Leveraging High Dimensional Spatial Graph Embedding as a Heuristic for Graph Algorithms

Peter Oostema Franz Franchetti
{poostema, franzf}@andrew.cmu.edu
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania

***Abstract*—Spatial graph embedding is a technique for placing graphs in space used for visualization and graph analytics. The general goal is to place connected nodes close together while spreading apart all others. Previous work has looked at spatial graph embedding in 2 or 3 dimensions. These used high performance libraries and fast algorithms for N-body simulation. We expand into higher dimensions to find what it can be useful for. Using an arbitrary number of dimensions allows all unweighted graph to have exact edge lengths, as n nodes can all be one distance part in a $n-1$ dimensional simplex. This increases the complexity of the simulation, so we provide an efficient GPU implementation in high dimensions. Although high dimensional embeddings cannot be easily visualized they find a consistent structure which can be used for graph analytics. Problems this has been used to solve are graph isomorphism and graph coloring.**

I. Introduction

Graph algorithms are typically discrete. Transforming graphs into a continuous space may allow for tractable solutions to non-polynomial problems for some graphs. To transform graphs into a continuous domain, the nodes can be placed into Euclidean space with forces applied to them.

Finding the iterations between all particles requires a N-body algorithm. N-body simulation is an approach to approximate the state over time of systems with multiple bodies all interacting. Using a N-body approach on graphs involves applying either attractive or repulsive forces between all nodes and a form of spring force for edges.

Graph embedding techniques generally seek to preserve the structure of the graph [3]. The embeddings are used for visualization or to extract important info. Spatial graph embedding has these same features. It creates a physical structure of the graph, and where a node's location represents how similar it is to the nodes around it.

Spatial graph embedding requires knowledge of what constitutes a *good* embedding, and how to solve for one. Current methods attempt to embed a graph in 3 dimensions. There are various issues with this, explained in [2]. Using a N-body approach to find a minimum energy state is an optimization problem, and many techniques apply from gradient descent [5]. N-body simulations of large graphs need to be run in a reasonable amount of time, so a high performance implementation is required for solving large and interesting cases [4].

Contributions. This paper makes the following contributions:

- We show how to perform geometric graph embeddings using any number of dimensions from 1 to $n-1$ where n is the number of nodes. Using a large number of dimensions smooths out the optimization. This allows consistent high quality embeddings to be found.
- We provide an efficient high dimensional N-body implementation on GPU. This uses annealing to quicken convergence and obtain higher quality results.
- The resulting positions of the nodes can be related to closeness centrality. Using these we can construct a graph fingerprint to identify graphs. Our method can be used as a heuristic to speed up graph isomorphism problems. Graph embeddings can also solve the graph coloring problem.

This paper will discuss other similar techniques and describe implementing high dimensional graph embedding. The performance of the CUDA implementation will be shown. The usefulness of embedding for problems like graph isomorphism and coloring will be examined.

II. Related Work

Multidimensonal Scaling. Multidimensional Scaling (MDS) is a method for visualizing data [6]. Information comes from a distance matrix that specifies how far apart each node is from others. An example of this would be a graph of cities with their driving distances as edge weights. MDS takes in this distance matrix and a set integer N and reduces the data into N-dimensional space. Often implementations of MDS are based on eigenvector decomposition. These methods can fail to fully satisfy the distance measures in low dimensions or use an excess number of dimensions.

Graph Signal Processing. This area of research defines the notion of various signal processing techniques for graphs [11]. The Graph Fourier Transform (GFT), and shift operators allow for a new representation of graph as frequencies. This representation has aided various applications such at simulating the brain, and predicting data labels in classification problems.

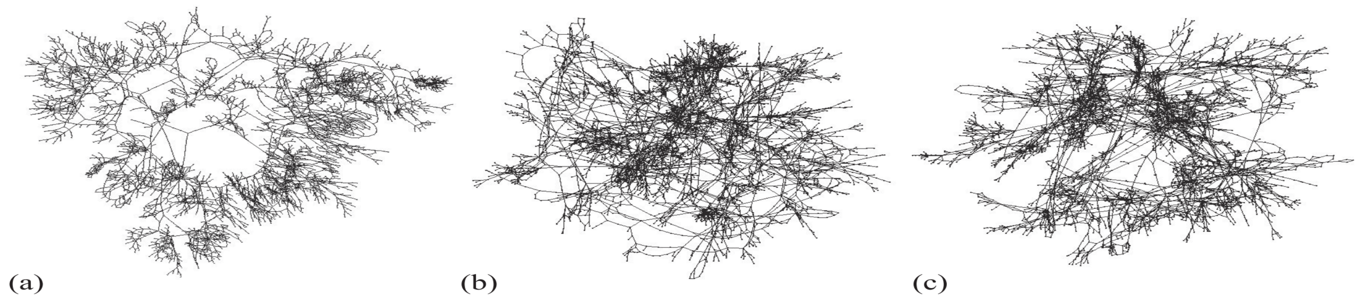


Fig. 1: (a) US power-grid with 4,941 nodes and 6,594 edges. (b) and (c) are quasi-stable configurations resulting from two random configuration as an initial state of the same network. [2]

Visualizing graph structures with N-body simulation. N-body simulations have been used before to explore large networks [2]. The paper looks at the production network of Japan, where each company is a node and each edge indicates the companies trade a non trivial amount of product. They focus on the largest unconnected component $\mathrm{G} = \{\mathrm{V}, \mathrm{E}\}$ with $|\mathrm{V}| = 100,000$ and $|\mathrm{E}| = 400,000$.

