Graph Layout Algorithms in Small Wide World

Joel R. Voss

Abstract

Small Wide World is open source graph visualization software designed to make writing algorithms fast and easy to test. It is written in Python, JavaScript, and C++ to combine the ease of programming of scripting languages and the speed of native languages. Many algorithms currently exist to layout graphs, but they come up short in various use cases. Small Wide World provides a handful of cheap layout algorithms as well as local and global minimization algorithms to improve the quality of graph layout.

Contents

1 Small Wide World

1.1 Introduction

Small Wide World presents an effective method of algorithm development and analysis for graph visualization. It was designed to make writing algorithms fast and easy to test. Many algorithms currently exist to layout graphs, but they come up short in various use cases. Small Wide World provides a handful of cheap layout algorithms as well as local and global minimization algorithms (L-BFGS-B, Metropolis Monte-carlo, and basin-hopping) to improve the quality of graph layout. As in literature, a graph is a set of vertices V connected by a set of edges E [1]. This paper uses the term node in place of verticies almost always because it was used in Dijkstra's 1959 paper on graphs [2] and is in common parlance. Small Wide World also uses the term network to describe the actual reason for creating a graph (i.e. social network) and the term map to describe the frame and the graph together.

Small Wide World is extensible and exists in its current form because writing and testing algorithms is made easy. The later algorithms described in this paper took less than 1 day to write. Debugging, testing, and evaluating the algorithms took much more time than getting a first prototype. This ease of prototyping will allow machine learning techniques to improve the state of the art in graph layout. These advances if written generally enough may be portable to molecular modeling as well as other similar problems.

Molecular modeling is more concerned with accurate models because many proteins will minimize when an appropriate model is chosen and global minimization is applied [3][4]. This is a reasonably fast and simple method, but could be improved. Local minimization runtime can be improved by providing values closer to the global minima. This value is shared with graph minimization, in fact graph minimization doesn't care about accurate models because there are often local minima that satisfy the requirements of the user. Since global minimization is far slower than local minimization, graphs can be though of as an easier problem than molecular modeling. The main reason for graphs with a large number of nodes being ugly is that global optimization is currently not solvable in polynomial time. If users were more patient and electricity was free, algorithms could be employed to ensure high quality graphs over a long period of time. Another reason for ugly graphs is that dynamic layout algorithms and local minimization algorithms are slow when working with a large number of nodes. Algorithms written for graph layout address the speed issue but at the cost of quality. If an algorithm was designed that was fast and also high quality (low potential energy computed by an effective cost function), graph layout would be a solved problem. This paper intends to start the work toward that goal by solving a large subset of small graphs (<100 nodes) and provide a library to make incremental progress on the complete set of graphs.

1.2 The Small Wide World Potential ($V_{SmallWideWorld}$)

Small Wide World uses a potential energy calculation which was designed for molecular modeling because it is intended to educate users how molecules look in their optimized state [5][6]. In molecular modeling literature, this function is referred to as a force field. The potential energy calculation works well for graphs in many respects. It prefers bonds angles to be as spread out as possible and it prefers non-bonded nodes at a distance r_0 (which is configurable, but set to 40 pixels in Small Wide World by default). As bonded or non-bonded nodes get closer, the potential becomes higher and beyond the sweet spot b_0 (configurable also, but set to 40 pixels in Small Wide World), potentials also grow rapidly. Non-bonded nodes have very little penalty being far away while bonded nodes have a very high potential far away from the ideal bond length. Bonded potentials are handled by the harmonic potential (a parabola with minimum at $b = b_0$ seen in Figure 1) [6]. Non-bonded potentials are handled by the Lennard-Jones potential [6]. The Lennard-Jones potential has minima of $-scale\_factor$ at r=r_0 and curves toward infinity as it approaches 0. Figure 2 shows the Lennard Jones Potential as found in Small Wide World. In Small Wide World, $scale\_factor$ is configurable, but is set to 0.055. The Lennard-Jones potential crosses 0 at 35.636 pixels. It is possible that a different potential would produce more reasonable graphs, but that is a research project for the future. One of the most important parts of the potential is the speed. The Small Wide World potential is approximately $O(n^2)$ operations due to the non-bonded potential. The Small Wide World potential was run on a set of random graphs and the timing was measured. With x being the number of nodes in the graph, a fit to $Ax^2 + Bx$ resulted in parameters A = 5.0e-8 and B = 2.6e-6 on a single core Intel Core i7 3700K @3.5GHz. A graph can be found in Figure 3. For clarity, these results predict that the Small Wide World potential should take 1 second for a graph with 4448 nodes. Due to overhead, the Small Wide World performs worse on such a graph (~1.24 seconds).

Harmonic Potential

$$V_H(b) = A_H * (b - b_0)^2 \tag{1}$$

Angular Harmonic Potential

$$V_{AH}(\theta) = A_{AH} * (\theta - \theta_0)^2 \tag{2}$$

Lennard-Jones 6-12 Potential

$$A_{LJ} = 2 * (r_0^6) * scale\_factor \tag{3}$$

$$B_{LJ} = (r_0^{12}) * scale\_factor \tag{4}$$

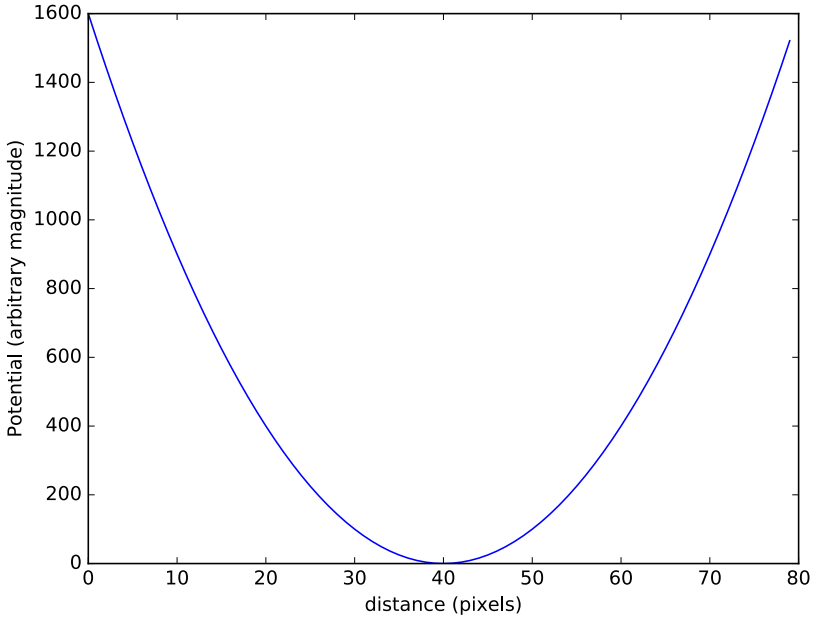

Figure 1: Harmonic Potential

$$V_{LJ}(r) = -A_{LJ} * (\frac{1}{r^6}) + B_{LJ} * (\frac{1}{r^{12}}) \tag{5}$$

$$A_{LJ} * (\frac{1}{root^6}) = B_{LJ} * (\frac{1}{root^{12}}) \tag{6}$$

$$\frac{A_{LJ}}{B_{LJ}} = \frac{1}{root^6} \tag{7}$$

$$root^6 = \frac{B_{LJ}}{A_{LJ}} \tag{8}$$

$$root^6 = \frac{(r_0^{12}) * scale\_factor}{2 * (r_0^6) * scale\_factor} \tag{9}$$

$$root^6 = 0.5 * r_0^6 \tag{10}$$

$$root = \sqrt[6]{0.5} * r_0 \tag{11}$$

$$root = 35.6 \tag{12}$$

Equations 6-12 give us a value for where the $V_{LJ}(r)$ equation crosses zero when $r_0 = 40$.

```
gnuplot> f(x) = A*x*x + B*x
gnuplot> A=1
gnuplot> B=1
gnuplot> fit f(x) 'sww_speed1.dat' using 1:3 via A, B
A               = 5.01617e-08      +/- 1.23e-10     (0.2452%)
B               = 2.61209e-06      +/- 3.548e-08    (1.358%)
```

The Small Wide World potential is a good cost function for a minimization algorithm. Drawbacks include speed, the existence of a large number of local minima on complex graphs, and the deviation from realistic potential outside a small range. Advantages include continuous functions inside the necessary range, similarity to implementations of molecular potentials in molecular modeling software, reasonably easy derivative computation, guaranteed solutions for simple graphs, no penalty for unconnected nodes beyond the minimum distance, finite values over a large space represented with 32-bit or 64-bit floats, and ease of programming and understanding.

