AN IMPROVED SPRING EMBEDDER LAYOUT ALGORITHM FOR COMPOUND GRAPHS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Alper Karaçelik

August, 2012

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Uğur Doğrusöz(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Burkay Genç

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Dr. Cevat Şener

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

AN IMPROVED SPRING EMBEDDER LAYOUT ALGORITHM FOR COMPOUND GRAPHS

Alper Karaçelik
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Uğur Doğrusöz
August, 2012

Interactive graph editing plays an important role in information visualization systems. For qualified analysis of the given data, an automated layout calculation is needed. There have been numerous results published about automatic layout of simple graphs, where the vertices are depicted as points in a 2D or 3D plane and edges as straight lines connecting those points. But simple graphs are insufficient to cover most real life information. Relational information is often clustered or hierarchically organized into groups or nested structures. Compound spring embedder (CoSE) of Chisio project is a layout algorithm based on a force-directed layout scheme for undirected, non-uniform node sized compound graphs.

In order to satisfy the end-user, layout calculation process has to finish fast, and the resulting layout should be eye pleasing. Therefore, several methods were developed for improving both running time and the visual quality of the layout. With the purpose of improving the visual quality of CoSE, we adapted a multi-level scaling strategy. For improving the performance of the CoSE, the grid-variant algorithm proposed by Fruchterman and Reingold and parallel force calculation strategy by using graphics processing unit (GPU) were also adopted. Additionally, tuning of the parameters like spring constant and cooling factor were considered, as they affect the behavior of the physical system dramatically. Our experiments show that after some tuning and adaptation of the methods above, running time decreased and the visual quality of the layout improved significantly.

Keywords: Interactive graph editing, Automated graph layout, Force-directed placement, Compound graph, FR-Grid Variant, Multi-level scaling, Parallel programming.

ÖZET

İYİLEŞTİRİLMİŞ BİR BİLEŞİK ÇİZGE YERLEŞTİRME ALGORİTMASI

Alper Karaçelik
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Doçent Dr. Uğur Doğrusöz
Ağustos, 2012

Etkileşimli çizge düzenleme, bilgi görselleştirme sistemlerinde önemli bir rol oynamaktadır. Eldeki verinin kaliteli analizini yapmak için otomatikleştirilmiş yerleşim hesaplaması yapmak gerekmektedir. Basit çizgelerin otomatik yerleşiminin yapılmasına ilişkin bir çok çalışma yapılmıştır. Genellikle bu basit çizgelerde köşeler iki ya da üç boyutlu düzlemde nokta, kenarlar da bu noktaları bağlayan doğru ya da eğriler olarak gösterilmektedir. Ancak, ilişkisel veriler, genellikle hiyerarşik veya kümelenmiş olarak organize edilmektedirler. CoSE ile, kuvvet yönelimli yerleşim şemasına dayalı bir bileşik çizge yerleştirme algoritması sunulmaktadır. CoSE ile birlikte yönsüz, birbirinden farklı büyüklükte köşelere sahip bileşik çizgelerin yerleşimi sağlanmaktadır.

Kullanıcı memnuniyetini sağlamak için çizge yerleştirme işinin kısa sürede tamamlanması ve çizgenin göze hoş gelen bir biçimde ekranda yer alması gerekmetedir. Hem performans hem de görsel kalitenin iyileştrilmesi amacıyla sunulan bir çok method bulunmaktadır. Bu tez çalışmasında CoSE'nin görsel kalitesinin geliştirilmesi amacıyla çok seviyeli ölçeklendirme stratejisini adapte ettik. Ayrıca, çalışma zamanının iyileştirilmesi için de Fruchterman ve Reingold'un kareleme yöntemi ile grafik işleme unitesi (GPU) üzerinde eş-zamanlı programlama stratejisini CoSE'ye uyguladık. Ek olarak, yay sabiti, serinleme faktörü ve benzeri parametrelerin ayarlanması işini de fiziksel sistemin davranışını önemli ölçüde etkilediği için dikkate aldık. Yaptığımız deneyler gösterdi ki, parametre ayarlamaları ve yukarıda bahsedilen metodların adaptasyonu ile birlikte CoSE algoritmasının çalışma süresi önemli ölçüde azalırken, sonuç çizgelerin görsel kalitesi ise önemli miktarda iyileştirildi.

Anahtar sözcükler: Etkileşimli çizge düzenleme, Otomatik çizge yerleşimi, Kuvvet yönelimli yerleşim, Bileşik çizge, FR kareleme yöntemi, Çok-seviyeli ölçeklendirme, Eş zamanlı programlama.