They give several criteria for a *good* visualization of a graph. The nodes in the graph should be close to others they are connected with. The nodes should also be well spread apart and in an equilibrium state. They attempt to solve these objectives by applying a spring force at each edge and a Coulomb charge at each node. Spring forces keep connected nodes close together, while electric charges make nodes push each other apart. The simulation uses momentum so that the minimum energy state can be found quicker. Drag was introduced so that it would reach equilibrium. The authors saw results where communities cluster close by, verified by their clustering analysis.

Positions x_i of the nodes are determined by the following equations. Here q is the coulomb charge, ℓ is the proper length of the edge, and γ is the drag coefficient.

$$m_i \frac{d^2 x_i}{dt^2} = \text{Coulomb} + \text{Spring} + \text{Frictional} \tag{1}$$

$$\text{Coulomb} = C q_i \sum_{i \neq j}^{N} q_j \frac{x_i - x_j}{|x_i - x_j|^3} \tag{2}$$

$$\text{Spring} = \sum_{(i,j) \in |E|}^{M} K_{i,j} (|x_i - x_j| - \ell_{i,j}) \tag{3}$$

$$\text{Frictional} = -\gamma_i \frac{dx_i}{dt} \tag{4}$$

There were several issues identified in the simulation. Random initializations resulted in random final configurations with some nodes unnecessarily entangled inside other structures. This can be seen in Figure 1. Here large branches of the graph can get stuck inside the broader structure. They propose starting the nodes in positions that best minimize the spring forces.

The issues with consistently getting a good structure can be seen in this small example graph of twelve nodes, which falls into three local minima Figure 2. The embedding on the right has each edge at length one, while the others have error on the edges. The two on the right are in equilibrium at higher energy states, meaning they are not maximally separated.

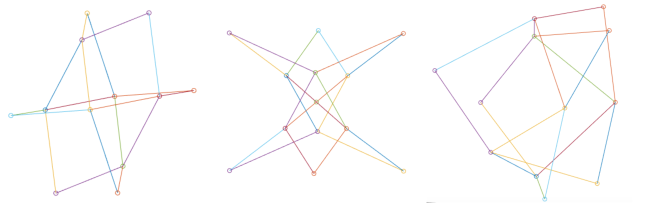


Fig. 2: Three resultant configurations of the same graph randomly initialized.

III. Background

Gradient descent optimization. The forces in N-body act similarly to gradients in optimization problems. Both find an optimum or minimum energy state.

Many methods have been developed to speed up gradient descent [5]. Adding momentum can speed up the computation as it moves like a ball down a slope. However it is inefficient to have the ball roll back up so Nesterov accelerated gradient acts like a ball that understands were it is moving next. This method performs the gradient update before applying momentum, so when the ball goes up a hill again it is not pushed further up it.

Methods that help gradient descent find minima faster also apply to N-body. Including momentum and drag drastically improves the rate of convergence. The same challenges also affect optimization with N-Body. It is important to find a good learning rate schedule to find the minimum as well as to reach an equilibrium state. N-Body is an expensive computation, so minimizing how many iterations it takes to find the solution is important.

***N*-Body Simulation with CUDA.** Computing the interactions of all pairs of bodies is the brute force approach, but hierarchical methods still require an all pair solution at each leaf or cell.

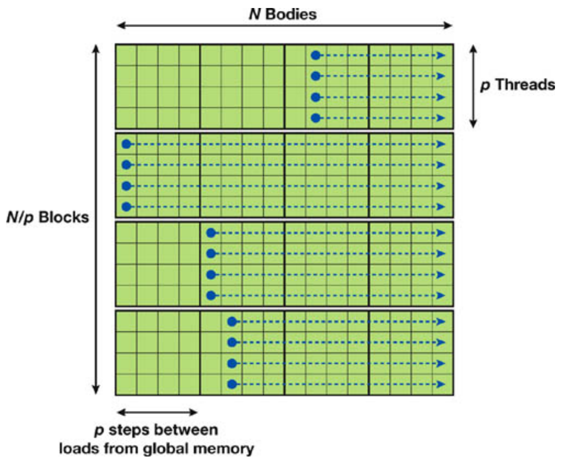


Fig. 3: The N-body execution pattern in CUDA has each thread find the force on one body.

The all pairs approach has $\mathrm{O(n^2)}$ independently commutable forces, so $\mathrm{O(n^2)}$ way parallelism, but would require as much memory [4]. Computation is instead parallelized $\mathrm{O(n)}$ ways and performed in tiles of dimension p as shown in Figure 3. This allows $\mathrm{p^2}$ interaction computations per $\mathrm{2p}$ reads from memory. The tiles must be computed sequentially by threads, but syncing on tiles ensures the shared memory is not overwritten by any thread progressing faster than others.

The code presented in this paper achieves performance near the theoretical peak of the 8800 GTX GPU. The performance analysis is in relation to the binary instructions compiled to, and so shows that the implementation does not lose time to memory stalls. The performance was improved by altering the tile size to insure the GPU's computation units were saturated.

Performing a high dimensional N-body simulation needs to use many of the same techniques.

Faster Algorithms for N-body. There exist several algorithms that take the all pairs complexity of $O(n^2)$ to an approximate solution with $O(n\log(n))$. They do not generalize well to high dimensions. Tree methods break down the computation into 2^d parts for d dimensions. Particle mesh methods do not simulate close field interaction, which for this problem are often the most important. Clustering the graph during the simulation would give similar execution times, but this has not been implemented.

