5-2009

Understanding sources and consequences of variation in self-assembly

Arjun Dasgupta

University of Arkansas, Fayetteville

Follow this and additional works at: http://scholarworks.uark.edu/csceuht

Part of the Numerical Analysis and Scientific Computing Commons

Recommended Citation


http://scholarworks.uark.edu/csceuht/23

This Thesis is brought to you for free and open access by the Computer Science and Computer Engineering at ScholarWorks@UARK. It has been accepted for inclusion in Computer Science and Computer Engineering Undergraduate Honors Theses by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu, ccmiddle@uark.edu.
UNDERSTANDING SOURCES AND CONSEQUENCES OF VARIATION IN SELF-ASSEMBLY
UNDERSTANDING SOURCES AND CONSEQUENCES OF VARIATION IN SELF-ASSEMBLY

A thesis submitted in partial fulfillment of the requirements for the honors degree of Bachelor of Science in Computer Engineering

By

Arjun Dasgupta

May 2009
University of Arkansas
Self-assembly is the process of constructing a larger structure from small component parts that randomly collide and bond through weak interactions [2]. It is a potentially important method in the field of nanoscale manufacturing. However, because of the nature of self-assembly, randomness and the design of self-assembly compete with each other, and unforeseen things can be produced. In this thesis, the tiling assembly model of Winfree [7] was simulated. The tiling assembly model has two bonds which assemble planar nanostructures. The goal of the simulations was to determine what probabilities would produce an average degree of four (the ideal) for each node in the structure. It was found that large differences in bond formation probabilities tended to produce structures with higher average degree, in accordance with the theory of [1], but also that large probabilities in general did the same thing. In addition, the simulations did not produce degrees of four because the simulations did not impose sufficient geometric constraints upon bond formation.
This thesis is approved for recommendation
to the Honors College.

Thesis Director:

____________________________________
Russell J. Deaton

Thesis Committee:

____________________________________
James P. Parkerson

____________________________________
Jin-Woo Kim
THESIS DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this thesis when needed for research and/or scholarship.

Agreed ______________________________________

                              Arjun Dasgupta

Refused ______________________________________

                              Arjun Dasgupta
ACKNOWLEDGEMENTS

This thesis is an outcome of the support and good wishes of many people. I am indebted to Dr. Russell Deaton for his patience and watchful guidance throughout the entire duration of the project. He has provided me tremendous support and advice. Without him, the project and the honors thesis would not be possible. I thank Dr. Jin-Woo Kim and Dr. Pat Parkerson for kindly serving on my thesis committee, and Dr. Craig Thompson, for uploading the much helpful thesis template on his website.

I thank my family and friends for the encouragement, care and support at every stage of my work.

Lastly, I thank the Honors College for supporting me with undergraduate research grants.
# TABLE OF CONTENTS

1. **Introduction** .......................................................................................................................... 1
   1.1 Problem .............................................................................................................................. 1
   1.2 Thesis Statement ................................................................................................................ 1
   1.3 Approach .......................................................................................................................... 1
   1.4 Organization of this Thesis ............................................................................................ 3

2. **Background** .......................................................................................................................... 4
   2.1 Self-assembly .................................................................................................................... 4
      2.1.1 Wang tiles .................................................................................................................. 4
      2.1.2 Self-assembly and biomolecular computing .............................................................. 5
      2.1.3 Random energy model and probabilistic self-assembly ............................................. 6

3. **Methodology** ....................................................................................................................... 8
   3.1 System design .................................................................................................................... 8
      3.1.1 Features of the simulation ......................................................................................... 8
      3.1.2 Variations in types of particles in system ................................................................. 9
   3.2 Implementation ................................................................................................................ 11
      3.2.1 Simulations with one pair of vertical-horizontal bond probabilities ................. 12
      3.2.2 Simulations with vertical-horizontal bond probability matrix ....................... 13

4. **Results** .................................................................................................................................. 15
   4.1 Data .................................................................................................................................. 15
   4.2 Analysis ............................................................................................................................... 25

5. **Conclusion** ............................................................................................................................ 28
   5.1 Summary ........................................................................................................................... 28
5.2 Contributions .......................................................................................................... 28
5.3 Future Work ............................................................................................................ 29

6. References ............................................................................................................................ 30

7. Appendix A: Code .............................................................................................................. 31

7.1 Bond .......................................................................................................................... 31
    7.1.1 bond.h ............................................................................................................... 31
    7.1.2 bond.cpp ........................................................................................................... 31

7.2 Particle ......................................................................................................................... 32
    7.2.1 particle.h ........................................................................................................... 32
    7.2.2 particle.cpp for System 1 and System 3 ........................................................... 32
    7.2.3 particle.cpp for System 2 ................................................................................. 35

7.3 Random ....................................................................................................................... 39
    7.3.1 random.h .......................................................................................................... 39
    7.3.2 random.cpp ....................................................................................................... 39

7.4 System ......................................................................................................................... 40
    7.4.1 System 1 with probability pairs ....................................................................... 40
    7.4.2 System 1 with probability matrix ..................................................................... 44
    7.4.3 System 2 with probability pairs ....................................................................... 47
    7.4.4 System 2 with probability matrix ..................................................................... 51
    7.4.5 System 3 with probability pairs ....................................................................... 55
    7.4.6 System 3 with probability matrix ..................................................................... 60

7.5 assembly.h .................................................................................................................. 64

7.6 Makefile ..................................................................................................................... 64

7.7 graph_plot.plt (Gnuplot script file) ............................................................................ 65
LIST OF FIGURES

Figure 2.1: 8 Wang tiles that can tile a plane. A sample of the tiling is to the right........................................4
Figure 2.2: The logical structure of a 2D lattice using A-B tiling .................................................................5
Figure 2.3: System containing 8 different types of Penrose tiles that can be used to generate a perfect tiling6
Figure 2.4: Tiling generated using a cluster of 10 tiles (a) made up of the 8 different tiles in Figure 2.3.......7
Figure 3.1: Particles in System 1. The particles are identical and 2 pairs of different bonds......................9
Figure 3.2: Self-assembly in System 2.................................................................9
Figure 3.3: Particles in System 2. The particles are identical and have four different bonds...............10
Figure 3.4: Self-assembly in System 2.................................................................10
Figure 3.5: The four types of particles in System 3......................................................................................11
Figure 3.6: Self-assembly in System 3.................................................................11
Figure 4.1: Final aggregates in System 1 for bond probabilities 0.1 and 0.9 ................................................16
Figure 4.2: Final aggregates in System 1 for bond probabilities 0.5 and 0.5 ..............................................16
Figure 4.3: Final aggregates in System 1 for bond probabilities 1 and 1 ......................................................17
Figure 4.4: Plot of average degree of particle in System 1..........................................................................18
Figure 4.5: Final aggregates in System 2 for bond probabilities 0.1 and 0.9 ................................................19
Figure 4.6: Final aggregates in System 2 for bond probabilities 0.5 and 0.5 ..............................................19
Figure 4.7: Final aggregates in System 2 for bond probabilities 1 and 1 ......................................................20
Figure 4.8: Plot of average degree of particles in System 2 .......................................................................21
Figure 4.9: Final aggregates in System 3 for bond probabilities 0.1 and 0.9 ................................................22
Figure 4.10: Final aggregates in System 3 for bond probabilities 0.5 and 0.5 ...........................................22
Figure 4.11: Final aggregates in System 3 for bond probabilities 1 and 1 ...............................................23
Figure 4.12: Plot of average degree of particles in System 3 ...................................................................24
Figure 4.13: Final aggregates in System 1 for bond probabilities 0 and 1 ...................................................25
1. INTRODUCTION

1.1 Problem

Self-assembly with DNA has been used to assemble small electronic circuits, simple computers, and materials with interesting properties [2-5]. Nevertheless, the ability of DNA-guided self-assembly to assemble defect-free and scalable nanostructures is unknown. Thus, the effects of noise on the self-assembly should be characterized. It is unclear at what point randomness overcomes the designed energetic interactions that produce the assembly in a bottom-up way. Inevitably, as the size and complexity of the assembly grows, randomness will have a larger effect on the system. Randomness in the assemblies is mainly a product of kinetic effects, that is, the random collision of component parts while diffusing in the reaction mixture.

1.2 Thesis Statement

To test how variations in probability of bond formations between particles affect their ability to assemble tiling structures with average degree four.

1.3 Approach

The thesis was an attempt to simulate self-assembly of 2-dimensional tiling as shown by Winfree et al. [9]. Primarily, the project focused on generating A-B tiling structure using variations in probability of bond formation in the constituent particles of self-assembly systems. It also tried to simulate a result for Penrose tiling [5] that a large difference in bond probability may produce more perfect tiling structures.
Simulations were carried out for three types of systems: System 1, System 2 and System 3. System 1 contained identical particles with two pairs of different types of bonds – bond type 0 and bond type 1. Only the same types of bond between two particles were allowed to connect with other particles and aggregates. Each bond type was assigned a probability of bond formation. A higher probability meant that the bond was more likely to be formed. This system was used to simulate the original A-B tile structure proposed by Adleman [1]. System 2 contained identical particles with four different types of bonds – bond type 0, bond type 1, bond type 2, and bond type 3. Bond types 0 and 2 were allowed to combine with each other and each was assigned the same probability of bond formation. Similarly bond types 1 and 3 were allowed to combine with each other and each was assigned the same probability of bond formation. System 3 consisted of four different types of particles based on the number and types of bonds. This system was used to simulate self-assembly using seed tiles. Two different simulations were carried with each system: first with fixed probability bond pairs and the second with variations in probability pairs by increasing them in increments of 0.1 from 0 to 1.

