

Expanders and ST-connectivity

1 Introduction

Perhaps the most interesting theory result in recent years is the resolution of the space complexity of the problem of testing *ST*-Connectivity in undirected graphs (USTCON). That is, given a graph $G = (V, E)$ and vertices $s, t \in V$, USTCON returns ‘yes’ if there exists a path from s to t in G and ‘no’ otherwise. The time complexity of USTCON is trivial: both breadth-first search (BFS) and depth-first search (DFS) solve USTCON in linear time. The space complexities of BFS and DFS are (in the general case) polynomial in the size of the input, and a number of algorithms are known which require less space. The true space complexity of this problem, however, had been undecided until this recent breakthrough.

1.1 Log-Space

The complexity class L (log-space) is the set of problems which can be solved using a section of memory that is at most logarithmic in size of the input. Intuitively, this means that all computations must be done on a constant number of pointers into the input and a logarithmic number of boolean variables; the input itself is held in a *read-only* section of memory (placing it in writable memory is already linear in the size of the input). In fact, USTCON was such an important problem, a complexity class was invented for those problems which reduce to it, named SL . Thus it is trivial that $L \subseteq SL$, but it was an important open question as to whether the two classes were equal.

Another known result was that $L \subseteq SL \subseteq RL$, where RL is the set of problems that can be solved (arbitrarily well) in logarithmic space with a randomized algorithm. In fact, one can build an algorithm which uses only a random walk: thus only requiring enough space for a pointer to the current position and a counter which tells you when to stop the walk (and guess that the answer is ‘no’). Thus as long as the the number of steps is polynomial in the input, the total space needed is logarithmic (and it can be shown that $|V|^3$ steps are sufficient).

Thus one way to approach the problem is to attempt to *derandomize* the random algorithm above (using at most a logarithmic amount of space), and this is, in fact, the direction that we shall take.

1.2 Expanders

Expanders are a natural way of attempting derandomization: they can be built deterministically but they act like random multigraphs. Note that, even though the original input is a graph, we will find it useful to view it as a multigraph (and then use transformations that will form new multigraphs). First, a definition:

Definition 1. Given a multigraph $G = (V, E)$ and vertices $x, y \in V$, we define $\mathbf{dist}(\mathbf{x}, \mathbf{y})$ to be the shortest path from x to y in G and $\mathbf{diam}(\mathbf{G}) = \max_{x, y \in V} \mathbf{dist}(x, y)$.

One property of expanders that will prove to be useful is that we can use the high connectivity property of an expander to bound its diameter. In particular, if we can show that $\mathbf{diam}(G) \leq c \log |V|$ for some constant c , then we can test *st*-connectivity by testing all paths with length less than $c \log |V|$ rooted at s . Furthermore, if G has maximum degree D , then there are at most $D^{c \log |V|} = O(|V|)$ paths to check and each one can be tested in $O(cD \log V)$ space (which is safely

within log-space as long as D is a constant). Thus the question arises: can we turn an arbitrary graph into an expander?

The most obvious way of getting an expander is to add edges. This has inherent problems, however, since adding edges will have the adverse effect of increasing the maximum degree (which we want to remain constant). Furthermore, we have the problem of actually *changing* the representation of our input graph: the input tape (as mentioned before) is read-only and so adding vertices and edges into the representation could require a linear amount of space in the input (more space than we have available). Thus we must come up with some system in which a much larger multigraph is uniquely represented by the input graph and so that we can compute which edges are in the multigraph “on the fly”.

To do so, we will look at a correspondence between directed and undirected multigraphs:

Definition 2. We denote the set $\{1, 2, \dots, N\}$ as $[N]$ and we say that a multigraph G is in the class $\mathcal{G}(n, d, \lambda)$ if

1. $|V(G)| = N$,
2. every vertex in $V(\vec{G})$ has degree d (all loop-edges supply 1 degree – not two), and
3. the spectral expansion of G is $\leq \lambda$.