Graphs Models. This paper uses several random graph models for embedding. First is the Erdős–Rényi model ($G_{n,p}$) [12]. These graphs have n nodes and all of the possible edges from pairs of nodes are present with equal probability p. The Barabási–Albert model (BA) [13] generates a graph by adding in nodes one by one while adding a set amount of edges for each node, preferring the heavy nodes. BA graphs have power law degree distributions, which simulate some real world graphs better than the exponential distribution of the Erdős–Rényi model. Other graphs used to showcase features of the embedding are random split graphs. Split graphs are those made up of a fully connected clique and an independent set of nodes. The paper also uses graphs made specifically to be hard on which to solve graph isomorphism [1].

Graph Isomorphism Problem. A graph isomorphism is a one-to-one mapping between two graphs G and H such that each pair of adjacent vertices in G are adjacent in H. The graph isomorphism problem asks whether or not two graphs are isomorphic to each other. The problem is known to be in NP, but not known to be part of P or NP-complete. Recent developments give it a ”psuedo polynomial” runtime [7].

Several fast solvers exist such as Traces [9] and Conauto [8]. This paper compares performance to MATLAB's implementation, which does not have state of the art speed. Improving on it does not change the state of the art speed, but demonstrates the potential for finding a solution faster.

Graph Coloring. Graph coloring is a diverse set of problems, but this paper will focus on vertex coloring. This problem asks how to color each vertex in a graph such that adjacent vertices do not have the same color. Typically the goal is to find the least number of colors needed, called the chromatic number. This is a well known NP-complete problem [10]. The greedy algorithm for graph coloring runs in linear time and looks at each node one at a time assigning new colors as necessary. Here the order matters and determines how many are used. The worst ordering gives the Grundy number, or the maximum number of colors a greedy coloring will use on a graph.

IV. Problem Statement

Previous work has struggled to produce consistent high quality embeddings. They suffer from random configurations and results that do not always satisfy that connect nodes are close with others farther apart. So we define a setup to find consistent embeddings.

It should be possible for graphs of edges with lengths that do not violate the triangle equality to be placed perfectly into Euclidean space using a bounded number of dimensions. Perfect implies two conditions. First, nodes connected by an edge should be exactly their edge weight apart. This paper focuses on unweighted graphs were all distances are set to one. Secondly while the edge distances need to maintain correctness, the system should find the global maximally separated state of the nodes. The nodes can be separated by adding repulsive forces between all nodes, similar to electric charges at each node. These constraints are specified in the following equations, where $\mathrm{n_i}$ is a node, $\mathrm{x_i}$ is a node's position, and $\mathrm{w_{i,j}}$ is the weight between nodes i and j.

For constants c_0 and c_1 the force from each particle is:

$$F(n_i) = F_{Elec}(n_i) + F_{Edge}(n_i) \tag{5}$$

$$F_{Elec}(n_i) = \sum_{i,j\leq|V|} \frac{c_0}{||x_i - x_j||_2^2} \tag{6}$$

If $w_{i,j} \in E$ then add to the force

$$F_{Edge}(n_i, n_j) = -c_1\sqrt[3]{||x_i - x_j||_2 - w_{i,j}} \tag{7}$$

V. High Dimensional Implementation

The goal of spatial graph embedding is to find an alternative representation to find interesting features or solve graph problems. Large graphs are not easily placed in low dimensional space, so we use as many dimensions as necessary.

The graph $\mathrm{G} = \{\mathrm{V}, \mathrm{E}\}, |\mathrm{V}| = \mathrm{n}, |\mathrm{E}| = \mathrm{m}$ is initialized to a $n - 1$ dimensional simplex of n nodes. This solves the issue of random initializations arriving at suboptimal solutions. This works for unweighted graphs where each edge is the same length. A simplex solves the problem of correct edge lengths as each node is the same distance apart from all others. The nodes can move from the simplex down convex paths to a lower energy state. It is conjectured that the repulsive electrical forces push the system to a unique rotationally invariant global optimum.

The N-body simulation is implemented as an all pairs computation. This algorithm makes good use of the GPU architecture as there is potential for $O(n^2)$ way parallelism. The code is optimized for GPUs and written in CUDA. One of the most costly parts of the computation has been rewritten as a matrix multiplication, and implemented with a cuBLAS call. The rest of the computation has code written with access patterns that effectively use the memory system.

The algorithm makes use of several well known optimization techniques. Momentum is used to quicken the computation as nodes often move in the same direction for multiple iterations. Dampening is used to make sure the movements do not get too chaotic, and to bring the structure into an equilibrium state. Simulated annealing lowers the error on edges below a threshold.

Hyper parameters need to be set correctly to find an embedding quickly. The forces need to be small enough so that the edges are not pulled too far each iteration. The comparative strength of the forces creates an overshoot from the optimum edge length. For this reason the constant on repulsion needs to be small, while it needs to be strong enough to push all other nodes further apart.