It was observed that increasing the gap between the pair of probabilities increased the average degree of particles in the system and the size of the final aggregates formed. The average degree also increased when both the probabilities increased. We concluded by noting that the increase in average degree and the increase in final aggregates’ size was analogous to a smoother tiling. Since the average degree of particles did not attain the value of 4, perfect tiling was not observed.
1.4 Organization of this Thesis

The thesis is organized into four more sections: Background, Methodology, Results, and Conclusions. The code used for the thesis has been attached as an Appendix. The Background section discusses previous related work on self-assembly of particles that this thesis was based on. The Methodology section described the construction and implementation of the experiments for the thesis. The Results and Conclusion section highlighted the data generated by the experiments, what was inferred from them, and future work that could be carried out to improve the finding of the thesis project.
2. BACKGROUND

2.1 Self-assembly

Self-assembly is the process of generating large structures through the random collision of the smaller structures that collide to attain lower free energy. It is an important process for biomolecular computing and the fabrication of nanomaterials. Research in the field of self-assembly is highly interdisciplinary with researcher from backgrounds in biology, material science, mathematics, physics and computer science. In this thesis, we have primarily used the self-assembly research in mathematics and material science as the background.

2.1.1 Wang tiles

In 1961, H. Wang proposed the concept of infinite tiling in a plane using tiles with differently shaded edges [6]. Rules for tiling were based on the color of the edges. For example the figure below categorizes planar tiling through a set of eight different tiles. A tile can only attach to another tile through a similarly colored edge. The Wang tiles are a class of formal systems and have computational power equivalent to a Turing machine.

![Figure 2.1: 8 Wang tiles that can tile a plane. A sample of the tiling is to the right](http://research.microsoft.com/en-us/um/people/cohen/wangfinal.pdf)
2.1.2 Self-assembly and biomolecular computing

2-dimensional self-assembly of biomolecules have been implemented to generate simple tiles [7, 9]. Work has also been carried out to show that self-assembly of biomolecules like DNA can be used for computational purposes [1, 8]. While the computations are significantly slower than those carried out by silicon based computers, the potential of carrying out massive parallel computations using biomolecular computing has made this from of computing an interesting research area.

Figure 2.2: The logical structure of a 2D lattice using A-B tiling

---

2.1.3 Random energy model and probabilistic self-assembly

Self-assembly, essentially a form of cellular automata, has been determined to be equivalent to other systems in statistical mechanics such as Ising models and directed percolations [3, 4]. Self-assembly models using the theory of Wang tiles and random energy models have been proposed. One such model proposed was self-assembly using two tiles that in turn formed eight different types of Penrose tiles [5]. The Penrose tiles bind with each other under certain constraints to generate smooth planar tiling.

![Figure 2.3: System containing 8 different types of Penrose tiles that can be used to generate a perfect tiling](image)

The proposed tiling would be achieved through the variation of probabilities of bonding at the various edges of the Penrose tiles, and also depend on the tiling state of the aggregates (Figure 2.4).

Figure 2.4: Tiling generated using a cluster of 10 tiles (a) made up of the 8 different tiles in Figure 2.3

---

3. METHODOLOGY

3.1 System design

Off-lattice simulations of the self-assembly of particles were carried out with off-lattice Monte Carlo simulations. The system for the simulation consisted of randomly generated particles. Each particle was free to form bonds with other particles under certain rules. The rules enforced a least distance for binding between two particles. In terms of the types of particles present, three types of systems were simulated – System 1, where each particle had 2 pairs of different types of bonds; System 2, where each particle had 4 different bonds; and System 3, where there were 4 different kinds of particles.

3.1.1 Features of the simulation

Each system was a cube containing particles of negligible size. Particles in each system followed rules to bind with other particles and form aggregates. Aggregates could form bonds with other aggregates and particles. At the time of generation, each particle was randomly assigned its x- and y- coordinates. This allowed each particle to diffuse freely in the system. In each “diffusion step”, every particle moved in incremental distance of 1 unit in either x- or y- direction. Consequently, each aggregate also moved in the system while maintaining the relative distance between its constituent particles. By generating the final aggregates in equilibrium from individual particles in a bottom up approach, the design of the simulation incorporates Monte Carlo simulation methods.

The parameters that could be varied in the system were the number of particles in the system, the dimensions of the system and the minimum distance required for binding. All three parameters controlled the concentration of particles in the system.
3.1.2 Variations in types of particles in system

The three systems were composed of different types of particles. System 1 contained identical particles with two pairs of different types of bonds – bond type 0 and bond type 1. Only similar bonds between two particles could “connect” to bind two particles or aggregates. Each bond type was assigned a probability of bond formation. A higher probability meant that the bond was more likely to be formed. The probability of formation of a bond was used to control the growth of aggregates in size and shape. This system was constructed to simulate a simple A-B tiling self-assembly [1].

Figure 3.1: Particles in System 1. The particles are identical and 2 pairs of different bonds.

System 2 consisted of identical particles with four types of bonds – bond type 0, bond type 1, bond type 2 and bond type 3. Bond type 0 could only connect to bond type 2, and both bond types were assigned the same probability of bond formation. Similarly, bond type 1 could only connect to bond type 3, and both bond types were assigned the same probability of bond formation. This system was created to simulate the DNA tiling
in the origami system described by Winfree et al. [9] while restricting the number of bonds each particle could form.

Figure 3.3: Particles in System 2. The particles are identical and have four different bonds.

System 3 consisted of four different types of particles. Particle type 1 had two bonds: a type 0 bond and a type 1 bond. Particle type 2 had three bonds: a type 1 bond and two type 0 bonds. Particle type 3 had three bonds: a type 0 bond and two type 1 bonds. Particle type 4 had four bonds: two type 0 bonds and two type 1 bonds. A bond could only connect with another bond of its own type. This system was constructed to simulate the assembly of particles in a grid using the concept of seed tiles [7].
3.2 Implementation

The simulations were implemented using C++ programming. Graph libraries from the Boost C++ Library [10, 11] were used to store and manipulate the graph structures in the programs. These were used to plot graphs to visually analyze the self-assemblies. Each of the three systems was implemented in two types of simulations. The first type of simulation was carried out with individual pairs of horizontal and vertical bond probabilities. Graphs of the final aggregates after the completion of all diffusion steps were printed out in Graphviz readable .dot files [12]. Graphs were generated from these files using the NEATO utility in Graphviz [13]. In the second type of simulation, particles were assigned bond probabilities from a matrix of probabilities. In the matrix the
bond probabilities were incremented by 0.1 from 0 to 1. Two particles joined to form an edge. The degree of a particle was defined as the number of edges adjacent to the particle. After the simulation for each probability pair, the average degree of a particle in the system was calculated. The average degree of particles in each system was plotted for all the probability pairs of bond formation using Gnuplot 4.2 [14].

3.2.1 Simulations with one pair of vertical-horizontal bond probabilities

In this simulation the total number of diffusion steps, the dimensions of the system, the number of particles, the minimum distance for binding, and the bond probabilities in particles were first defined. After running tests with different values for the parameters, 10,000, 10 units x 10 units x 10 units, 100, and 1 unit were determined to be respective stable parametric values for the total number of diffusion steps, the dimensions of the system, the number of particles, and the minimum distance for binding for the simulations. Particles were then generated and assigned bonds based on the specifications of the system. At the time of generation, each particle was assigned random double values for x- and y- coordinates within the dimensions of the system.

At each diffusion step the position of every particle and aggregate was changed by a unit distance, which is the minimum distance for bonding. Every particle and aggregate was treated as a “component” of the system. Essentially each particle is a vertex and each component is a connected graph of these vertices. For each component in the system, the simulation iterated through constituent vertices. For each vertex the simulation iterated through vertices in other components and checked if the two vertices could form a bond.

The checking process consisted of:
1. Calculating whether the two vertices were within minimum binding distance.

2. Determining if the two particles contained bonds that were allowed to bind.

3. Comparing the probability of bond formation to a random number between 0 and
1. If the probability of bond formation was greater than the random number
   generated then the two particles were connected through their respective bonds.

   Once the particles were connected, the available bonds within the two particles were
   adjusted to reflect that each particle had used the corresponding bond. The components
   of the two particles were joined to form a new single component. This was
   accomplished by doing a union of the graphs of each component to generate a new
   connected graph.

   At the end of all diffusion steps, each remaining component of the system was printed
   out in .dot format to be generated as a visual graph using Graphviz NEATO.

3.2.2 Simulations with vertical-horizontal bond probability matrix

   Simulations with bond probability matrices were implemented similarly to the
   simulations with one pair of bond probabilities. At the beginning of the simulations, the
   dimensions the total number of diffusion steps, the dimensions of the system, the number
   of particles, and the minimum distance for binding were defined similarly to the first type
   of simulations. Each of the two bonding probabilities was incremented by 0.1 from 0 to 1.
   For each pair of the probability of bond formation, diffusion steps were carried out
   according to the method described in Section 3.2.1. The average degree of particles in
   each system was calculated for each bond probability pair and printed out at the end of all
diffusion steps. The printed results were used to plot 3-dimensional surface graphs of average degree with the two probabilities using Gnuplot [14].
4. RESULTS

Based on the two forms of implementations of the simulations, two types of results were generated for each system. For each system, visual graphs were generated for single bond probability pairs. For each system, surface plots for average degree of particles vs. bond probability pairs were generated.

4.1 Data

For all simulations, the number of diffusion steps was set to 10,000, the minimum distance for bond formation was set to 1 unit, and the dimensions of the system were set to 10 units x 10 units x 10 units. Each simulation was carried out five times, and the arithmetic means of the average degree of particles from the five simulations were used as data.

