Semester Project Network Mappings

Yves Jacoby

July 23, 2006

Contents

1 Introduction

Position estimation in networks is growing in importance. The first reason is surely the rapid growth of the Internet and many other computer networks and the services offered on these networks. It is important to be able to locate yourself in such networks in order to be able to find the nearest host providing some service or find the closest mirror to download some files. Algorithms for position estimation make it possible to find the nearest location to download some files without user interaction. Or in peer to peer networks, to locate efficiently the nearest host that can provide you with files of interest.

There are many other relations that can be represented as networks and on which one can use an algorithm for position estimation presented in the following chapters. An example of such a *virtual* network is the network of music artists. Artists are influenced by other artists, they play with other artists, they play in different groups, etc. All these relations can be represented in a network. Using the algorithms for position estimation in networks, one could imagine making queries like find artists whose music style ressembles some other artists style. You can read more about this in Appendix D.

The goal of this term project was to implement, compare and test both algorithms on networks. This was done in two separate steps. First a framework was implemented, which is independent of the algorithms used. In a second step, I implemented the algorithms and a simple method to generate random networks. In the following, I will present two methods to do position estimation in networks. A precise and computationally intensive one for smaller networks, described in [5] and a fast, but less precise one for much larger networks, described in [2]. Both these methods try to embed the network graph in a high-dimensional Euclidean space, giving each node a position in that space, in order to be able to compute distances between nodes.

In Chapter 2, I will describe the testing framework. It is a small framework, which is very easy to extend, such that in the future, maybe one can implement and test other algorithms. An application, $NetMap$, built using this framework is presented in the Appendix C.

Chapter 3 contains a description of both algorithms. The numerical algorithms are shortly explained in Appendix A, you will find them explained in greater detail in [4]. In Chapter 4 I list my observations regarding the two methods. The two algorithms are very different, so that a real comparison of both is not really possible. In Chapter 5 some improvements to these methods are suggested.

You can read the conclusion of this term project in Chapter 6. It was difficult to find a common conclusion to these two algorithms, which appear to have nothing in common, except that they estimate positions of nodes in networks.

2 Framework

The main goal of this term project was to compare two algorithms for estimating network node positions. To be able to do this and maybe later implement new algorithms to be compared to, an extensible framework is needed, which on one hand supports a sufficiently complex representation of networks and on the other hand, allows for simple addition of new algorithms for generation of networks and estimation of distances in a network.

In the following sections, I explain in greater detail the features of the framework, its architecture and how it works.

The types and method names are not explained in the following, except if the classes are important for the extension of the program with new algorithms and features. If you have some doubts about what exactly some method does or some type is, please have a look at the javadoc or source code of *NetMap*.

2.1 Framework Features

The main features I implemented in this framework were:

- Customizable network generation algorithms
- Customizable estimation algorithms
- A sufficiently complete network representation

- Extendability

There are many different network topology generators, so it is not sufficient to implement only one algorithm. It should be simple to implement and test or use other generation algorithms. The implementation is is described in chapter 2.2. For a short overview of different methods (random graph generators, structural generators and degree-based generators) for network topology generation and a qualitative comparison I recommend reading [6].

As for topology generators, there are many different algorithms for distance estimation, some references are [5], [2], [3]. This part of the framework should be just as interchangeable as the topology generator part. A detailed description of the implementation can be found in chapter 2.3.

The third main component of the framework, which holds everything together, is the network representation. The requirements for the network representation are:

- Algorithms working on it should be able to do their task efficiently
- Some simple routines, like shortest path computation, finding neighbours, etc. need to be part of the basic framework
- The nodes should contain all the needed information for many different distance estimation algorithms.
- The generator should be able to generate its network in a convenient format, without having to do conversions or other manipulations afterwards

The implementation of the network is described in 2.4.

The main part of the framework, like topology generation algorithm, distance estimation algorithm, etc. are clearly defined by interfaces. Classes that implement these interfaces can be integrated without additional work in the framework, by adding them to the *netmap.properties* file. The framework then dynamically loads them at startup and uses reflection to instantiate objects or call methods. In chapter 2.5 you can find the format of this file and additional information on the internals of netmap.

Some basic metrics are included in the framework. These can be used between or after the runs of algorithms to test different results and the quality of the approximations. It has to be noted here, that it is difficult to find a satisfying measure of quality for these approximations, especially if we want to measure the quality between two algorithms and two different networks.

2.2 Network Generation Implementation

The main problem I encountered when implementing this part of the framework, is that we want to be able to use a graphical interface, as well as a console interface and we have to be able to pass parameters in both cases. After some testing, I decided to let the programmer of the algorithm implement the input interfaces himself. This is done in the GUI case by exporting a panel where the user can enter the options. In the console case, a string is passed to the generator class.

The interface for the generators looks as follows:

```
public interface INetGenerator {

  /**
   * Generates the network.
   *
   */
  public void generate();

  /**
   * Can only be called once the network has been generated. Returns the set
   * of nodes that have been generated.
   *
   * @return the set of generated nodes.
   */
  public ArrayList<NetNode> getNodes();
```

```
  /**
   * Can only be called once the network has been generated. Returns the set
   * of edges that have been generated.
   *
   * @return the set of generated edges.
   */
  public ArrayList<NetLink> getLinks();

  /**
   * Sets the options of the generator using a string.
   *
   * @param s
   * @return
   */
  public boolean setOptions(String s);

  /**
   * Sets the options as the object returned by the panel.
   *
   * @param o
   *            the object containing the options.
   * @return true is everything is correct.
   */
  public boolean setOptions(Object o);

  /**
   * Returns a panel which asks the user to set the generator's parameters.
   *
   * @return a property panel.
   */
  public INetGenPanel getOptionPanel();
}
```

A class implementing this interface, can be declared in the configuration file to be loaded as a network generator at the start of the program. The constructor of the generator needs to have one parameter, a double *size* , which specifies the size of the surface to place the network on as $size \times size$. The constructor is called using the java reflection API.

The algorithm should generate nodes at a position on the surface given by the parameter to the constructor and edges between these nodes. The resulting edges and nodes have to be passed back when calling *getNodes*() or *getLinks*(). You can find a simple implementation of such a generator in the source code of *NetMap*.

The `INetGenPanel` class from which all the panel classes need to be derived looks as follows.

```
public abstract class INetGenPanel extends JPanel {

  private JDialog dialog;

  /**
   * The panel asks the user the different parameters for the network
   * generation. Once the OK button was pressed, the panel is hidden and the
   * <code>getResult()</code> has to return <code>null</code> if the input
   * was incorrect or an object that can be passed to the algorithm.
   *
   * @return an object to pass to the algorithm.
   */
  public abstract Object getResult();

  /**
   * Sets the parent dialog.
   *
   * @param dialog
   *            a dialog window.
```

```
     */
    public void setDialog(JDialog dialog) {
            this.dialog = dialog;
    }

    /**
     * Has to be called by the implementing class when the input is finished.
     *
     */
    protected void inputFinished() {
            this.dialog.setVisible(false);
    }
}
```

This abstract class for the parameter panel makes it especially convenient to implement customized panels. The derived class only needs to extend the panel, and a *getResult*() method, which returns an object specifying the options set by the user. Displaying the panel, retrieving the options and passing them to the adequate algorithm is handled internally by the program.

One thing that could be changed in a future version is the description of the parameters, which would be given in an XML file. The program could parse the file and automatically generate the appropriate panel or text to ask the user. However, I have not been able to find a way to change the input parameters depending on the input of some previous parameters, so I opted for the presented solution.

2.3 Network Distance Estimation Algorithm Implementation

The implementation of the estimation algorithms poses problems similar to the implementation of generation algorithms. There is one additional problem, which is that some algorithms are interactive. Asking the user for parameters can be handled as in the case of the generation algorithms. However, the user interactivity while the algorithm is running is a problem.

Implementation

The estimation algorithms are in a bubble-like environment. They are managed by an console interaction class or a panel.

The algorithms work directly on the network, having access to nodes, edges and the facilities provided by the `Network` class. The programmer should be careful not to change the network while accessing it. The main reason for direct access to the network is efficiency.

To implement a new algorithm, you need a central class, which will be the class you write in the configuration file. This class should have two static methods:

- *getPanel*() which should return a panel to interact with the user.
- *getConsole*() which should return a class to interact with the user on a console level.