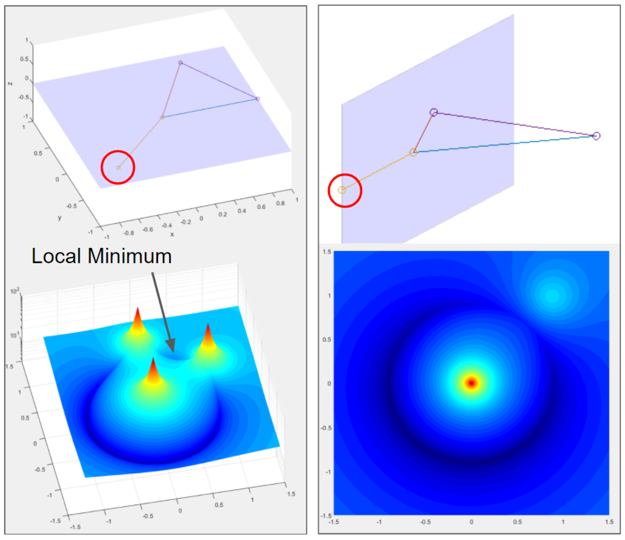


Fig. 4: Forcing functions in 2D and 3D of the repulsive and edge forces on a node. The triangle and line can embed in 2D, but is convex in 3D.

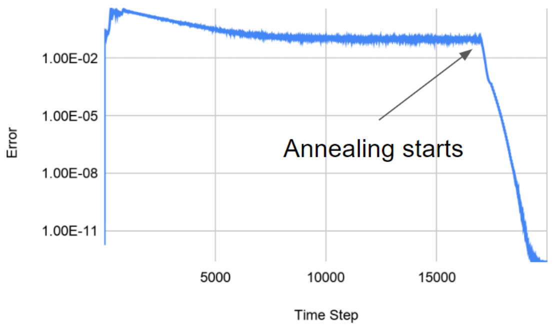


Fig. 5: Convergence of embedding error (edge forces) on a 400 node graph.

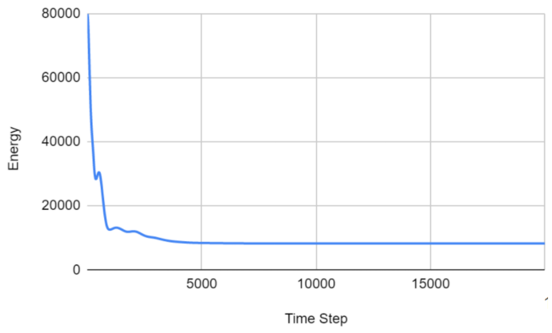


Fig. 6: Convergence of embedding energy (repulsive forces) on a 400 node graph.

Convexity in High Dimensions. A large number of dimensions can be required to properly embed a graph. It is often necessary for at least one additional dimension than required to give a convex optimization space. A simple example can be seen in Figure 4. This shows the intensity of the forcing functions as the point circled in red moves in the drawn plane. This point is strongly pushed away from the nodes in the triangle. This can cause it to fall into a local minimum internal to the triangle. Adding a third dimension gives a convex path to a lower energy state. It can find a lower energy state by maintaining its edge length and rotating outside the triangle.

Convergence and annealing. Embeddings are initialized to a simplex, so that the edge lengths constraints are fulfilled. To find the maximally separated embedding the nodes need to rotate around each other maintaining the edge lengths. After the maximally separated state has been found, annealing can start. This *cools* the structure making it possible for the nodes to move exactly to fulfill the edges lengths, while maintaining the low energy state. This process is shown in Figure 5 and Figure 6. These show the error (different of length and required length) and energy (total repulsive force) after every time step. The overall process seeks to find the minimum energy state while keeping the error on the edge

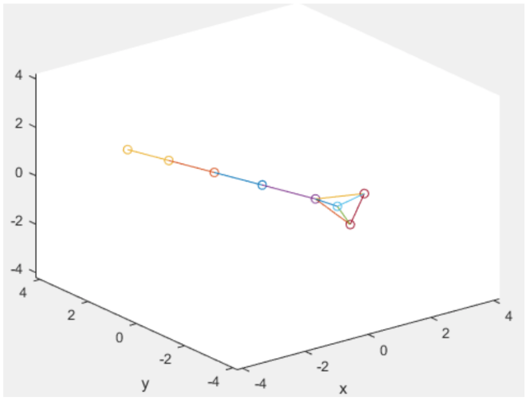


Fig. 7: An example graph with mass rotated along the Y-axis.

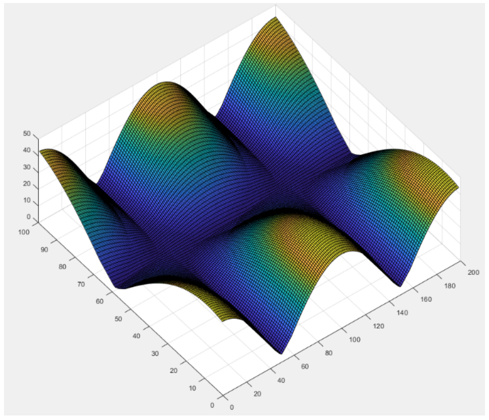


Fig. 8: Loss function on angles of rotation in X and Z Dimensions.

length small. The error starts at zero because it is initialized to a simplex. It quickly jumps up from the step size of the simulation, but remains within one percent of the required length. Lowering the step size allows the nodes to get to the exact distance apart. The measure of separation, energy, shows that the structure separates greatly at first, but take a long time to settle to a minimum.

Orientation Optimization. A possible rotationally invariant position can be found for each embedding. It works by rotating the structure to minimize the sum of positions in the highest dimension. It then goes through each dimension performing this operation. An example can be seen in Figure 7. The graph is made up of a four point line graph attached to a clique of four nodes. Here the amount of mass not on the Y-axis is minimized. The line can be rotated along this axis, but the tetrahedron cannot. The loss function of this is in Figure 8. It shows the amount of mass times distance not on the Y-axis as the graph is rotated around the other two axes. The peaks indicate misalignment of the structure. The contour repeats as the structure is symmetrical by rotation. It was found to consistently find the same orientation for some small graphs. But larger graphs have not been tested as this naive approach has a complexity of $O(n^4)$.