For each vertex v in a graph, we will give unique labels (from a ground set size d) to the edges that are incident to v . We will do this for each v ; thus each non-loop edge $\{u, v\}_i$ will be given two (not necessarily distinct) labels – one from the perspective of v and the other from the perspective of u . Given an edge $v \in V$ and a label x , we will define the **rotation map** $Rot_G[v, x] = [u, y]$ if the edge $\{u, v\}_i$ has label x with respect to v and label y with respect to u (the subscripts i are used to emphasize that there could be multiple edges between two vertices).

To get an intuitive idea of the rotation map, consider the edges to be roads between vertices (which are cities). There are d names available for each road, but each city gets to decide what it wants to name the roads leaving its city (as long as no two roads leaving the city have the same name!). Hence two cities may name the same road two different things (this is common in real life!!). Then the rotation map is just a way to translate between road names: if city u gives the road $\{u, v\}$ the name x and city v gives the road $\{u, v\}$ the name y , then $Rot_G([u, x]) = [v, y]$.

The labellings of multigraphs will not be arbitrary; we will need to be systematic about how we choose them, since we need them to be uniquely defined (that way we can compute labellings “on the fly” and not have to spend any space recording them).

We will use two different transformations on multigraphs, both of which were discussed in class. We will assume that the multigraphs given to us have a proper labelling (and when we use the transformations, we will make sure a labelling is predefined).

Definition 3. Given multigraphs $G \in \mathcal{G}([N], D, \lambda_1)$ and $H \in \mathcal{G}([D], d, \lambda_2)$, we define the **zigzag product** $G \otimes H \in \mathcal{G}(N \times [D], d^2, \lambda_3)$ (where λ_3 is a function of λ_1 and λ_2) as discussed in class: for each vertex v in G , we replace the vertex with a copy of H . In particular, we will place vertex i (from H) on the edge $Rot_G[v, i]$ (this is why the edge labellings are necessary).

We now need to label the edges of the new graph $G \otimes H$; let $f = \{(v, a), (w, b)\}$ be an edge in $G \otimes H$. Then f was formed by a short edge $\{(v, a), (v, a')\}$, a long edge $\{(v, a'), (w, b')\}$ and another short edge $\{(w, b'), (w, b)\}$. Then if $Rot_H([a, i]) = [a', i']$ and $Rot_H([b, j]) = [b', j']$, then $Rot_{G \otimes H}([(v, a), (i, j)]) = [(w, b), (j', i')]$.

Again, some intuition. We think about the edges as roads: then the edges in $G \circledast H$ are shortcuts that span three roads – a short, then long, then short. Let’s say the new shortcut connects vertices u and v . If we traveled the long way from u to v , we would traverse three roads, and each would get a name from the city we are leaving: say x, y, z (in order). Then u will give the new shortcut the name (an ordered pair) (x, z) . Now if we went back the long way, we would get different names, but also traverse the roads in the reverse order, so we would get names z', y', x' . Hence v names the shortcut (z', x') .

Definition 4. Given a multigraph $G \in \mathcal{G}([N], D, \lambda)$, we define the **k^{th} power of G** , written $G^k \in \mathcal{G}([N], D^k, \lambda^k)$, to be the multigraph on vertices N where the pair $\{x, y\}$ has an edge in G^k for each walk length k from x to y in G (note that a walk can have repeated edges and vertices, unlike a path). Furthermore, for a walk $(v_0, v_1, \dots, v_{k-1}, v_k)$ in G we will have $\text{Rot}_{G^k}([v_0, (a_0, a_1, \dots, a_k)]) = [v_k, (b_0, b_1, \dots, b_k)]$, where $\text{Rot}_G([v_{i-1}, a_i]) = [v_i, b_i]$ for all $i \in [k]$.

Intuitively, an edge in G^k is a shortcut for a path P , length k , in G ; the name given to the shortcut is a k -tuple of the names of the roads in the path, in the order they are traversed.