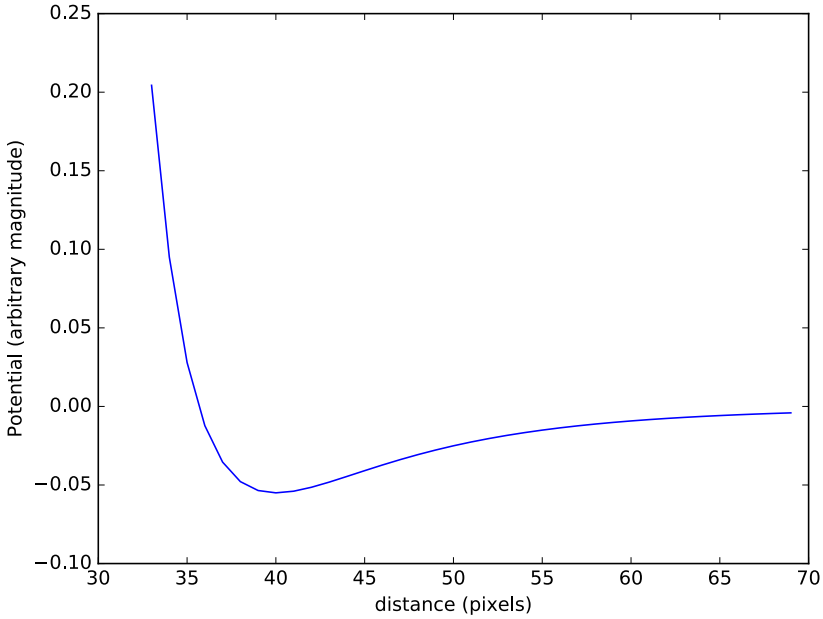

Figure 2: Lennard-Jones 6-12 Potential

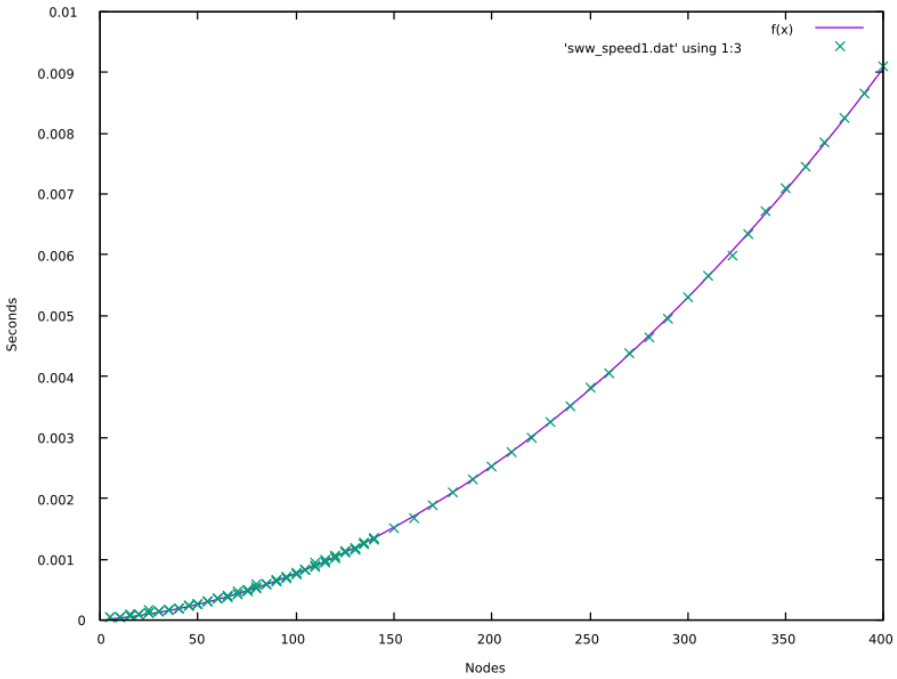

Figure 3: $V_{SmallWideWorld}$ Speed

A following paper in development will discuss the derivitive of the Small Wide World potential and its possibility to improve graph layout algorithms. Unfortunately, solving for minima in large graphs is a significantly more difficult problem than minimization of small graphs.

This paper concerns six cheap algorithms for 2-dimensional graph layout using a potential function. The goal of these algorithms is to produce low potential configurations at a lower cost than known algorithms. All six algorithms are deterministic and not guaranteed to find optimal solutions. Instead they intend to provide good solutions to a common subset of graphs in a very short amount of time and to provide lower potentials on the full set of graphs. Their correctness or incorrectness can be easily checked by potential energy computation. I am testing these algorithms against naive algorithms[1] as well as popular and ergodic algorithms (algorithms guaranteed to find global minima given enough time) to ensure that the algorithms perform better than slower and faster algorithms for a great many common graphs. These algorithms fit a different niche than local optimization algorithms and global optimization algorithms, but can be made more similar at a cost. The main benefit of these algorithms is that unlike global optimization algorithms, they can be run in polynomial time and are fast. The first algorithm is Branch sort, the second algorithm is Loop sort, the third is Grid sort, the fourth is Lattice sort, the fifth is Spiral sort, and the last is Right Loop sort. Right Loop sort is the best in most cases but Loop sort, Grid sort, Lattice sort, Spiral sort, and Branch sort produce lower potentials for certain graphs. In the cases of Lattice sort and Spiral sort, both are naive algorithms that were not expected to work well, but each produce the best layout more often than Branch sort. This is likely due to Branch sort's weakness on large complex graphs (12-210 nodes, 13-270 edges) on which these algorithms are being tested. Testing these algorithms against small or simple graphs would be far less interesting because cheap algorithms can provide optimal solutions as does local minimization with random initial placement.

1.3 Branch Sort

Branch sort is a simple algorithm in Small Wide World which picks locations for nodes in a graph assuming that all nodes are connected in a star configuration with no loops. To achieve this, it counts a node's connections and picks locations for neighboring nodes using trigonometry, creating regular stars. To decide which node to place first, Branch sort uses a simple constraint that one neighboring node be known at the time of its placement. For loops, this falls through, so branch has a second mode where it accepts placement if more neighbors are not known. This works well with many simple graphs. Larger graphs and certain simple graphs have overlaps that cause the graphs to be unoptimized. This

[1] My definition of a naive algorithm in this context may differ from other definitions, it is meant to convey that the algorithm does not take into account connections during its operation except for the backbone.

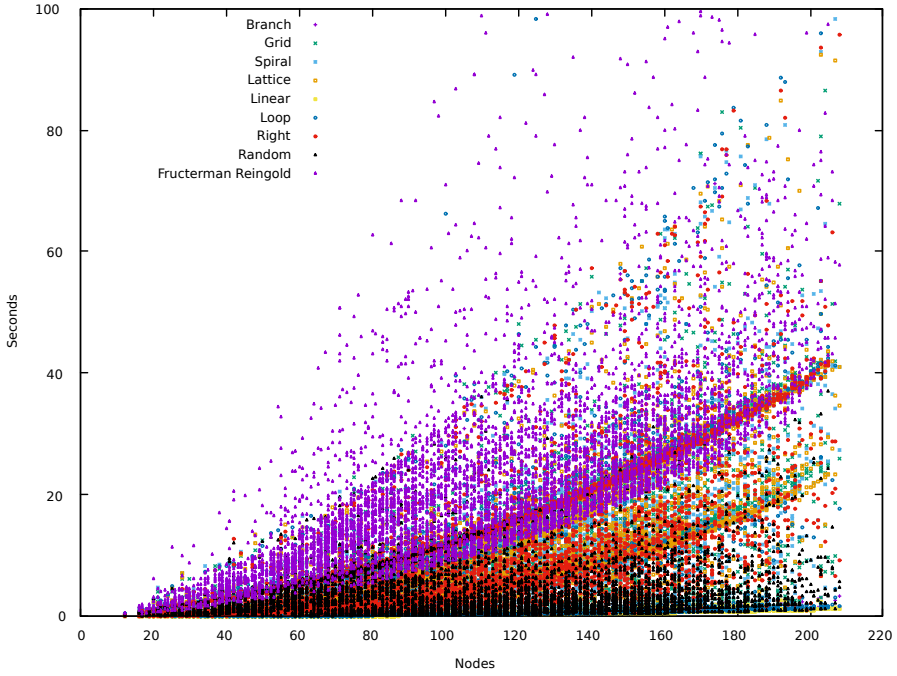

Figure 4: Algorithm Speed

causes branch to fail local optimization. One workaround used sparingly during testing is to add a small amount of randomization to the graph so that no node lays on top of one another. Besides this weakness, graphs with loops are poorly optimized by the Branch sort algorithm. One of the main benefits of branch sort is its speed. Since it is approximately 500-4000 times faster than local minimization with L-BFGS-B (using SciPy which uses Fortran underneath and C++)[7][8][9], it can be run with practically no downside.[10][2]

In tests of small graphs, Branch sort improves the speed of L-BFGS-B local minimization considerably over random or naive placement of nodes (such as linear, lattice, or spiral placement) in many cases. For larger graphs, Branch sort is less useful but still provides some benefit due to nodes on the backbone being placed at the correct distance from one another. Graphs in this paper that discuss speed will be comparing the speed of local optimization using L-BFGS-B after the algorithm was used. A lower speed does not always represent a better algorithm, it just shows that the algorithm produced a configuration close to the local minima. Similarly, tables and graphs that compare potential will often compare potential after local minimization to show the quality of the minima found.

Algorithm	comparison	Comparator	Result
Branch	better than	Lattice	3.717%
Branch	2x better than	Lattice	2.834%
Branch	3x better than	Lattice	2.200%
Branch	4x better than	Lattice	1.934%
Branch	5x better than	Lattice	1.784%
Branch	6x better than	Lattice	1.717%
Lattice	better than	Branch	96.283%
Lattice	2x better than	Branch	95.366%
Lattice	3x better than	Branch	94.882%
Lattice	4x better than	Branch	94.616%
Lattice	5x better than	Branch	94.516%
Lattice	6x better than	Branch	94.416%

Branch sort will only place a node when it knows at least one of its neighbor's positions except for the first node. This decision makes it possible to set a group of nodes to known and let Branch sort solve branches off that group. It doesn't take into account neighbors nodes, so it will often pick incorrectly which direction to place a branch. This appears to be inevitable without sacrificing speed, so modifications to this algorithm will create different algorithms that trade speed for accuracy and optimization. When thinking about graph layout, it is often more important that the algorithm be fast than accurate since basin-

[2]Local minimization can be much slower than Branch sort or possibly just as fast if the initial position is close to a minima. In reality, random initial position causes local minimization to be orders of magnitude slower than Branch sort, on the order of 1000-1000000 times slower.