VI. CUDA Implementation

N-Body simulation can be an expensive operation with complexity $O(n^2)$. Using many dimensions brings the complexity up to $O(n^3)$. So an efficient parallel implementation of the algorithm is necessary for solving large problems.

The code for embedding focuses on two major considerations when writing CUDA code. First the memory access needs to be properly coalesced. To avoid bank conflicts every piece of data a warp requests at the same time need to come from distinct addresses modulus 128. Memory should be read sequentially when possible to reduce excess writes to cache.

The second consideration is to make sure there are enough warps per Simultaneous Multiprocessor (SM) to hide instruction and memory latencies. Most computations here have a n^2 level of parallelism, which even for a 100 node graph means 300 warps. Enough for the 40 SM's on the RTX 2070 Super.

Listing 1: Electric Forces Kernel

```
__global__
void NBodyMul(float* N, float* V, float* partials,
      unsigned d, unsigned n){
  int k = blockIdx.y * 8;
  int i = blockIdx.x * 1024 + threadIdx.x;
  int iTile = threadIdx.x / 8;
  int iTileDim = threadIdx.x % 8;
  __shared__ float NShd[64*8];
  float myNPos0 = N[i*d + k + 0];
  float myNPos1 = N[i*d + k + 1];
  float myNPos2 = N[i*d + k + 2];
  float myNPos3 = N[i*d + k + 3];
  float myNPos4 = N[i*d + k + 4];
  float myNPos5 = N[i*d + k + 5];
  float myNPos6 = N[i*d + k + 6];
  float myNPos7 = N[i*d + k + 7];
  for (int tileID = 0; tileID < n; tileID += 64){
    int tileBoundary = tileID + 64;
    int tileBoundary = (n < 64) ? n % 64 : 64
    __syncthreads();
    if (iTile < tileWidth){
      NShd[threadIdx.x] = N[(iTile+tileID)*d+(k+iTileDim)];
    }
    __syncthreads();
    if (i < n){
      for (int j = 0; j < tileWidth; j++){
        float partialComp = partials[i*n+tileID+j];
        force0 += (myNPos0-NShd[0+j*8])*partialComp;
        force1 += (myNPos1-NShd[1+j*8])*partialComp;
        force2 += (myNPos2-NShd[2+j*8])*partialComp;
        force3 += (myNPos3-NShd[3+j*8])*partialComp;
        force4 += (myNPos4-NShd[4+j*8])*partialComp;
        force5 += (myNPos5-NShd[5+j*8])*partialComp;
        force6 += (myNPos6-NShd[6+j*8])*partialComp;
        force7 += (myNPos7-NShd[7+j*8])*partialComp;
      }
    }
    if (i < n){
      V[i*d + k + 0] += force0; V[i*d + k + 1] += force1;
      V[i*d + k + 2] += force2; V[i*d + k + 3] += force3;
      V[i*d + k + 4] += force4; V[i*d + k + 5] += force5;
      V[i*d + k + 6] += force6; V[i*d + k + 7] += force7;
    }
  }
}
```

The computation of electric forces is shown in Listing.1. Each thread handles eight dimensions of data from one node, comparing them to the same dimensions of all other nodes. Blocks work on different dimensions so data can be reused within a block. The computation iterates through tiles of the matrix reading into shared memory for fast access. Tile size is determined by the size of the cache to prevent overflow. The for loops are unrolled to increase performance, but anymore would overflow the registers needed for a thread block.

Distance Calculations as a Matrix Multiply. The largest bottleneck in the computation comes from calculating the all pairs forces. This can be found with a high performance matrix multiply kernel.

Nearly half the computation of this problem involves finding the distances between all pairs of points. This computation can be performed as a matrix multiply. For all nodes $\forall i, j \in N$ the distance $\mathrm{d_{i,j}} = ||\mathrm{x_i} - \mathrm{x_j}||_2$ equals:

$$= \sqrt{\sum_{k=1}^{d} (x_{i,k} - x_{j,k})^2} \tag{8}$$

$$= \sqrt{\sum_{k=1}^{d} (x_{i,k}^2 - 2x_{i,k}x_{j,k} + x_{j,k}^2)} \tag{9}$$

$$= \sqrt{\sum_{k=1}^{d} (x_{i,k}^2) + \sum_{k=1}^{d} (-2x_{i,k}x_{j,k}) + \sum_{k=1}^{d} (x_{j,k}^2)} \tag{10}$$

The middle summation is a scaled dot product of two vectors. Computing the dot product of all pairs of columns is exactly a matrix multiplied by its transpose. Finding all pairs of distances can be implemented as a call to a cuBLAS kernel.

The rest of the computation involves computing the forces on each node given the distance between them. As with finding distances these are $O(n^3)$ operations with $O(n^2)$ parallelism. They make up the current bottleneck as seen in Table.1.

A. *Performance*

Tables 1 and 2 show the performance of the implementation in detail. GFLOPs indicate how many floating point operations are performed in the computation and GFLOPS measures how many operations are performed per second. Most of the run time is spent computing the electric forces from the distances and edge forces. Both take similar time, but the electric forces are a denser computation and run more efficiently.