The aggregates generated by the first implementation of simulation of System 1 with bond probabilities 0.1 and 0.9 were one large cluster of particles and rest small clusters (Figure 4.1). The large aggregate generated was a group of smaller clusters joined by linear structures. Similar aggregates were generated for other bond probability pairs if one of the probabilities was significantly greater than the other probability. For simulations where both bond probabilities were small numerous small clusters were generated as final aggregates, while for simulations where both bond probabilities were large one large final aggregate with less than five smaller aggregates was generated.
Figure 4.1: Final aggregates in System 1 for bond probabilities 0.1 and 0.9

Figure 4.2: Final aggregates in System 1 for bond probabilities 0.5 and 0.5
The second implementation of simulation of System 1 generated a surface plot where average degree of particles in the system increased smoothly as the bond probability pairs increased (Figure 4.2). There was a definite increase in average degree when both bond probabilities increased; although it did not attain the ideal value of 4 even when both bond probabilities were 1.
The aggregates generated by the first implementation of simulation of System 2 with bond probabilities 0.1 and 0.9 were multiple clusters of varying sizes (Figure 4.3). There was one large aggregate along with a group of smaller clusters. The large aggregate too appeared to be a group of small clusters joined by linear structures. Linear substructures were observed in the smaller clusters too. Similar aggregates were generated for other bond probability pairs if one of the probabilities was significantly greater than the other probability. For simulations where both bond probabilities were small numerous clusters of smaller sizes were generated as final aggregates, while for simulations where both bond probabilities were large one large final aggregate with less with five smaller aggregates was generated.
Figure 4.5: Final aggregates in System 2 for bond probabilities 0.1 and 0.9

Figure 4.6: Final aggregates in System 2 for bond probabilities 0.5 and 0.5
Figure 4.7: Final aggregates in System 2 for bond probabilities 1 and 1

The second implementation of simulation of System 2 generated a surface plot where average degree of particles in the system increased as the bond probability pairs increased (Figure 4.4). There was a definite increase in average degree when both bond probabilities increased. However, the increase in average degree was not as smooth as for System 1, and the maximum average degree reached (when both bond probabilities were 1) was lower than that for System 1.
The aggregates generated by the first implementation of simulation of System 3 with bond probabilities 0.1 and 0.9 were very different from the ones generated for the previous two systems. Many discrete clusters were generated with each particle forming few bonds (Figure 4.5). Many of the aggregates were linear and many others had linear or ring sub-structures. Similar aggregates were generated for other bond probability pairs if one of the probabilities was significantly greater than the other probability. For simulations where both bond probabilities were small numerous clusters of smaller sizes were generated as final aggregates, while for simulations where both bond probabilities were large one large final aggregate along with a two to three small aggregates with less than ten particles were generated.
Figure 4.9: Final aggregates in System 3 for bond probabilities 0.1 and 0.9

Figure 4.10: Final aggregates in System 3 for bond probabilities 0.5 and 0.5
The second implementation of simulation of System 3 generated a surface plot where average degree of particles in the system increased as the bond probability pairs increased (Figure 4.6). There was a definite increase in average degree when both bond probabilities increased. The surface plot for System 3 was different than the surface plot for the other systems in respect the slope of the surface. The average degree was very low for small values of both probabilities, and in general lower than corresponding average degree values for System 1 and System 2. The maximum value attained by the average degree (when both bond probabilities were 1) was less than that of maximum value for System 2.
The graphs generated by all systems for the first simulation with bond probabilities 0 and 1 were of aggregates that were either linear or rings. The linear structures were generally very long and the rings were generally composed of two to ten particles. Below is a graph generated for the first simulation of System 1 with bond probability pair of 0 and 1:
4.2 Analysis

The data from the simulations of each system highlighted interesting results. Simulations of all systems confirmed that the average degree of particles in the systems increased as the bond probabilities of the particles increased. Essentially the average degree of particles was greatest in systems where the particle had both bond probabilities 1. However, the average degree was significantly below 4 even when both bond probabilities of the constituent particles were set to 1. The fact that the maximum average degree attained is below 4 signifies that the particles in the systems are not able to use all their bonds.

Looking at the graphs generated, we notice the large number of particles in larger aggregates that have only formed two or three bonds. Although there are a large number of such particles, with unused bonds, they cannot bind with each other because of the inherent rigidity of the aggregate components in the simulations. In the simulations, the
relative distance between all particles in a component is maintained when the component moves or binds. This restricts the particles from binding with the other particles within the same component if they aren’t within the minimum distance of binding. In reality, the bonds between particles are flexible, even breakable. The flexibility and dissociability of bonds allows particles to rotate about their bonds and also allows the bonds to stretch. With their bonds allowed to bend, particles within components can move more freely. Consequently, they can bind with other particles within their parent structure. Hence the simulations do not accurately depict reality.

Comparing System 1 with System 2, we observe that there are fewer clusters formed in System 1. Moreover the clusters in System 1 are categorized by the presence of one large aggregate with the rest small aggregates, while in System 2 there are more mid-sized clusters. This can be explained by the binding rules imposed on the particles of the two systems. In System 1 each bond in a particle has the choice to bind with one of two bonds in other particles, whereas in System 2 each bond in a particle can only bind with one bond in another particle. Due to these reasons the probability of chances of bond formation are higher in System 1. The rules of binding in the two systems coupled with the absence of bond stretching features in the simulation cause the average degree of System 2 to be significantly less than the average degree of System 1.

System 3 gives some very interesting results. While the aim of constructing the system was to be able to generate smooth A-B tiling using mechanisms similar to ones induced by seed tiles, the simulations did not give conforming results. The aggregates formed at the end of all diffusion steps in System 3 did not resemble the smooth A-B tiling. This was primarily because the particles in System 3 bonded with unintended
particles. At the time of construction of the particles, particle type 2 was designed to only bond with particles of type 1 and 3. Similarly particles of type 1 and particles of type 3 were designed to form bonds with their own kind of particles, and particles of type 2 and type 4. However as observed in the simulation data, the unintended bonding of particles took place. Because of a larger number of bonds, particles of type 4 had the inherent advantage to form more bonds. Therefore in the simulations, the particles of type 4 formed the most number of bonds, even binding with particles of type 2. Implementation of bond flexibility and better designs of the constituent particles of System 3 may have solved this problem.

Hence we see that while the simulations were somewhat inaccurate due to the absence of some geometric constraints, their average degrees increased with increase in probability of bond formations.
5. CONCLUSION

5.1 Summary

In conclusion, we observed that the simulations of the systems produced larger aggregates with increase in the difference between the bond probability pairs in the particles, and with increase in bond probabilities in general. Larger aggregates also meant that more bonds were formed between the particles in the system, and hence an increase in the average degree of the particles. The increase in average degree and aggregate size also meant that structures being formed by the particles were more getting more similar to tiling structures where each particle forms four bonds. However, the average degree of particles in all the simulations remained below four even for highest bond probabilities. This was because of the absence of geometrical constraints in the simulations, like bond folding and rotation of particles about their bonds.

5.2 Contributions

The simulations were carried out using off-lattice techniques to mimic the diffusion of particles in physical space. Although the inclusion of some additional features in the simulation would increase its accuracy, the simulation was helpful in verifying that the average degree and size of aggregates in systems increase with the increase in difference between the bond probabilities of constituent particles, and with increase in the probabilities of bond formation.
5.3 Future Work

The project generated a few interesting ideas, which could not be pursued due to lack of time. To make the simulations more realistic physical factors such as bond angles in particles and size-dependent rate of diffusion could be incorporated into the model. The bond angles would restrict the formation of bonds between two particles if either of the particles was crowded with other neighbors. A size-dependent rate of diffusion in the simulation would incorporate the fact that rate of diffusion of a particle is a function inversely proportional to the mass of the particle. The stretching and folding of bonds could also be incorporated to allow particles within the same component to bind with each other. The simulations could also be carried out in three-dimensional space.

Moreover, further research into optimal design of the particles could be carried for generation of required structures from particles. This would make the simulations useful to carry out the design of nano-structures in real life.
6. REFERENCES


7. APPENDIX A: CODE

7.1 Bond

7.1.1 bond.h

#ifndef BOND_H
#define BOND_H

#include <iostream>

class Bond {
  public:
    Bond();
    ~Bond();
    Bond(int t);
    int get_type() const;
  
  private:
    int type;
  
};

#endif

7.1.2 bond.cpp

#include "bond.h"

Bond::Bond() //Gives the type of bond
{  
type = 0;
}

Bond::Bond(int t) //Instantiates the type of bond to t
{  
type = t;
}

int Bond::get_type() const //Returns the type of bond
{  
return type;
}

Bond::~Bond()
{  
  std::cout << "Exiting Bond" << std::endl;
}
7.2 Particle

7.2.1 particle.h

#ifndef PARTICLE_H
#define PARTICLE_H

#include <vector>
#include <iostream>
#include "bond.h"

class Particle {

public:
    Particle();
    Particle(double t, double u, double p1, double p2);
    ~Particle();
    double get_x() const;
    double get_y() const;
    int get_bond_size() const;
    int get_fill_size() const;
    int get_bond(int b) const;
    bool get_fill(int f) const;
    void set_x(double xx);
    void set_y(double yy);
    void set_prob1(double p1);
    void set_prob2(double p2);
    std::vector<int> get_Bonds() const;
    std::vector<bool> get_Fill() const;
    void set_bond(int k);
    void fill_bond(int h);
    void make_fill();
    double calc_distance(Particle *p);
    bool bind(Particle *p);
    void displayBonds();
    bool calc_angle(Particle *p);

private:
    std::vector<int> bonds;
    std::vector<bool> fill;
    double x;
    double y;
    double prob1;
    double prob2;

};
#endif

7.2.2 particle.cpp for System 1 and System 3

#include <math.h>
#include "particle.h"
#include "rand.h"
#include <vector>
Particle::Particle()
{
    x = 0;
    y = 0;
    prob1 = 1;
    prob2 = 1;
}