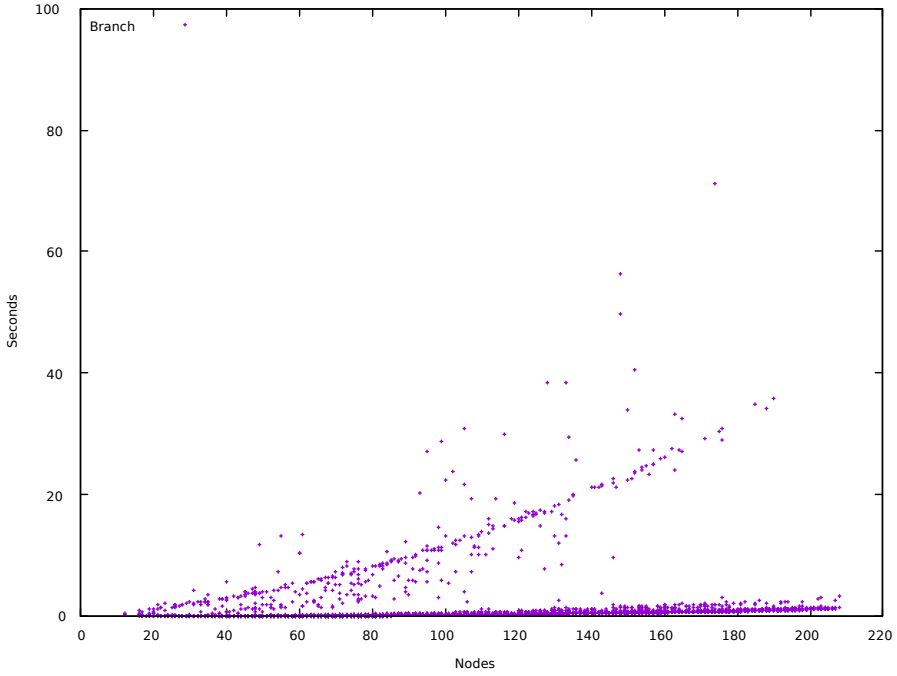

Figure 5: Branch Sort Speed

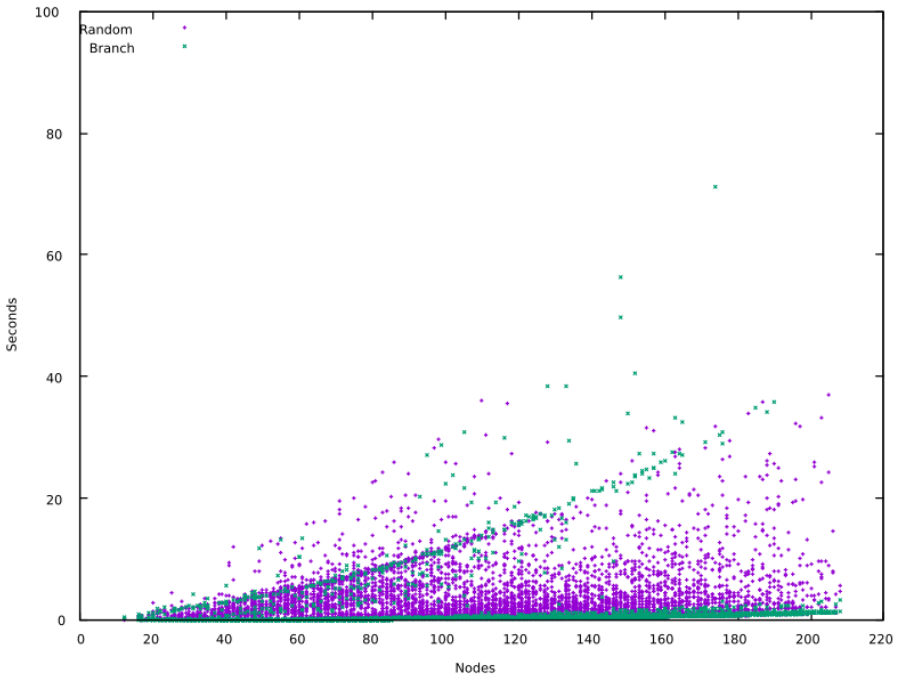

Figure 6: Branch Sort vs Random Local Minimization Time

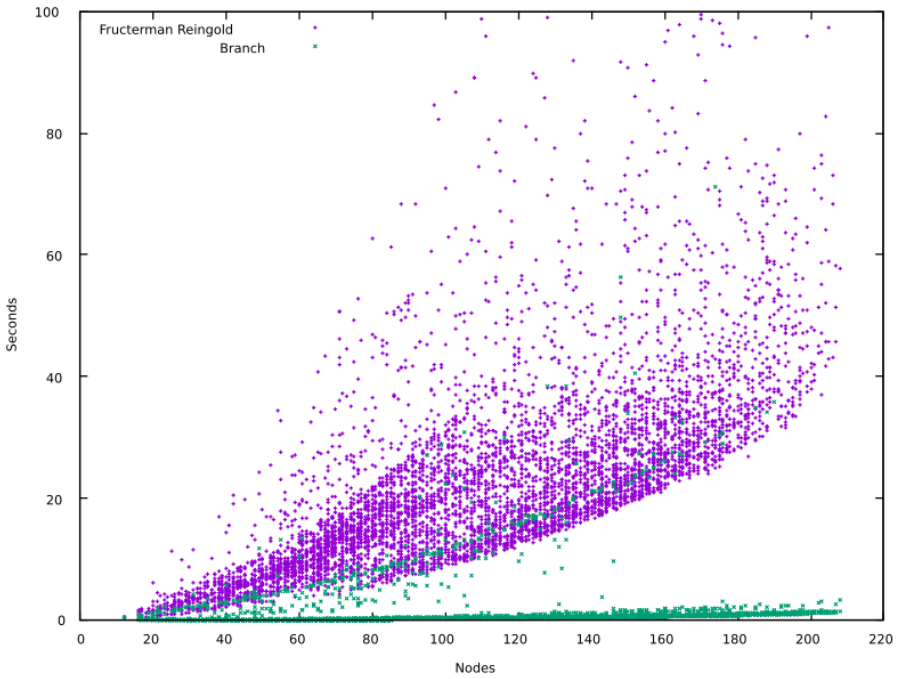

Figure 7: Branch Sort vs Fruchterman-Reingold Local Minimization Time

hopping with L-BFGS-B can solve many problems accurately given enough time [11][12][13][14]. As mentioned before, the closer a configuration is to the global minimum, the quicker local minimization (i.e. L-BFGS-B) will find it. Thus it is only important to save CPU time over the basin-hopping algorithm if your user is waiting for it or if you are dealing with a large number of inputs (such as machine learning, molecular modeling, and brute force graph research). The end goal is certainly to find the global minimum (or a point on a slope toward it) quickly and in polynomial time, but this goal is surprisingly difficult for complex graphs. It is trivial for simple graphs. However, social networks that have dense networks can also benefit from branch sort, see Figure 13. While Grid sort will normally perform better than Branch sort, Branch sort provides a lower potential which speeds up local minimization.

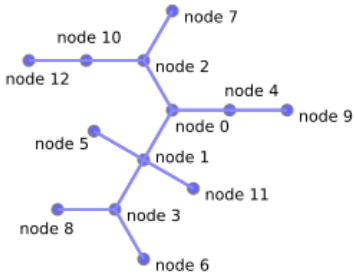

Figure 8: Graph with 13 nodes, 12 edges Branch sort local minimum

It is very common for graphs to have a large number of inputs, but to also have an overarching theme. Social networks are often very densely connected, which means Branch sort and Loop sort are poor choices for optimization. Right Loop sort may be a reasonable choice for social networks. Networks based on computer data networks are not often looped, so branch sort makes more sense. Networks based on x86 assembly branching and looping are very regular, so right loop and specialized algorithms will work better than Branch sort or Loop sort. Certain logic graphs (e.g. state machine graphs and dependency graphs)

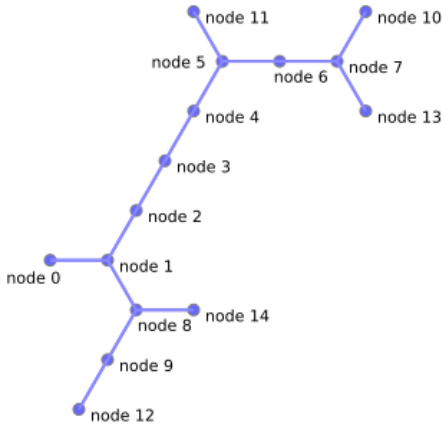

Figure 9: Graph with 15 nodes, 14 edges Branch sort local minimum

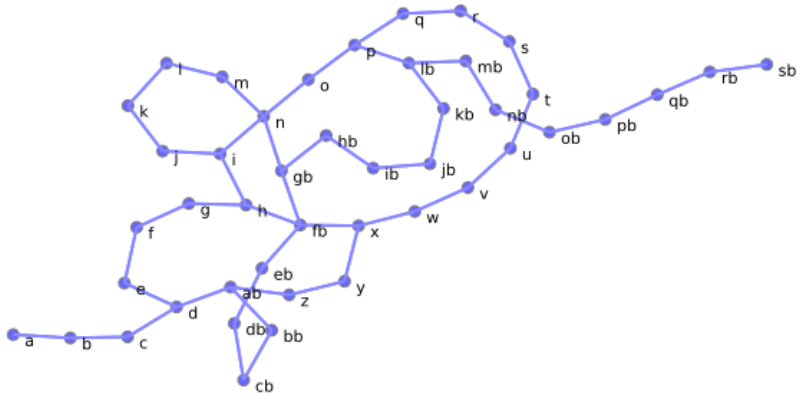

Figure 10: Graph with 45 nodes, 50 edges Branch sort local minimum

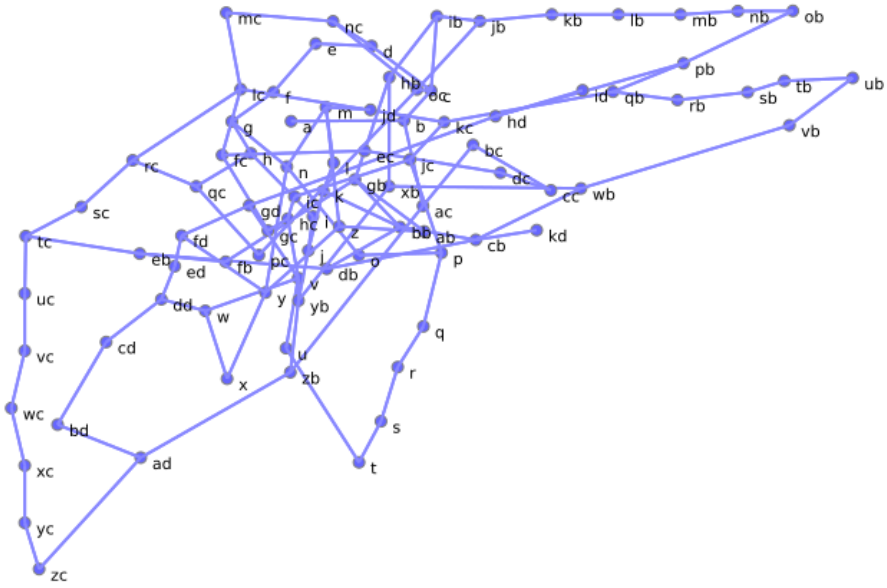

Figure 11: Graph with 89 nodes, 116 edges Branch sort near local minimum

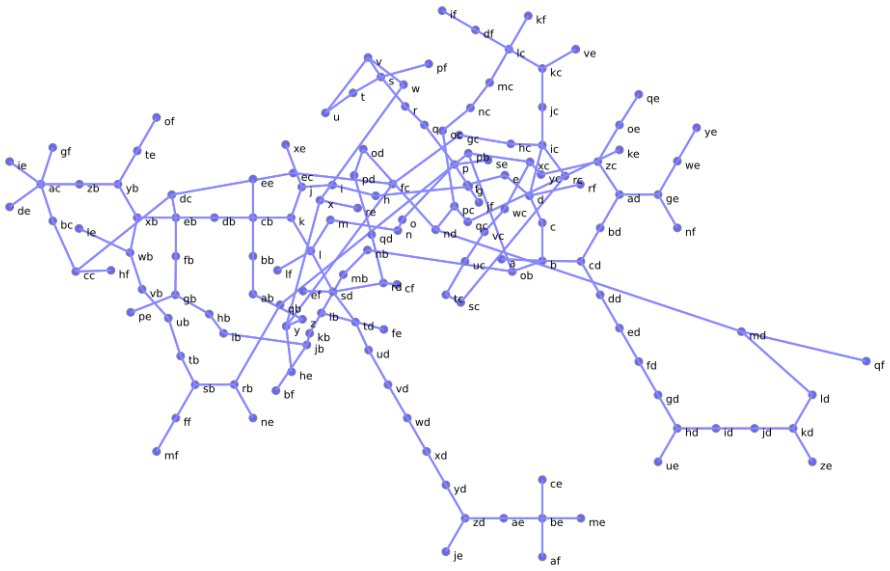

Figure 12: Graph with 148 nodes, 159 edges Branch sort near local minimum

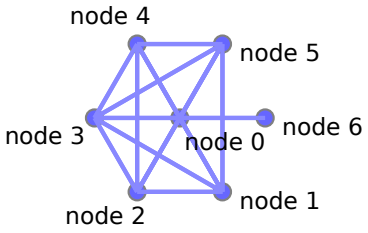

Figure 13: Dense Graph with 7 nodes, 16 edges Branch sort

are best handled by a unidirectional optimized graph algorithms like Right Loop sort. Constraint graphs can also be solved with Right Loop sort depending on its complexity. Loop sort can produce reasonable solutions for a certain set of simple flow graphs and constraint graphs. Graphviz's `dot` has been commonly used for x86 assembly graphs, logic graphs, and social network graphs, and uses an algorithm similar to Right Loop sort. Branch sort makes more sense for chemical bond networks such as hydrocarbons. Loop sort makes more sense for chemical bond networks such as proteins and aromatic molecules. While Loop sort is not yet ready to take on complex structures, more work will eventually be done to improve its quality. Since chemical bond networks are almost always 3-dimensional, 3d versions of the algorithms will have to be developed to move past the initial proof of concept phase which deals with 2d graphs.

