't have a dynamic algorithm with polynomial preprocessing, $O(n^{2-\varepsilon})$ time query and $O(n^{1-\varepsilon})$ time updates.

OTD and OMv. Last lecture, we gave a fine-grained reduction from All-Pairs triangle detection to triangle detection (and in particular we used this generic reduction for "negative triangle"). All-Pairs triangle detection (for unweighted graphs) is equivalent to Boolean Matrix Multiplication (BMM) in the same way as All-Pairs Min triangles is equivalent to $(\min, +)$-product.

Thus, our reduction is also a reduction from BMM to triange detection. We will now revisit our reduction and show that a minor modification reduces "online BMM" to OTD.

The online BMM problem, also known as *OMv* (Online Matrix Vector), is as follows:

OMv: Preprocess a given $n \times n$ matrix A, so that if one is given n Boolean column vectors $b_1, \ldots, b_n$, one can output Ab_i before seeing b_{i+1}.

The **OMv Hypothesis** asserts that there is no algorithm that solves the OMv problem in total time $O(n^{3-\varepsilon})$ for any $\varepsilon > 0$.

In other words, the hypothesis says that there are no Strassen-like algorithms that solve BMM in truly subcubic time if B is given column by column, and not in batch.

The fastest known algorithm for OMv runs in $n^3/\exp(\sqrt{\log n})$ time [LW17]. (Also, it turns out that there can be no high cell probe lower bound since there is a cell probe algorithm that does $O(n^{1.5})$ "time" per vector [LW17, CKL18].)

We will show that the OMv Hypothesis implies the OTD Hypothesis, using the same ideas as in our previous lecture.

Let's assume that we have an algorithm for OTD that given a tripartite graph with $|J| = |K| = N$ can perform $n = N^a$ vertex insertions into I and their associated triangle queries in total time $O(N^{2-\varepsilon}n)$ for $\varepsilon > 0$.

Let A be a given $n \times n$ Boolean matrix, and let $v_1, \ldots, v_n$ be the Boolean vectors that are given online. We will show how to use the OTD algorithm to compute each Av_i after seeing v_i so that the total computation time will be truly subcubic.

We'll create a graph G_i representing Av_i: First we create a bipartite graph G_A representing A:

- for every row j of A there is a node j in partition J,
- for every column k of A there is a node k in partition K,
- for every $j \in J, k \in K$ such that $A[k, j] = 1$, there is an edge (k, j).

When vector v_i comes, we add a node v with edges to G_A forming G_i:

- for every k such that $v_i[k] = 1$, we add an edge (k, v),
- for every $j \in J$ we add an edge (j, v).

This graph G_i represents Av_i: There is a triangle in G_i using edge (j, v) for $j \in J$ if and only if $Av_i[j] = 1$.

Computing Av_i thus amounts to, given G_i, find all $j \in J$ that appear in a triangle together with v, i.e. for every edge (j, v) determine whether it appears in a triangle.

We now want to reduce this all-edge triangle computation to triangle detection. We did this in the previous lecture! The only difference is that in the previous lecture the three parts of the tripartite graph had size n. Now, one of the parts, $\{v\}$, has size 1, and the other two have size n.

Nevertheless, we proceed in the same way, except that we do not bucket $\{v\}$. We first claim that finding can be efficiently reduced to detection (similarly to the previous lecture):

Claim 1. *Suppose we have an OTD algorithm A that on query returns whether v is in a triangle in amortized $T(n) = O(n^{2-\varepsilon})$ time for $\varepsilon > 0$. Then we also have an algorithm that can return a triangle through v (if one exists) in amortized $O(n^{2-\varepsilon})$ time.*

Hint: prove this claim similarly to how you would reduce finding to detection for Triangle.

Now the reduction proceeds in the same way as last time.

The correctness should be clear. If we find a triangle b_J, c_K, v, then $Av_i[b_J] = 1$. If we remove (b_J, v), it's because we already computed $Av_i[b_J] = 1$ and the edge isn't needed anymore.

Algorithm 1: Reducing OMv to OTD

Begin Reduction:

We are given a graph $G_A = (J \cup K, E)$, bipartite graph representing A

Partition J, K into $\{J_1, \ldots, J_{n/L}\}, \{K_1, \ldots, K_{n/L}\}$.

for *all pairs* (a, b)*, where* a, b *range from* 1 *to* n/L **do**

  Consider G_{ab}, the subgraph of G_A induced by $J_a \cup K_b$.

  Initialize an OTD data structure for G_{ab}

On query v_i to OMv:

Initialize C to an n length vector of all zeros.

(At the end of the algorithm, $C[j] = Av_i[j]$.)