Particle::Particle(double t, double u, double p1, double p2)
{
    x = t;
    y = u;
    prob1 = p1;
    prob2 = p2;
}

double Particle::get_x() const
{
    return x;
}

double Particle::get_y() const
{
    return y;
}

int Particle::get_bond_size() const
{
    return bonds.size();
}

int Particle::get_fill_size() const
{
    return fill.size();
}

void Particle::set_x(double xx)
{
    x += xx;
}

void Particle::set_y(double yy)
{
    y += yy;
}

void Particle::set_prob1(double p1)
{
    prob1 = p1;
}

void Particle::set_prob2(double p2)
{
    prob2 = p2;
}

std::vector<int> Particle::get_Bonds() const
{
    return bonds;
}

std::vector<bool> Particle::get_Fill() const
{
return fill;
}

void Particle::set_bond(int i)
{
    bonds.push_back(i);
    fill.push_back(false);
}

void Particle::make_fill()
{
    for (std::vector<int>::iterator it = bonds.begin(); it!=bonds.end(); ++it) {
        fill.push_back(false);
    }
    //std::cout << "bonds " << bonds.size() << " fill " << fill.size() << std::endl;
}

void Particle::fill_bond(int i)
{
    fill[i] = true;
}

double Particle::calc_distance(Particle *p)
{
    return sqrt(pow((x - p->get_x()),2.0) + pow((y - p->get_y()),2.0));
}

bool Particle::calc_angle(Particle *p)
{
    float tan_angle = fabs((x - p->get_x())/(y - p->get_y())); // tan of angle between the 2 particles
    if((tan_angle >= (1/sqrt(3))) && (tan_angle <= (sqrt(3)))) // angle is between 30 and 60 degrees
        return true;
    else
        return false;
}

void Particle::displayBonds()
{
    for (std::vector<int>::iterator it = bonds.begin(); it!=bonds.end(); ++it) { //for each bond in the bond vector of this particle
        std::cout << "BONDS: " << *it << std::endl; //print out the type of bond as the int bond type stored at the current index of the bond vector
    }
}

bool Particle::bind(Particle *p)
{
    std::vector<int> b = p->get_Bonds();
    std::vector<bool> f = p->get_Fill();
    rand_seed();
    double test1 = rand_double(1);
    double test2 = rand_double(1);
    int countFirst = 0;
    int countSecond = 0;
    }
//std::cout << "O size " << bonds.size() << ' ' << fill.size() << std::endl;
//std::cout << "P size " << b.size() << ' ' << f.size() << std::endl;
for (std::vector<int>::iterator it = bonds.begin(); it!=bonds.end(); ++it) {
    countSecond = 0;
    for (std::vector<int>::iterator itt = b.begin(); itt!=b.end(); ++itt) {
        std::cout << "BONDS: " << *it << ' ' << *itt << std::endl;
        std::cout << "CF " << countFirst << ' ' << countSecond << std::endl;
        if((*it == *itt) && (*itt != 4) && (!fill.at(countFirst)) && (!p->get_Fill()).at(countSecond)) {
            if ((*it == 0) && (prob1 > test1)){
                *it = 4;
                *itt = 4;
                fill_bond(countFirst);
                p->fill_bond(countSecond);
                return true;
            }
            if ((*it == 1) && (prob2 > test2)){
                *it = 4;
                *itt = 4;
                fill_bond(countFirst);
                p->fill_bond(countSecond);
                return true;
            }
        }
        countSecond++;
    }
    countFirst++;
}
return false;
}
}
Particle::~Particle()
{
    bonds.clear();
    fill.clear();
}

7.2.3 particle.cpp for System 2

#include <math.h>
#include "particle.h"
#include "rand.h"
#include <vector>

Particle::Particle()
{
    x = 0;
    y = 0;
    prob1 = 1;
    prob2 = 1;
}
Particle::Particle(double t, double u, double p1, double p2)
{
    x = t;
    y = u;
    prob1 = p1;
    prob2 = p2;
}

double Particle::get_x() const
{
    return x;
}

double Particle::get_y() const
{
    return y;
}

int Particle::get_bond_size() const
{
    return bonds.size();
}

int Particle::get_fill_size() const
{
    return fill.size();
}

void Particle::set_x(double xx)
{
    x += xx;
}

void Particle::set_y(double yy)
{
    y += yy;
}

void Particle::set_prob1(double p1)
{
    prob1 = p1;
}

void Particle::set_prob2(double p2)
{
    prob2 = p2;
}

std::vector<int> Particle::get_Bonds() const
{
    return bonds;
}

std::vector<bool> Particle::get_Fill() const
{
    return fill;
}

void Particle::set_bond(int i)
{
    bonds.push_back(i);
    fill.push_back(false);
}
void Particle::make_fill()
{
    for (std::vector<int>::iterator it = bonds.begin(); it!=bonds.end(); ++it) {
        fill.push_back(false);
    }
    //std::cout << "bonds " << bonds.size() << " fill " << fill.size() << std::endl;
}

void Particle::fill_bond(int i)
{
    fill[i] = true;
}

double Particle::calc_distance(Particle *p)
{
    return sqrt(pow((x - p->get_x()),2.0) + pow((y - p->get_y()),2.0));
}

bool Particle::calc_angle(Particle *p)
{
    float tan_angle = fabs((x - p->get_x())/(y - p->get_y())); // tan of angle between the 2 particles
    if((tan_angle >= (1/sqrt(3))) && (tan_angle <= (sqrt(3)))) // angle is between 30 and 60 degrees
        return true;
    else
        return false;
}

void Particle::displayBonds()
{
    for (std::vector<int>::iterator it = bonds.begin(); it!=bonds.end(); ++it) //for each bond in the bond vector of this particle
        std::cout << "BONDS: " << *it << std::endl; //print out the type of bond as the int bond type stored at the current index of the bond vector
}

bool Particle::bind(Particle *p)
{
    std::vector<int> b = p->get_Bonds();
    std::vector<bool> f = p->get_Fill();
    rand_seed();
    double test1 = rand_double(1);
    double test2 = rand_double(1);
    int countFirst = 0;
    int countSecond = 0;

    //std::cout << "O size " << bonds.size() << " " << fill.size() << std::endl;
    //std::cout << "P size " << b.size() << " " << f.size() << std::endl;
    for (std::vector<int>::iterator it = bonds.begin(); it!=bonds.end(); ++it) {
        countSecond = 0;
        etc...
for (std::vector<int>::iterator itt = b.begin(); itt != b.end(); ++itt) {
    //std::cout << "BONDS: " << *it << " " << *itt << std::endl;
    //std::cout << "CF " << countFirst << " CS " << countSecond
    if((*it != 4) && (*itt != 4) && 
        (!(fill.at(countFirst))) && (!(p->get_Fill()).at(countSecond))) {
        if (((*it == 0) && (*itt == 2)) && (prob1 > test1)){
            *it = 4;  
            *itt = 4; 
            fill_bond(countFirst); 
            p->fill_bond(countSecond); 
            return true; 
        } 
        if (((*it == 2) && (*itt == 0)) && (prob1 > test1)){
            *it = 4;  
            *itt = 4; 
            fill_bond(countFirst); 
            p->fill_bond(countSecond); 
            return true; 
        } 
        if (((*it == 1) && (*itt == 3)) && (prob2 > test2)){
            *it = 4;  
            *itt = 4; 
            fill_bond(countFirst); 
            p->fill_bond(countSecond); 
            return true; 
        } 
        if (((*it == 3) && (*itt == 1)) && (prob1 > test1)){
            *it = 4;  
            *itt = 4; 
            fill_bond(countFirst); 
            p->fill_bond(countSecond); 
            return true; 
        } 
    }
    countSecond++;
}
}
 Particle::~Particle()
{
    bonds.clear();
    fill.clear();
}
7.3 Random

7.3.1 random.h

```cpp
#include <cstdlib>
#include <ctime>

using namespace std;

void rand_seed();
int rand_int(int a, int b);
double rand_double();
double rand_double(double num);
```

7.3.2 random.cpp

```cpp
#include "rand.h"

void rand_seed()
{
    int seed = static_cast<int>(time(0));
    srand(seed);
}

int rand_int(int a, int b)
{
    return a + rand() % (b - a + 1);
}

double rand_double()
{
    return (double) rand()/RAND_MAX;
}

double rand_double(double num)
{
    return ((double) rand()/RAND_MAX) * num;
}
```
7.4 System

All the files below are system.cpp.

7.4.1 System 1 with probability pairs

```cpp
#include <string>
#include <iostream>
#include <fstream>
#include <boost/graph/adjacency_list.hpp>
#include <cassert>
#include <boost/graph/graph_utility.hpp>
#include <boost/pending/disjoint_sets.hpp>
#include <boost/graph/incremental_components.hpp>
#include "assembly.h"

int main(int argc, char* argv[]) {
    using namespace boost;
    typedef adjacency_list<vecS, vecS, undirectedS> Graph;
    typedef graph_traits<Graph>::vertex_descriptor Vertex;
    typedef graph_traits<Graph>::vertices_size_type size_type;
    register int i,j,k,l;
    int b, p;
    double x, y;
    time_t ltime; /* calendar time */
    std::ofstream testfile;
    string testname = string(argv[1]);
    cout << "How many iterations ?" << std::endl;
    int iterations;
    std::cin >> iterations;
    cout << "How many diffusion steps per iterations of alpha?" << std::endl;
    int converge;
    std::cin >> converge;
    cout << "How many particles?" << std::endl;
    int n_particles;
    std::cin >> n_particles;
    int n_bonds = n_particles * 4;
    double alpha = 2.0*n_particles;
    int n_bondtypes = 2;
    std::cout << "Enter dimension?" << std::endl;
    double dimension;
    std::cin >> dimension;
    std::cout << "Enter distance for binding?" << std::endl;
    double distance;
```
```
std::cin >> distance;
float prob1;
float prob2;
std::cout << "Enter horizontal probability (prob1):" << std::endl;
std::cin >> prob1;
std::cout << "Enter vertical probability (prob2):" << std::endl;
std::cin >> prob2;

/*
  int tests = 1;
  int iterations = 10000;
  int n_particles = 100;
  int n_bondtypes = 2;
  double dimension = 10;
  double distance = 1;
*/

double total_edges = ((double) n_bonds/2.0);

rand_seed();

int intermediate = 0;
double sum_square = 0.0;

double avg_components = 0.0;
double degree = 0.0;
int n_edges = 0;
int n_components = 0;
Particle *part;

for(l = 0; l < iterations; l++) {
    Graph g(n_particles); //Check the Boost library for this one.
    std::vector<Particle*> particles; //instantiate a vector of Particle class called "particles"

    for (i = 0; i < n_particles; i++) { //for each of the "n_particles" number of particles do the following:
        //Generate random x,y co-ordinates for a particle
        x = rand_double(dimension);
y = rand_double(dimension);
part = new Particle(x, y, prob1, prob2); //instantiate the particle with the co-ordinates generated above
part->set_bond(0); //set bond b to particle p
part->set_bond(1);
part->set_bond(0);
part->set_bond(1);
particles.push_back(part); //push the particle "part" into the vector "particles"
}

std::vector<size_type> rank(num_vertices(g));
std::vector<Vertex> parent(num_vertices(g));
typedef size_type* Rank;
typedef Vertex* Parent;
disjoint_sets<Rank, Parent> ds(&rank[0], &parent[0]);
initialize_incremental_components(g, ds);
incremental_components(g, ds);
typedef component_index<unsigned int> Components;

for (k = 0; k < converge; k++) {
```
Components components(&parent[0], &parent[0] +
parent.size());

