Implementation of the undirected-s-t-connectivity log-space algorithm

Tom Gur, Liron David

September 9, 2010

1 Introduction

In 2005, Omer Reingold proved that $L = SL$ by showing that the $SL-Complete$ problem $Undirected - S - T - Connectivity$ is in fact in L (Omer Reingold, Undirected ST-Connectivity in Log-Space, STOC 2005). We have implemented the mentioned algorithm. The program gets as an input either an adjacencies matrix of a graph, or alternately, it lets the user draw a graph. In addition, two vertices are selected (denote s, t). The algorithm will then decide whether there is a path between s and t, and if so - it will display this path. All this is being done by using only logarithmic space.

Practically speaking, although the algorithm is in L, implying that the algorithm runs in polynomial time - it turns out, after a careful analysis, that the polynomial which represents the runtime is very large, rendering the algorithm practically intractable: we've managed to find a very loose lower bound of $\Omega(n^{d*10^9})$ Where $d \geq 4$ (There is no time complexity analysis in the paper, and the exact analysis is very tedious, and depends on the specific base expander that is being used). Thus our program does not actually halt in reasonable time.

2 Overview of the algorithm

Roughly speaking, the algorithm works in the following stages, each stage uses logarithmic space only and the connectivity property (in each stage, s, t are connected iff they were connected in the original graph) is being preserved:

2.1 Input

An undirected graph $G = (V, E)$ is explicitly given by a standard adjacencies matrix in addition to two vertices $s, t \in V$. Denote $N = |V|$.

2.2 Switching the representation of G to a rotation map Rot_G

For a D-regular graph G, the rotation map $Rot_G : [N] \times [D] \to [N] \times [D]$ is defined as follows: $Rot_G(v,i) = (w,j)$ if the i'th edge incident to v leads to w, and this edge is the j'th edge incident to w. We construct such function from the adjacencies matrix of G, which given a pair of vertex and edge (label) returns the vertex which is reach by traversing from v through i and the edge that leads back to v.

2.3 Transforming G into a constant degree non-bipartite regular graph G_{reg}

The most efficient way to transform G into a constant degree no-bipartite regular graph G_{reg} while preserving connectivity is by replacing each vertex of G by a ring (set of subsequent vertices each connected to the previous vertex and the next vertex) of N vertices. Then adding a self loop for each vertex, and finally connecting the ring according to the connections in G, while maintaining that each vertex can have at most on edge that leads to a different ring (vertices without such edge, will just have another self-loop instead. Formally speaking, the rotation map of G_{reg} should be as such:

- $Rot_{G_{reg}}((v,w),1) = ((v,w\acute{}),2)$, where $w\acute{} = w+1$ if $w < N$ and $w\acute{} = 1$ otherwise.
- $Rot_{G_{reg}}((v,w),2) = ((v,w\acute{}),1)$, where $w\acute{} = w{-}1$ if $w > 1$ and $w\acute{} = N$ otherwise.
- $Rot_{G_{reg}}((v,w),3) = ((w,v),3)$ in case there is an edge between v and w in G. Otherwise, $Rot_{G_{reg}}((v,w),3) = ((v,w),3)$.
- $Rot_{G_{reg}}((v,w),i) = ((v,w),i)$, for $i > 3$.

2.4 Recursively turning G_{reg} to a constant degree regular graph with a logarithmic diameter G_l

This stage is the main transformation and the core of the algorithm. In this stage we use a combination of two graph products: powering & the zigzag product, in order to increase the expansion (measured by the spectral gap) upto where the diameter of the graph is logarithmic, while keeping the degree a constant (and preserving the connectivity, of course). The main transformation is being applied recursively until these properties are obtained. Roughly speaking, powering the graph increases the expansion (as long as the graph has self-loops, as we've insured) while increasing the degree, and the zigzag product reduces the degree while not harming the expansion too much.

2.4.1 Matrix powering

Raising the adjacencies matrix of a graph to the power of k results in a matrix describing all of the path in length k. If the graph has self-loops, as we've insured, the product describes all of the path of length upto k. Thus the expansion increases, but so does the degree. Connectivity is obviously held as power works separately on each connection component by its definition.

