Spring 2014

Construction Algorithms for Expander Graphs

Vlad S. Burca
Trinity College, vlad.burca@trincoll.edu

Construction Algorithms for Expander Graphs

VLAD Ș. BURCĂ '14
Adviser: Takunari Miyazaki

Graphs are mathematical objects that are comprised of nodes and edges that connect them. In computer science they are used to model concepts that exhibit network behaviors, such as social networks, communication paths or computer networks. In practice, it is desired that these graphs retain two main properties: sparseness and high connectivity. This is equivalent to having relatively short distances between two nodes but with an overall small number of edges. These graphs are called expander graphs and the main motivation behind studying them is the efficient network structure that they can produce due to their properties. We are specifically interested in the study of k-regular expander graphs, which are expander graphs whose nodes are each connected to exactly k other nodes. The goal of this project is to compare explicit and random methods of generating expander graphs based on the quality of the graphs they produce. This is done by analyzing the graphs' spectral property, which is an algebraic method of comparing expander graphs. The explicit methods we are considering are due to G. A. Margulis (for 5-regular graphs) and D. Angluin (for 3-regular graphs) and they are algebraic ways of generating expander graphs through a series of rules that connect initially disjoint nodes. The authors proved that these explicit methods would construct expander graphs. Moreover, the random methods generate random graphs that, experimentally, are proven to be just as good expanders as the ones constructed by these explicit methods. This project's approach to the random methods was influenced by a paper of K. Chang where the author evaluated the quality of 3 and 7-regular expander graphs resulted from random methods by using their spectral property. Therefore, our project implements these methods and provides a unified, experimental comparison between 3 and 5-regular expander graphs generated through explicit and random methods, by evaluating their spectral property. We conclude that even though the explicit methods produce better expanders for graphs with a small number of nodes, they stop producing them as we increase the number of nodes, while the random methods still generate reasonably good expander graphs.

1. INTRODUCTION

The construction of expander graphs has been an interesting challenge to mathematicians and theoretical computer scientists since the first proposal of explicit construction by G. A. Margulis in 1973 [1]. These graphs, having the special property of being sparse and highly connected, started attracting more attention once research in the field proved their usefulness in various applied topics, including sorting networks, computation of linear transformations, construction of good error-correcting codes, and more importantly, construction of efficient computer networks [2].

In order to apply expander graphs to these applications, efficient construction methods had to be found. Following Margulis' first explicit construction, D. Angluin proposed a simplification of the algorithm [3]. The goal of Margulis' method was to provide an explicit method of constructing expanders. He was able to produce such a method for creating a 5-regular expander and he generalized his results through a theorem [3] whose statement is highly depended on a constant d, referenced by his theorem. As noted by Angluin, the actual value of d was still unknown, so Angluin's goal was to come up with the simplest explicit construction that Margulis' theorem would still hold for. Thus, in his paper [3], he presents an algebraic method for generating 3-regular expanders.

Along with these explicit ways of generating expander graphs, there were also attempts of generating random expander graphs – starting with a random graph and modifying it such that it will eventually have the special properties of an expander. Given the complicated nature of the explicit methods for generating really good expanders, K. Chang's and C. Hammond's goal was to test the conjecture according to

Author's address: Computer Science Department, Trinity College, 300 Summit Street, Hartford, Connecticut 06106-3100 (E-mail: vlad.burca.2014@trincoll.edu).

which the simple random 3-regular bipartite graphs may be just as good expanders as the results of the explicit methods. They approached this question using a series of computational and numerical experiments and their results [4], [5] proved that it is possible to achieve very good expanders, in terms of their expansion constant, by using a randomized strategy when creating edges between nodes.

The goal of this project is to experimentally compare the expansion properties of expanders by using results of the previously mentioned methods (Margulis', Angluin's and the random methods). It does this by using the existing explicit and random construction methods in order to generate expander graphs and then numerically compare their quality. While Margulis' and Angluin's goal was to come up with explicit construction algorithms, and Chang's and Hammond's was to test whether or not random methods can produce good expanders too, our goal is to combine these methods, compare them and experimentally conjecture which type of methods (explicit or random) generates the best expanders over 3 and 5-regular graphs with various number of nodes.

First, we are going to give an informal definition of Ramanujan graphs. These are k-regular graphs that have a large spectral gap. The spectral gap is defined by the difference of the absolute values of the 2 largest eigenvalues of the adjacency matrix that defines the graph. Moreover, the Ramanujan graphs are considered to be the best expander graphs in terms of the spectral property defined through the eigenvalues.