for (Components::size_type c = 0; c < components.size();
  ++c) {
  Components::value_type::iterator
  ii = components[c].begin(),
  iiend = components[c].end();
  int direction = rand_int(0, 1);
  for ( ; ii != iiend; ++ii) {
    if(direction == 0)
      particles[*ii]->set_x(distance);
    else
      particles[*ii]->set_y(distance);
  }
}

for (Components::size_type c = 0; c < components.size();
  ++c) {
  Components::value_type::iterator
  ii = components[c].begin(),
  iiend = components[c].end();
  for ( ; ii != iiend; ++ii) {
    for (Components::size_type cc = c; cc <
        components.size(); ++cc) {
      Components::value_type::iterator
      jj = components[cc].begin(),
      jjend = components[cc].end();
      for ( ; jj != jjend; ++jj) {
        if(*ii != *jj && calc_distance(particles[*ii]) < distance) &&
        bind(particles[*ii]) {
          // edge(*ii, *jj, g);
          if(!edge(*ii, *jj, g).second){
            add_edge(*ii, *jj, g);
            ds.union_set(*ii, *jj);
          }
          cout << l << " Edge added: " << *ii << " " << *jj << std::endl;
        }
      }
    }
  }
}

n_components = components.size();
incremental_components(g, ds);

intermediate = num_edges(g);
n_edges += intermediate;
sum_square += (double) intermediate * intermediate;
avg_components += n_components;
graph_traits<Graph>::vertex_iterator pi, pi_end;
for(tie(pi, pi_end) = vertices(g); pi != pi_end; ++pi) {
    degree += out_degree(*pi, g);
}

//delete components;
for(i = 0; i < n_particles; i++)
    delete particles[i];

if(l == (iterations - 1)) {
    std::ofstream fout("figs/component.dot");
    fout << "graph A {\n"
        << " rankdir=LR\n"
        << " size="56,56"\n"
        << " ratio="filled"\n"
        << " graph[fontsize="56"]\n"
        << " edge[style="bold"]\n"
        << " node[shape="circle"]\n"
    
    graph_traits<Graph>::edge_iterator eiter, eiter_end;
    for (tie(eiter, eiter_end) = edges(g); eiter != eiter_end;
        ++eiter) {
        fout << source(*eiter, g) << " -- " << target(*eiter, g) << " [\n"
            << "[color="black" << "\n"
        }
    fout << "}\n"
    }

    std::cout << "DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD"
        << std::endl;

double average_edges = ((double) n_edges/iterations);
    double fraction = average_edges/total_edges;
    double fraction_square = sum_square/(total_edges*total_edges);
    double error = sqrt(1.0/((double) iterations))*sqrt((1.0/((double)
        iterations))*(fraction_square - iterations*fraction*fraction));

    std::cout << "num_particles = " << n_particles << " num_bonds = " << n_bonds
        " n_bondtypes = " << n_bondtypes << std::endl;
    std::cout << "alpha = " " alpha " << std::endl;
    std::cout << "The fraction in ISET was: " fraction " std::endl;
    std::cout << "The standard error in ISET was: " error << std::endl;
    std::cout << "The average number of components was: " <<
        (avg_components/((double) iterations)) << std::endl;
    std::cout << "The average degree was: " (degree/((double)
        iterations*n_particles)) << std::endl;

    testfile.open(testname.c_str(), ofstream::app);
    ltime = time(NULL); /* get current cal time */
    string current_time = asctime( localtime(&ltime) ); // timestamp for test
    testfile << "Time of test = " current_time;
    testfile << "Iterations = " iterations << " Diffusion steps = " converge;
    testfile << " Binding Distance = " distance << std::endl;
    testfile << "Dimension = " dimension << " Probl = " prob1 << " Prob2 = " prob2;
    testfile << "Num_particles = " n_particles << " Num_bonds = " n_bonds;
    testfile << " N_bondtypes = " n_bondtypes << std::endl;
    testfile << "Alpha = " alpha << std::endl;
    testfile << "The fraction in ISET was: " fraction << std::endl;
    testfile << "The standard error in ISET was: " error << std::endl;
}
### 7.4.2 System 1 with probability matrix

```cpp
#include <string>
#include <iostream>
#include <fstream>
#include <boost/graph/adjacency_list.hpp>
#include <cassert>
#include <boost/graph/graph_utility.hpp>
#include <boost/pending/disjoint_sets.hpp>
#include <boost/graph/incremental_components.hpp>
#include "assembly.h"

int main(int argc, char* argv[]) {
    using namespace boost;

    std::ofstream iset_file;
    std::ofstream degree_file;
    string iset_filename = string(argv[1]);
    string degree_filename = string(argv[2]);
    float iset_deg_array[121][4] = { 0 };
    int index = 0; //index for iset_deg_array

    cout << "How many iterations ?" << std::endl;
    int iterations;
    std::cin >> iterations;
    cout << "How many diffusion steps per iterations of alpha?" << std::endl;
    int converge;
    std::cin >> converge;
    cout << "How many particles?" << std::endl;
    int n_particles;
    std::cin >> n_particles;
    int n_bonds = n_particles * 4;
    double alpha = 2.0*n_particles;
    int n_bondtypes = 2;
    cout << "Enter dimension?" << std::endl;
    double dimension;
    std::cin >> dimension;
    cout << "Enter distance for binding?" << std::endl;
    double distance;
    std::cin >> distance;

    //initialize each probability to 0
    float prob1;
```
float prob2;

/*
 * int tests = 1;
 * int iterations = 10000;
 * int n_particles = 100;
 * int n_bondtypes = 2;
 * double dimension = 10;
 * double distance = 1;
 */

for(prob1 = 0; prob1 < 1.1; prob1 = prob1 + 0.1) {
    for(prob2 = 0; prob2 < 1.1; prob2 = prob2 + 0.1) {
        typedef adjacency_list<vecS, vecS, undirectedS> Graph;
        //boost::graph_traits<G>::vertex_descriptor - The type for vertex
        //representative objects.
        typedef graph_traits<Graph>::vertex_descriptor Vertex;
        //boost::graph_traits<G>::vertices_size_type - The unsigned integer
        //type for number of vertices in the graph.
        typedef graph_traits<Graph>::vertices_size_type size_type;

        register int i, j, k, l;
        int b, p;
        double x, y;

        double total_edges = ((double) n_bonds/2.0);
        rand_seed();

        int intermediate = 0;
        double sum_square = 0.0;

        double avg_components = 0.0;
        double degree = 0.0;
        int n_edges = 0;
        int n_components = 0;
        Particle *part;

        for(l = 0; l < iterations; l++) {
            Graph g(n_particles); //Check the Boost library for this one.
            std::vector<Particle*> particles; //instantiate a vector of
            Particle class called "particles"

            for (i = 0; i < n_particles; i++) { //for each of the
                "n_particles" number of particles do the following:
                //Generate random x,y co-ordinates for a particle
                x = rand_double(dimension);
                y = rand_double(dimension);
                part = new Particle(x, y, prob1, prob2); //instantiate
                the particle with the co-ordinates generated above
                part->set_bond(2); //set bond b to particle p
                part->set_bond(1);
                part->set_bond(2);
                part->set_bond(1);
                particles.push_back(part); //push the particle "part"
                into the vector "particles"
            }

            std::vector<size_type> rank(num_vertices(g));
            std::vector<Vertex> parent(num_vertices(g));
        }
typedef size_type* Rank;
typedef Vertex* Parent;
disjoint_sets<Rank, Parent> ds(&rank[0], &parent[0]);
initialize_incremental_components(g, ds);
incremental_components(g, ds);
typedef component_index<unsigned int> Components;
for (k = 0; k < converge; k++) {
  Components components(&parent[0], &parent[0] + parent.size());
  for (Components::size_type c = 0; c < components.size(); ++c) {
    Components::value_type::iterator ii = components[c].begin(),
      iiend = components[c].end();
    int direction = rand_int(0, 1);
    for (; ii != iiend; ++ii) {
      if (direction == 0)
        particles[*ii]->set_x(distance);
      else
        particles[*ii]->set_y(distance);
    }
  }
  for (Components::size_type c = 0; c < components.size(); ++c) {
    Components::value_type::iterator jj = components[c].begin(),
      jjend = components[c].end();
    for (; jj != jjend; ++jj) {
      if (*ii != *jj) &&
        particles[*ii]->calc_distance(particles[*jj]) < distance)
        if (!edge(*ii, *jj, g).second) {
          add_edge(*ii, *jj, g);
          ds.union_set(*ii, *jj);
        }
    }
  }
  n_components = components.size();
}
incremental_components(g, ds);

intermediate = num_edges(g);
n_edges += intermediate;
sum_square += ((double) intermediate * intermediate);
avg_components += n_components;
graph_traits<Graph>::vertex_iterator pi, pi_end;

for(tie(pi,pi_end) = vertices(g); pi != pi_end; ++pi) {
    degree += out_degree(*pi, g);
}

//delete components;
for(i = 0; i < n_particles; i++)
    delete particles[i];

double average_edges = ((double) n_edges/iterations);
double fraction = average_edges/total_edges;
double fraction_square = sum_square/(total_edges*total_edges);
double error = sqrt(1.0/((double) iterations))*(fraction_square - iterations*fraction*fraction));
double av_degree = (degree/((double) iterations*n_particles));

//set array for iset and average degree
iset_deg_array[index][0] = prob1;
iset_deg_array[index][1] = prob2;
iset_deg_array[index][2] = fraction;
iset_deg_array[index][3] = av_degree;
index++; //increment index
}

degree_file.open(degree_filename.c_str(), ofstream::app);
iset_file.open(iset_filename.c_str(), ofstream::app);
for (index = 0; index < 121; index++){
    degree_file << iset_deg_array[index][0] << "        " << iset_deg_array[index][1] << "        " << iset_deg_array[index][3] << std::endl;
    iset_file << iset_deg_array[index][0] << "        " << iset_deg_array[index][1] << "        " << iset_deg_array[index][2] << std::endl;
}
iset_file.close();
degree_file.close();

return 0;