2.4.2 Zig-Zag product

The zig-zag product $G \circ H$ of regular graphs G, H is a regular graph of smaller degree than G, with a spectral gap that is bounded as a function of the spectral gap of G and H. The zigzag product was introduced by Reingold, Vadhan and Wigderson ("Entropy waves, the zig-zag graph product, and new constant-degree expanders", 2002).

Roughly speaking, the zig-zag product $G \circ H$ replaces each vertex of G with a copy (cloud) of H, and connects the vertices by moving a small step (zig) inside a cloud, followed by a big step (zag) between two clouds, and finally performs another small step inside the destination cloud.

More formally:Let G be a D-regular graph on $[N]$ with rotation map Rot_G and let H be a d-regular graph on $[D]$ with rotation map Rot_H. The zig-zag product $G \circ H$ is defined to be the d^2-regular graph on $[N] \times [D]$ whose rotation map $Rot_{G \circ H}$ is as follows:

$Rot_{G \circ H}((v, a), (i, j))$:

1. Let $(a', i') = Rot_H(a, i)$.
2. Let $(w, b') = Rot_G(v, a')$.
3. Let $(b, j') = Rot_H(b', j)$.
4. Output $((w, b), (j', i'))$.

The zigzag product holds these properties:

1. Reduction of the degree: It is immediate from the definition of the zigzag product that it transforms a graph G to a new graph which is d^2-regular. Thus if G is a significantly larger than H, the zigzag product will reduce the degree of G. Roughly speaking, by amplifying each vertex of G into a cloud of the size of H the product in fact splits the edges of each original vertex between the vertices of the cloud that replace it.

2. Spectral gap preservation: The expansion of a graph can be measured by its spectral gap. An important property of the zigzag product is the preservation of the spectral gap. That is, if H is a "good enough" expander (has a large spectral gap) then the expansion of the zigzag product is close to to the original expansion of G. Formally: Define a (N, D, λ)-graph as any D-regular graph on N vertices, whose second largest eigenvalue (of the associated random walk) has absolute value at most λ. Let G_1 be

a (N_1, D_1, λ_1)-graph and G_2 be a (D_1, D_2, λ_2)-graph, then $G_1 \circ G_2$ is a $(N_1 \cdot D_1, D_2^2, f(\lambda_1, \lambda_2))$-graph, where $f(\lambda_1, \lambda_2) < \lambda_1 + \lambda_2 + \lambda_2^2$).

3. Connectivity preservation: The zigzag product $G \circ H$ preserve each connected component of G, so every two connected vertices in G are also connected in $G \circ H$ and vice verse. Formally speaking, given two graphs G, H if s, t are vertices of G, and s', t' are the corresponding vertices in $G \circ H$ (in fact, s' and t' could be any vertices inside the corresponding "cloud" that the zigzag product induced), then s', t' are in the same connected component of $G \circ H$ if and only if s, t are in the same connected component in G. It's easy to see that this property holds, since the zig-zag product essentially transform each vertex of G into a "cloud" isomorphic to H, and the edges between the vertices of G are by definition transformed into edges between the "clouds" in $G \circ H$.

2.4.3 Main Transformation

The main transformation is basically a careful recursive composition of these two graph products, which keeps a very gentle balance between degree and expansion. It's defined as follows:

On input G and H, where G is a D^{16}-regular graph on $[N]$ and H is a D-regular graph on $[D^{16}]$, both given by their rotation maps, the transformation T outputs the rotation map of a graph G_l defined as follows:

- Set l to be the smallest integer such that $(1-1/DN^2)^{2l} < 1/2$.
- Set G_0 to equal G, and for $i > 0$ define G_i recursively by the rule: $G_i = (G_{i-1} \circ H)^8$.
- Denote by $T_i(G, H)$ the graph G_i, and $T(G, H) = G_l$.

2.5 Exhaustive search over all possible routes of G_l

Now, that we've got G_l a constant degree (denote d) expander of size n, with a logarithmic diameter - there are $O(d^{logn})$ possible routes, thus it is possible to perform an exhaustive search over all routes in logarithmic time.

3 Software operation instructions

The user interface is very simple. The user can choose between drawing a graph by hand or inputting the graph through a file.