The interface of a new algorithm can be implemented to fit the programmers needs. This is due to the fact that the algorithm is controlled by the console or GUI interface directly and has no other interaction with the framework than working on the network.

Again, an implementation using XML for parameters would be a desirable feature in a future version.

Algorithm Interaction

As an example, we can have a closer look at the PIC algorithm. The PIC algorithm allows you to choose some reference points, however, depending on the order of computation of these reference points, you can get different results. This implies that the user should be able to select in which order the reference points are computed.

First I am going to discuss the GUI case. Following the previous idea, I let the choice to the programmer of the algorithm whether to react to user input or not and take the appropriate action. The programmer can have

a class implement the `NetPanelListener` interface and register that class to the panel displaying the network map. The `NetPanelListener` looks as follows

```
public interface NetPanelListener {
  /**
   * This method is called by the network panel when one node is clicked by
   * the user. It passes as parameter the node on which the user clicked.
   *
   * @param netNode
   *             the node the user clicked.
   */
  public void selectAction(NetNode netNode);

  /**
   * This method is called when the user clicked one node and then drag the
   * cursor to another node and released it. The first node is the one on
   * which the user pushed the button and the second one is the one on which
   * he released the button.
   *
   * @param n1
   *             the node over which the user pressed the button.
   * @param n2
   *             the node over which the user released the button.
   */
  public void selectAction(NetNode n1, NetNode n2);
}
```

As you can see, most of what needs to be done externally to the algorithm is already handled by the framework when the interface methods are called. The handler only needs to deal with the nodes.

In the console case, it was far more difficult to find a user friendly solution. I decided therefore that each algorithm should also implement an automatic mode. This need is especially expressed when running the PIC algorithm, which needs a selection of starting nodes. The console version should allow for the same specific input as the GUI version. These features should all be the implemented in a separate class, which is returned by the call to *getConsole*().

2.4 Network Implementation

The network is represented by nodes and links, defined in classes `NetNode` and `NetLink`. The complete representation is encapsulated in the `Network` class.

The content of a network node is:

- A unique ID, which is given to it when, it is entered in the network.
- A set of all the edges which it is part of.
- The network it is associated to.
- A position vector that can have any dimension, for algorithms which associate with each node a position in a higher-dimensional space (e.g. PIC).
- A color which is used to draw the node.
- Optionally a name, which can be used for the simulation of virtual networks, like a network of artists.

The representation of an edge is much simpler, since it has a length, two end-nodes and a color. The network class manages all the nodes and links. It provides possibilities to locate nodes or find the node that is nearest to a point, the number of edges and nodes and a possibility to register a listener for network changes (see `NetChangeListener`).

After the network has been built, the programmer should not try to change links or nodes. This can result in an inconsistent state of the internal structures.

2.5 Modularity of the Framework

Modularity of the framework is achieved using the Java reflection API. You can load an additional algorithm or measurement tool by adding a line to the *netmap.properties* file. The file is parsed at program start, the classes loaded and eventually they will be initialized using reflection. The format of the file is:

```
Generator1=netmap.network.generators.SimpleNetGenerator
Generator2=netmap.network.generators.NetGen2

Algorithm1=netmap.algorithms.bigbang.BigBangAlgorithm
Algorithm2=netmap.algorithms.pic.PICAlgorithm
```

The two available categories are *Generator*, *Algorithm*. Every record in each category needs to have a different number.

2.6 File Format

This section describes the program's file format. The nodes are described by a unique identifier, their position in an euclidean space and a name. The edges are described by their length. Nodes and edges have an additional color property for nicer display.

The current format is:

```
NN: 22 # Number of nodes
EN: 30 # Number of edges
# Description of nodes
# ID X-position Y-position Color Name
0 507.07 474.81 -26804480 The name can be written like this # This is a comment

# Description of edges
# Length  Start-node End-node Color
51.069714284642245 0 8 -33528116
```

Everything that follows a # is taken as a comment. The attributes are space separated.

3 Algorithm Descriptions

The two algorithms described in the following sections are the Big-Bang algorithm, which was described in detail in [5] and the PIC Algorithm, which was described in detail in [2].

There are at least 3 different versions of the Big-Bang paper that can be found on the Internet. I will base the description of the algorithm on the version [5], which is the most complete. The equations to compute the force induced on a particle are wrong (example: the formula for $F_{ij}^{\langle 2 \rangle}$ on page 1005 in [5]), so I recompute them completely at the end of the description.

In the following, i and j denote nodes in the network graph. Δ_{ij} denotes the weighted shortest path between node i and j. The current estimated position of node i is v_i.

You can read about the results and observations of the experiments in 4.

3.1 Description of the Bigbang Algorithm

The Big-Bang Simulation algorithm is an adaptation of spring simulations to network mapping and embeddings. Each node is represented as a particle in an Euclidean space. The particles travel through that space, influenced by a force field. The force field reduces the potential energy of the particles. The potential energy is chosen such that it decreases if the quality of the embedding increases according to different metrics. In addition to this driving force, they are subject to friction, in order to permit the system to stabilize itself with time.

This algorithm's purpose is to compute positions for the nodes of a network. The information necessary to the computation are the currently estimated distances between each pair of nodes and the weighted shortest path in the network graph between each pair of nodes. The weight of an edge is the network distance between its two endpoints.

It is not described in any version of the paper where exactly the computation is done or how all the data is passed to the nodes that compute the positions. I therefore assume that we have a computation node, which may be part of the network. This machine knows the complete topology of the network in order to be able to compute the distance matrix among network node pairs. After finishing the computation, this node sends each node its estimated position.

The Euclidean space in which the positions of the nodes are estimated can have any dimension. However, the authors of [5] noticed in their experiments that a dimension of 7 was an appropriate choice for this algorithm.

The name of Big-Bang Simulation comes from the fact that at the beginning of the simulation, all the nodes are placed at the origin. After starting the simulation, they move away from each other very quickly, to later slow down due to friction and try to find a minimum of the potential energy. In this simulation, the network graph nodes are considered particles with a certain position.

In the following description, some additional definitions are needed. d_{ij} denotes the symmetric pair distortion, which is defined as the maximum between the expansion and the contraction ratios between two nodes i and j as

$$d_{ij} = \max\left(\frac{\|v_i - v_j\|}{\Delta_{ij}}, \frac{\Delta_{ij}}{\|v_i - v_j\|}\right)$$

f_i denotes the force that is currently applied to node/particle i. $\dot{v}_i$ denotes the speed of i. The mass of each node is assumed to be 1 unit, so that the value of the acceleration $\ddot{v}_i$ is the same as the force's value. The algorithm is divided into 4 phases explained later. Using the expression for the energy field in phase $\star$, an expression $\mathcal{F}^{\star}_{ij}$ denoting the force acting between two particles can be derived. The derivation of $\mathcal{F}^{1}_{ij}$, $\mathcal{F}^{2}_{ij}$, $\mathcal{F}^{3}_{ij}$ and $\mathcal{F}^{4}_{ij}$ is explained in detail later.

The algorithm is divided into 4 phases. In each phase a complete simulation is run, using different definitions of the potential energy field and the position of the nodes v_i, which were computed in the last phase. In the first phase, the positions are all initialized to $\mathbf{0}$. One could say that the phases go from coarse to fine. The first phase minimizes the error between particle v_i and v_j by using the difference between $\|v_i - v_j\|$ and Δ_{ij} as an error metric. The second and following phases minimize the symmetric pair distortion d_{ij} between node i and j using each time more sensitive error functions. I noticed and the authors wrote that it is important to respect the order of the phases, because of numerical instabilities that may happen if the forces that apply to nodes become too large. This effect is due to the increasing sensitivity of the error function.

Algorithm 3.1 describes in pseudo-code the procedure to compute a phase of the algorithm. In the first part, for each particle, the total force f_i that applies to particle i is computed using the expression of the potential energy field corresponding to the current phase. In the second part, for each particle, an integration is done, to compute the new positions of the particle using f_i and $\dot{v}_i$. We iterate over part 1 and 2 for a selected number of times, which is empirically chosen. The phases are executed in order.