Acknowledgement

First and foremost, I would like to thank to my advisor, Assoc. Prof. Uğur Doğrusöz for his guidance and patience during my thesis study. He always inspired me for pushing myself harder and motivated me for finding the best solutions all the time.

I would also like to thank to the authority of Bilkent University for providing me with a good environment and facilities to complete my thesis. I am very grateful to all the instructors that helped me to improve my academic knowledge.

During the two years, I have shared my office with fun and friendly colleagues. Thanks to them, I always felt like at home. But my special thanks are to one of my best friends, Selçuk Onur Sümer. My graduate period would not be the same without him. Also, I would like to thank to Salim Arslan for his assistance during my graduation.

Finally, yet most importantly, I would like to thank my parents with all my heart for their endless love and support and to my friends for their companionships.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Information visualization is a study focusing on constructing visual representation of large-scale textual information. It is widely used in research and analysis of data because visual materials prevent people from losing interest on a subject, and more importantly, make it easier to analyze and understand the underlying data. Information visualization has applications in scientific research (Figure 1.1), marketing analysis, cost optimizations, crime mapping (Figure 1.6), etc.

Graphs are widely used for visualizing the relational data such as social or biological networks. Using graphs for visualization allows one to run queries on the data as well. These queries include: "Find all the paths between two nodes", "Find the shortest path between two nodes" and "Find the n^{th} degree neighbour of a node". Using graphs for visualization also allows integration with other systems via pre-defined standards like BioPAX [12] or GraphML [13].

Drawing a graph is basically producing a picture of a graph topology. Generally, nodes are drawn in a circular or rectangular form or simply as dots, while edges can be drawn as straight line segments, arcs, or arrows (if the graph is directed) that connect the nodes. A graph can be drawn in infinitely many ways,

Figure 1.1: Radiographs showing different parts of the human body [1].

so aesthetics is an important issue for both manual and automated analysis. In a quality layout of a graph, number of edge crossings and overlapping nodes should be low, all nodes should be distributed evenly on the drawing area, adjacent nodes should be near each other and size of the drawing area should be small [7]. Moreover, picture of the graph should exhibit structural properties like symmetry, or there should not be any edge crossings for planar graphs. Beyond the visual quality measures, layout calculations should take a short time. Execution time is desired to be less than the user interaction time, which is about two seconds.

As the complexity and the size of the data to be analyzed increases, compound and clustered graphs are used more frequently. A textual representation of pathway of ATM mediated phosphorylation of repair proteins is shown in Figure 1.2. This snapshot is taken from Pathway Commons [2] which provides a wide biological network data in BioPAX standard. It also allows to run simple queries like: "In which networks does a specific protein exist?".

It is necessary to compound and clustered graphs instead of the simple ones

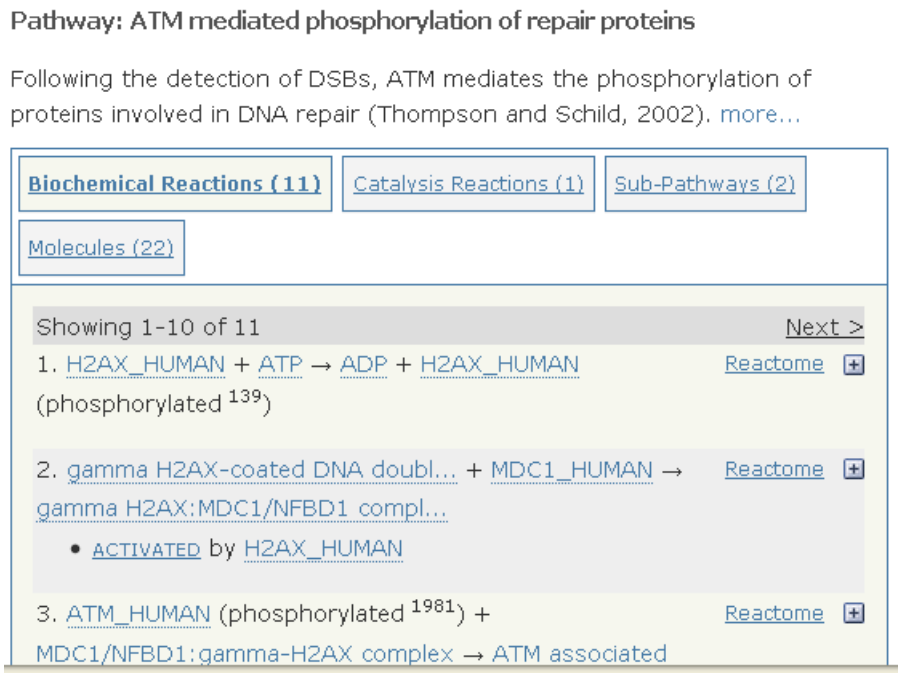

Figure 1.2: Pathway Commons representation of the pathway of ATM mediated phosphorylation of repair proteins [2].