1.3.1 Metropolis

The Metropolis Monte-carlo algorithm is a naive global optimization algorithm [15][16]. It has been used for decades to minimize chemical structures[17][18][19]. Its main weakness is that it requires unnecessarily large amount of computational power to get past high potential barriers near local minima. Small Wide World provides an intuitive interface to run Metropolis Monte-carlo. Attempting Metropolis Monte-carlo algorithm for 10 seconds (15k rounds on a single core with 55 nodes and 59 edges, 19k rounds with 45 nodes and 59 edges) provides a good comparison for the efficiency of the algorithm. While I haven't tuned the Metropolis Monte-carlo algorithm for this problem, a factor of 1000 in time is probably enough to provide a reasonable margin of error in my test with 12-70 nodes and 12-80 edges. 644 graphs were tested.

On average, Branch sort took 0.002 seconds.
On average, Metropolis took 10.004 seconds (5000 times longer).

Algorithm	comparison	Comparator	Result
Branch	better than	Metropolis	0.311%

Algorithm	comparison	Comparator	Result
Branch	2x better than	Metropolis	0.0%
Metropolis	better than	Branch	99.689%
Metropolis	2x better than	Branch	98.137%
Metropolis	3x better than	Branch	96.429%
Metropolis	4x better than	Branch	94.410%
Metropolis	5x better than	Branch	93.012%
Metropolis	6x better than	Branch	91.460%
Metropolis	10x better than	Branch	88.509%
Metropolis	100x better than	Branch	78.416%
Metropolis	200x better than	Branch	77.484%
Metropolis	1000x better than	Branch	76.553%
Metropolis	10000x better than	Branch	74.379%
Metropolis	100000x better than	Branch	71.739%
Metropolis	1000000x better than	Branch	66.770%
Metropolis	10000000x better than	Branch	61.180%
Metropolis	100000000x better than	Branch	56.211%

This graph shows that Metropolis for 10 seconds beats Branch significantly as you might expect if you have used Metropolis Monte-carlo and Branch sort. This test allows us to consider how much more effective it would be to use Metropolis than any of the algorithms presented here.

A limitation of Loop sort and Branch sort are that they have modes of failure that are far worse than 1 second of Metropolis Monte-carlo. This provides us with a reasonable solution: run a tuned Metropolis Monte-carlo algorithm on any result from Loop sort or Branch sort which is higher than a certain value. Branch being one of the least effective algorithms, it is an excellent example of Metropolis being used to improve the quality of a map given a layout algorithm as a starting point instead of random placement. In this second test, I am performing the same test but after Branch sort, I am running 0.2 seconds of Metropolis. This choice of timing allows us to make an assumption about a realistic scenario where maps must be rendered in less than a second. 488 graphs were tested.

On average, Branch sort + 0.2 seconds of Metropolis (BranchM) took 0.205 seconds.
On average, Metropolis took 10.004 seconds.

Algorithm	comparison	Comparator	Result
BranchM	better than	Metropolis	10.959%
BranchM	2x better than	Metropolis	2.055%
Metropolis	better than	BranchM	89.041%
Metropolis	2x better than	BranchM	56.164%
Metropolis	3x better than	BranchM	32.877%

Algorithm	comparison	Comparator	Result
Metropolis	4x better than	BranchM	13.014%
Metropolis	5x better than	BranchM	9.589%
Metropolis	6x better than	BranchM	5.479%
Metropolis	10x better than	BranchM	1.370%
Metropolis	100x better than	BranchM	0.000%

This experiment requires a control. To control properly, I randomized the output and ran Metropolis for 0.2 seconds. 285 graphs were tested.

Algorithm	comparison	Comparator	Result
Metropolis	better than	Metropolis(0.2)	100.000%
Metropolis	2x better than	Metropolis(0.2)	99.298%
Metropolis	3x better than	Metropolis(0.2)	97.544%
Metropolis	4x better than	Metropolis(0.2)	95.439%
Metropolis	5x better than	Metropolis(0.2)	89.123%
Metropolis	6x better than	Metropolis(0.2)	81.754%
Metropolis	10x better than	Metropolis(0.2)	55.439%
Metropolis	100x better than	Metropolis(0.2)	0.000%

This result shows that 10 seconds of Metropolis is still better than Branch + 0.2 seconds of Metropolis (BranchM) in 89% of cases, but surprisingly that bias falls off quickly after 2x. Comparing with the control which does not fall off rapidly until 10x, this shows that the Branch sort algorithm can improve the quality of Metropolis over random initial placement of nodes.

Metropolis Monte-carlo can be run for a long amount of time with no large detriment to the end result potential energy. While most Metropolis Monte-carlo solutions look worse than Loop sort and Branch sort solutions, potential is generally more valuable than aesthetics in graph layout. For many uses, aesthetics is more important than potential, so this should be considered a benefit of these algorithms.

1.4 Loop Sort

Loop sort is a more complex algorithm with the second best performance in the tests. It contains many constraint solving algorithms and uses concepts that humans employ when working with graphs containing loops. Its last step is to run the Branch sort on any nodes not handled by the Loop sort, which means that Branch sort is a prerequisite of Loop sort. Other algorithms could be substituted, but it must be capable of running on a subset of nodes using a set of givens to produce correct results. The first step of Loop sort is Linear

sort, the naive algorithm of placing all nodes on the X axis using the backbone. Linear sort is done cheaply with polar coordinates which has the drawback of placing some nodes in the same place. This organization of nodes on the x axis allows a simple function to group nodes together. The second step of Loop sort is to group nodes. This simple algorithm uses position on the x axis to decide whether a group can be made out of a subset of nodes in the graph. It does this by first querying the position of connected nodes. If there are no connected nodes to the left, it must be a left bracket (See Figure 14 for illustration of the grouping concept). If there are no connected nodes to the right, it must be a right bracket. If a node only has one connection, it is in a group by itself – a leaf node. If a node has only one connection to the left and multiple to the right, it is possibly a left bracket. We can check whether any nodes on its left (unconnected and connected) are connected with any nodes on its right. If any are connected, then it is part of a group. If none are connected, then it is a left bracket. If a node has only one connection to the right and multiple to the left, it is possibly a right bracket. Since right brackets aren't used in loop sort at the moment, we ignore this. In the future, it may be desirable to use right brackets for some purpose. For optimization, the current group is kept in memory to reduce the number of groups needed to be searched.

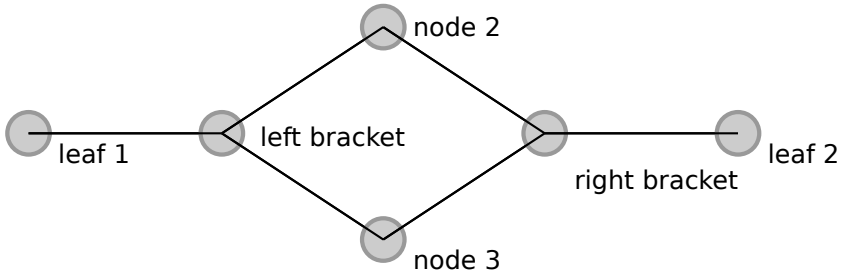

Figure 14: Grouping with Linear Sort Part 1

Loop sort is much slower and more complex than Branch sort, but provides much higher quality solutions with low potentials and high reliability. As its name suggests, Loop sort was designed to find small loops and create a regular polygon from the nodes. It does this by sorting the nodes linearally (based on their connections) and finding which nodes are groups. A group is defined by having only single connections to other nodes. This is explained in Figure 14 and 15. An example of a group would be {left bracket, node 2, node 3, right bracket} or any of the nodes in Figure 15. This works especially well on chemical-style networks because of their simplicity (most elements will not bond to more than 3 atoms, the carbon group being a notable exception: carbon, silicon, germanium, tin, and lead).

Loop sort is not yet fully tested and has several weaknesses. Loop sort is especially bad at dense networks due to bugs in routing, grouping, and basic design. Loop sort also doesn't work on many graphs with complex branching. As improvements are made, Loop sort may eventually be able to solve all manner of graphs, providing a certain solution to graph layout without any local optimization or long processing required. Loop sort is by far the slowest algorithm and performs far worse on large graphs owing to the numerous high complexity functions that are needed to deal with the complexity of all possible graphs.

Figure 16 and 17 show two complex graphs that Loop sort produced the lowest local minima. Both graphs are solvable by humans and global optimization software in a reasonable amount of time given the local minima found by Loop sort.