Algorithm 3.1: Compute phase $\star$

Input: A set of particles $\mathcal{P}$ initialized to their starting position.
Input: The number i of iterations.

```
1 for k := 0 to i do
2     for Particle p in P do
3         f_p := 0;
4         for Particle q in P \ {p} do
5             f_p := f_p + F*_{p,q};
6     for Particle p in P do
7         Integrate f_p to get new v_p;
```

In algorithm 3.1 a *for* loop with a fixed number of iterations is used in the computation of each phase. Depending on the version of the paper,the authors speak about termination criteria or a total number of iterations I. This is what leads the authors to give this algorithm a running time of $O(In^2)$. In earlier version of the paper, the following termination criteria are cited:

1. the distortion satisfies $\max_{i,j}(d_{ij}) < 1 + \epsilon$ - The particles are almost perfectly embedded.
2. the maximum velocity of a particle decreases below a threshold - The particles are almost at halt.
3. the difference between local minimum and maximum of potential enery decreases below a threshold - The particles are near an equilibrium.
4. the reduction speed of the potential energy is below a certain threshold - Small convergence rate of Energy
5. the maximum velocity grows over a certain threshold - The particles start flying away from each other.

Points (2) to (5) of this list should only be checked for after a certain minimum number of iterations of each phase. Furthermore, in earlier version of the paper it says that the step size for the integration should be adapted while running a phase of the algorithm. At the beginning of the phase, the step size should be kept very small and slowly increased as the energy decreases.

Induced Force Expressions

It remains to describe $F_{ij}^{\star}$ for $\star$ equals 1 to 4 and how to compute it. The equations for the potential force field are given by the authors in the paper, unfortunately, in the newest version of the paper, the force equations are not given and in the original paper, they are computed in the appendix, but with some mistakes. First, I will show the equations for the potential force field, then explain how to derive the force and finally, derive the four expressions for the force.

The energy force field is defined between each two nodes in the graph. To get the total field at a node, we need to sum up the induced field over all other nodes in the network. To get the force at a certain node, given the properties of the derivative, we need to sum up over all the derivatives of the field.

The potential energy force field expressions are defined for phases 1 to 4 as follows:

1. $E_{ij}^{\langle 1 \rangle}(v_i, v_j) = (\|v_i - v_j\| - \Delta_{ij})^2$ - The force field between the two nodes n_i and n_j is defined as the squared distance error between. The error is computed between the distance of the currently estimated positions and the shortest weighted path between these two nodes.
2. $E_{ij}^{\langle 2 \rangle}(v_i, v_j) = (d_{ij} - 1)^2$ - The force field in phase 2 and up always try to minimize the symmetric pair distortion. The -1 term has to be there, so that when the embedding is perfect, the field value equals 0.
3. $E_{ij}^{\langle 3 \rangle}(v_i, v_j) = \exp^{E_{ij}^{\langle 2 \rangle}{}^{\frac{3}{4}}} - 1$ - In phase 3, you can see that basically, we are still minimizing the $E_{ij}^{\langle 2 \rangle}$, but this expression is much more sensitive to small errors, because $E_{ij}^{\langle 2 \rangle}$ is in the exponent of the expression.
4. $E_{ij}^{\langle 4 \rangle}(v_i, v_j) = \exp^{(E_{ij}^{\langle 2 \rangle})} - 1$ - Phase 4 is the most sensitive phase to the symmetric pair distortion error phases.

The first phase is numerically stable, because the values do not really grow too much, especially because in the force, the exponent disappears as we will see. Phase 2-4 are increasingly sensitive to errors of the symmetric pair distortion. The reason of this is that when the system is nearly stabilized, the convergence speed decreases, because the forces become very small. A way to make the convergence faster again is to change the force field to make it much stronger again, so that we arrive at the desired result faster.

Given the expression for the force field, rewritten as $\mathcal{F}(\|v_i - v_j\|, \Delta_{ij})$ a function of two parameters, namely $\|v_i - v_j\|$ written shortly $\|v_{ij}\|$ and Δ_{ij}, we can derive the expression for the force F_{ij} between two particles i and j using the following fact

$$F_{ij} = \frac{d}{dx}\mathcal{F}(x, \Delta_{ij})\Big|_{x=\|v_{ji}\|} \tag{3.1}$$

Namely, the field force induced on a particle by another particle is given by the derivative of the pair embedding error with respect to the Euclidean distance between the particles. The exact derivation of this can be read in [5].

Using equation (3.1) one can derive the formulas (3.2), (3.3) for the forces between node i and j. Here I show how to compute the formulas for the first and second phase. The 3^{rd} and 4^{th} are very similar.

$$\begin{aligned} F_{ij}^{\langle 1\rangle} &= \frac{d}{d\left\|v_{ij}\right\|}\mathcal{F}^{\langle 1\rangle}(\left\|v_{ij}\right\|, \Delta_{ij}) \\ &= \frac{d}{d\left\|v_{ij}\right\|}(\left\|v_{ij}\right\| - \Delta_{ij})^2 \\ &= 2(\left\|v_{ij}\right\| - \Delta_{ij}) \end{aligned} \tag{3.2}$$

$$\begin{aligned} F_{ij}^{\langle 2\rangle} &= \frac{d}{d\left\|v_{ij}\right\|}\mathcal{F}^{\langle 2\rangle}(\left\|v_{ij}\right\|, \Delta_{ij}) \\ &= \frac{d}{d\left\|v_{ij}\right\|}(d_{ij} - 1)^2 \\ &= \begin{cases} 2(\left\|v_i - v_j\right\| - \Delta_{ij})\frac{1}{\Delta_{ij}^2} & \text{if } \left\|v_{ji}\right\| > \Delta_{ij} \\ 2(\left\|v_i - v_j\right\| - \Delta_{ij})\frac{\Delta_{ij}}{\left\|v_{ji}\right\|^3} & \text{if } \left\|v_{ji}\right\| < \Delta_{ij} \end{cases} \\ &= F_{ij}^{\langle 1\rangle}\begin{cases} \frac{1}{\Delta_{ij}^2} & \text{if } \left\|v_{ji}\right\| > \Delta_{ij} \\ \frac{\Delta_{ij}}{\left\|v_{ji}\right\|^3} & \text{if } \left\|v_{ji}\right\| < \Delta_{ij} \end{cases} \end{aligned} \tag{3.3}$$

Here is a summary of the forces used in each phase.

1. $F_{ij}^{\langle 1\rangle}(v_i, v_j) = 2(\left\|v_{ij}\right\| - \Delta_{ij})$
2. $F_{ij}^{\langle 2\rangle}(v_i, v_j) = \begin{cases} 2(\left\|v_i - v_j\right\| - \Delta_{ij})\frac{1}{\Delta_{ij}^2} & \text{if } \left\|v_{ji}\right\| > \Delta_{ij} \\ 2(\left\|v_i - v_j\right\| - \Delta_{ij})\frac{\Delta_{ij}}{\left\|v_{ji}\right\|^3} & \text{if } \left\|v_{ji}\right\| < \Delta_{ij} \end{cases}$
3. $F_{ij}^{\langle 3\rangle}(v_i, v_j) = \frac{3}{4}\left(\mathcal{F}^{\langle 2\rangle}(\left\|v_{ij}\right\|, \Delta_{ij})\right)^{-\frac{1}{4}} F_{ij}^{\langle 2\rangle}(v_i, v_j) \cdot \exp^{E_{ij}^{\langle 2\rangle}\frac{3}{4}}$
4. $F_{ij}^{\langle 4\rangle}(v_i, v_j) = F_{ij}^{\langle 2\rangle}(v_i, v_j) \cdot \exp^{E_{ij}^{\langle 2\rangle}}$

3.2 Description of the PIC Algorithm

This algorithm does not try to find a global best position for all the nodes, but only a local best position given some reference points (called landmarks). The theoretical parts of the paper [2] lack many details, like how to position the initial landmarks.

The algorithm maps each node to a point in a d-dimensional euclidean space. It tries to do that, by minimizing the sum of the pair distance errors between the node whose position we want to estimate and a selected set of so called landmarks, which are chosen in advance. These landmarks already have a position in the d-dimensional space.

As I said, the paper lacks many details, so before I go on explaining the algorithm, I want to explain the main assumption I had to take. It is not explained in the paper how to chose the positions of the starting set of landmarks. These landmarks should be optimally placed in a d-dimensional space and at the beginning, it is a small set, so I decided to use the Big-Bang algorithm to set the starting landmarks. To select the elements of this set, I used the uniform distribution. Since each node needs to agree on a value for d, I assume that this value is known to all the nodes in advance.