7.4.3 System 2 with probability pairs
```cpp
#include <boost/pending/disjoint_sets.hpp>
#include <boost/graph/incremental_components.hpp>
#include "assembly.h"

int main(int argc, char* argv[]) {
    using namespace boost;
    typedef adjacency_list<vecS, vecS, undirectedS> Graph;
    typedef graph_traits<Graph>::vertex_descriptor Vertex;
    typedef graph_traits<Graph>::vertex_descriptor Vertex;
    typedef graph_traits<Graph>::vertices_size_type size_type;

    register int i, j, k, l;
    int b, p;
    double x, y;
    time_t ltime; /* calendar time */

    std::ofstream testfile;
    string testname = string(argv[1]);

    cout << "How many iterations ?" << std::endl;
    int iterations;
    std::cin >> iterations;
    cout << "How many diffusion steps per iterations of alpha?" << std::endl;
    int converge;
    std::cin >> converge;
    cout << "How many particles?" << std::endl;
    int n_particles;
    std::cin >> n_particles;
    int n_bonds = n_particles * 4;
    double alpha = 2.0*n_particles;
    int n_bondtypes = 2;
    cout << "Enter dimension?" << std::endl;
    double dimension;
    std::cin >> dimension;
    cout << "Enter distance for binding?" << std::endl;
    double distance;
    std::cin >> distance;
    float prob1;
    float prob2;
    cout << "Enter horizontal probability (prob1):" << std::endl;
    std::cin >> prob1;
    cout << "Enter vertical probability (prob2):" << std::endl;
    std::cin >> prob2;

    double total_edges = ((double) n_bonds/2.0);
    rand_seed();
    int intermediate = 0;
}
```
double sum_square = 0.0;
double avg_components = 0.0;
double degree = 0.0;
int n_edges = 0;
int n_components = 0;
Particle *part;

for(l = 0; l < iterations; l++) {
    Graph g(n_particles); //Check the Boost library for this one.
    Graph g has n_particles (?)
    std::vector<Particle*> particles; //instantiate a vector of
    Particle class called "particles"
    for (i = 0; i < n_particles; i++) { //for each of the
        "n_particles" number of particles do the following:
            //Generate random x,y co-ordinates for a particl e
            x = rand_double(dimension);
y = rand_double(dimension);
          part = new Particle(x, y, prob1, prob2); //instantiate
          the particle with the co-ordinates generated above
          part->set_bond(0); //set bond b to particle p
          part->set_bond(1);
          part->set_bond(2);
          part->set_bond(3);
          particles.push_back(part); //push the particle "part" into
          the vector "particles"
         }
    std::vector<size_type> rank(num_vertices(g));
    std::vector<Vertex> parent(num_vertices(g));
typedef size_type* Rank;
typedef Vertex* Parent;
disjoint_sets<Rank, Parent> ds(&rank[0], &parent[0]);
initialize_incremental_components(g, ds);
incremental_components(g, ds);
typedef component_index<unsigned int> Components;
for (k = 0; k < converge; k++) {
    Components components(&parent[0], &parent[0] +
    parent.size()));
    for (Components::size_type c = 0; c < components.size();
    ++c) {
        Components::value_type::iterator
        ii = components[c].begin(),
        iiend = components[c].end();
        int direction = rand_int(0, 1);
        for ( ; ii != iiend; ++ii) {
            if(direction == 0)
                particles[*ii]->set_x(distance);
            else
                particles[*ii]->set_y(distance);
        }
    }
}

for (Components::size_type c = 0; c < components.size();
    ++c) {
    Components::value_type::iterator
for ( ; ii != iiend; ++ii) { for (Components::size_type cc = c; cc <
components.size(); ++cc) {
    Components::value_type::iterator
    jj = components[cc].begin(),
    jjend = components[cc].end();
    for ( ; jj != jjend; ++jj) {
        if(*ii != *jj && particles[*ii]-
>calc_distance(particles[*jj]) < distance) && particles[*ii]-
>bind(particles[*jj]) {
            //edge(*ii, *jj, 
g);
            if(!edge(*ii, *jj, g).second){
                add_edge(*ii, *jj, g);
                ds.union_set(*ii, *jj);
            }
        }
    }
}
}

n_components = components.size();
incremental_components(g, ds);

intermediate = num_edges(g);
n_edges += intermediate;
sum_square += ((double) intermediate * intermediate);
avg_components += n_components;
graph_traits<Graph>::vertex_iterator pi, pi_end;
for(tie(pi, pi_end) = vertices(g); pi != pi_end;  ++pi) {
    degree += out_degree(*pi, g);
}
//delete components;
for(i = 0; i < n_particles; i++)
    delete particles[i];

if(l == (iterations - 1)) {
    std::ofstream fout("figs/component.dot");
    fout << "graph A \n"
        << " rankdir=LR\n"
        << " size="56,56"\n"
        << " ratio="filled"\n"
        << " graph[fontsize="56"]\n"
        << " edge[style="bold"]\n"
        << " node[shape="circle"]\n";
    graph_traits<Graph>::edge_iterator eiter, eiter_end;
    for (tie(eiter, eiter_end) = edges(g); eiter != eiter_end; ++eiter) {
        fout << source(*eiter, g) << " -- " << target(*eiter, g) <<
"[color="black" << "\n"];\n";"}


    fout << "\n";

}

std::cout << "DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD" << std::endl;

double average_edges = ((double) n_edges/iterations);
double fraction = average_edges/total_edges;
double fraction_square = sum_square/(total_edges*total_edges);
double error = sqrt(1.0/((double) iterations))*sqrt((1.0/((double)
iterations))*(fraction_square - iterations*fraction*fraction));

std::cout << "num_particles = " << n_particles << " num_bonds = " <<
n_bonds << " n_bondtypes = " << n_bondtypes << std::endl;
std::cout << "alpha = " << alpha << std::endl;
std::cout << "The fraction in ISET was: " << fraction << std::endl;
std::cout << "The standard error in ISET was: " << error << std::endl;
std::cout << "The average number of components was: " <<
(avg_components/(double) iterations)) << std::endl;
std::cout << "The average degree was: " << (degree/(double)
iterations*n_particles)) << std::endl;

testfile.open(testname.c_str(), ofstream::app);
ltime=time(NULL); /* get current cal time */
string current_time = asctime( localtime(&ltime) ); // timestamp for
test
testfile << "Time of test = " << current_time;
testfile << "Iterations = " << iterations << " Diffusion steps = " <<
converge << " Binding Distance = " << distance << std::endl;
testfile << "Dimension = " << dimension << " Prob1 = " << prob1 << "
Prob2 = " << prob2 << std::endl;
testfile << "Num_particles = " << n_particles << " Num_bonds = " <<
n_bonds << " N_bondtypes = " << n_bondtypes << std::endl;
testfile << "Alpha = " << alpha << std::endl;
testfile << "The fraction in ISET was: " << fraction << std::endl;
testfile << "The standard error in ISET was: " << error << std::endl;
testfile << "The average number of components was: " <<
(avg_components/(double) iterations)) << std::endl;
testfile << "The average degree was: " << (degree/(double)
iterations*n_particles)) << std::endl;
testfile << "\n";
testfile.close();

std::cout << "DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD" << std::endl;

return 0;

7.4.4 System 2 with probability matrix

#include <string>
#include <iostream>
#include <fstream>
#include <cassert>
#include <boost/graph/adjacency_list.hpp>
#include <cassert>

#include <boost/graph/graph_utility.hpp>
#include <boost/pending/disjoint_sets.hpp>
#include <boost/graph/incremental_components.hpp>
#include "assembly.h"

int main(int argc, char* argv[]) {

    using namespace boost;

    std::ofstream iset_file;
    std::ofstream degree_file;
    string iset_filename = string(argv[1]);
    string degree_filename = string(argv[2]);
    float iset_deg_array[121][4] = { 0 };
    int index = 0; //index for iset_deg_array

    cout << "How many iterations ?" << std::endl;
    int iterations;
    std::cin >> iterations;
    cout << "How many diffusion steps per iterations of alpha?" << std::endl;
    int converge;
    std::cin >> converge;
    std::cout << "How many particles?" << std::endl;
    int n_particles;
    std::cin >> n_particles;
    int n_bonds = n_particles * 4;
    double alpha = 2.0 * n_particles;
    int n_bondtypes = 2;
    std::cout << "Enter dimension?" << std::endl;
    double dimension;
    std::cin >> dimension;
    std::cout << "Enter distance for binding?" << std::endl;
    double distance;
    std::cin >> distance;