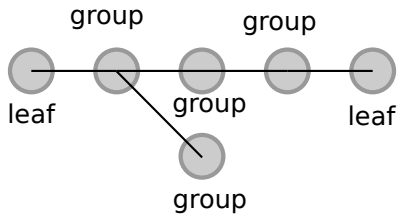

Figure 15: Grouping with Linear Sort Part 2

Algorithm	comparison	Comparator	Result
Loop	better than	Lattice	38.540%
Loop	2x better than	Lattice	29.838%
Loop	3x better than	Lattice	25.138%
Loop	4x better than	Lattice	21.870%
Loop	5x better than	Lattice	19.970%
Loop	6x better than	Lattice	18.370%
Lattice	better than	Loop	61.460%
Lattice	2x better than	Loop	55.743%
Lattice	3x better than	Loop	53.259%
Lattice	4x better than	Loop	52.125%
Lattice	5x better than	Loop	51.575%
Lattice	6x better than	Loop	51.125%

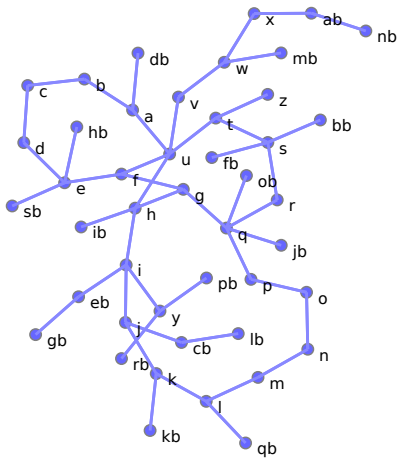

Figure 16: Graph with 45 nodes, 48 edges Loop Sort local minimum

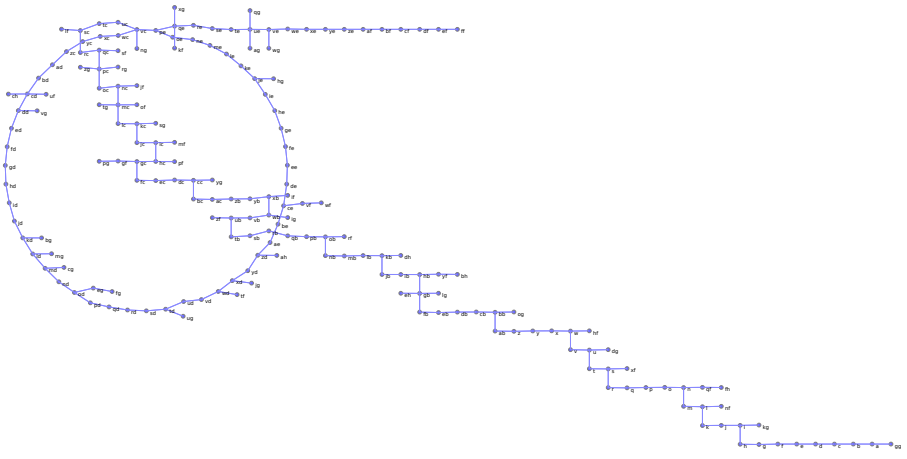

Figure 17: Graph with 188 nodes, 188 edges Loop Sort local minimum

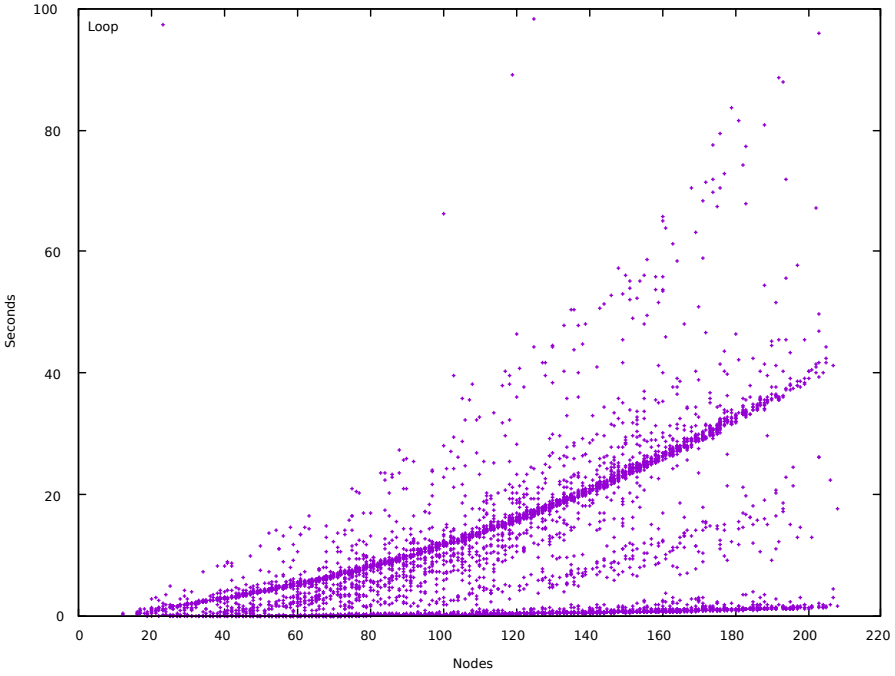

Figure 18: Loop Sort Speed

1.5 Grid Sort

Grid sort is a complex algorithm with the third best performance of the six algorithms. Unlike Loop sort, it does not depend on any other algorithm. Grid sort creates a 2-dimensional array (a grid) which is extensible in all 4 directions. The first node is put at (0, 0) and connected nodes are placed in neighboring cells. Like Branch sort, it uses simple facts about the graph to make good decisions on where to place nodes. Unlike Branch sort it can query whether a node can be placed without being too close to another node. This allows Grid sort to ensure that it never produces a graph with two nodes in the same cell. The most common problem with Grid sort are loops. To solve this, an expensive function called rearrangeGrid was created to reduce these large potentials. The function uses a cached version of the Small Wide World potential ($V_{SmallWideWorldCache}$) to pick the lowest potential given the grid. Improvements can be made to this algorithm to speed it up and to improve the quality of solutions. The current setup provides a very good result in many cases at a reasonable cost. Minimized Grid sort has the lowest potential in 9.8% of test graphs, more than Lattice sort, Spiral sort, and Branch sort.

One function that has not been fully realized is toGrid which can turn a graph into a grid. This makes it possible to improve many graphs cheaply. The reason this is not finished is because there doesn't yet exist an algorithm to effectively deal with nodes that are too close together. Instead, as a temporary fix the function places nodes on top of one another. An algorithm that correctly spread out nodes would be a significant benefit to Branch sort which commonly places nodes on top of each other. Once that modification was finished, the resulting code would look like:

```
map.sortBranch()
map.toGrid()
```

Algorithm	comparison	Comparator	Result
Grid	better than	Lattice	35.056%
Grid	2x better than	Lattice	18.103%
Grid	3x better than	Lattice	12.969%
Grid	4x better than	Lattice	10.518%
Grid	5x better than	Lattice	9.268%
Grid	6x better than	Lattice	8.435%
Lattice	better than	Grid	64.944%
Lattice	2x better than	Grid	44.824%
Lattice	3x better than	Grid	32.289%
Lattice	4x better than	Grid	24.671%
Lattice	5x better than	Grid	19.570%
Lattice	6x better than	Grid	16.469%