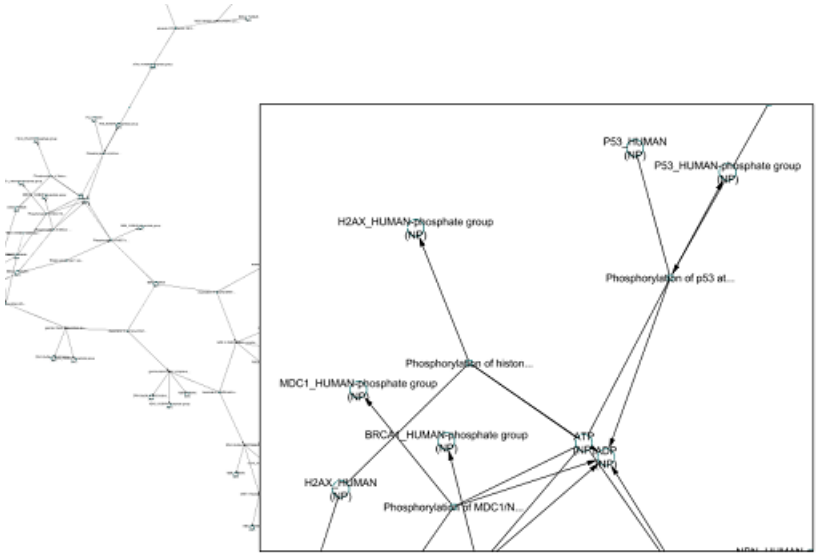

Figure 1.3: Cytoscape representation of the pathway of ATM mediated phosphorylation of repair proteins [3].

in order to indicate the classifying information and manage the complexity of the data. Clustering and hierarchically organizing the input data is required because most of the time they are in the nature of complex data, and play an important role in analysis.