1.3 Eigenvalues

As we have seen before, many of the properties of expanders can be measured by the eigenvalues of its adjacency matrix. Some of these properties are given in the next lemma:

Lemma 1. *Let $G \in \mathcal{G}([N], d, \lambda)$ be a multigraph with eigenvalues $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_k$. Then the following are true:*

1. $\lambda_1 = d$.
2. $|\lambda_i| \leq d$ for all $i \in [k]$.
3. $\lambda_2 = d$ if and only if G is disconnected.
4. $\lambda_k = -d$ if and only if G is bipartite.
5. $\text{diam}(G) \leq c \log_{d/\lambda} N$ for some constant c (recall that $\lambda = \max\{|\lambda_2|, |\lambda_k|\}$).

As mentioned earlier, if we can ensure that $\lambda(G)$ is bounded away from 1 (independent of the size of G), then we can run DFS search in a logarithmic amount of space. So we will take our original graph G and build a multigraph G' (using the replacement product and power operation) which is ensured to have a small value for λ and we will test the ST-connectivity in G' .

2 The Algorithm

If we plan to transform the graph G into some other graphs, we must be careful of three things:

1. We must build the multigraph G' without writing down any edges or vertices (and so that we can compute edges and vertices “on the fly”).
2. We must maintain a polynomial number of vertices with a set maximum degree (so that when DFS takes $O(\log |V(G')|)$ steps, we have that $\log |V(G')| = O(\log |V(G)|)$).
3. We must keep the maximum degree small (also needed for DFS)

Hence we will need to find a transformation that takes these things into account.

2.1 The Transformation

The particular transformation we will use is a combination of the two products mentioned in the previous section.

Definition 5. Given multigraphs $G \in \mathcal{G}([N], D^{16}, \lambda_1)$ and $H \in \mathcal{G}([D^{16}], D, 1/2)$, we define the multigraph $T(G, H) = (G \otimes H)^8$. Furthermore, we define a series of graphs G_i where $G_0 = G$ and $G_i = T(G_{i-1}, H)$ for all i .

Of course, the definition of T somewhat depends on whether any multigraphs in $\mathcal{G}([D^{16}], D, 1/2)$ exist, and luckily they do (see [?] for details on such a construction). It is easy to check that $G_i \in \mathcal{G}([N] \times [D^{16}]^i, D^{16}, \lambda_3)$ for some λ_3 , which we can bound using the next lemma:

Lemma 2. *If $G \in \mathcal{G}([N], D, \lambda)$, then $T(G, H) \in \mathcal{G}([N] \times [D^{16}], D^{16}, \lambda^2)$.*

Proof. See Lemma 11 in [?]. □

As mentioned previously, it is important to keep the number of vertices a polynomial in n , and so we can only hope to use G_k where $k = O(\log n)$. The next lemma shows that this is sufficient to create the value of λ we want.

Lemma 3. *There exists a $k = O(\log n)$ such that $G_k \in \mathcal{G}(N \times [D^{16}]^k, D^{16}, 1/2)$.*

Proof. It is known that for $G \in \mathcal{G}(N, D^{16}, \lambda)$, which are connected and non-bipartite, we have that $\lambda(G) \leq (1 - (1/N^2 D^{16}))$. We count the number of times we need to use T before the resulting graph G' has $\lambda(G') \leq 1/2$. Assume that such a G' exists for the first time after the k^{th} step. Then by Lemma ??, we have that

$$\lambda(G') \leq (1 - 1/N^2 D^{16})^{2^k} \leq 1/2$$

so, since D is a constant, we have that $k = O(\log N)$. □

Thus the transformation T does everything we need it to do.

2.2 The Algorithm in Action

We are finally ready to discuss the running of the algorithm.