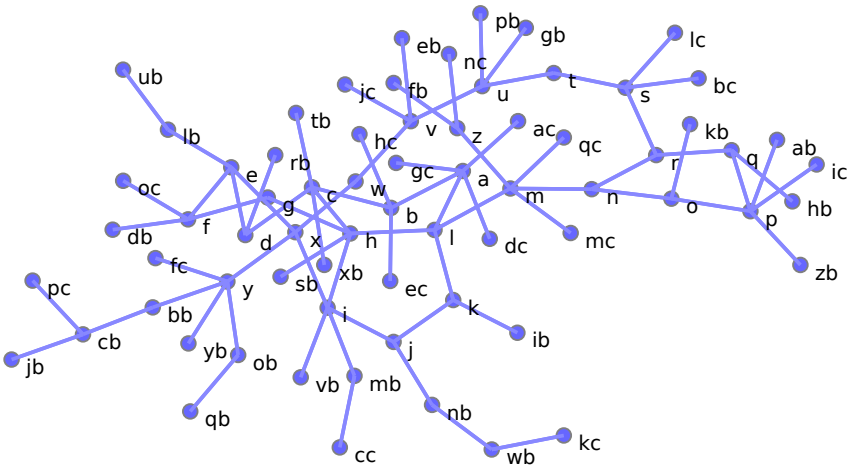

Figure 19: Graph with 69 nodes, 74 edges Grid sort near local minimum

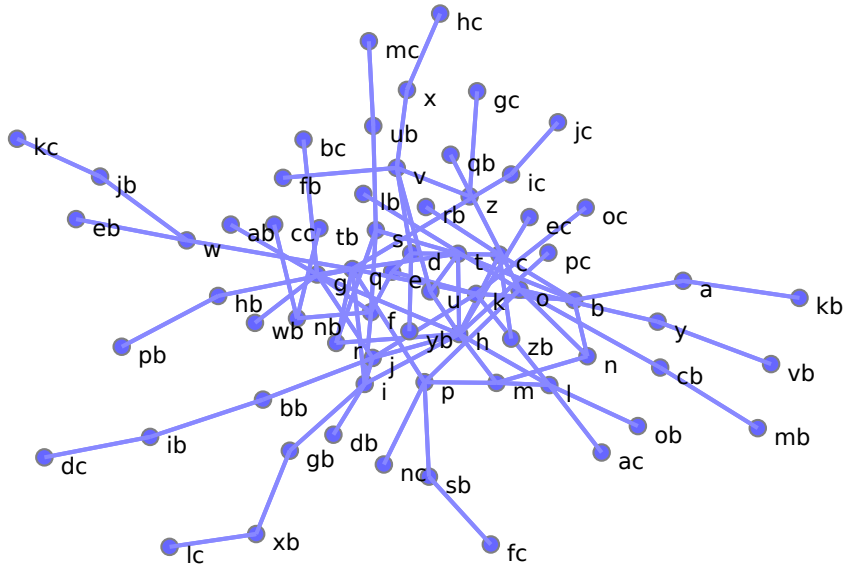

Figure 20: Graph with 68 nodes, 87 edges Grid sort near local minimum

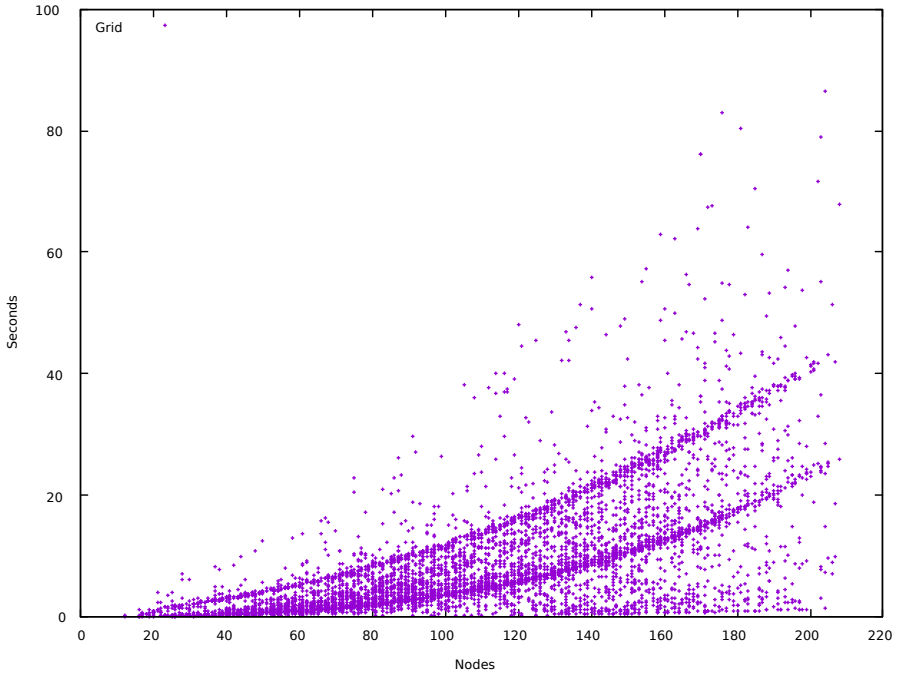

Figure 21: Grid Sort Speed

1.6 Lattice Sort

Lattice sort is a fast and simple algorithm with the fourth best performance of the six algorithms. This is an unexpected result because of its naivety. Lattice sort creates a grid similar to Grid sort, but offsets each row by $b_0 * \frac{\sqrt{3}}{2} = b_0 * 0.866$. This forms the basis of a lattice with vectors $(b_0, 0)$ and $(b_0 * 0.866, b_0)$. This makes bond lengths from a node to 6 adjacent cells equal to b_0. Grid sort's bond lengths are b_0 for 4 adjacent cells. This is a big deal because it reduces the potential of 2 diagonally adjacent nodes significantly. The potential of two connected nodes diagonally in lattice formation is 0. The potential of two connected nodes diagonally in grid formation is 247 $(1 * ((b_0 * \sqrt{2} - b_0)^2)$ where $b_0 = 40$). Another minor optimization in this algorithm is to layout nodes from top to bottom back and forth so that any backbone nodes are placed next to one another. Since finding the backbone of a graph can be done in linear time using Dijkstra's algorithm, this optimization is cheap. Lattice configuration mimics crystal formation which is ordered by lowest potential. Because of its speed, this algorithm is especially useful for many different graphs. Because it is naive graphs it produces are of very low quality because many lines are crossed. This occurs because nodes that are connected are not placed preferentially. Besides being aesthetically unpleasing and difficult to understand, this affects

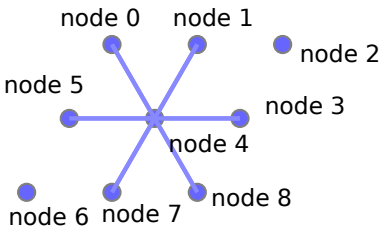

Figure 22: Lattice sort

the functionality. Because lines are crossed, it finds a large number of local minima, making local minimization far less effective. Figure 31 shows one graph where local minimization was effective on a result of lattice sort reducing the potential from 1.02e6 to 2.35e5.

In order to significantly improve the potential in the general case, different orders must be tested. Because testing different orders costs linearally more than a single iteration and there are exponential orders, finding better solutions is difficult. While you might expect this to result in finding the best configuration in exponential time, this may not be true for the average case. An entire paper could be dedicated to finding minimum configurations with lattice sort in less than exponential time, but suffice that it may be true and move on.[3] This is similar to the problem faced by Grid sort's rearrangeGrid, so it has not been tested rigorously. Future work may involve converting lattices to grids or visa-versa. It may make sense to combine the two but for now the two together are quite powerful.

A valuable use for Lattice sort is to give a guaranteed reasonable baseline to test other algorithms against. Because it is naive and guaranteed to give a solution that has no overlapping nodes, it can replace Metropolis and random placement as a baseline for testing (as seen in 1.3.1). Figure 32 shows that indeed lattice presents a better baseline than random placement in a large number of important cases especially those with a high number of nodes. Figure 33 shows that random goes off the graph as the number of nodes increases. This is consistent with Lattice being a better overall algorithm than random placement.

[3] A number of methods to this end make it reasonable to assume that picking random and non-random lattice configurations will end up in non-exponential time finding of a global minimum. Using information about the distance between connected nodes is by far the most beneficial. This information reduces exponential orders by a significant amount. Contraint solving is an algorithm that could possibly solve the Lattice minimization problem in less than exponential time. The first algorithm I considered was one where the highest potential nodes were moved and the rest of the nodes would flow around them. Another reasonable algorithm to this end is Basin-hopping. It may not solve in polynomial time in every case, but it is often enough to solve in the average case.

Algorithm	comparison	Comparator	Result
Random	better than	Lattice	2.700%
Random	2x better than	Lattice	1.817%
Random	3x better than	Lattice	1.700%
Random	4x better than	Lattice	1.584%
Random	5x better than	Lattice	1.500%
Random	6x better than	Lattice	1.467%
Lattice	better than	Random	97.300%
Lattice	2x better than	Random	95.549%
Lattice	3x better than	Random	93.699%
Lattice	4x better than	Random	92.199%
Lattice	5x better than	Random	90.948%
Lattice	6x better than	Random	89.982%

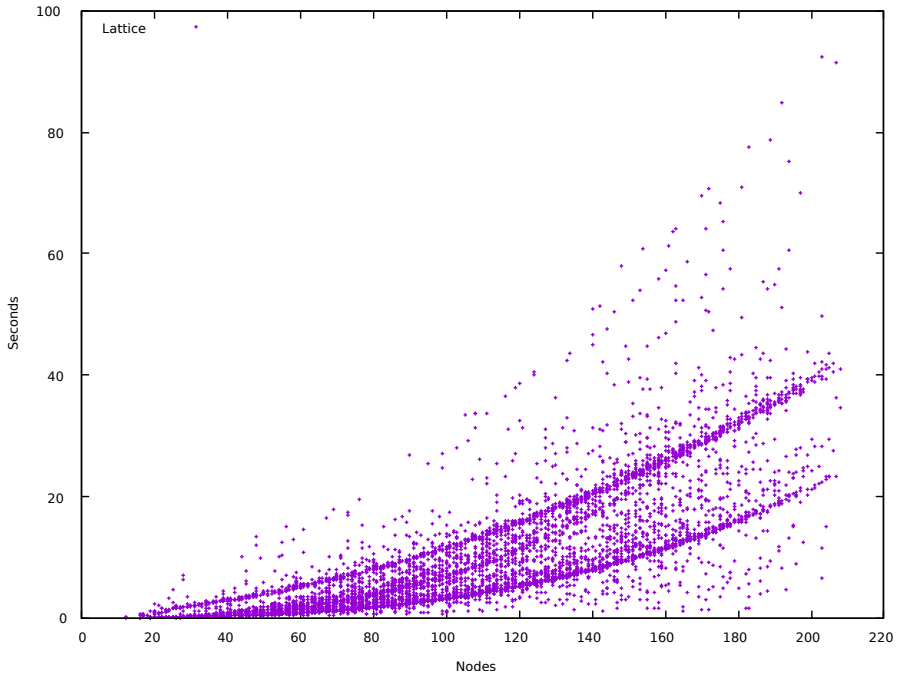

Figure 23: Lattice Sort Speed

1.7 Spiral Sort

Spiral sort is similar to Lattice sort. It is a fast and simple algorithm with the fifth best performance in the grid tests with a very similar function to Lattice

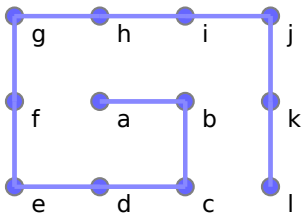

Figure 24: Linear 12 node graph Spiral sort

sort. Instead of sorting the backbone back and forth in a rhombus, spiral sort sorts the backbone in a spiral starting at the center and rotating clockwise. Each node is placed in cardinal direction instead of using trigonometry because it's simpler and uses the grid infrastructure available. Minor improvements could be made to this algorithm by using math to reduce the distance between adjacent nodes, but this is not a high priority. Spiral sort suffers from a similar problem to Lattice sort in that it finds local minima. Unlike Lattice sort, the start of the backbone is placed at the center and the end of the backbone is placed near the outside. The top of the graph is filled with the backbone while branches are at the bottom. One would not expect this to be particularly good, but the packing is better than grid when grid chooses very long bond lengths. Each very long bond length costs $A * dr^2$, a single diagonal can cost 247 potential. Thus packing is preferred in a great many cases. The reason Spiral sort has similar performance to Lattice sort is probably because they have similar positive and negative attributes. If we were to test 10 random orders of lattice, we should expect them to also tie with Lattice sort and Spiral sort. The fact that Lattice sort and Spiral sort have such high performance means that all six algorithms tested here have considerable weaknesses.

Algorithm	comparison	Comparator	Result
Spiral	better than	Lattice	52.225%
Spiral	2x better than	Lattice	10.752%
Spiral	3x better than	Lattice	7.068%
Spiral	4x better than	Lattice	5.951%
Spiral	5x better than	Lattice	5.318%
Spiral	6x better than	Lattice	5.034%
Lattice	better than	Spiral	47.775%
Lattice	2x better than	Spiral	10.602%
Lattice	3x better than	Spiral	7.851%
Lattice	4x better than	Spiral	6.868%
Lattice	5x better than	Spiral	6.418%
Lattice	6x better than	Spiral	6.118%

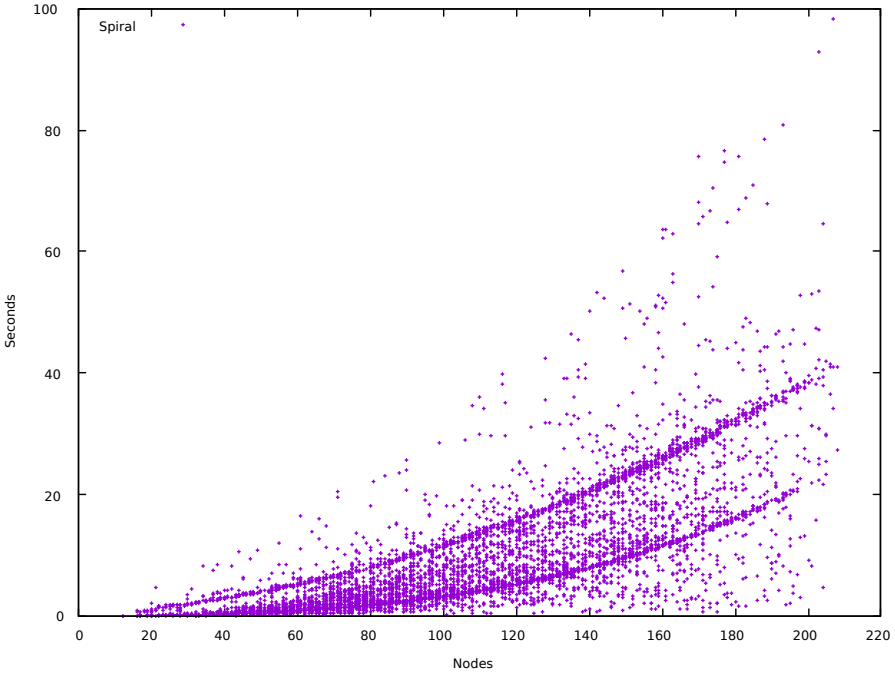

Figure 25: Spiral Sort Speed

1.8 Right Loop Sort

The best performing algorithm is the most recent entry, Right Loop sort. It works well on loops and branches and is biased toward the postive x direction (the right), which is where it derives its name, Right Loop sort. It is implemented in 26 lines of Python and uses a breadth first algorithm to choose the position of nodes. The output of this algorithm is similar to Graphviz's most popular algorithm for directed graphs, `dot` [20]. It starts by putting the first node in the first column. It then puts all its neighbors in the second column. It then puts its neighbors' neighbors that haven't been placed into the third column. It continues until all nodes have been placed. One benefit of this design is that it visualizes Dijkstra's algorithm [2].