    //initialize each probability to 0
    float prob1;
    float prob2;

    for(prob1 = 0; prob1 < 1.1; prob1 = prob1 + 0.1) {
        for(prob2 = 0; prob2 < 1.1; prob2 = prob2 + 0.1) {
            typedef adjacency_list<vecS, vecS, undirectedS> Graph;
            //boost::graph_traits<G>::vertex_descriptor - The type for vertex
            //representative objects.
            typedef graph_traits<Graph>::vertex_descriptor Vertex;
            //boost::graph_traits<G>::vertices_size_type - The unsigned integer
            //type for number of vertices in the graph.
            typedef graph_traits<Graph>::vertices_size_type size_type;

            register int i, j, k, l;
            int b, p;
            double x, y;

            double total_edges = ((double) n_bonds/2.0);
rand_seed();
int intermediate = 0;
double sum_square = 0.0;
double avg_components = 0.0;
double degree = 0.0;
int n_edges = 0;
int n_components = 0;
Particle *part;

for(l = 0; l < iterations; l++) {

    Graph g(n_particles); //Check the Boost library for this one.
    std::vector<Particle*> particles; //instantiate a vector of
    Particle class called "particles"
    for (i = 0; i < n_particles; i++) { //for each of the
        "n_particles" number of particles do the following:
        //Generate random x,y co-ordinates for a particle
        x = rand_double(dimension);
        y = rand_double(dimension);
        part = new Particle(x, y, prob1, prob2); //instantiate
        the particle with the co-ordinates generated above
        part->set_bond(1); //set bond b to particle p
        part->set_bond(2);
        part->set_bond(3);
        part->set_bond(4);
        particles.push_back(part); //push the particle "part"
    }
    std::vector<size_type> rank(num_vertices(g));
    std::vector<Vertex> parent(num_vertices(g));
    typedef size_type* Rank;
    typedef Vertex* Parent;
    disjoint_sets<Rank, Parent> ds(&rank[0], &parent[0]);
    initialize_incremental_components(g, ds);
    incremental_components(g, ds);
    typedef component_index<unsigned int> Components;
    for (k = 0; k < converge; k++) {
        Components components(&parent[0], &parent[0] +
        parent.size());
        for (Components::size_type c = 0; c <
        components.size(); ++c) {
            Components::value_type::iterator
            ii = components[c].begin(),
            iiend = components[c].end();
            int direction = rand_int(0, 1);
            for (; ii != iiend; ++ii) {
                if(direction == 0)
                    particles[*ii]->set_x(distance);
                else
                    particles[*ii]->set_y(distance);
            }
        }
    }
}
for (Components::size_type c = 0; c < components.size(); ++c) {
    Components::value_type::iterator
    ii = components[c].begin(),
    iiend = components[c].end();
    for ( ; ii != iiend; ++ii) {
        for (Components::size_type cc = c; cc < components.size(); ++cc) {
            Components::value_type::iterator
            jj = components[cc].begin(),
            jjend = components[cc].end();
            for ( ; jj != jjend; ++jj) {
                if(*ii != *jj &&
                    particles[*ii]->calc_distance(particles[*jj]) < distance) &&
                    particles[*ii]->bind(particles[*jj]) {
                    //edge(*ii, *jj, g);
                    if(!edge(*ii, *jj, g).second){
                        add_edge(*ii, *jj, g);
                        ds.union_set(*ii, *jj);
                    }
            }
        }
    }
    n_components = components.size();
    incremental_components(g, ds);
}

intermediate = num_edges(g);
n_edges += intermediate;
sum_square += ((double) intermediate * intermediate);
avg_components += n_components;

for(tie(pi, pi_end) = vertices(g); pi != pi_end; ++pi) {
    degree += out_degree(*pi, g);
}

//delete components;
for(i = 0; i < n_particles; i++)
    delete particles[i];

double average_edges = ((double) n_edges/iterations);
double fraction = average_edges/total_edges;
double fraction_square = sum_square/(total_edges*total_edges);
double error = sqrt((1.0/((double) iterations))*(fraction_square - iterations*fraction*fraction));
double av_degree = (degree/((double) iterations*n_particles));
//set array for iset and average degree
iset_deg_array[index][0] = prob1;
iset_deg_array[index][1] = prob2;
iset_deg_array[index][2] = fraction;
iset_deg_array[index][3] = av_degree;
index++; //increment index
}

degree_file.open(degree_filename.c_str(), ofstream::app);
iset_file.open(iset_filename.c_str(), ofstream::app);
for (index = 0; index < 121; index++){
degree_file << iset_deg_array[index][0] << "        " <<
iset_deg_array[index][1] << "        " << iset_deg_array[index][3] <<
std::endl;
iset_file << iset_deg_array[index][0] << "        " <<
iset_deg_array[index][1] << "        " << iset_deg_array[index][2] <<
std::endl;
}iset_file.close();
degree_file.close();

return 0;

7.4.5 System 3 with probability pairs

#include <string>
#include <iostream>
#include <fstream>
#include <boost/graph/adjacency_list.hpp>
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include <boost/algorithm/random.hpp>
#include <boost/pending/disjoint_sets.hpp>
#include <boost/graph/incremental_components.hpp>
#include "assembly.h"

int main(int argc, char* argv[])
{

using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS> Graph;
//boost::graph_traits<Graph>::vertex_descriptor - The type for vertex

representation.

typedef graph_traits<Graph>::vertex_descriptor Vertex;
//boost::graph_traits<Graph>::vertices_size_type - The unsigned integer type for

number of vertices in the graph.

typedef graph_traits<Graph>::vertices_size_type size_type;

register int i,j,k,l;
int b, p;
double x, y;
time_t ltime; /* calendar time */

int n_type1;       //number of particles with 2 bonds (1 type-1 and 1 type-0)
int n_type2;       //number of particles with 3 bonds (1 of type-1 and 2 of

type-0)
int n_type3;       //number of particles with 3 bonds (2 of type-1 and 1 of

...
//the rest of the parciles = 15 - 12 = 3 will have 4 bonds
( 2 type-1 and 2 type-0)

std::ofstream testfile;
string testname = string(argv[1]);

//...
std::vector<Particle*> particles; // instantiate a vector of
Particle class called "particles"

for (i = 0; i < n_particles; i++) { // for each of the
"n_particles" number of particles do the following:

// Generate random x, y co-ordinates for a particle
x = rand_double(dimension);
y = rand_double(dimension);
part = new Particle(x, y, prob1, prob2);  // instantiate
the particle with the co-ordinates generated above
if (i < n_type1) {  // create 1st type of particles with
1 type-1 bond and 1 type-0 bond
    part->set_bond(4);  // set bond b to particle p
    part->set_bond(4);
    part->set_bond(0);
    part->set_bond(1);
    n_bonds = n_bonds + 2;
}
else if ((i < n_type2 + n_type1) && (i >= n_type1)) {  // create 2nd type of particles with
// 1 type-1 bond and 2 type-0 bonds
    part->set_bond(4);
    part->set_bond(1);
    part->set_bond(0);
    part->set_bond(0);
    n_bonds = n_bonds + 2;
}
else if ((i < n_type3 + n_type2 + n_type1) && (i >=
    n_type2 + n_type1)) {  // 3rd type of particles w/ 2 type-1 & 1 type-0 bonds
    part->set_bond(4);
    part->set_bond(1);
    part->set_bond(1);
    part->set_bond(0);
    n_bonds = n_bonds + 3;
}
else {  // the rest of the particles
    part->set_bond(1);
    part->set_bond(0);
    part->set_bond(1);
    part->set_bond(0);
    n_bonds = n_bonds + 4;
}
particles.push_back(part); // push the particle "part" into
the vector "particles"
}
std::vector<size_type> rank(num_vertices(g));
std::vector<Vertex> parent(num_vertices(g));
typedef size_type* Rank;
typedef Vertex* Parent;
disjoint_sets<Rank, Parent> ds(&rank[0], &parent[0]);
initialize_incremental_components(g, ds);
incremental_components(g, ds);
typedef component_index<unsigned int> Components;
for (k = 0; k < converge; k++) {
    Components components(&parent[0], &parent[0] +
    parent.size());
    for (Components::size_type c = 0; c < components.size();
    ++c) {
        Components::value_type::iterator

ii = components[c].begin(),
iend = components[c].end();
int direction = rand_int(0, 1);
for ( ; ii != iiend; ++ii) {
    if(direction == 0)
        particles[*ii]->set_x(distance);
    else
        particles[*ii]->set_y(distance);
}

for (Components::size_type c = 0; c < components.size(); ++c) {
    Components::value_type::iterator
        ii = components[c].begin(),
iend = components[c].end();
    for ( ; ii != iiend; ++ii) {
        for (Components::size_type cc = c; cc < components.size(); ++cc) {
            Components::value_type::iterator
                jj = components[cc].begin(),
jjend = components[cc].end();
            for ( ; jj != jjend; ++jj) {
                if((*ii != *jj)&&(particles[*ii]-
                    >calc_distance(particles[*jj]) < distance) && particles[*ii]-
                    >bind(particles[*jj])) {
                    //edge(*ii, *jj, g);
                    if(!edge(*ii, *jj, g).second){
                        add_edge(*ii, *jj, g);
                        ds.union_set(*ii, *jj);
                        cout << l << "Edge added: " << *ii << " " << *jj << std::endl;
                    }
                }
            }
        }
    }
}

n_components = components.size();
incremental_components(g, ds);

intermediate = num_edges(g);
n_edges += intermediate;
sum_square += ((double) intermediate * intermediate);
avg_components += n_components;
graph_traits<Graph>::vertex_iterator pi, pi_end;
for(tie(pi,pi_end) = vertices(g); pi != pi_end;  ++pi) {
    degree += out_degree(*pi, g);
}

//delete components;
for(i = 0; i < n_particles; i++)
    delete particles[i];
if(l == (iterations - 1)) {
    std::ofstream fout("figs/component.dot");
    fout << "graph A {" << " rankdir=LR" << " size="56,56"\n" << " ratio="filled"\n" << " graph[fontsize="56"]\n" << " edge[style="bold"]\n" " node[shape="circle"]\n";
    graph_traits<Graph>::edge_iterator eiter, eiter_end;
    for (tie(eiter, eiter_end) = edges(g); eiter != eiter_end;
        ++eiter) {
        fout << source(*eiter, g) << " -- " << target(*eiter, g) << "[color="black"]\n" << " [\n";
        fout << "}\n";
    }
}

std::cout << "DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD" << std::endl;

double average_edges = ((double) n_edges/iterations);
double fraction = average_edges/total_edges;
double fraction_square = sum_square/(total_edges*total_edges);
double error = sqrt(1.0/((double) iterations))*(sqrt((1.0/((double) iterations))*(fraction_square - iterations*fraction*fraction)));

std::cout << "num_particles = " << n_particles << " num_bonds = " << n_bonds << " n_bondtypes = " << n_bondtypes << std::endl;
std::cout << "alpha = " << alpha << std::endl;
std::cout << "The fraction in ISET was: " << fraction << std::endl;
std::cout << "The standard error in ISET was: " << error << std::endl;
std::cout << "The average number of components was: " << (avg_components/((double) iterations)) << std::endl;
std::cout << "The average degree was: " << (degree/((double) iterations*n_particles)) << std::endl;

std::cout << "Time of test = " << current_time;
testfile << "Time of test = " << current_time;
testfile << "Iterations = " << iterations << " Diffusion steps = " << converge << " Binding Distance = " << distance << std::endl;
testfile << "Dimension = " << dimension << " Prob1 = " << prob1 << " Prob2 = " << prob2 << std::endl;
testfile << "Num_particles = " << n_particles << " Num_bonds = " << n_bonds << " N_bondtypes = " << n_bondtypes << std::endl;
testfile << "Alpha = " << alpha << std::endl;
testfile << "The fraction in ISET was: " << fraction << std::endl;
testfile << "The standard error in ISET was: " << error << std::endl;
testfile << "The average number of components was: " << (avg_components/((double) iterations)) << std::endl;
testfile << "The average degree was: " << (degree/((double) iterations*n_particles)) << std::endl;
testfile << "\n";
testfile.close();

std::cout << "DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD" << std::endl;
7.4.6 System 3 with probability matrix