Let G_i be the graph consisting of G_A and a vertex v with edges (v, j) for every $j \in J$ and (v, k) for all $k \in K$ s.t. $v_i[k] = 1$

for *all pairs* (a, b)*, where* a, b *range from* 1 *to* n/L **do**

  Consider the OTD data structure for G_{ab}

  Insert a vertex v' into G_{ab} with the same edges as v in $G_i[J_a \cup K_b]$.

  while G_{ab} *contains a triangle containing* v' *(⋆ A query ⋆)* **do**

    Let $b_J \in J$, $c_K \in K$ be the nodes of the triangle with v' returned by the OTD alg.

    Set $C[b_J] = 1$.

    Delete (b_J, v) from G_i (this deletes it from all the induced subgraphs $G_i[J_a \cup K_b]$).

    Insert a vertex into G_{ab} with the same edges as v in $G_i[J_a \cup K_b]$; call this node v'.

return C

End OMv to OTD reduction

Runtime. Consider one of the Av_i computations. The runtime is dominated by the number of times a update/query call to an OTD data structure happens (and each such call takes $O(L^{2-\varepsilon})$ time by assumption). There are two types of such calls. The first type are those that return a triangle. The total number of such calls is no more than n because C only has n elements, and on each iteration we're setting one of them to 1 (and removing the edge so we can't set it to 1 again).

The second type of calls are those that do not find a triangle. The total number of such calls is exactly one for each pair (a, b), making sure that no updates to the data structure for G_{ab} are needed. Thus the number of such calls is $(n/L)^2$.

The amortized running time for computing Av_i is thus

$$O(L^{2-\varepsilon})\left(n + \left(\frac{n}{L}\right)^2\right).$$

To minimize the runtime, we set $L = n^{1/2}$, which gives an amortized runtime of $O(n^{2-\varepsilon/2})$.

To solve OMv we do n such operations, one for each v_i separately, getting a running time of $O(n^{3-\varepsilon/2})$.

Thus under the OMv Hypothesis, OTD requires amortized time $n^{2-o(1)}$ even with arbitrary polynomial preprocessing.

Thus any lower bound for a dynamic problem that is based on OTD can be based on OMv.

There are many reductions from online triangle detection to dynamic problems! Some of the first papers that contain such reductions are [RZ04] and [P̆10]. The latter paper gives very nice reductions from 3SUM to a triangle reporting problem in sparse graphs and then reduces that problem to dynamic graph problems. Abboud and V. Williams [AV14] give many reductions from dynamic versions of triangle deteciton, including OTD; they also initialize the study of fine-grained hardness of dynamic problems from other hypotheses such as SETH/OV. The OMv problem was defined in [HKNS15] who also adapt the BMM to Triangle reduction of [VW18] to obtain the hardness of OTD from OMv. They call OTD uMv since the OTD problem is equivalent to the problem where you are given an $n \times n$ Boolean matrix M to preprocess (corresponding to

the $J \times K$ subgraph) and then pairs of boolean vectors come in $(u_1, v_1), (u_2, v_2), \ldots$ (these correspond to the indicator vectors of the neighbors of v_i in OTD in J and K respectively) and you need to output $u_i M v_i^T$ after each (u_i, v_i) is seen.

4 A reduction from OV to a dynamic problem.

Besides from OMv and OTD, there are many reductions to dynamic problems from APSP (typically to weighted problems such as dynamic $s-t$ shortest paths), and also many from OV (and hence from SETH). Some dynamic problems have reductions from all three problems.

We will see a reduction from OV to a simple problem for which we do not have such strong hardness from OMv or APSP.

The problem is **Dynamic Two Strongly Connected Components** (SCC2): Given a directed graph $G = (V, E)$, maintaing G under edge insertions and deletions to be able to answer the query, does G have more than 2 strongly connected components (SCCs)?

(Recall that a SCC is an inclusion-maximal subgraph H that is strongly connected, i.e. for every pair of vertices s, t in H, s can reach t and t can reach s.)

The SCCs of an m-edge, n-node graph can be computed from scratch in $O(m+n)$ time. Thus one can always recompute the SCCs after each edge update in linear time. Thus, the question is, can one get *truly sublinear update time*?

We will show that under the OV Hypothesis, there is no fully dynamic algorithm for SCC2 that has amortized update time $O(m^{1-\varepsilon})$ for $\varepsilon > 0$.

The reduction proceeds as follows. Let $S \subseteq \{0,1\}^d$ be an instance of OV, $|S| = n$, where $d = n^{o(1)}$.

We create a graph G as follows.

- Add a node u for every vector $u \in S$.
- Add a node c for every coordinate $c \in [d]$.
- For every $u \in S, c \in [d]$ such that $u[c] = 1$, add a directed edge (c, u).
- Add a node s and a directed edge (u, s) from every $u \in S$ to s.
- Add a new node t.

We can preprocess this graph G.