In order to quantify how good a graph is in terms of being an expander, the theorem of Friedman [6] was used. This states that random k-regular graphs on n vertices are almost Ramanujan, that is, they satisfy

$$\lambda \leq 2\sqrt{k-1} \qquad (1)$$

where λ is the second largest eigenvalue of the adjacency matrix that defines the graph. This implies that Ramanujan graphs have a smallest possible λ, bounded by the given inequality.

Even though the graphs generated by Margulis' or Angluin's methods are explicit, a consistent way of evaluating their results compared to the random methods was needed so we decided to use the same Lubotzky-Phillips-Sarnak inequality, (1). This decision was made due to the fact that it is a straightforward, computationally easy way of quantifying how close a graph is to being a Ramanujan graph. Moreover, since the Ramanujan graphs are the expander graphs with the optimal expander constants, it made sense to compare all graphs constructed through the expander generating algorithms to the best expanders that can be achieved.

In order to accomplish this goal, implementations of all the expander generating methods mentioned earlier, were wrote and tests were ran by generating expanders through them and evaluating their eigenvalues, according to inequality (1). Most of the project package is done in Python, using the NumPy library for various numerical computations. The computation of eigenvalues (for relatively large matrices) was achieved through a C implementation of the Power Method – a slightly modified implementation of the algorithm presented in Chang's thesis [4]. This was due to the slowness of Python and NumPy in generating the eigenvalues for relatively large matrices. Moreover, the representation of the generated graphs is being done through adjacency list matrices (n x k, where n is the number of nodes and k the number of edges). The NumPy method for generating eigenvalues was not

making use of this optimization and this was causing it to be even slower. Therefore, we decided to use a faster language, C, and a method (Power Method) that would fully take advantage of the way the graph was being stored, through adjacency list matrices.
The package can run 4 different algorithms for generating expanders: Angluin's, Margulis', method for random 3-regular bipartite graphs and method for random 5-regular bipartite graphs. In practice, the implemented random generating method can generate k-regular bipartite graphs, for any positive k. The reasoning behind picking k = 3, 5 is that Angluin's method produces 3-regular bipartite graphs and Margulis' does 5-regular bipartite graphs. Therefore, it was desired to pair these two explicit constructions to equivalent random expanders. The package is implemented entirely and can produce sets of data for all 4 methods. From the experimental results, it can be seen that the random expanders tend to perform better for an increasing value of n, while the explicit constructions tend to do the opposite, performing worse. This could also be due to the fact that, unlike the random methods, the explicit ones generate a higher number of multi-edges between pairs of nodes in the constructed graph.

2. PROJECT DESCRIPTION

The project's main objective is to be able to construct expander graphs for any given input, n (number of nodes). In order to achieve this, we decided to select a couple of already researched algorithms that produce expander graphs. Since, through the literature reading that was done, it was found that there are two types of algorithms that could generate graphs with these special properties – explicit methods and random methods – we decided to implement both and test their performances. At this point in the project we realized that, since Angluin's method produces a 3-regular bipartite expander, it could be tested against a 3-regular bipartite random expander; we would proceed with a similar approach for Margulis' method and a 5-regular bipartite random expander. In the rest of this section, we will briefly describe how each of the algorithms works.

2.1 Definition

Throughout the project, k-regular bipartite expander graphs were used. *Bipartite graphs* are graphs with nodes that can be divided into two disjoint sets such that no edge is incident to any two nodes in the same set. Additionally, the graph G is called *k-regular* if each node has *exactly k* edges incident to it.

An *expander graph* is a highly connected and sparse graph. It is an undirected graph, G = (V, E) characterized by an *expander constant*, with V = the set of nodes and E = the set of edges. There are several ways of defining this constant, through *edge expansion*, *vertex expansion* or *spectral expansion*. In the research I did for this project I focused on k-regular graphs and I only used the *spectral property*. This is measured by the *spectral gap*, which is defined in the following way:

$$spectral\_gap(G) = k - \lambda_2$$

where λ_2 represents the second largest eigenvalue of the adjacency matrix used to define the k-regular G.