The PIC algorithm is run locally at each node that wants to determine it's position. The position in a d-dimensional space is computed using reference nodes, so called landmarks, whose positions are already known in the d-dimensional euclidean space. A new node approximates it's current position, by trying to minimize as much as possible the mean squared error (MSE) between the weighted shortest path and the current high-dimensional positions of the landmarks. The authors of the paper use the simplex algorithm to minimize the MSE.

Algorithm (3.2) is a pseudo-code description of what a node has to run in order to determine it's position in the d-dimensional space. The algorithm is divided into two parts. First, the node has to determine a set of landmarks which it can then, in a second step, use to estimate its position.

Algorithm 3.2: PIC algorithm

Input: A dimension d in which to estimate the position
Data: The current node is node a
`/* Step 1: Find a set` $\mathcal{L}$ `of landmarks */`
1 Select a set of landmarks $\mathcal{L}$ with $|\mathcal{L}| \geq d+1$
`/* Step 2: Estimate the current node's position */`
2 Use the simplex algorithm as described in [4] to minimize $\sum_{l \in \mathcal{L}} \left(\frac{\Delta_{al} - \|v_a - v_l\|}{\|v_a - v_l\|} \right)^2$

Selecting Landmarks

At the beginning, the authors assume that a set of landmarks is placed in the network. They however do not describe how these nodes' positions are to be chosen. The nodes that have a position can be taken as possible landmarks.

The authors showed using their empirical results that a selection strategy choosing some close landmarks and some random landmarks was the best choice for estimating a node's position. The random landmarks will mostly be further away than the close ones. The main problem in this step is to select the close landmarks. There are two algorithms described in the paper to find close landmarks. The tradeoff is overhead vs. precision.

The first method has very low overhead, but is not very precise. Assuming we already know one landmark c, we can measure the round-trip-time (RTT) to it. Then, we ask it for its neighborn and measure the RTT to each one of them. If one is closer, we set c to this one and iteratively go on until we do not find any closer node.

The second method uses the PIC algorithm to estimate and refine in each step its current estimated position. It start by using a random selection strategy to select a set $\mathcal{L}$ of landmarks and uses it to estimate its position. Using this position, the algorithm can select a new set of landmarks to compute its position. This process iterates until we have reached a point where we do not find any closer nodes.

Parameter Values

The following observations about the parameter values of the algorithm have been made by the authors of the paper, given the results they obtained. The dimension should be chosen around 8 and dimensions over 12 seem to give no benefit. To choose the landmarks, it is good to choose in a hybrid way, which means some close and some far. In the presented results, they choose 4 close nodes and 12 far nodes. To choose more landmarks has nearly no effect on the hybrid or close selection scheme, but gives better results for the random scheme.

4 Comparison between PIC and Big-Bang Algorithms

The biggest difference between these two algorithms is the domain of application. The Big-Bang algorithm is used to compute the positions of all the nodes in a network at once. The PIC algorithm is used to compute the position of a new node in the network using positions previously computed for the other nodes.

The chapter 4.3 tries to compare both algorithms, list their strength and weaknesses. The next two chapters however will concentrate on observations on both algorithms.

The average symmetric pair distortion plots have been transformed such that the best value is not 1, but 0, by substracting 1 from every datapoint.

4.1 Observations: Big-Bang Algorithm

The Big-Bang algorithm is a slow algorithm. In figure 1 you can see the running time of the algorithm drawn against the number of nodes in the network graph. The exact time is not important, only that the algorithm has a quadratic running time. This is a main problem when running this algorithm. The authors of the paper indicated that the algorithm was usable for less than 750 nodes. With today's computers, it is possible to do more (depending on the computation time one is willing to spend), but because of the quadratic term it will not be much more.

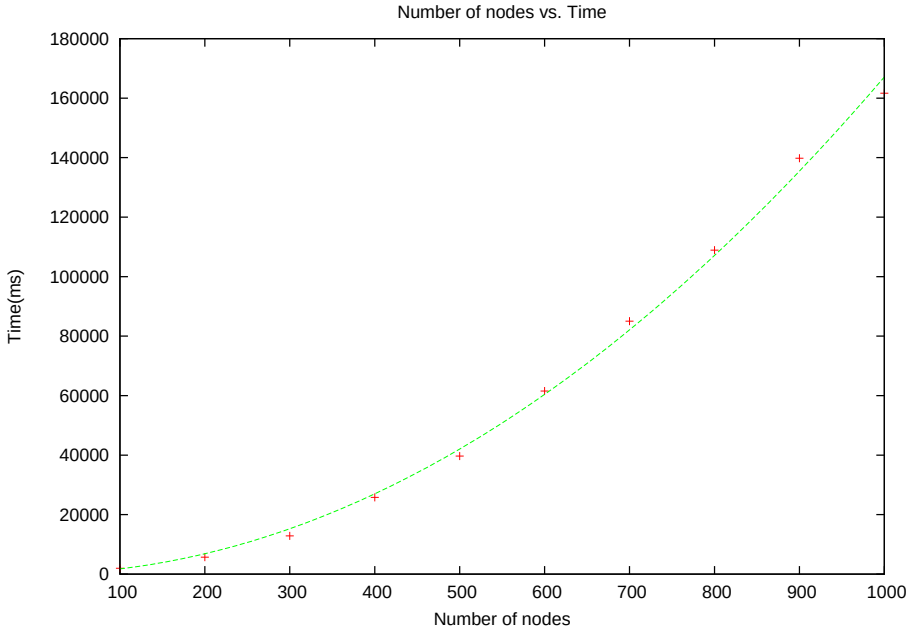

Figure 1: Big-Bang: Number of nodes vs Time

Another observation is that the algorithm runs in $O(d \cdot n^2)$ where d is the dimension and n the number of nodes. Figure 2 shows a plot of the running time against the dimension.

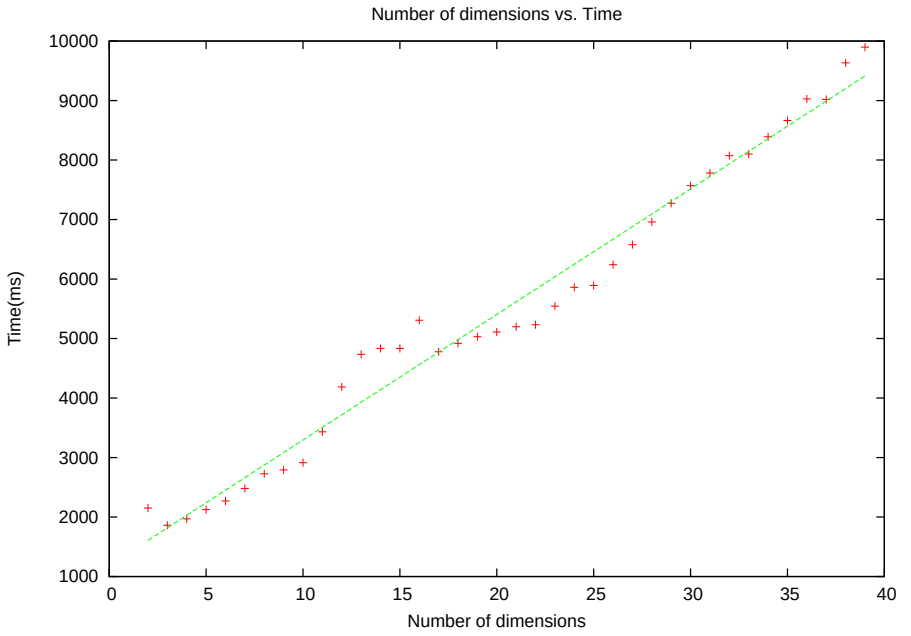

Figure 2: Big-Bang: Number of dimensions vs Time

A problem I encountered when running this algorithm is that it often crashed because of numerical instabilities or too large integration steps. The four phases are stable when computing them, the problem appears when switching from phase 1 to 2 and 2 to 3. 3 to 4 is not a large difference anymore if 3 worked well. I therefore made the step size dynamic in order to be able to have faster convergence, which means that less iterations are needed. The step size parameter is the upper limit for the step size. The number of iterations is very important to a stable phase switch, because if the previous phase did not converge enough, the simulation is probably going to explode (i.e. the forces are going to be too large for a stable integration). In Figure 3 you can see the maximum values of the force in a typical run of the algorithm. In this figure, each phase has 500 iterations. You can see the jumps at the beginning of the second and third phase. Handling these is probably the most

difficult part of that algorithm. Unfortunately, no information was given in the paper about how the authors dealt with that problem.