Now we begin a phase for every vector $v \in S$:

1. For every coordinate c such that $v[c] = 1$, add a directed edge (s, c) to G.
2. For every coordinate c' such that $v[c'] = 0$, add directed edges (t, c') and (c', t) to G.
3. Query whether G has more than 2 SCCs. If so, return "Orthogonal Pair" and stop. Otherwise continue.
4. Delete all edges inserted in steps 1 and 2.

Let's first see the correctness. Consider the phase for some $v \in S$.

Suppose that there is no vector $u \in S$ that is orthogonal to v. Then for every $u \in S$ there is a $c \in [d]$ so that $u[c] = v[c] = 1$, and hence there is a path from s to c to u. Due to the edge (u, s) that exists for every u, we have that s is in the same SCC as every u in S. Some of the coordinate nodes c will be in that SCC, namely those for which $v[c] = 1$. The coordinates c' for which $v[c'] = 0$ are not in this SCC since s can never reach them: s goes to coordinates that v is 1 in and those only go to the nodes in S which go back to v. The coordinates c' for which $v[c'] = 0$ are all in a single SCC together with the node t since they all have edges to and from t. Thus, if v is not orthogonal to any $u \in S$, G has 2 SCCs.

Now suppose that there is some vector $u \in S$ that is orthogonal to v. Then s cannot reach u because the only way to get to u would be through some coordinate node c such that $v[c] = u[c] = 1$ and no such

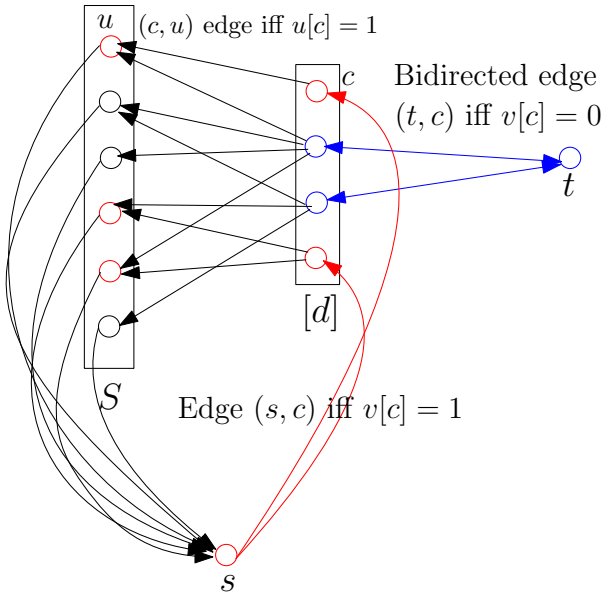

Figure 3: In this figure we present a reduction from OV to SCC2. The red and blue edges are the edges inserted during the phase for a vector v. The red nodes form one SCC, the blue nodes a second SCC and the black nodes are the additional SCCs that correspond to vectors orthogonal to v.

coordinate exists. Meanwhile, u also isn't in the SCC that contains t since it cannot reach t. u only has an edge to s and s cannot reach t by constrution. Thus there are at least three SCCs: one that contains s, one that contains t and one that contains u and they are distinct. (In fact, the number of SCCs is the number of vectors orthogonal to v, plus 2.)

Let's see the running time. Let's ignore the preprocessing time. (It's an exercise to show that one can assume that no polynomial preprocessing time will help.) We perform a total of $O(nd)$ updates and $O(n)$ queries. The number of vertices in the graph is $O(n + d)$ and the number of edges is $m = O(nd)$. If the update and query time is $O(m^{1-\varepsilon})$ for some $\varepsilon > 0$, then the total running time over all $v \in S$ is

$$O((nd)^{2-\varepsilon}),$$

and this refutes the OV hypothesis.

References

[AV14] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443. IEEE Computer Society, 2014.

[CKL18] Diptarka Chakraborty, Lior Kamma, and Kasper Green Larsen. Tight cell probe bounds for succinct boolean matrix-vector multiplication. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 1297–1306. ACM, 2018.

[HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplica-

tion conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30. ACM, 2015.

[KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142. SIAM, 2013.

[LW17] Kasper Green Larsen and R. Ryan Williams. Faster online matrix-vector multiplication. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2182–2189. SIAM, 2017.

[LY23] K. Green Larsen and H. Yu. Super-logarithmic lower bounds for dynamic graph problems. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1589–1604, 2023.

[PD06] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[PŤ10] Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 603–610. ACM, 2010.

[RZ04] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. In *Algorithms - ESA 2004, 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004, Proceedings*, volume 3221 of *Lecture Notes in Computer Science*, pages 580–591, 2004.

[RZ08] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008.

[RZ16] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM J. Comput.*, 45(3):712–733, 2016.

[San04] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 509–517. IEEE Computer Society, 2004.

[vNS19] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 456–480. IEEE Computer Society, 2019.

[VW18] Virginia Vassilevska Williams and R. Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *J. ACM*, 65(5):27:1–27:38, 2018.