Since the project evaluates expander graphs in terms of how close they are to *Ramanujan graphs*, a definition what this kind of graph means is necessary. A *Ramanujan graph* is a graph with the largest spectral gap possible. Moreover, a more practical definition of a *Ramanujan graph*, and the definition that will be used for

the rest of the project description, is that a graph is Ramanujan if the inequality (1) is satisfied.
Given the previous definition of expander graphs, note that all the methods (both explicit and random) presented in this project construct graphs that satisfy the *spectral expansion* property of expanders.
Before proceeding with the descriptions, some comments have to be made on the notations that will be used from now on, through the paper (this notation will be consistent to the variables and constants that are used in the actual project code package). Therefore:

Table I. Variable notations

n	Refers to the modulo of the group $\mathbb{Z}_n$
k	Degree of the k-regular graphs
size = n^2	Number of elements in matrix A or B
H	Adjacency list matrix of expander graph
size_H = 2 * size = 2 * n^2	The dimension of H

2.2 Methods

2.2.1 Margulis' Method

As previously stated in the Introduction, Margulis' goals were to give an explicit, algebraic construction for expander graphs. His results were concluded into a more general theorem about concentrators. Concentrators are expander graphs that are defined by Margulis as:

Definition 1 [3]. Let H consist of an ordered pair (A, B) of sets of nodes, together with some edges between the nodes of A and the nodes of B. Let c and α be positive, with $c > 1$ and $\alpha < 1$. Then H is called a *(c, α)-concentrator* if and only if for every nonempty proper subset X of A with $|X| < \alpha |A|$, the set $Y = \{b \in B: b \text{ is adjacent to some } a \in X\}$ satisfies $|Y| > c|X|$.

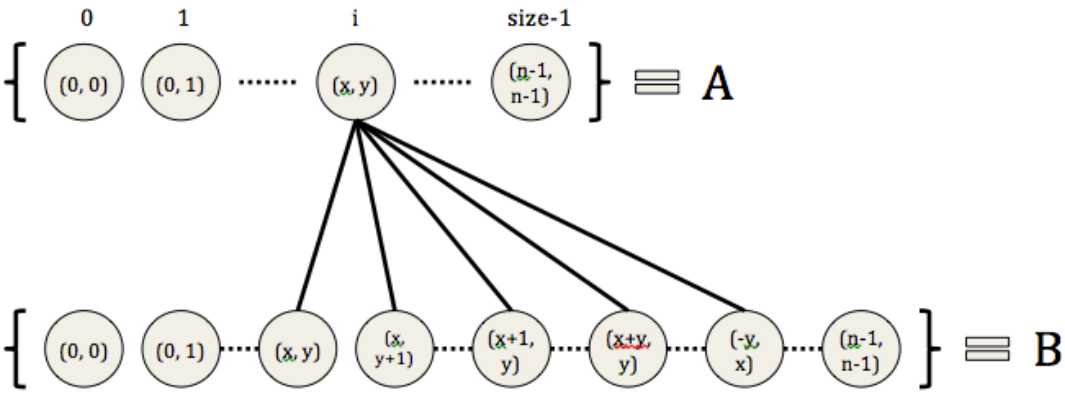
Informally, concentrators are graphs that hold desirable connectivity properties similar to the ones of the expander graphs. Following this definition, Margulis states that the graphs constructed through his method, H_m, where each of the elements of H_m are modulo m, follow the theorem:

Theorem 1 [3]. There exist a positive constant d such that for any integer $m > 1$ and any α such that $0 < \alpha < 1$, H_m is a *$(1 + d(1 - \alpha), \alpha)$-concentrator*.

Being the first explicit construction [1] of an expander graph, it is also the simplest one, involving just several rules of connecting nodes between two sets of disjoint nodes. At the beginning of the algorithm, none of the nodes are connected. The elements of the two disjoint sets, A and B, are tuples of the following form $(x, y) \in \mathbb{Z}_n$. Therefore, the number of elements in A (or B) is equal to n^2 (`size`). The rules for connecting the nodes from A to nodes in B are the following:

- Take a node $(x, y) \in A$ and connect it to the following nodes:
  - $(x, y) \in B$
  - $(x + 1, y) \in B$
  - $(x, y + 1) \in B$
  - $(x + y, y) \in B$
  - $(-y, x) \in B$

A note has to be made regarding the operations seen within the tuple-nodes. Since the two elements of the tuple always have to be from $\mathbb{Z}_n$, the operations have to be done modulo n.