Testing was performed on a RTX 2070 Super, which has a theoretical peak performance of 8.2 TFLOPS for single precision floating point arithmetic. The matrix multiplication is performed by a cuBLAS kernel and reaches 6 TFLOPS, which shows it making nearly full use of the GPU. It must be noted that a fused multiply-add operation has a latency of one cycle and so is treated as one operation. The electric force functions reaches 1 TFLOP. The edge forces are found with

	Run Time (s)	GFLOPs	GFLOPS
Distances	0.16	1000	6,180
Electric Forces	1.93	2004	1,040
Edge Forces	1.82	32	17.5
Force Update	0.04	2	52.4
Memory Alloc/Copy	0.11		
Total	4.07	3038	747

Table 1: Performance over 1,000 iterations on $G_{n,p}$ where $n = 1,000$, $p = 8/n$

	Run Time (s)	GFLOPs	GFLOPS
Distances	0.16	1000	6,210
Electric Forces	1.92	2004	1,050
Edge Forces	2.97	514	173
Force Update	0.04	2	52.0
Memory Alloc/Copy	0.11		
Total	5.19	3520	678

Table 2: Performance over 1,000 iterations on $G_{n,p}$ where $n = 1,000$, $p = 128/n$.

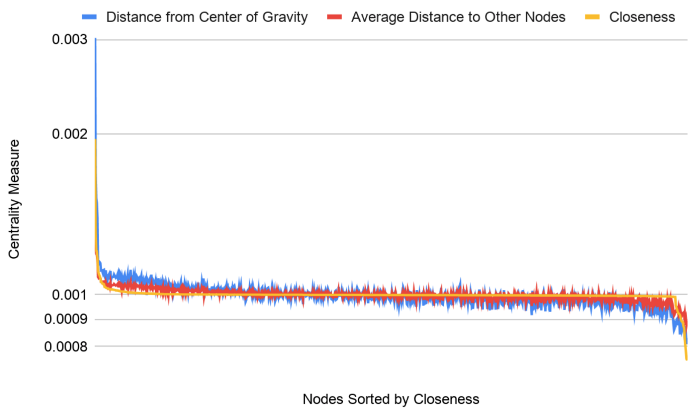


Fig. 9: Sorted centrality of a PAM graph with $n = 1,000$ and 10 edges per node.

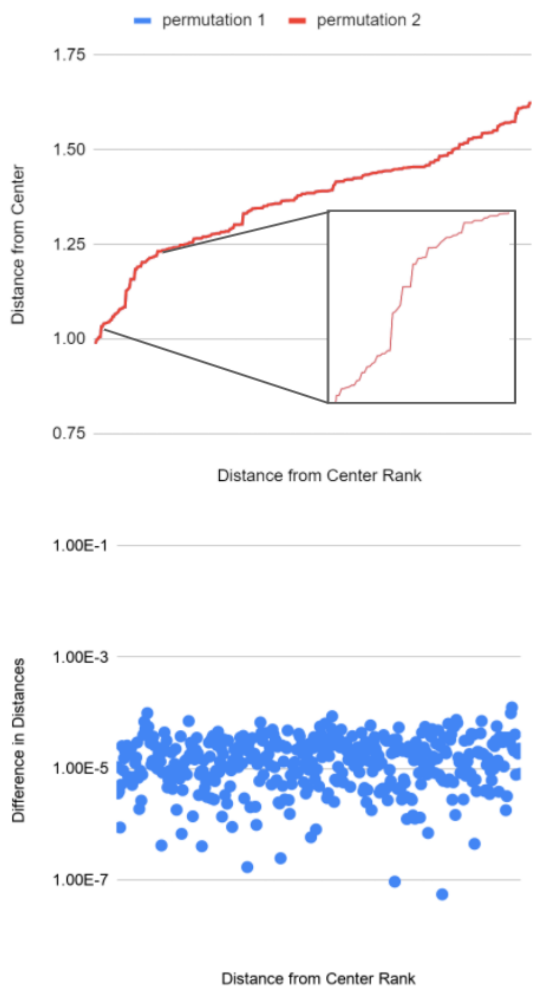
a dense computation and so do not achieve great performance for sparse graphs. These operations average out to the overall performance to around 700 GFLOPS in these graphs.

VII. Applications

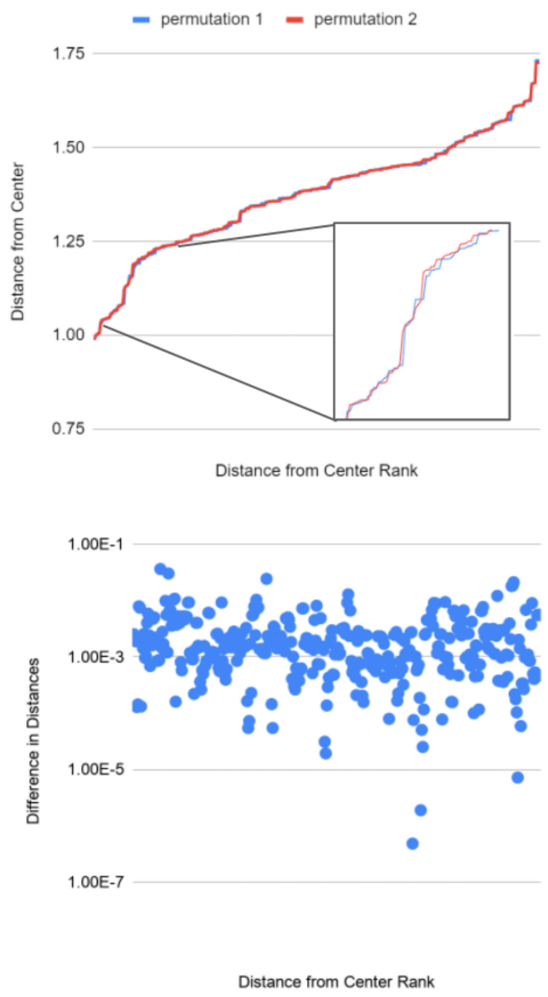
A. *Closeness Centrality*

Closeness is a measure of centrality in graphs. It is computed by finding all pairs of shortest paths in the graph. The closeness of each node is a measure of how large the sum of its shortest paths are.