In the case of a ring with 7 nodes it produces the graph found in Figure 20. The graph has a potential of 85.5. The global minima for this graph found by Loop sort is 22.53.

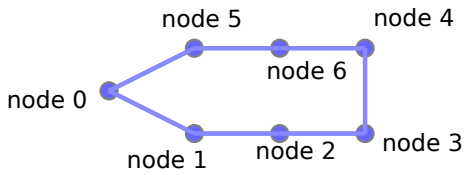

Figure 26: Ring with 7 nodes Right Loop Sort

While Loop sort wins against Right Loop sort in simple graphs, Right Loop sort wins in the vast majority of large complex graphs. While Loop sort was designed for organic chemistry, Right Loop sort was designed to solve the problem of complex graphs elegantly. A simple graph that Loop sort is currently unable to solve elegantly that Right Loop sort can solve reasonably is optimize1a_3_16. Figure 27 shows Loop sort's solution which contains at least two significant errors (fd and fg are too short and too long respectively). Figure 28 shows Right Loop sort's solution which only has a handful of small errors – cd, ce, df, ef, gh, gj, ji and kl are too long. Potential computations from the output of the algorithm can be found in the column labeled Initial in the table below. Potential computations from the minimized output using L-BFGS-B are found in the Minimized column. This shows that Right Loop sort, Loop sort, Grid sort, and Spiral sort all find the global minimum while Lattice sort and Fruchterman-Reingold are both stuck at a local minima.

Algorithm	Initial	Minimized
Right Loop	498.87	54.203
Loop	1184.1	54.165

Algorithm	Initial	Minimized
Grid	3650.4	54.193
Spiral	8397.1	54.160
Lattice	963.90	229.16
Fruchterman-Reingold	20271.	183.5

Small Wide World has a fully functional graphical interface in the browser. To work with the optimize1a_3_16 graph and all graphs presented in this paper, visit https://www.small-wide-world.com/paper/graphs/

Algorithm	comparison	Comparator	Result
Right Loop	better than	Lattice	59.493%
Right Loop	2x better than	Lattice	45.308%
Right Loop	3x better than	Lattice	39.073%
Right Loop	4x better than	Lattice	35.423%
Right Loop	5x better than	Lattice	32.705%
Right Loop	6x better than	Lattice	30.838%
Lattice	better than	Right Loop	40.507%
Lattice	2x better than	Right Loop	24.604%
Lattice	3x better than	Right Loop	16.669%
Lattice	4x better than	Right Loop	12.502%
Lattice	5x better than	Right Loop	10.052%
Lattice	6x better than	Right Loop	8.618%

1.9 Fruchterman-Reingold Sort (Force-Directed Placement)

Fruchterman and Reingold produced a very popular dynamic graph layout algorithm loosely based on a spring model. It was originally written in 1991 and continues to be used to this day by software such as Graphviz (`fdp` and `sfdp`), NetworkX, D3.js, Sigma.js, and Linkurious. It is my hope that my research will convince software developers to use the set of algorithms in Small Wide World before attempting this expensive algorithm to benefit their users. In this study, 100 rounds of Fruchterman-Reingold was applied to a graph with random layout and compared against the other algorithms. The winner was chosen in 10000 unminimized graphs and 6000 locally minimized graphs. The results of how many winners each algorithm produced are below.

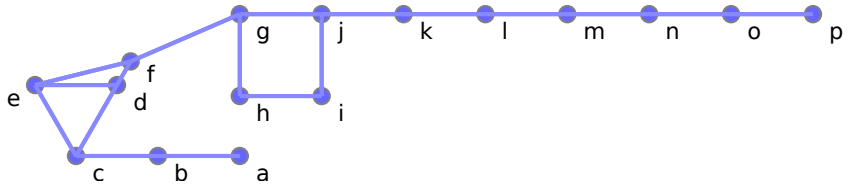

Figure 27: optimize1a_3_16 16 nodes Loop Sort

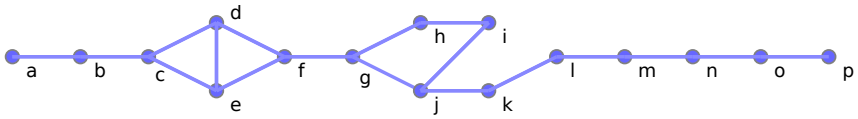

Figure 28: optimize1a_3_16 16 nodes Right Loop Sort

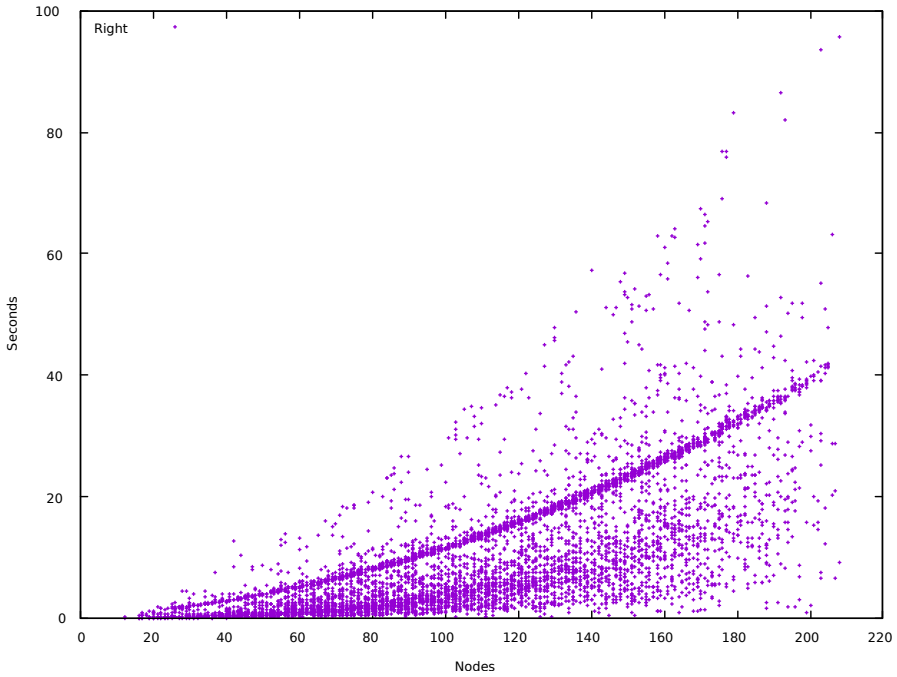

Figure 29: Right Loop Sort Speed

Minimized

Count	Algorithm
2735	Right Loop
1506	Loop
585	Grid
403	Spiral
365	Lattice
365	Fruchterman-Reingold
35	Branch
6	Random

Unminimized

Count	Algorithm
6090	Right Loop
1242	Fruchterman-Reingold
1142	Grid
662	Loop
461	Lattice
357	Spiral
46	Branch

These tables should illustrate that not only is Right Loop a better algorithm than 100 rounds of Fruchterman-Reingold, but also that each algorithm has merits. Figure 33 shows a much more important message than these tables – as the number of nodes increases, Fruchterman-Reingold performs worse than every other algorithm. While Fruchterman-Reingold performs reasonably well on graphs with less than 40 nodes, its performance decreases dramatically compared even to naive algorithms. This means that while Fruchterman-Reingold can be run for longer on larger or more complex graphs, it cannot compete in the Potential-computational time efficiency metrics that all algorithms must compete on.

Algorithm	comparison	Comparator	Result
Fruchterman-Reingold	better than	Lattice	17.703%
Fruchterman-Reingold	2x better than	Lattice	13.986%
Fruchterman-Reingold	3x better than	Lattice	11.219%
Fruchterman-Reingold	4x better than	Lattice	8.635%
Fruchterman-Reingold	5x better than	Lattice	7.251%
Fruchterman-Reingold	6x better than	Lattice	6.434%
Lattice	better than	Fruchterman-Reingold	82.297%
Lattice	2x better than	Fruchterman-Reingold	78.580%

Algorithm	comparison	Comparator	Result
Lattice	3x better than	Fruchterman-Reingold	76.196%
Lattice	4x better than	Fruchterman-Reingold	74.546%
Lattice	5x better than	Fruchterman-Reingold	72.845%
Lattice	6x better than	Fruchterman-Reingold	71.329%

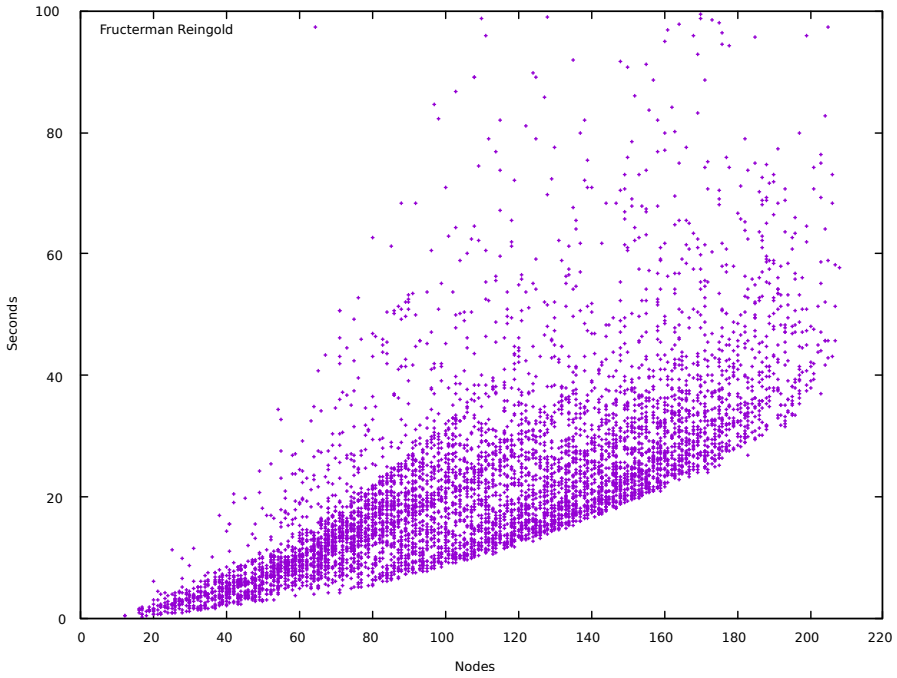

Figure 30: Fruchterman-Reingold Sort Speed

1.10 Future work

Many algorithms will likely result from Small Wide World due to its ease of development. The most important algorithm will probably be far more complex than Loop sort or Right Loop sort. Specifically I am looking at modified versions of Branch sort, Grid sort, Loop sort, and Right Loop sort that incorporate machine learning. With fast classification algorithms, algorithms could make good choices on where to place nodes to ensure lowest potential.