(Fig. 1) Illustration of Margulis' method

Here is the pseudocode for this method:

ALGORITHM 1. Margulis' method of generating expander graphs

```
    INPUT: The size of the permutation set (corresponding to the value n from the description)
    OUTPUT: The adjacency list of an expander graph
1   generate matrices A and B as a permutation of the {0 … size-1} elements; resize the
    permutation to size x size
2   initialize empty matrix H of size size_H x k
3   for each index i in matrix A
4           get the corresponding tuple (x0, y0) of the index i
5           construct each of the five tuples that (x0, y0) can connect to
6           get the corresponding index j of the newly constructed (x, y) tuple
7           save the indices in the adjacency list matrix: H[i][k] = j; H[j][k] = i
8   return H
```

(Code 1) Pseudocode for Margulis' method of generating expander graphs

Even though the algorithm seems simple, there are some small implementation details that have to be taken into account to improve performance. They involve transitioning from the index of the tuple representation of a node to the actual tuple (x, y), and vice-versa.

We will first address the first direction: *going from the index i of a tuple to the actual tuple (x, y)*. The indexing of the elements from $\mathbb{Z}_n \times \mathbb{Z}_n$ is being done as illustrated in (Fig. 1). Therefore, the following rule of recovering the tuple (x, y) from index I can be obtained:

$$x = i/n \qquad y = i \,\%\, n \tag{2}$$

Moreover, in order to recover the index i from the tuple (x, y):

$$i = x * n + y \,\%\, n \tag{3}$$

Using equations (2) and (3), the operations from lines {4} and {6} of (Code 1) can be optimally achieved. Before using this approach, an array (which in Python is represented as a *list*) was used to store the entire collection of $\mathbb{Z}_n \times \mathbb{Z}_n$ and then

access it based on the needed index. Also, such Python list has a method that would return the index of a given element, so that was very helpful. The problem with it is that, being a list, the operations were very slow on such a data structure with a relatively big number of elements stored in it. This is why this approach was reconsidered and it resulted in the improved one, described above.

2.2.2 Angluin's Method

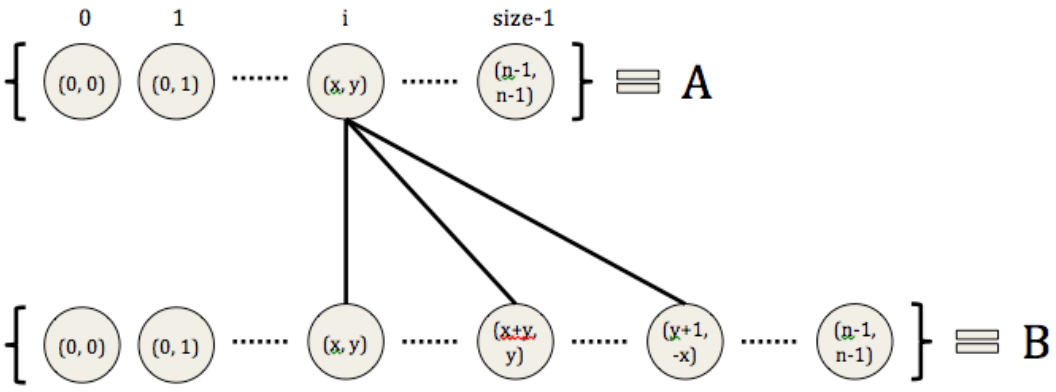
Angluin's goal was to improve Margulis' method by finding the simplest construction for which **Theorem 1** would still hold. She managed to do this by formulating an explicit method for 3-regular expander graphs, similar to Margulis' method for 5-regular graphs. She proved [3] that **Theorem 1** holds with her resulted 3-regular expander graphs in place of Margulis' H_m.

This method is inspired by Margulis' algorithm, but, as stated in the author's paper [3], it provides an even simpler explicit construction of expander graphs. The initial setup with the two disjoint sets is identical to the one of the previously presented method, so I am not going to re-state it over here. Instead, I will present the way of constructing the edges for Angluin's method:

- Take a node $(x, y) \in A$ and connect it to the following nodes:
  - $(x, y) \in B$
  - $(x + y, y) \in B$
  - $(y + 1, -x) \in B$

A note has to be made regarding the operations seen within the tuple-nodes. Since the two elements of the tuple always have to be from $\mathbb{Z}_n$, the operations have to be done modulo n.

The pseudocode for Angluin's method is identical to the one presented in (Code 1). Moreover, the implementation optimizations mentioned earlier through equations (2) and (3) apply here as well, since the algorithm involves the same transformations from index to tuple and vice-versa.