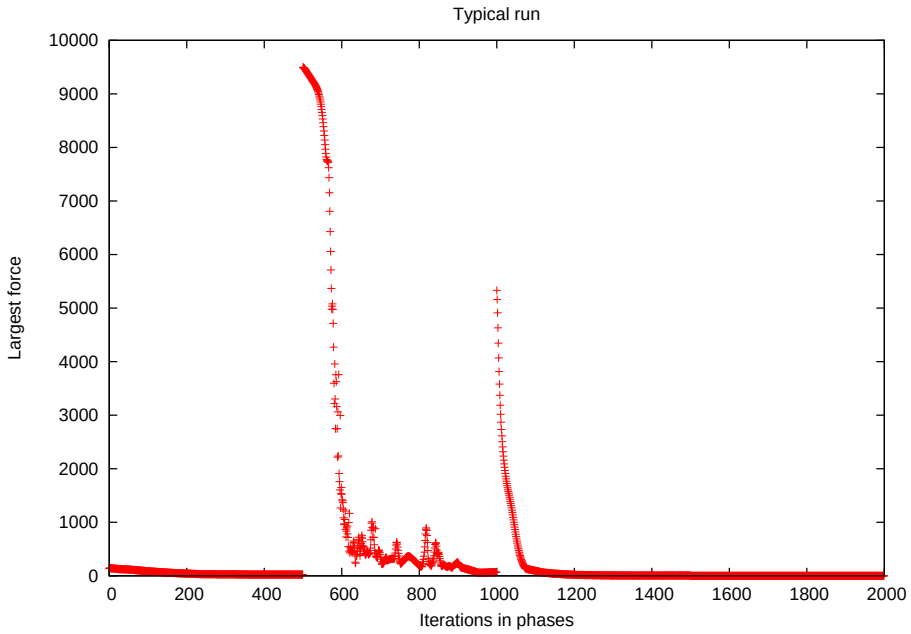

Figure 3: Big-Bang: Largest force in a run of the algorithm. Each phase has 500 iterations. One can clearly see the switch from phases $1 \rightarrow 2$ and $2 \rightarrow 3$.

Now, I am going to concentrate on the quality of the results given a certain set of parameters. For the following examples, I generated a hierarchical network graph with 133 nodes. I generated 2 layers, one with 100 nodes and another one with 33 nodes. The link probability inside the large layer is 0.1 (to represent hosts on a network that are losely connected) the small layer has a link probability of 1 and the inter-layer links have a probability of 0.5. I used 500 iterations in each phase. The network is drawn on a $[0:1] \times [0:1]$ board. This means that all the direct links between network nodes have at most $\sqrt{2}$.

In figure 4 you can see the different metrics, average symmetric pair distortion, average relative error and the average square error plotted with respect to the dimension. This confirmes what the authors recommended as value for the dimension into which to embed the network. The value should be chosen between 7 and 10.

Another important parameter of the Big-Bang algorithm is the friction. The more friction we have, the faster the approximation converges. The simulation plots in figure 5 show that the friction has in a large range no significant effect on the results (as the authors already observed). This means that the friction can be freely chosen to stabilize the simulation. My tests showed that good values for the friction are between 0.5 and 0.9, depending on the size and topology of the network graph.

4.2 Observations: PIC Algorithm

The PIC algorithm is an algorithm that can be run locally on each node that wants to find it's position, so the interesting questions are how to choose the number of landmarks and what is a sufficient number of iterations. These two questions will be answered in this chapter.

The plots in figure 6 are taken for an embedding of a network in 2 dimensions. Looking at the plots of figure 6, we see that a number of iterations between 60 and 80 is enough. More iterations do not improve the results anymore. Another observation we can make looking at figure 6 is that 1 iteration of the algorithm does not significantly increase the total time of the algorithm. However, as you can see in figure 7 the number of iterations needed to arrive at the point where more iterations don't increase the result significantly increases with the dimensionality of the embedding space. The network used for this example is not embeddable in a 2 dimensional space.

The dimension of the embedding is very important for this algorithm, unfortunately using higher dimensions and getting better approximations uses a lot more iterations. The plot of figure 8 shows the importance of the

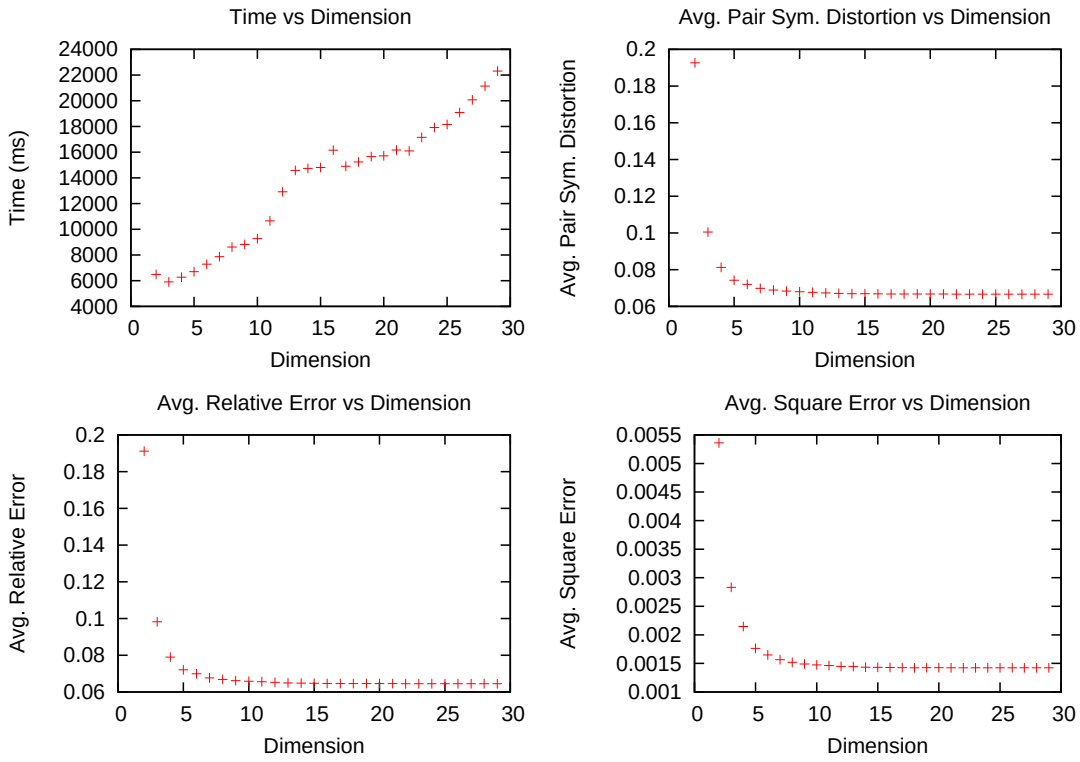

Figure 4: Big-Bang: Effects of dimensions.