#include <string>
#include <iostream>
#include <fstream>
#include <boost/graph/adjacency_list.hpp>
#include <cassert>
#include <boost/graph/graph_utility.hpp>
#include <boost/pending/disjoint_sets.hpp>
#include <boost/graph/incremental_components.hpp>
#include "assembly.h"

int main(int argc, char* argv[]) 
{
using namespace boost;

std::ofstream iset_file;
std::ofstream degree_file;
string iset_filename = string(argv[1]);
string degree_filename = string(argv[2]);
float iset_deg_array[121][4];
int index = 0; //index for iset_deg_array

cout << "How many iterations ?" << std::endl;
int iterations;
std::cin >> iterations;
cout << "How many diffusion steps per iterations of alpha?" " std::endl;
int converge;
std::cin >> converge;
std::cout << "How many particles?" " std::endl;
int n_particles;
std::cin >> n_particles;
std::cout << "How many type-1 particles?" " std::endl;
int n_type1; //number of particles with 2 bonds (1 type-1 and 1 type-0)
std::cin >> n_type1;
std::cout << "How many type-2 particles?" " std::endl;
int n_type2; //number of particles with 3 bonds ( 1 of type-1 and 2 of type-0)
std::cin >> n_type2;
std::cout << "How many type-3 particles?" " std::endl;
int n_type3; //number of particles with 3 bonds ( 2 of type-1 and 1 of type-0)
std::cin >> n_type3;
//the rest of the parciles = 15 - 12 = 3 will have 4 bonds ( 2 type-1 and 2 type-0)
int n_bonds = 0;
double alpha = 2.0*n_particles;
int n_bondtypes = 4;
std::cout << "Enter dimension?" " std::endl;
double dimension;
std::cin >> dimension;
std::cout << "Enter distance for binding?" " std::endl;
double distance;
std::cin >> distance;
//initialize each probability to 0
float prob1;
float prob2;

/*
int tests = 1;
int iterations = 10000;
int n_particles = 100;
int n_bondtypes = 2;
double dimension = 10;
double distance = 1;
*/

for(prob1 = 0; prob1 < 1.1; prob1 = prob1 + 0.1){
    for(prob2 = 0; prob2 < 1.1; prob2 = prob2 + 0.1){
        typedef adjacency_list<vecS, vecS, undirectedS> Graph;
        //boost::graph_traits<G>::vertex_descriptor - The type for vertex
        //boost::graph_traits<G>::vertices_size_type - The unsigned integer
        for(i = 0; i < n_particles; i++) {
            //Generate random x,y co-ordinates for a particle
            x = rand_double(dimension);
y = rand_double(dimension);
            part = new Particle(x, y, prob1, prob2); //instantiate the particle with the co-ordinates generated above
            if( i < n_type1){ //create 1st type of particles with
                part->set_bond(4); //set bond b to particle p
                part->set_bond(4);
                part->set_bond(0);
                part->set_bond(1);
n_bonds = n_bonds + 2;
            }
        }
    }
}

Graph g(n_particles); //Check the Boost library for this one.
Graph g has n_particles (?)
std::vector<Particle*> particles; //instantiate a vector of
Particle class called "particles"

for (i = 0; i < n_particles; i++) { //for each of the
    "n_particles" number of particles do the following:

    //Generate random x,y co-ordinates for a particle
    x = rand_double(dimension);
y = rand_double(dimension);
    part = new Particle(x, y, prob1, prob2); //instantiate the particle with the co-ordinates generated above
    if( i < n_type1){ //create 1st type of particles with
        part->set_bond(4); //set bond b to particle p
        part->set_bond(4);
        part->set_bond(0);
        part->set_bond(1);
        n_bonds = n_bonds + 2;
    }
}
else if ((i < n_type2 + n_type1) && (i >= n_type1)) {
    //create 2nd type of particles with 1 type-1 bond and 2 type-0 bonds
    part->set_bond(4);
    part->set_bond(1);
    part->set_bond(0);
    n_bonds = n_bonds + 3;
}
else if ((i < n_type3 + n_type2 + n_type1) && (i >= n_type2 + n_type1)) {
    //3rd type of particles w/ 2 type-1 & 1 type-0 bonds
    part->set_bond(4);
    part->set_bond(1);
    part->set_bond(1);
    part->set_bond(0);
    n_bonds = n_bonds + 3;
}
else {
    //the rest of the particles
    have 2 type-1 and 2 type-0 bonds
    part->set_bond(1);
    part->set_bond(0);
    part->set_bond(1);
    part->set_bond(0);
    n_bonds = n_bonds + 4;
}
particles.push_back(part); //push the particle "part" into the vector "particles"
}
total_edges = ((double) n_bonds/2.0);

std::vector<size_type> rank(num_vertices(g));
std::vector<Vertex> parent(num_vertices(g));
typedef size_type* Rank;
typedef Vertex* Parent;
disjoint_sets<Rank, Parent> ds(&rank[0], &parent[0]);
initialize_incremental_components(g, ds);
incremental_components(g, ds);
typedef component_index<unsigned int> Components;
for (k = 0; k < converge; k++) {
    Components components(&parent[0], &parent[0] + parent.size());
    for (Components::size_type c = 0; c < components.size(); ++c) {
        Components::value_type::iterator ii = components[c].begin(),
        iiend = components[c].end();
        int direction = rand_int(0, 1);
        for (; ii != iiend; ++ii) {
            if(direction == 0)
                particles[*ii-]
            >set_x(distance);
            else
                particles[*ii-]
            >set_y(distance);
        }
    }
}
Components::value_type::iterator
ii = components[c].begin(),
iiend = components[c].end();
for ( ; ii != iiend; ++ii) {
    for (Components::size_type cc = c; cc <
    components.size(); ++cc) {

Components::value_type::iterator
jj = components[cc].begin(),
jjend = components[cc].end();
for ( ; jj != jjend; ++jj) {
    if(*(ii != *jj)&&(particles[*ii]→calc_distance(particles[*jj]) < distance) &&
    particles[*ii]→bind(particles[*jj])) {
        //edge(*ii, *jj, g);
        if(!edge(*ii, *jj, g).second){
            add_edge(*ii, *jj, g);
            ds.union_set(*ii, *jj);
        }
    }
}
}
}

n_components = components.size();
incremental_components(g, ds);
}

intermediate = num_edges(g);
n_edges += intermediate;
sum_square += ((double) intermediate * intermediate);
avg_components += n_components;
graph_traits<Graph>::vertex_iterator pi, pi_end;
for(tie(pi,pi_end) = vertices(g); pi != pi_end; ++pi) {
    degree += out_degree(*pi, g);
}

//delete components;
for(i = 0; i < n_particles; i++)
    delete particles[i];
}

double average_edges = ((double) n_edges/iterations);
double fraction = average_edges/total_edges;
double fraction_square = sum_square/(total_edges*total_edges);
double error = sqrt(1.0/((double) iterations))*sqrt((1.0/((double)
    iterations))*(fraction_square - iterations*fraction*fraction));
double av_degree = (degree/((double) iterations*n_particles));

//set array for iset and average degree
iset_deg_array[index][0] = prob1;
iset_deg_array[index][1] = prob2;
iset_deg_array[index][2] = fraction;
iset_deg_array[index][3] = av_degree;
index++; //increment index
}

degree_file.open(degree_filename.c_str(), ofstream::app);
iset_file.open(iset_filename.c_str(), ofstream::app);
for (index = 0; index < 121; index++){
    degree_file << iset_deg_array[index][0] << "        " << iset_deg_array[index][1] << "        " << iset_deg_array[index][3] << std::endl;
    iset_file << iset_deg_array[index][0] << "        " << iset_deg_array[index][1] << "        " << iset_deg_array[index][2] << std::endl;
}
iset_file.close();
dergee_file.close();

return 0;

7.5 assembly.h

#ifndef ASSEMBLY_H
#define ASSEMBLY_H

#include <iostream>
#include <vector>
#include <algorithm>
#include <utility>

#include "rand.h"
#include "bond.h"
#include "particle.h"

#endif

7.6 Makefile

CC = g++
CFLAGS = -O2 -I /home/arjun/boost_1_36_0
#CFLAGS = -ggdb -I /home/arjun/boost_1_36_0
OBJECTS = system.o particle.o bond.o rand.o

all: system

particle.o: particle.cpp particle.h
$(CC) $(CFLAGS) -c particle.cpp
bond.o: bond.cpp bond.h
$(CC) $(CFLAGS) -c bond.cpp
rand.o: rand.cpp rand.h
$(CC) $(CFLAGS) -c rand.cpp
system.o: system.cpp
$(CC) $(CFLAGS) -c system.cpp
system: $(OBJECTS)
$(CC) $(CFLAGS) -o system $(OBJECTS)
### 7.7 graph_plot.plt (Gnuplot script file)

```
set terminal jpeg nocrop enhanced font verdana 12 size 640,480
set output "degree_surface.jpeg"
set size 1.0,1.0

set data style line

set clabel '%8.2f'
set key right

set isosamples 60

set parametric
set key below
set data style line

# View Angle set view x_rot, y_rot, z_scale,
set view 60, 30, 0.85, 1.1

# Surface
#set nosurface
set surface

set contour both
set cntrparam bspline
set hidden3d
set dgrid3d

set ticslevel 0

set xlabel "Prob1"
set ylabel "Prob2"
set xrange [0:1]
set yrange [0:1]

# Projected to 0+
set xlabel "Avg.\n degree" offset 1, 0
set zrange [0.0:4.0]
set title 'Average degree vs. Prob1 vs. Prob2'
```