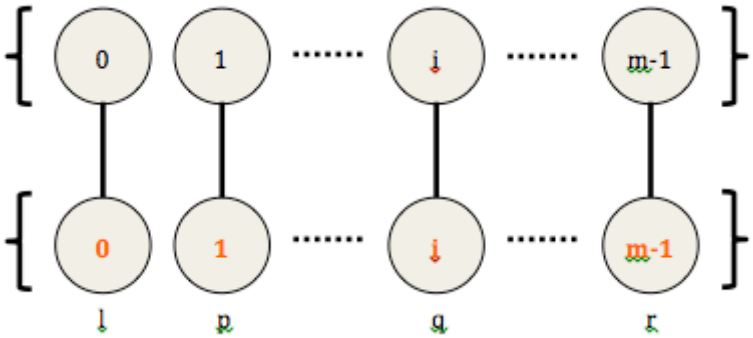


(Fig. 2) Illustration of Angluin's method

2.2.3 Random Method

The random method, inspired by Chang's work and results in his thesis [4], generates k-regular bipartite graphs without building edges using specific rules, as the previously two methods did. Instead, it does so by splitting the initial set of `size_H` nodes into `{0 … size_H/2 - 1}` and `{size_H/2 … size_H – 1}`. Afterwards, it randomly permutes the second set of nodes and, finally, it creates the edges between

the nodes in the first set and the nodes in the second set in order, correspondingly, as illustrated below (*for ease of notation, let* `m = size_H/2`):



(Fig. 3) Illustration of the Random method

The indices inside the nodes are the ones after permutation, while the ones on the outside, are the original ones, after splitting ($l, p, q, r \in \{(size\_H)/2 \ldots size\_H\}$).
This permute-and-create edges process is repeated k times, generating a k-regular bipartite random graph. It is bipartite because the edges from within each of the 2 halves are not connected to each other – they are only connected to nodes from the other half.

Here is the pseudocode for this method:

ALGORITHM 2. Random method of generating expander graphs

```
    INPUT: Size of desired matrix H.
    OUTPUT: The adjacency list of an expander graph.
1   initialize empty matrix H of size size_H x k
2   generate list of size_H elements and split it into half
3   for edge in k
4            randomly permute the second half of the split
5            for each node i in the first half of the split
                     H[i][k] = element_i_of_the_permuted_half
                     H[element_i_of_the_permuted_half][k] = i
8   return H
```

(Code 2) Pseudocode for the Random method of generating expander graphs

Since the graphs are randomly generated, a sampling factor was used (given as an input) in order to generate multiple such graphs and then take the average of their eigenvalues. In this way, the result is more reliable to what would happen in general, with randomly generated expanders.

2.3 Deliverables

Once the graphs are generated through one of these methods, in order to compare their quality as expander graphs, inequality (1) was used. This provided a consistent way of evaluating which method produces better expanders as the number of nodes increases. Further discussion on how these methods are structured within the package will be provided in the next section.

The deliverables of the project consist of a Python & C package (that will be further illustrated in the next section) that can generate expander graphs through any of the 4 methods mentioned earlier. Moreover, the package summarizes the quality of each of the generated expanders by computing the eigenvalue of each of them and comparing it through the inequality mentioned at (1).