A good optimization that would possibly improve the speed of potential calculation is to use an estimation of density. This could also work with specialized optimizations of the potential where the algorithm searches for nodes with high potential energy (see A2 below).

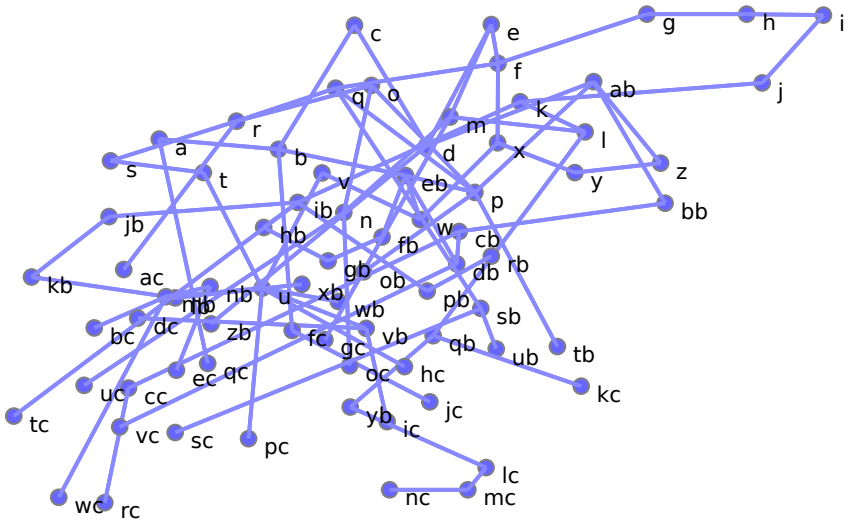

Figure 31: Grid with 75 nodes, 85 edges Lattice Sort near local minimum

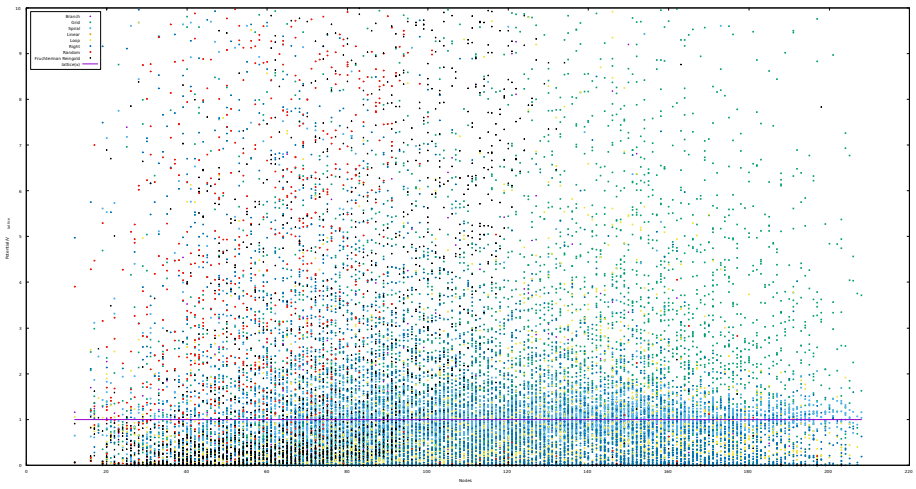

Figure 32: Algorithms vs Lattice sort

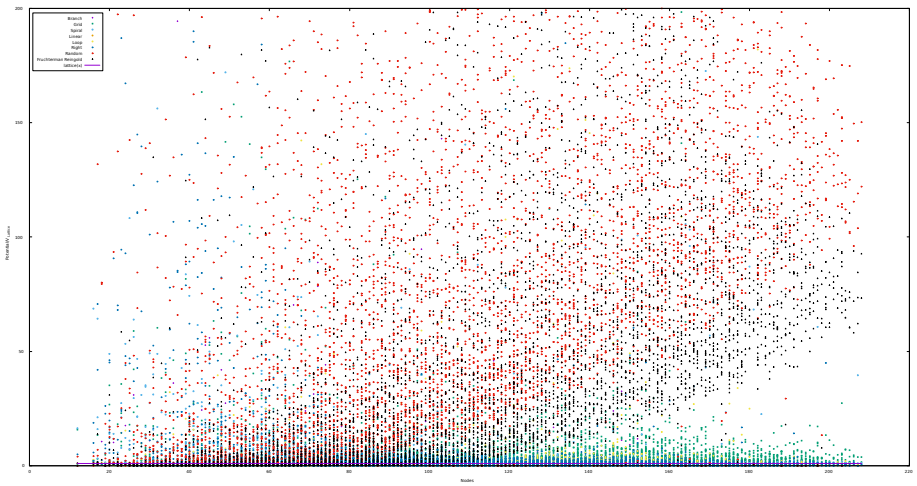

Figure 33: Algorithms vs Lattice sort zoomed out

Three possible improvements that could be made involve rethinking the optimization problem. The first improvement would be to consider rewriting the Metropolis and Basinhopping algorithms to use lattice or grid methods and cached potentials. A lot of effort is wasted in Metropolis and Basinhopping algorithms making poor random choices. Time spent in local minimization (L-BFGS-B) could be greatly reduced by only picking values that provide values very near b_0 for bond lengths.

Constraint solving is a machine learning technique that can be employed to improve the quality of graphs. This is a topic of ongoing research, when bond length is constrained to a small value by known potentials, the search space (configuration space) becomes very small. Any graph that has a bond length greater than the maximum can be filtered based on that information. Using this information to speed up the global minimization has not yet been fruitful, but most time has been spent improving the algorithms in preparation of this paper.

The second improvement is the topic of recent research, a pair of algorithms use detail from the potential to optimize a single node with the highest potential. Using this information, we can avoid attempting to minimize any other potentials. This works especially well with graphs that have a small number of flaws while the rest of the graph is minimized. It is possible to make this algorithm ergodic at a reasonable cost: all valid values that satisfy the constraints of molecular potential in two dimensions can be very small. A2 is one of the two algorithms which is available in Small Wide World.

The third improvement would be to attempt to implement Particle-Mesh Ewald (PME). Particle-Mesh Ewald has proven effective at providing a faster implementation of Lennard-Jones potential calculation in 3 dimensions. This lattice

equation is very complex and not well described. A significant improvement in minimization would mean that the average speed of 10 steps of global minimization on a large graph would reduce from 100 seconds to 10 seconds. While this is still unacceptable for real-time applications, it is acceptable for many applications such as visualizing a graph in design time.

References

[1] T.M. Fruchterman, E.M. Reingold, Graph drawing by force-directed placement, Software: Practice and Experience. 21 (1991) 1129–1164.

[2] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik. 1 (1959) 269–271.

[3] S. Lifson, A. Warshel, Consistent force field for calculations of conformations, vibrational spectra, and enthalpies of cycloalkane and n-alkane molecules, The Journal of Chemical Physics. 49 (1968) 5116–5129.

[4] J.L. Miller, Chemistry nobel honors computer simulation of biomolecules, Physics Today. 66 (2013) 13.

[5] M. Levitt, S. Lifson, Refinement of protein conformations using a macromolecular energy minimization procedure, Journal of Molecular Biology. 46 (1969) 269–279.

[6] T. Schlick, Molecular modeling and simulation: An interdisciplinary guide: An interdisciplinary guide, Springer Science & Business Media, 2010.

[7] R.H. Byrd, P. Lu, J. Nocedal, C. Zhu, A limited memory algorithm for bound constrained optimization, SIAM Journal on Scientific Computing. 16 (1995) 1190–1208.

[8] C. Zhu, R.H. Byrd, P. Lu, J. Nocedal, Algorithm 778: L-bFGS-b: Fortran subroutines for large-scale bound-constrained optimization, ACM Transactions on Mathematical Software (TOMS). 23 (1997) 550–560.

[9] J.L. Morales, J. Nocedal, Remark on "algorithm 778: L-bFGS-b: Fortran subroutines for large-scale bound constrained optimization", ACM Transactions on Mathematical Software (TOMS). 38 (2011) 7.

[10] E. Jones, T. Oliphant, P. Peterson, others, SciPy: Open source scientific tools for Python, (2001–2001--).

[11] D. Wales, Energy landscapes: Applications to clusters, biomolecules and glasses, Cambridge University Press, 2003.

[12] D.J. Wales, J.P. Doye, Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms, The Journal of Physical Chemistry A. 101 (1997) 5111–5116.

[13] Z. Li, H.A. Scheraga, Monte carlo-minimization approach to the multiple-minima problem in protein folding, Proceedings of the National Academy of Sciences. 84 (1987) 6611–6615.

[14] D.J. Wales, H.A. Scheraga, Global optimization of clusters, crystals, and biomolecules, Science. 285 (1999) 1368–1372.

[15] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, The Journal of Chemical Physics. 21 (1953) 1087–1092.

[16] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, others, Optimization by simmulated annealing, Science. 220 (1983) 671–680.

[17] M.P. Allen, D.J. Tildesley, Computer simulation of liquids, Oxford university press, 1989.

[18] M. Levitt, M. Hirshberg, R. Sharon, V. Daggett, Potential energy function and parameters for simulations of the molecular dynamics of proteins and nucleic acids in solution, Computer Physics Communications. 91 (1995) 215–231.

[19] W.L. Jorgensen, J. Tirado-Rives, Monte carlo vs molecular dynamics for conformational sampling, The Journal of Physical Chemistry. 100 (1996) 14508–14513.

[20] E.R. Gansner, E. Koutsofios, S.C. North, K.-P. Vo, A technique for drawing directed graphs, IEEE Transactions on Software Engineering. 19 (1993) 214–230.