In order to use a file, just click the "Enter graph path" checkbox, and either insert the path manually, or use the browse button to select it graphically. the file format is a normal, space separated, adjacencies matrix format. The vertices will by named by their order in the adjacencies matrix $0, \ldots, n-1$. Then the user should insert the number of the start and end vertices (s, t) and click "Ok".

Alternately, the user can click the "Draw graph" checkbox and click "Ok". Then a new window will appear (with instructions). In order to draw a vertex, one should simply click on an empty space inside the window. Double-clicking a vertex will set it as s, double-clicking on another vertex will set it as t. Finally, clicking on two consecutive vertices will draw an edge between them. Clicking "Ok" will run the algorithm.

4 Implementation details

We would like to open this section with a note: One of the most prominent qualities of this algorithm is the fact it only uses logarithmic space. This quality dictates a very unorthodox coding style, as non of the data is being stored. i.e at each step of each walk over G_l the entire calculation of the main transformation (recursion over recursion over recursion...) is being performed. Thus we don't actually explicitly ever build any of the graphs mentioned above, we only calculate labels over labels each time a graph is being accessed. This style of structure enforce a very strict coding style which is not very consistent with the standard object-oriented-java approach.

With that behind us, let's get into the schematics of the program. After the user inputs the graph G and the vertices s, t, the static "solve_ustconn" method from the Solver class is being called. Than, the base expander, H is being created. Although H does not depend on the input, thus is $O(1)$, it's size has to be of at least 4^{16} vertices, which is too large to be stored in the memory. In the paper H is being found by an exhaustive search (as it's only $O(1)$). Although deprecated, we've left the code for this exhaustive search in our code (which also calculates the spectral gap, and complete spectral analysis of the matrices - done by a combination of numerical linear algebra we wrote and algorithms from the JAMA library which we modified for our needs) - In practice, we've created an abstract class "Expander", of which we had several derived classes of quite a few explicit constructions of expanders. We've ended up choosing the explicit construction of the Gabber-Galil expander, due to the combination of small degree (8) and simplicity.

After H is create. The algorithm makes an instance of the "MainTransformation" class, which contains all of the functions that are needed to generate the rotation map of G_l, the final constant degree expander with a logarithmic diameter. The program then perform an exhaustive search for a path between s and t over the rotation map of G_l, and returns whether such path exists, and in the latter case, shows the path.

We also have a function that can explicitly build the adjacencies matrix from each rotation map, and another function that visually displays the graphs (more in section 5), so we can display the graph in each step of the transformations, but since the size of the graphs is enormous, this is not practical.

5 Code documentation

In this section we will describe the role of each important class (There are some trivial classes such as "Pair" which does not need any explanation) in the program. Each non-trivial method inside of each class is documented within the code.

5.1 graph_utils

5.1.1 Graph.java

The basic explicit graph class. Includes whatever you'd expect from an adjacencies matrix represented graph. You will also find the function Rot, which calculates the rotation map of the graph from the adjacencies matrix. Notice that this is the only place that the input is actually accessed. Also includes a method that can calculate the spectral gap of the graph and additional utils.

5.1.2 Expander.java

An abstract class inherited from Graph. Meant to be used to represent graphs by rotation map only, as these graphs are too big to store in memory.

5.1.3 GabberGalilExpander.java

The chosen implementation of the mentioned expander.

5.1.4 GraphOperators.java

This class includes the implementation of the zig-zag product and matrix powering. In addition, it includes the mentioned implementation of the rotation map of G_{reg}, the transformation that makes G into a constant degree regular graph.

5.1.5 GeneralUtils.java

Includes a method that extract the adjacenices matrix of a graph from its rotation map (The rest of the deprecated util have moved to the Junkyard file).

5.2 ustconn_solver_algorithm

5.2.1 Solver.java

The main class with the static method that allows to solve a ust-conn problem, and the exhaustive search method.

5.2.2 MainTransformation.java

This class holds everything that is needed to perform the recursive calculation of the main transformation. Actually stores the rotation map of the final graph G_l.

5.3 misc

5.3.1 Label.java

A simple vertex + index pair.

5.4 gui

5.4.1 USTCONN_Applet.java

The main applet.

5.4.2 IO_Handler.java

Includes the graph drawing applet and visualization of graphs given by adjacencies matrices.