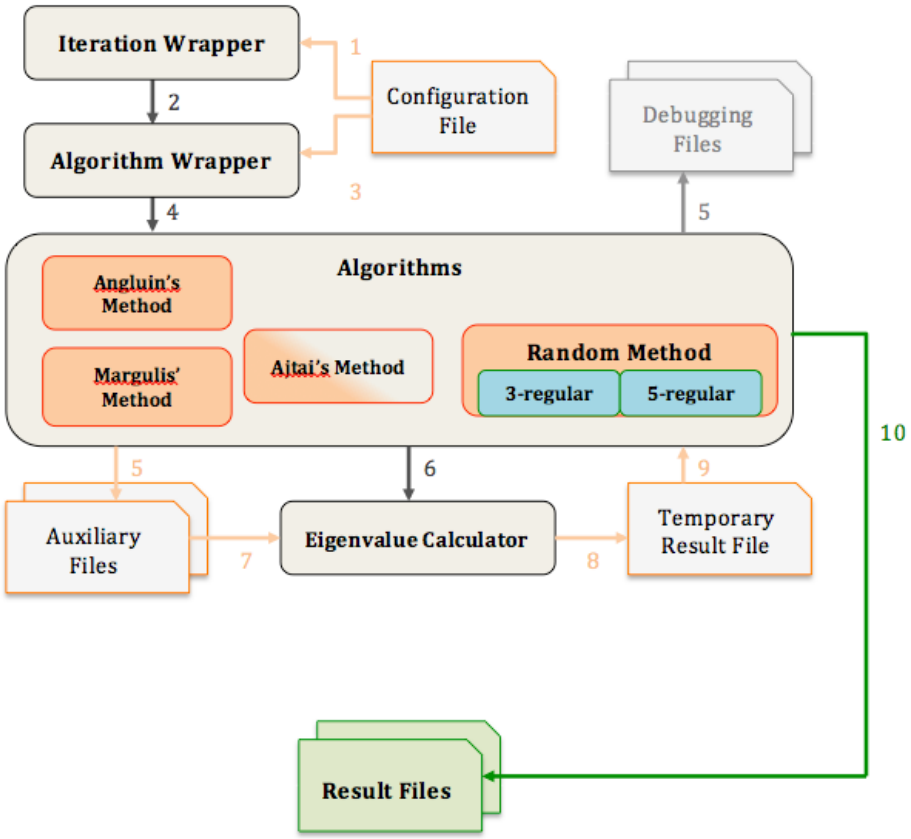
The package also contains a 5th method, explicit as well, due to M. Ajtai [7], that is not fully implemented. His approach is more complicated than the ones presented so far and we will talk more about it at the end of this paper.

3. IMPLEMENTATION

The expander graph-generating package was implemented with the 4 methods mentioned before, along with an evaluating method (through eigenvalues) for the constructed graphs. As mentioned above, this section will give a detailed explanation of the overall implementation design of the package.
The package is composed of several modules that will be explained in more detail:

1. Iteration wrapper
2. Algorithm wrapper
3. Algorithms
4. Eigenvalue Calculator

The flow of data through the package is illustrated in (Fig. 4), below.



(Fig. 4) Program execution flow of the expander package

Here is a step-by-step description of the illustrated flow, pointing out the main features of each of the modules.

Configuration File – Provides parameters input (n, number of samples to be generated for the random graphs, etc.), debugging options (printing generator matrices with indices, printing the adjacency matrix of the expander, etc.) along with the option of choosing which algorithms should the package run (Angluin's method, Margulis' method, Random 3-regular, Random 5-regular). Moreover, it has a flag for cleaning up all the auxiliary or debugging files generated from previous runs.

Iteration Wrapper [*Python*] – This initial wrapper is the one that starts the entire package. It receives input from the *Configuration File* [1] and then it runs the ***Algorithm Wrapper*** [2] for different values of n. The *Iteration Wrapper* changes the value of n in the *Configuration File* and runs the *Algorithm Wrapper*. This process happens for all the hard-coded values of n.

Algorithm Wrapper [*Python*] – This wrapper is composed of 2 different Python scripts. The first one reads the earlier modified *Configuration File* [3] and saves all the parameters and the algorithms that have to be run. It then generates the generator matrices A and B (the ones used for Angluin's or Margulis' methods) and checks which algorithm should it call next – if it is one of the explicit methods or one of the random methods. The second script of this wrapper splits it further on, and decides exactly which of the four different algorithms to pick. If multiple algorithms were checked in the configuration file, the wrapper will run each of them, one by one.

Debugging Files [optional] – If the flags for debugging information are switched on in the *Configuration File*, the *Algorithm Wrapper* will write [5] the specified information in these files.

Algorithms [*Python*] – Once the *Algorithm Wrapper* decides on which algorithm to call, the corresponding method is being triggered [4]. Depending on the algorithm details, as described in the previous section, this module will generate the adjacency list matrix of an expander graph. It will then write it to an *Auxiliary File* [5] and call the ***Eigenvalue Calculator*** [6] module in order to evaluate the quality of the generated expander.

Auxiliary Files – Provide communication between the *Algorithms* module and the *Eigenvalue Calculator* module. One of the files saves the adjacency matrix of the earlier generated expander, while the other saves parameters specific to the generated expander.

Eigenvalue Calculator [*C*] – This module has one of the hardest jobs – this is also the reason why it was implemented in C, rather than Python (a more detailed explanation on this will be provided later on in this section). The *Eigenvalue Calculator* reads the matrix information from the *Auxiliary Files* [7] and then applies the Power Method algorithm to generate the second largest eigenvalue of the given expander. It then writes it to another auxiliary file, *Temporary Result File* [8].