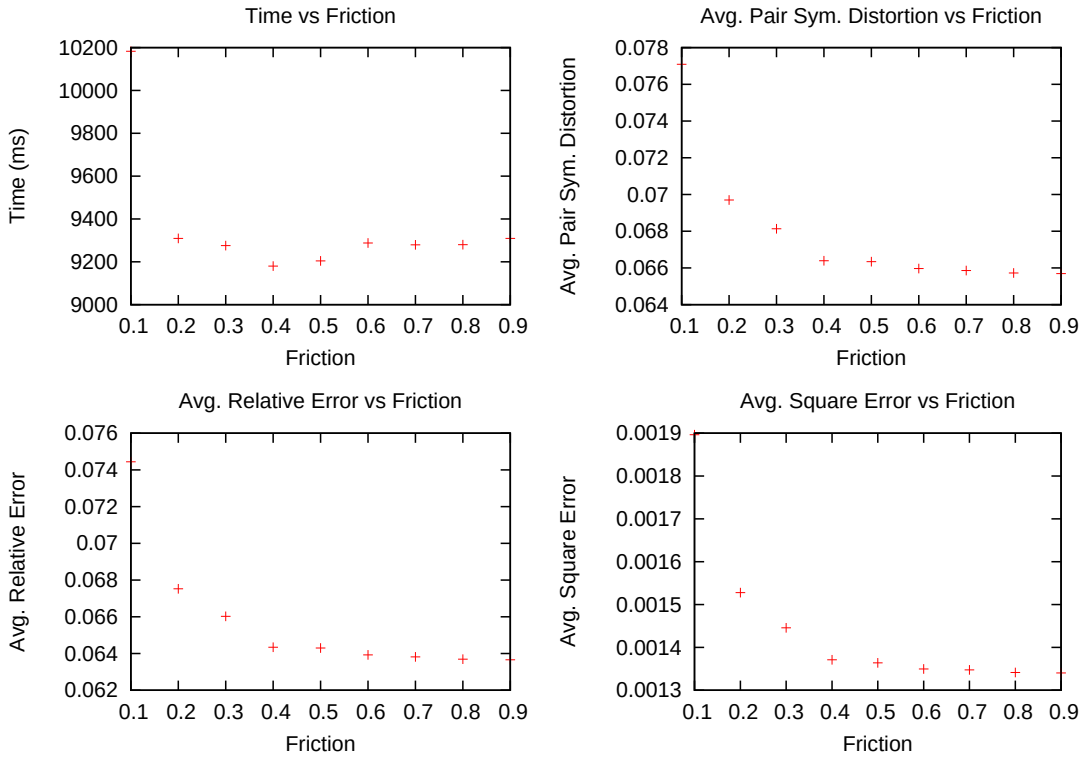

Figure 5: Big-Bang: Effects of friction.

dimension in the PIC algorithm. The network for these measurements was chosen such that it would not be embeddable in a 2-D space. The number of iterations is set to 100000, so that we can concentrate on the quality of the approximation. As you can see, optimal values are attained at a dimension around $8 - 12$. This value depends a lot on the complexity of the network. The best approach is to empirically test which dimension is

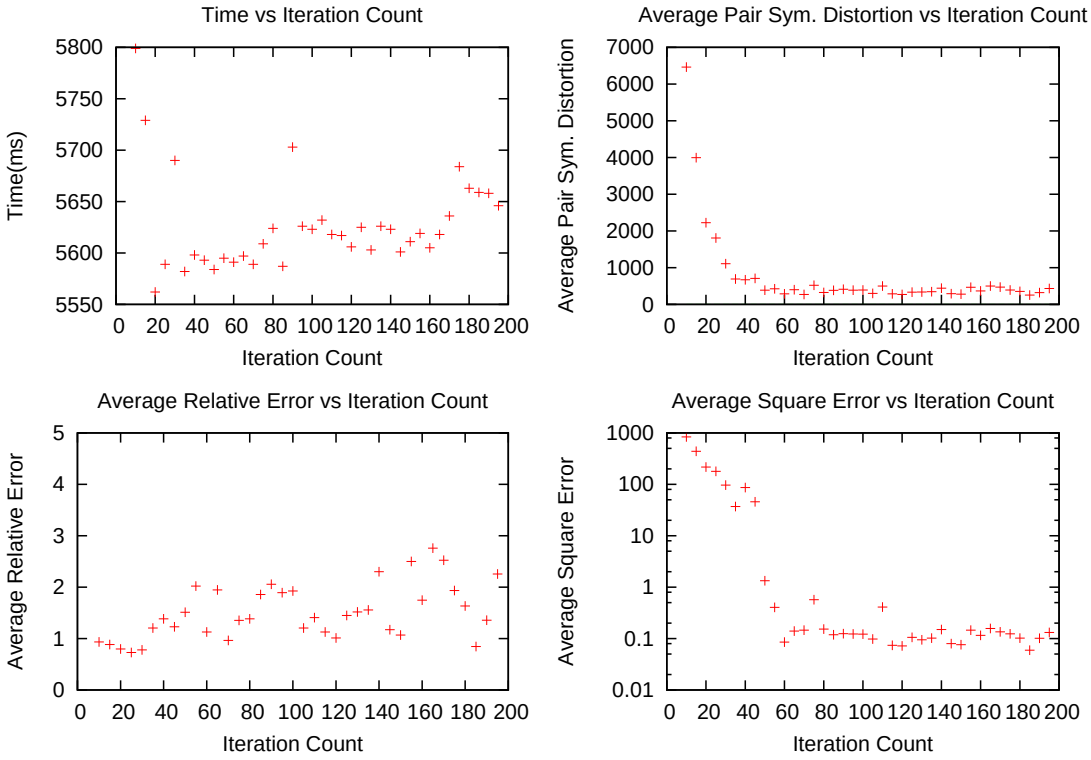

Figure 6: PIC: Influence of the number of iterations.

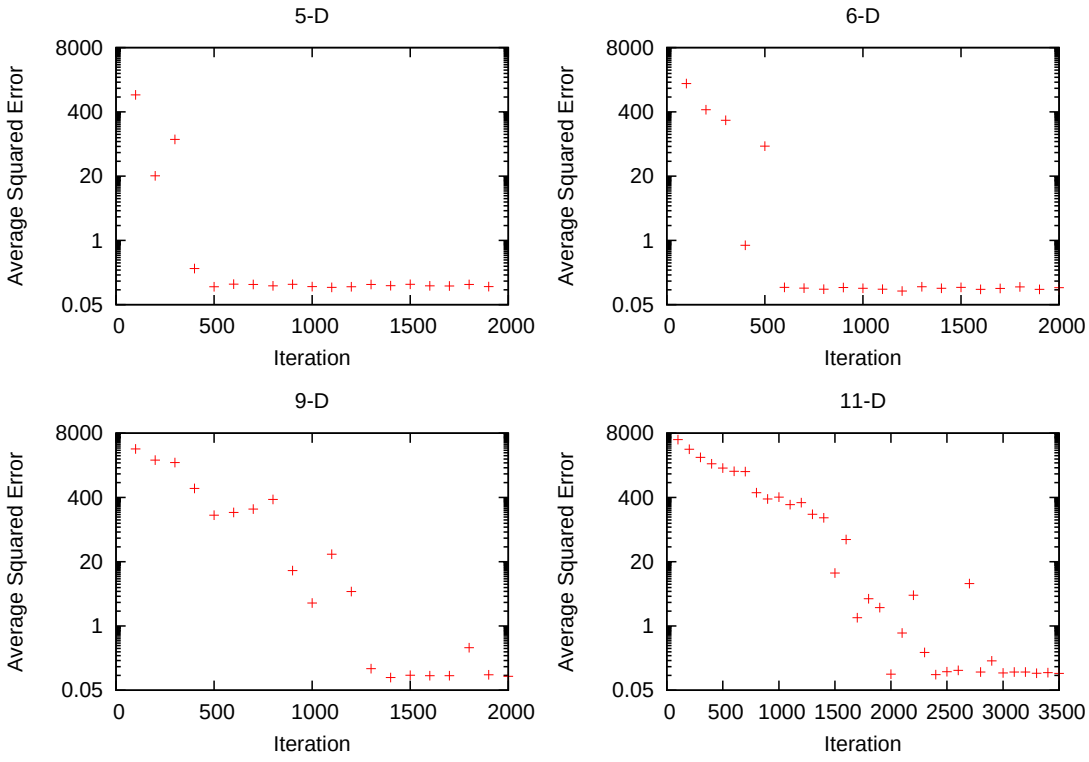

Figure 7: PIC: Influence of the number of iterations.

more appropriate to embed a certain network. The authors of the paper use an 8 dimensional space for their experiments (empirically chosen given the networks they are using).