Lemma 4. *Let G be a D -regular, undirected graph over n vertices, and let $\lambda < 1$ be a constant. There exists a space $O(\log D \cdot \log N)$ algorithm A' such that if s and t are vertices in the same connected component and this component is an $([N], D, \lambda)$ graph, then A' outputs ‘yes’.*

Proof. There are D^k paths of length $k = O(\log n)$ from s . A' decides connectivity by simply enumerating all of these paths and outputting ‘yes’ if one of these paths encounters t . A' thus requires $O(\log D \cdot \log N)$ space. □

Thus A' essentially decides the connectivity problem in the case of graphs which are expanders, or more precisely in cases where the starting vertex is in a connected component which is an expander. We will use this algorithm as a part of our main algorithm as below:

Algorithm (A). Given a multigraph $G \in \mathcal{G}([N], D, \lambda)$ and two vertices s and t define an algorithm A to decide connectivity in log-space, consisting of the following steps:

1. Transform G to G_{regular} , a D^{16} -regular graph as follows:

- (a) Replace every vertex in G with a cycle of length N .
- (b) For each pair of vertices (v, w) where v is a neighbour of w , connect them to (w, v)
2. Obtain a $([D^{16}], D, 1/2)$ graph H (recall that we know from [?] that this is possible).
3. Define $G_{expander}$ to be $T(G_{regular}, H)$
4. Invoke A' (from Lemma ??) on $G_{expander}$ and the vertices (s', t') where $s' = (s, 1^{k+1})$ and $t' = (t, 1^{k+1})$ (recall from Lemma ?? that $k = O(\log N)$).

Lemma 5. *The algorithm A above is correct.*

Proof. If s and t are in the same connected component S in G , then they are also in the same connected component $S' = S \times [N] \times [D^{16}]^k$ in $G_{expander}$. Let S be a connected component such that $s \in S$. It can be shown that $G_{expander}$ has a connected component $S \times N \times ([D_{16}])_l$, that contains s' . Hence, if s and t were in the same connected component in G , then s' and t' would be in the same connected component in $G_{expander}$ and the algorithm A' when applied on $G_{expander}$ and s' and t' would output "connected". Similarly, if $s \in S$ but $t \notin S$ then one of them would not be in the connected component $S \times N \times [D^{16}]^k$ and hence A' would output "not connected". Note that in our above discussion, the term "connected component" is used to mean a connected subset S , of vertices in G , such that there are no edges in G between vertices in S and vertices not in S . □

Lemma 6. *The algorithm A is Log-Space.*

Proof. Let us investigate the space complexity of each of the steps shown in Definition 7 :

1. Converting G to $G_{regular}$ is easily observed to be Log-Space.
2. Obtaining a graph H - one may use a (precomputed) constant size expander.
3. The transformation T : Recall that the transformation T is a recursive function on the graphs G and H which are represented in the form of their rotation maps. The evaluation of $Rot_{G_{i+1}}$ during each (recursive) phase of the transformation consists of a (constant) number of operations, which either evaluate Rot_{G_i} or require a constant amount of memory. The depth of the recursion tree is k (see Lemma ??) which is $O(\log n)$ and thus the recursion itself can be performed in space $O(\log n)$. Also, the evaluation of the Rotation Maps of G and H can be performed in $O(\log n)$ space. See the paper for a detailed proof of the space required for evaluating Rot_{G_i} .
4. The algorithm A' - this algorithm decides connectivity in the case of connected components which are expanders. This algorithm simply enumerates all paths from the source, and outputs "connected" if it encounters the required destination. Due to the property that expanders have logarithmic diameters, algorithm A' takes $O(\log n)$ space (see Lemma ??).

□

References

- [1] O. Reingold, Undirected ST-Connectivity in Log-Space, Proceedings of the 36th ACM Symposium on the Theory of Computing. ACM, New York, 2005 376–385.
- [2] O Reingold, S. Vadhan, A. Wigderson, Entropy waves, the zig-zag graph product, Ann. of Math. 155, no. 1 (2002), 157–187.