While the *Eigenvalue Calculator* is running, the *Algorithms* module is asleep, waiting for the eigenvalue to be calculated. It constantly sends signals to the *Eigenvalue Calculator* process until it makes sure that the process completed. It then opens the *Temporary Result File* [9], saves the computed eigenvalue and prints it in the *Result Files* [10]

Result Files – These files hold information for each of the 4 possible types of generated expanders. For each of these types, they hold information on the number of

nodes and the expansion value of the generated graph. The expansion value is defined, from inequality (1) as:

$$expansion\ value = 2\sqrt{k-1} - \lambda \quad (4)$$

Informally, this tells "how far away is a graph from being Ramanujan". In order for a given graph to be Ramanujan, the result of the difference in (4) has to be maximized.

Example of output result, for Angluin's method:

```
1  angluin:
2          8: 0.4855051247461901
3          18: 0.00437712474619012
4          32: -0.021004875253809896
```

(Code 3) Example of Result File output

3.1 Challenges

As mentioned earlier, there were a couple of challenges that had to be overcome while implementing this package. The first challenge was more related to the code structure – as it can be seen, the code is very modulated; this structure gave me a very easy way of debugging various parts of it and allowed me to work in parallel on different modules.

The small implementation challenges and optimizations with each of the constructing methods have been discussed in the previous section.

The main challenge of this project was computing the eigenvalues of the constructed graphs. The generated graphs have very big dimensions, up to 2,000,000 x 5. My initial approach was to use a specific numerical Python library. The implementation of the eigenvalue method was very slow for the needs of the project. Therefore, we decided to implement the Power Method algorithm that Chang uses as well in his thesis [4]. The implementation was still in Python and, although there was some visible improvement, it was not fast enough for big matrices. Thus, we decided to implement it again, in C this time. The implementation was able to compute the second largest eigenvalue for the largest matrix (2,000,000 x 5) in a matter of seconds and this is why we ended up using a cross language implementation of the package.

3.2 Results

The package was ran for values of `size_H` that are close to the values that Chang used to test his random methods for 3-regular and 7-regular bipartite graphs. The *Result Files* are attached in *Appendix A*. Moreover, Table 1 presents a summary of these results for easier analysis. It can be seen, as Chang concluded as well in his thesis, that the graphs generated through the Random Method converge to being Ramanujan as their size increases, as tested against the inequality in (1). On the other hand, the best expanders constructed are achieved at relatively low graph sizes, due to Explicit Methods. Once the size of the expander grows, the expander quality of the graphs produced by either of the methods is getting worse, but the Random Methods are still able to produce expander graphs – i.e. the value of inequality (4) is still positive most of the times.

RESULTS

	Expansion value			
Number of nodes	Angluin 3-regular	Random 3-regular	Margulis 5-regular	Random 5-regular
8	0.485505	-0.171572	1.267949	1.267949
18	0.004377	0.300356	0.511528	-0.147700
32	-0.021004	0.051901	0.030111	0.238354
162	-0.089250	-0.017769	-0.667582	0.004192
1250	-0.121959	0.007560	-0.880319	-0.006768
1800	-0.124888	0.000638	-0.894546	0.006628

(Table 1) Summary of results

Moreover, similar to Chang's results, it can be seen that the expansion value of the graphs generated through random methods gets very close to 0. His results also state that even with a large number of nodes the random methods are still generating Ramanujan graphs with a more than 79% chance (for graphs with 1000 nodes) [4]. Similarly, on our summary table, we can see that indeed, the expansion value for the graphs generated through random methods, having 1000 nodes, is still positive, meaning that the graphs are still in the Ramanujan spectrum. This observation does not stand for the explicit methods though, since the expansion value starts to grow negatively from a low number of nodes (aprox. 50 in Margulis' case).

4. FUTURE WORK

As mentioned earlier, the package included in this project contains an additional explicit method of generating expander graphs, due to M. Ajtai, that is not fully implemented yet. Briefly, we will present the main ideas behind this method as well as its current implementation status.

4.1 Ajtai's Method

This explicit construction method generates 3-regular expanders using a different approach than the previously described methods. It starts with a random graph and it modifies it by switching edges between pairs of nodes. Thus, it chooses edges (x, y), (u, v) in the graph, deletes them and adds the edges (x, v), (y, u) back to the graph [7]. The goal of the algorithm is to minimize the number of cycles of a given length, $s = c \log n$, where n is the number of nodes in the initial graph and c is a fixed absolute constant. The choice of the edges that are switched is also done based on the strategy of minimizing the number of cycles. Moreover, when the algorithm reaches a local minimum in the number of cycles of length s, the graph is an expander. Ajtai's paper that describes this algorithm states that the number of cycles of length s is a good measure of the expanding properties [7].