Another parameter of the algorithm is the choice of the landmarks (near or randomly distributed). Unfortu-

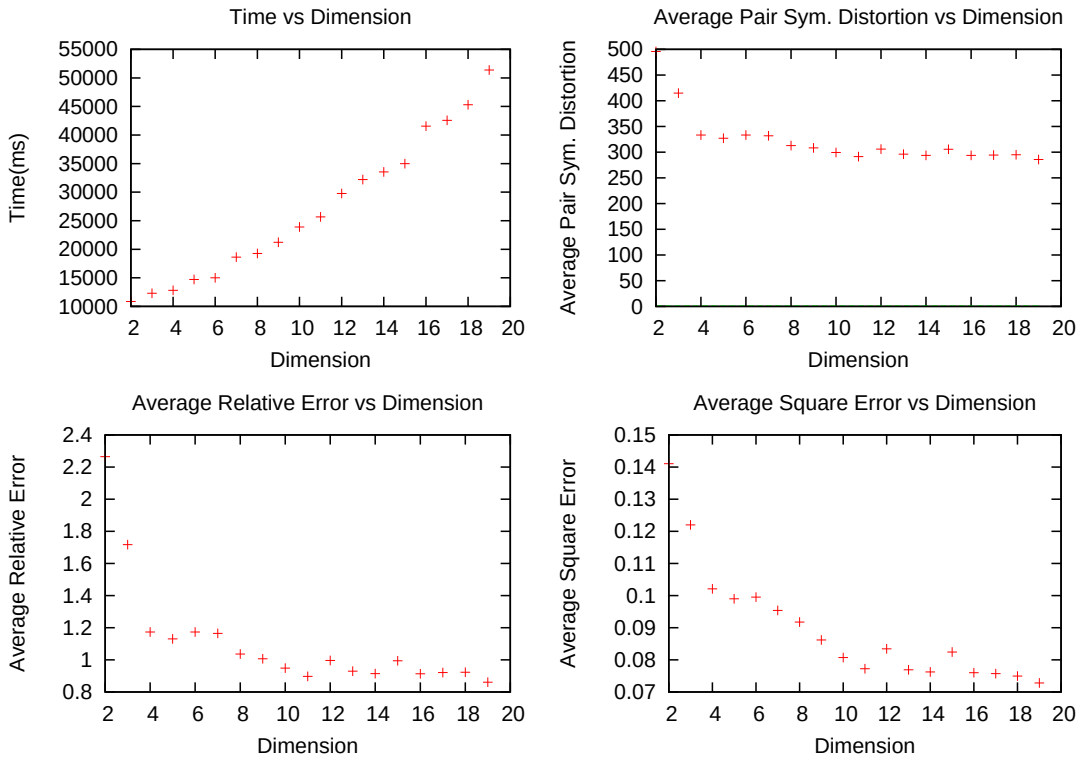

Figure 8: PIC: Influence of the dimension.

nately, I could not simulate large enough networks to see any significant changes in my simulation results. The authors of [2] recommend a hybrid solution, choosing half the landmarks from the near nodes and the other half randomly.

One additional observation I have to make about this algorithm is that it often falls into local minima. As one run of the algorithm is really fast on a single node, I would recommend running it some more times to get a better approximation of the position. This method is recommended in the paper too as one of the methods to determine the nearest landmarks (read in 3.2).

4.3 Big-Bang vs. PIC Algorithm

The main difference between these too algorithms is that the Big-Bang algorithm estimates the positions of all the nodes at once, on a central machine with complete information about the network. The PIC algorithm on the other hand computes the position of a single node, locally with very restricted information about the network. The results in the PIC paper were obtained using network graphs with more than 50′000 nodes and only 16 landmarks. This main difference implies the following two observations.

The domain of application of the Big-Bang algorithm is on network graphs between 30 and 1000 nodes (more needs too much time). The PIC algorithm is thought to be used on really large networks.

The precision of the Big-Bang algorithm is higher than the PIC algorithm's precision, this is due to the fact that it has a global view of the system and tries to find a global optimum. The PIC algorithm only estimates the position of a node given some landmarks, so it will always only find a local optimum.

The algorithm of choice depends of the available information. If one can use the Big-Bang algorithm (small number of nodes, complete information), one should use it, because it is much more precise than the PIC algorithm.

5 Possible Improvements

5.1 Bigbang Algorithm

It is difficult to make many improvements to this algorithm on the level of the complexity of the algorithm. Computing as much as possible in advance, like the shortest paths between each pair of nodes is a necessity.

This algorithm can very easily be parallelized, by partitioning the problem. Assume we have k processors. The problem consists of two steps:

1. We need to compute the new forces that apply to the nodes. In order to do that, we can partition the edges in E_1, ..., E_k groups. For each edge e in E_i, processor i computes the force that has to be applied on the particles at the end of the spring. This force is added to the current force applied at each node.
2. In the second step, the new positions needs to be computed. Here, we partition the node in N_1, ..., N_k groups. For each node $n \in N_i$, processor i computes the new position of the node n.

As the number of nodes and edges is the same over the complete algorithm, these partitions only need to be defined once. In chapter 3.1, I said that the bigbang algorithm had to be computed centrally. This seems like a contradiction to the parallelization just presented, but for growing complexity of the network, more and more data is needed at each node, which increases network traffic if the algorithm is computed in a distributed manner. However, on a shared memory machine or on a dual-core processor machine, this is not a problem and parallelizing the algorithm is not a problem.

5.2 PIC Algorithm

In [2] there is no description of how to choose the reference points in a "good" way. One should remember that the more reference points we have whose location is known exactly or precisely enough, the better our later approximation will be. To give a location to the reference points, the Bigbang algorithm is a good choice, as it is very precise for nearly complete graphs and we can assume that the reference points (in real networks) are directly linked together or linked together with nearly direct paths. Furthermore, their number should be rather small.

6 Conclusions

It is difficult to find a conclusion that can apply to both algorithms at the same time, however, both algorithms still have a striking similarity in the fact that they both need a lot of information in order to be able to find a node's position in a network. The PIC algorithm is probably a step in the right direction, but the lack of infrastructure and common protocols make it difficult to use them until now.

In both papers, the authors forgot to tell about the details and problems of a real implementation, like the numerical instabilities in the Big-Bang algorithm and the local minima problem for the PIC algorithm.

It was a very interesting term project.

A Numerical Methods

A.1 Euler's Method

As this chapter's main purpose is to describe the integration method used in the Bigbang algorithm, we will assume that we always want to solve the differential equation for the position, knowing the force that acts on the particle. The main criterium for chosing an integration method for the Big-Bang algorithm is that we do not want to evaluate the force more than once per step as this its evaluation costs $O(n^2)$, where n is the number of nodes.

In the following, $\mathbf{x}$ denotes the position, $\mathbf{v} = \dot{\mathbf{x}}$ and $\frac{\mathbf{F}}{m} = \mathbf{a} = \ddot{\mathbf{x}}$ the first resp. second derivative of it. $\mathbf{F}$ denotes the force and m the mass of a point. $\mathbf{x}$, $\mathbf{v}$, $\mathbf{a}$ and $\mathbf{F}$ are vectors. h is the integration step size.

The Euler method is a very simple and thus popular method. However, it has a large amount of truncation errors and it is neither stable nor accurate for many pratical examples. In the computation of the Bigbang algorithm, the main problem is the start, where the forces are the largest.

The computation for the Euler method is the implementation of the following equation:

$$\begin{aligned} \mathbf{x}_{i+1} &= \mathbf{x}_i + h \cdot \dot{\mathbf{x}}_i + \frac{h^2}{2} \cdot \ddot{\mathbf{x}}_i \\ &= \mathbf{x}_i + h \cdot \dot{\mathbf{x}}_i + \frac{h^2}{2} \cdot \frac{\mathbf{F}_i}{m} \end{aligned} \tag{A.1}$$

$$\dot{\mathbf{x}}_{i+1} = \dot{\mathbf{x}}_i + h \cdot \frac{\mathbf{F}_i}{m} \tag{A.2}$$

F has to be computed in each iteration, using the formulas defined in the algorithm.

A.2 Downhill Simplex

The downhill simplex is a simple and fast method for multidimensional optimization. It doesn't require derivatives of the function to minimize. In [4] this method is referred to as *the best method if the figure of merit is "get something working fast"*.

The simplex method's name is due to the fact that this method uses a simplex (a geometrical figure which in d dimensions has $d + 1$ points). The algorithm then moves the simplex through the multidimensional space by transforming it in each step. It can be shrinked, expanded or mirrored in order to approach all the $d + 1$ points of the simplex to the local optimum.

You can find a more detailed explanation and the algorithm in [4].

B Difficulties

B.1 Implementation Difficulties

The main difficulties encoutered in the implementation phase where:

- The missing explanation on how to place the reference points for the PIC algorithms. This was solved by using the Bigbang algorithms to determine good positions in higher dimensional space for the reference points.
- The overall architecture of the program, which should allow other methods of network generation or loading and other algorithms for computation.