This was shown to have a close parallel to the distance from center of gravity of the resulting spatial embedding. How similar these measures can be can be seen in Figure 9. The closeness centrality, distance from center of mass, and average distance to other nodes were found for each node. The data was then sorted on the closeness centrality. Many nodes have similar centrality, but those with high and low levels



(a) Same Graph



(b) One Swapped Edge

Fig. 10: Distances in embedding of a 400 node graph with permuted labels.

are accurately identified by either approach. The measure of centrality does not differ greatly at any node. This establishes the distance of a node from the center of gravity as a measure of centrality, although an expensive one as each iteration has the same complexity of an all pairs shortest path algorithm, which is used to compute closeness centrality. Using euclidean distance gives a measure of centrality in a continuous space, which can better distinguish nodes.

B. Graph Fingerprint

We have observed that embedding graphs with permuted node labels result in similar resulting structures. While it is difficult to definitively show them to be that same we can simply find the distance to the center of gravity for each point. By sorting these distances and comparing the series we can see how similar two graphs are.

In Figure 10 we see two permutations on the node labels of a graph result in identical graph *fingerprints*. These are found by finding the distance to center of mass for each node, and sorting the distances. After swapping one edge the fingerprints become distinct. The difference in the distances is also plotted. The differences are near 10^-5 for identical graphs, while the differences are 300 times greater after swapping an edge. We cannot expect the fingerprints to always be distinct for different graphs. But it does give an understanding of the graph and some concept of how similar two graphs are. This fingerprint could potentially be used as a signal of a graph.

C. Graph Isomorphism

The graph isomorphism problem tests to see if the structure of two graphs are the same. In other words given two adjacency matrices can the rows and columns of one be swapped to match the second. Checking graphs for the same structure should be easy in a spatial embedding. However checking that a geometric structure is the same under rotation is as intractable a problem. The embeddings of two permutations of the same graph do not come out exactly the same, but extremely close.
Embeddings can also be used to refute graph isomorphism. This can be seen in Figure 10. Here two permuted graphs

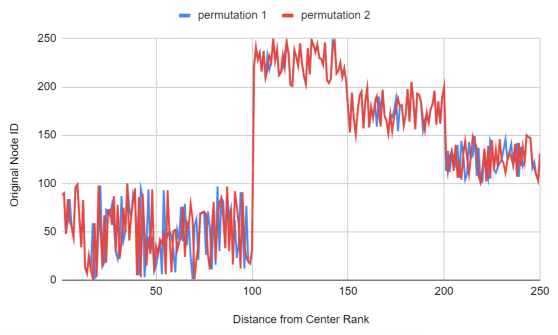


Fig. 11: Ranking of nodes in a 150 node split graph for two permutations by original index.

	Time (s)	Time after Reordering (s)
Average	137	45
Std Dev	225	87
Minimum	0.08	0.03
Maximum	Timeout	281

Table 3: Times for MATLAB's isomorphism() function. Reordering takes an additional 16 seconds in every case.

are embedded and the distances are seen to overlap. After swapping one edge the "fingerprints" distinctively no longer overlap, showing the graphs to be different.

Graph Isomorphism Speedup. As a heuristic the distance from the center of mass can be used to classify nodes. This works well for small graphs, but fails for similar nodes at the same distance from the center of mass. Figure 11 shows the ranking of nodes of a split graph, with a clique of size 100, and three groups of nodes in the independent set of degrees of 1, 2, and 3. Nodes are classified by their distance to center of mass, with the closest being assigned the label 1 and the last label n. It correctly classifies nodes of difference degrees, but struggles among nodes of the same degree. Nodes that are misclassified are labeled close to what they should be.

Table.3 shows the computational benefit of reordering the nodes from the embedding of a 200 node graph from [1]. 20 different permutations were run and solved using the MATLAB built in function isomorphism(). A timeout of 10 minutes was set. Several tests were faster to solve before reordering, but on average reordering is faster. This can be seen in Figure 12. Many permutations run significantly longer before reordering. MATLAB solves the isomorphism problem using a local search from the starting point. This shows that ranking the nodes by distance is in general closer to the original labeling than a random permutation.

D. Graph Coloring

Graph coloring, as described in Section 3.5, is equivalent to embedding a graph into a simplex where connected points may not be at the same point. This is true as nodes at the same point in space will make up an independent set and can be assigned the same color. Nodes connected by an edge will not be at the same position as the edge forces them apart. An

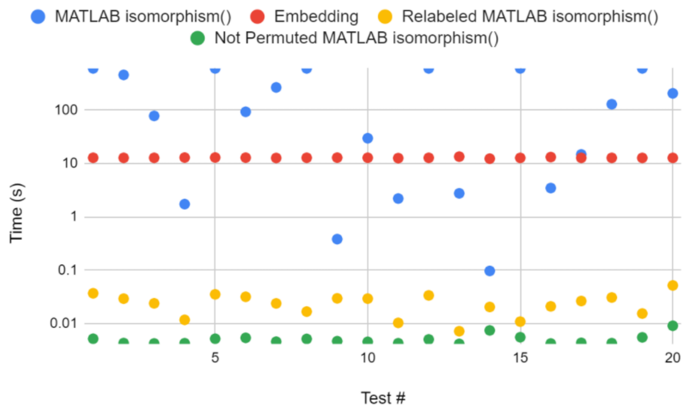


Fig. 12: Graph isomorphism times for 400 Node Graph run with MATLAB.