We designed the implementation of this method through 3 main steps:

1. Create H, a random 3-regular graph on n vertices.
2. Create method that swaps edges *(x, y)*, *(u, v)* with *(x, v)*, *(y, u)*.
3. Using the condition previously stated in the description of the algorithm, perform swaps on edges of graph H until the stopping condition occurs (the decrease in the number of cycles of length s is not significant anymore).

Currently, only steps **1** and **2** are implemented, with step **3** being the main goal of future work that can be done on this project. Moreover, we are considering future work that involves running the tests for even bigger values of `size_H` and logging the number of multi-edges and the maximum number of multi-edges per pair of nodes that is achieved through each of the methods. Ideally, one would want an expander

that would have a relatively low number of multi-edges - so exploring construction methods that achieve that also represents a topic of interest for this project.

5. CONCLUSIONS

The goal of this project was to experimentally compare the expander graphs generated by explicit and random methods by analyzing their spectral property. By implementing a package that would generate expanders through the methods mentioned in this paper (due to Margulis, Angluin and random methods), we were able to conclude that, as the number of vertices increases, the random methods tend to produce graphs that are within the bounds of being Ramanujan (by using inequality (1)), while the graphs constructed through explicit methods are failing the Ramanujan test.

REFERENCES

[1] Margulis, G. Explicit constructions of concentrations. *Problemy Peredachi Informatsii, 9* (4). 71-80. (English translation in: Problems of Information Transmission, Plenum. New York, 1975).

[2] Shlomo, H., Linial, N. and Wigderson, A. Expander graphs and their applications. *Bulletin of the American Mathematical Society, 43.4.* 439-561.

[3] Angluin, D. A note on construction of Margulis. *Information Processing Letters, 8* (1). 17-19.

[4] Chang, K. An experimental approach to studying Ramanujan graphs. *Math Junior Seminar Thesis* (2001).

[5] Hammond, C. Efficient algorithms for random expander graphs.

[6] Lubotzky, A., Phillips, R. and Sarnak, P. Ramanujan graphs. *Combinatorica 8* (1988). 261-277.

[7] Ajtai, M. Recursive construction for 3-regular expanders. *Combinatorica 14* (1994). 379-416.

APPENDIX A (Result files)

```
angluin:
  8: 0.4855051247461901
  18: 0.00437712474619012
  32: -0.021004875253809896
  50: -0.043587875253809916
  72: -0.06139087525380971
  98: -0.08292787525380984
  128: -0.10825287525380992
  162: -0.1092508752538097
  200: -0.13472487525380968
  450: -0.16013087525380971
  800: -0.18751387525380963
  1250: -0.2119598752538098
  1800: -0.23488887525380976
```

```
random3:
  8: -0.1715728752538097
  18: 0.3003561247461901
  32: 0.05190112474619024
  50: 0.04480512474619047
  72: -0.07115387525380967
  98: 0.05197312474619009
  128: 0.010014124746190234
  162: -0.017769875253809797
  200: 0.011234532746191572
  450: 0.00547112474619027
  800: 0.006104124746190376
  1250: 0.0075601247461905
  1800: 0.000638124746190182
```

```
margulis:
  8: 1.2679490000000002
  18: 0.5115280000000002
  32: 0.03011100000000022
  50: -0.2548469999999998
  72: -0.4262220000000001
  98: -0.5370369999999998
  128: -0.6130129999999996
  162: -0.6675820000000003
  200: -0.7082660000000001
  450: -0.8138629999999996
  800: -0.8572610000000003
  1250: -0.8803190000000001
  1800: -0.8945460000000001
```

```
random5:
  8: 1.2679490000000002
  18: -0.14770099999999964
  32: 0.23835499999999987
  50: 0.2881
  72: 0.1078030000000001
  98: -0.014101000000000141
  128: 0.057017000000000095
  162: 0.004192999999999891
  200: 0.043402918000000845
  450: 0.04419400000000007
  800: 0.012801000000000062
  1250: -0.006768000000000107
  1800: 0.006628000000000078
```

APPENDIX B (Source code)

The code can be found entirely at: https://github.com/vburca/Senior-Project

Submitted on May 9, 2014