B.2 Algorithmic Difficulties

The biggest problem here is choosing the reference node in an optimal way for the PIC algorithm and placing them. When approximating a node's position, a number of reference nodes is needed too, which can be small or large, choosing close nodes or nodes far away. It is not really clear yet, what exactly should be the criteria for the choices here, furthermore, determining closest neighbouring reference nodes is not trivial and is described in chapter 3 of [2].

The Bigbang algorithm doesn't have any special difficulties, it is a very straight forward algorithm.

C Program Description and Guide

Using the previously described framework, different inputs where programmed, GUI and console. Furthermore, for testing purposes and as main part of this term project, the two algorithms from [2] and [5] where implemented.

To use the program, you always have to define a network in the first step, either reading it in from a file or generating a random network. In a second step, you have to choose an algorithm, set it's parameters and apply it. The third step is to collect the statistics, which can be viewed on the screen or saved to a CSV file for further analysis.

In the following sections, you'll find a guide describing both, the GUI and the console input methods.

C.1 Using the GUI interface

The GUI has only very basic functions for the moment, and is mainly for visualization purposes. You can generate networks or read them from file and have them visualized. The GUI application also permits looking at evaluated distances between single nodes, which is not possible in the console version. In figure 9 you can see a screenshot of the application.

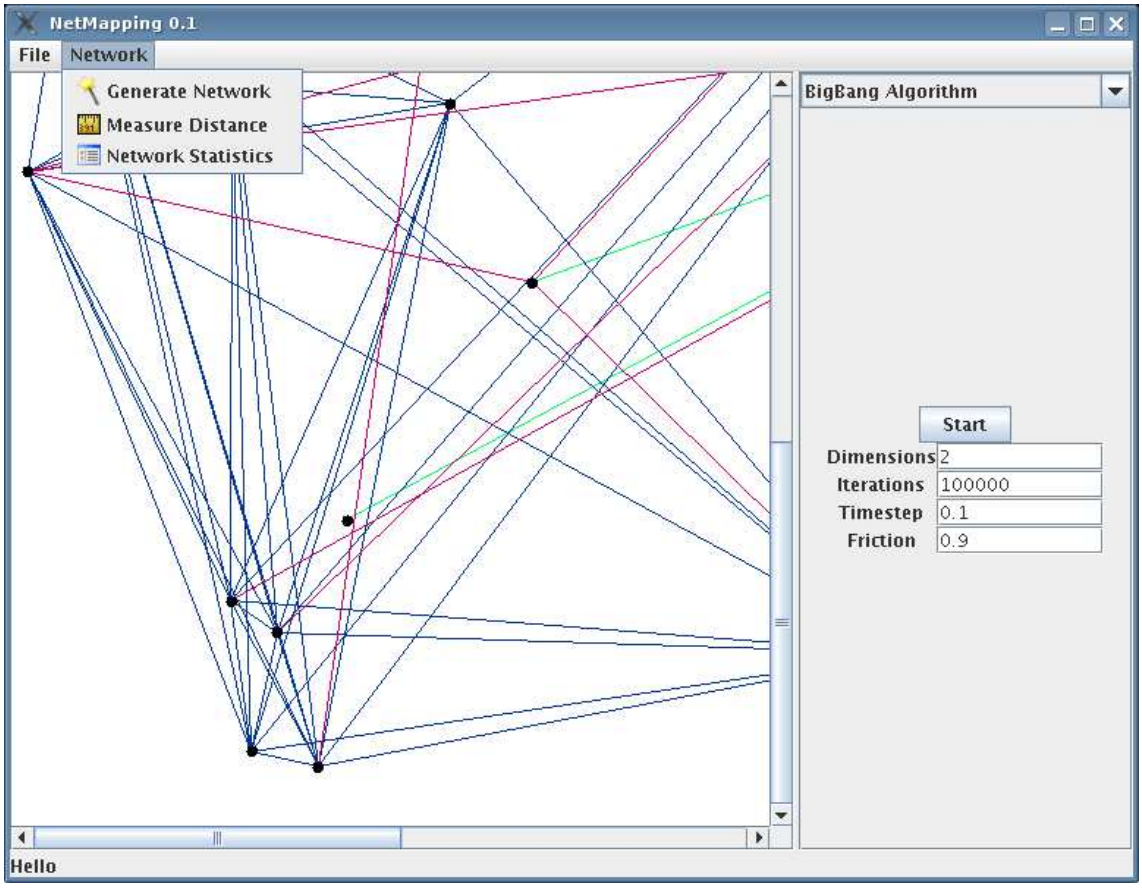

Figure 9: A screenshot of the GUI application.

The file menu contains the choices to load, save networks and save statistics. The network menu contains the choices for network generation, measuring distances and showing statistics.

The different fields should be intuitive enough to be used without further explanations.

C.2 Using the console interface

The console mode is enabled using the "–console" option when starting the program. The console interface is probably for many purposes the preferate way to work, as it allows batch jobs. You can use the "-f filename" option to specify a file to read the commands from and have them executed.

Here is a typical console session:

```
Welcome to NetMap command line.
```

```
->  generate 2 10 20 0.01 0.1 0.9
Options set to:
NetOptions:
# of layers: 2 # of rows: 3
Layer 1: 10 Nodes [0, 0, 0]
Layer 2: 20 Nodes [0, 0, 0]
Links from 1 1: p=0.01 color=[0, 0, 0]
Links from 1 2: p=0.1 color=[0, 0, 0]
Links from 2 2: p=0.9 color=[0, 0, 0]

generating network...

->  algo bigbang 0.9 0.1 1000 2
Bigbang algorithm successfully executed.
->  print netstats
Network statistics
Totals: N: 30 -- L: 192
Layer: N: 10 -- P: 0.01 -- L: 1
Layer: N: 20 -- P: 0.9 -- L: 168
From: 0 To: 1 : N:23 -- P: 0.1

->  print algostats
Total squared error: 80.2352034700344
```

C.2.1 Console application commands

generate generates a network with the given options. The parameters are the following

1. the number of layers in the network.
2. the nodes in each layer space separated.
3. the probability of a link between the nodes of the different layers, space separated. In the following order: $1->1$ $1->2$ ... $2->2$ ... $3->3$ ...

algo runs the algorithm specified as first parameter. The parameters of the algorithm follow. For more information, use *help algo*.

print can be used to show different statistics on the screen, like the number of nodes, links, ... using the parameter *netstats* or the mean squared error, using *algostats*. For more information use *help print*.

help without parameters displays all the available commands. With as a parameter some command, displays additional information to the given command.

D Music Network

To test the implementation of these algorithms on a real life network, Prof. Wattenhofer suggested using them to find closely related artists in an artist network. The reference page was *www.allmusic.com*, which has a huge list of artists, with their relations to other artists and groups.

I started by downloading the pages, parsing them and creating a network out of the data. The nodes in the network were the artists and the edges were created using relations like *played in group*, *influenced by*, ... These edges could be weighted according to some user specific criteria/preferences.

Unfortunately, my multi-threaded download program was too fast and soon all the IPs I was trying to download from where blocked by the site. This was the end of that part of the project. We asked the maintainers of the site for the database, unfortunately, this did not succeeded either.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms.* MIT Press, Cambridge, MA, USA, 2001.

[2] Costa, M., Castro, M., Rowstron, A. I. T., and Key, P. B. PIC: Practical internet coordinates for distance estimation. In *24th International Conference on Distributed Computing Systems (24th ICDCS'2004)* (Tokyo, Japan, Mar. 2004), IEEE Computer Society, pp. 178–187.

[3] Lim, H., Hou, J. C., and Choi, C.-H. Constructing internet coordinate system based on delay measurement. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2003), ACM Press, pp. 129–142.

[4] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. *Numerical recipes in C: the art of scientific computing.* Cambridge University Press, New York, NY, USA, 1988.

[5] Shavitt, Y., and Tankel, T. Big-bang simulation for embedding network distances in euclidean space. *IEEE/ACM Trans. Netw. 12*, 6 (2004), 993–1006.

[6] Tangmunarunkit, H., Govindan, R., Jamin, S., Shenker, S., and Willinger, W. Network topology generators: degree-based vs. structural. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2002), ACM Press, pp. 147–159.