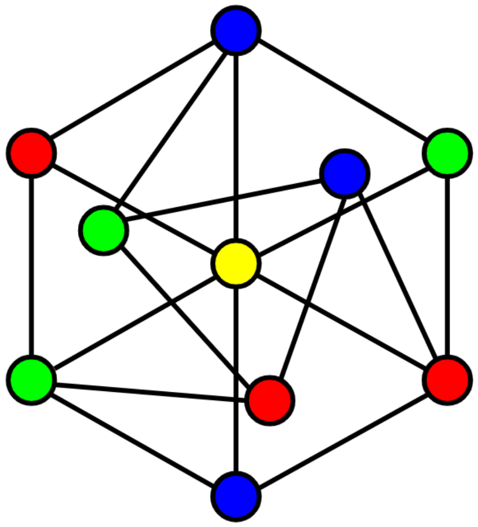


Fig. 13: Coloring of the Golomb Graph. From Wikipedia "Golomb Graph".

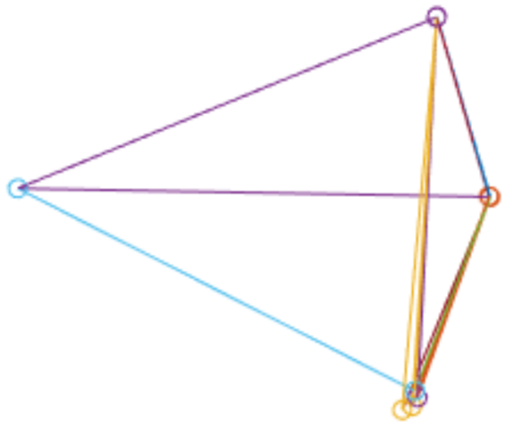


Fig. 14: Embedding for coloring of the Golomb Graph.

example of this is in 14. Here each node at the same position in the tetrahedron can be assigned the same color for a valid coloring using the chromatic number of colors.

This embedding can be found by minimizing the distance between nodes instead of maximizing for finding unique embeddings. In the computation the sign of the N-Body force can simply be flipped. Figure 14 shows a tetrahedron embedding of the Golomb graph with chromatic number four.

Future work will look at larger, more interesting graphs for coloring. Given that this is a heuristic approach it may not always give a coloring using the least number of colors. The usefulness of this approach will be determined by how well the results compares to the chromatic and Grundy number. It will also be compared to the speed to solve graph coloring with state of the art solvers. Here we present the potential use of this approach for graph coloring.

VIII. Conclusion

High dimensional spatial embeddings with exact edge lengths can be quickly found and used to solve graph isomorphisms and colorings. Embeddings can be used for solving existing graph algorithms faster or for understanding graphs in a new way. The uniqueness of the structure makes it useful for the graph isomorphism problem. Finding a minimally separated embedding can be used for graph coloring. A high dimensional N-body simulation is more expensive than a 3D problem, but this paper provides a way to run the high dimensional simulation efficiently.

References

[1] A. Dawar and K. Khan, ”Constructing hard examples for graph isomorphism.” J. Graph Algorithms Appl., 23(2), pp. 293–316, 2019.

[2] Y. Fujiwara, ”Visualizing a large-scale structure of production network by N-body simulation,” Progress of Theoretical Physics Supplement, 179, pp. 167–177, 2009.

[3] P. Goyal and E. Ferrara, ”Graph embedding techniques, applications, and performance: A survey.” arXiv preprint arXiv:1605.09096, 2017.

[4] L. Nyland, M. Harris, and J. Prins, ”Fast N-Body simulation with CUDA.” GPU Gems 3, Addison Wesley, pp. 677–795, 2007.

[5] S. Ruder, ”An overview of gradient descent optimization algorithms.” arXiv:1609.04747, 2016.

[6] J. D. Carroll, P. Arabie, ”Multidimensional scaling,” Annual Review of Psychology, 31, pp. 607-649, 1980.

[7] Babai L, ”Groups, graphs, algorithms: The graph isomorphism problem.” Proc Internat Congr of Mathematicians, 2018.

[8] J. L. López-Presa, A. F. Anta, and L. N. Chiroque, ”Conauto-2.0: Fast isomorphism testing and automorphism group computation.” CoRR, abs/1108.1060, 2011.

[9] B.D. McKay, and A. Piperno, ”Practical Graph Isomorphism, II” Journal of Symbolic Computation 60, pp. 94-112, 2014 https://doi.org/10.1016/j.jsc.2013.09.003.

[10] E. Lawler, ”A note on the complexity of the chromatic number problem,” Inform. Process. Lett. 5, pp. 66-67, 1976.

[11] A. Ortega, P. Frossard, J. Kovacevi, M.F. Moura, and P. Vandergheynst, ”Graph signal processing: Overview, challenges, and applications.” Proceedings of the IEEE, 2018.

[12] P. Erdo˝s and A. Re´nyi, ”On the evolution of random graphs,” Publ. Math. Inst. Hung. Acad.Sci. 5, 17, 1960; (Academic Press, London, 1985).

[13] R. Albertand, A. L. Barabasi, ”Statistical mechanics of complex networks,” Rev. Modern Phys. 74, pp. 47–97, 2002.