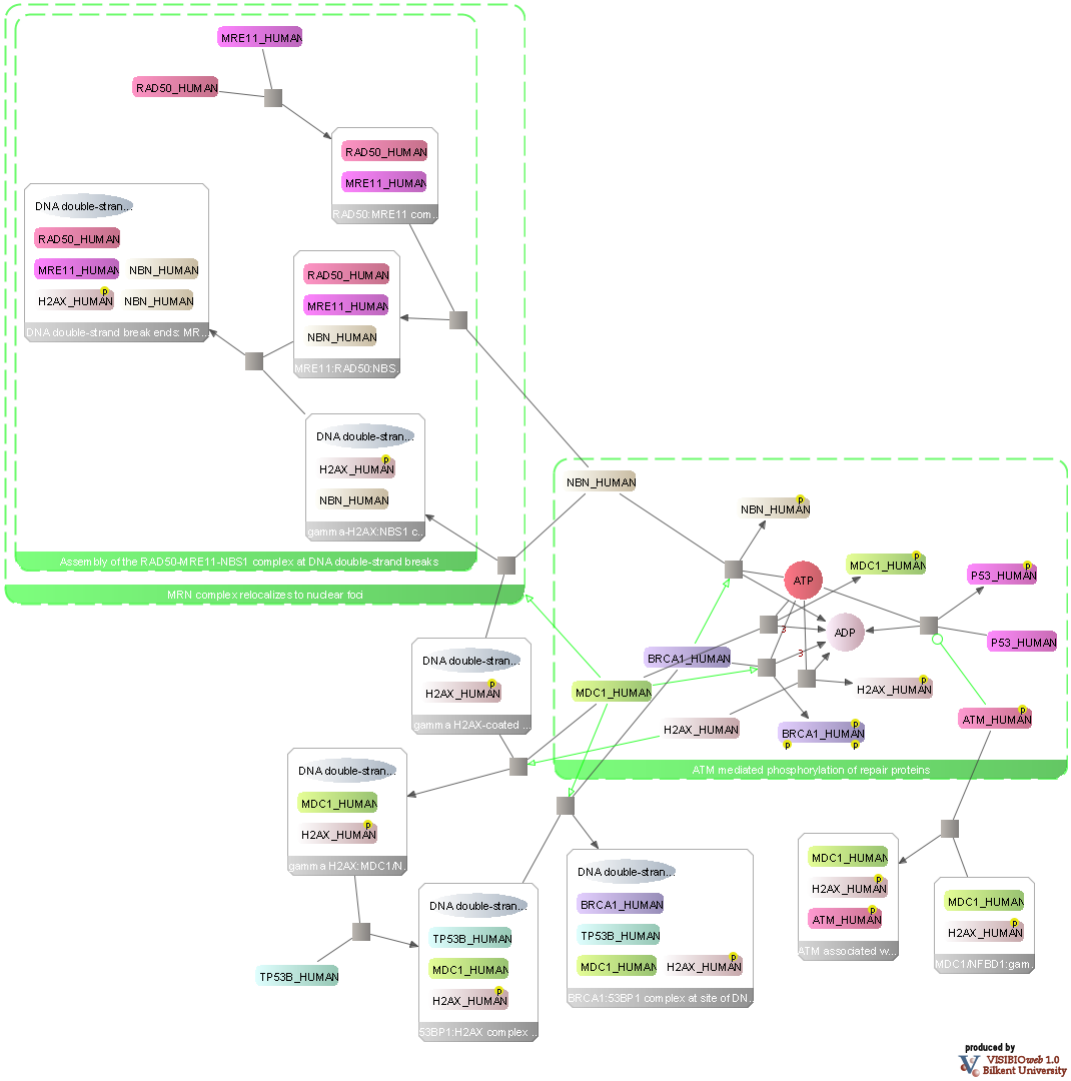

Figure 1.4: VISIBIOweb representation of the pathway of ATM mediated phosphorylation of repair proteins (abstractions at different levels are shown) [4].

Visual representation of the same pathway in Figure 1.2 is shown in Figure 1.3. Although compound and clustered structures are hidden, one can easily say that, visual representation will be more useful than the textual one. For example, results of the queries like "What is the shortest path between protein a and

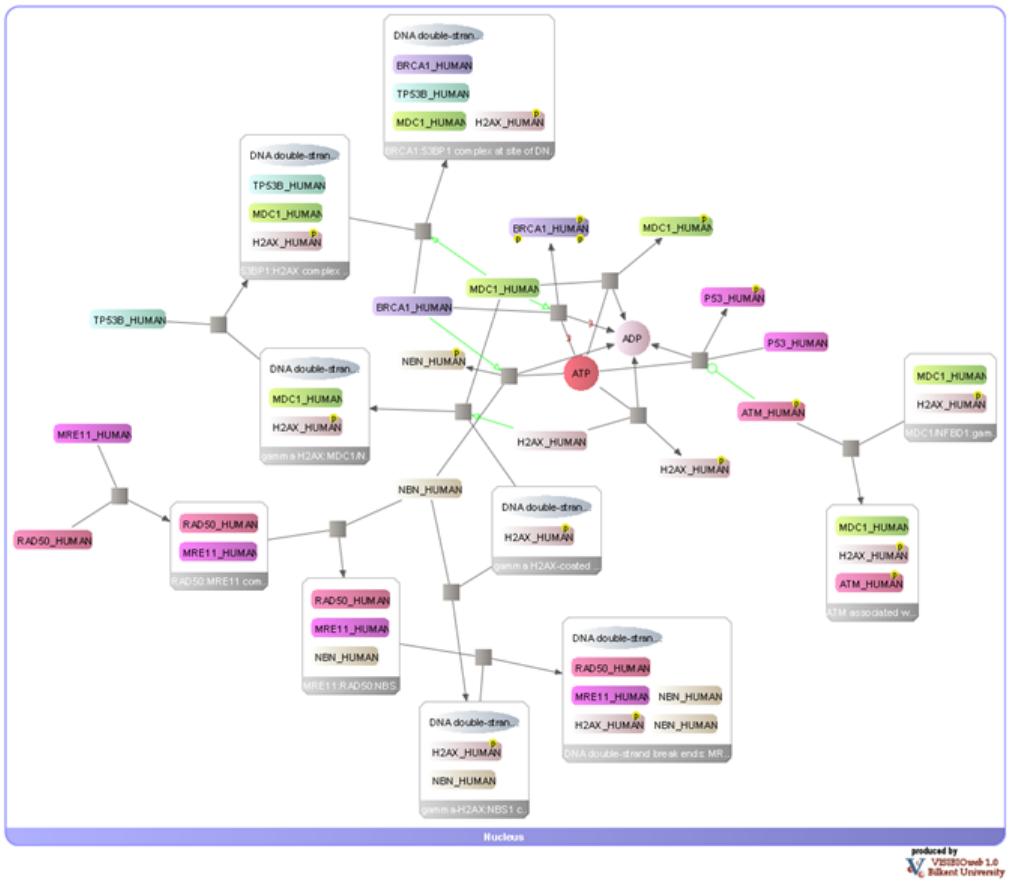

Figure 1.5: VISIBIOweb representation of the pathway of ATM mediated phosphorylation of repair proteins (cellular compartments are shown with clustered structure) [4].

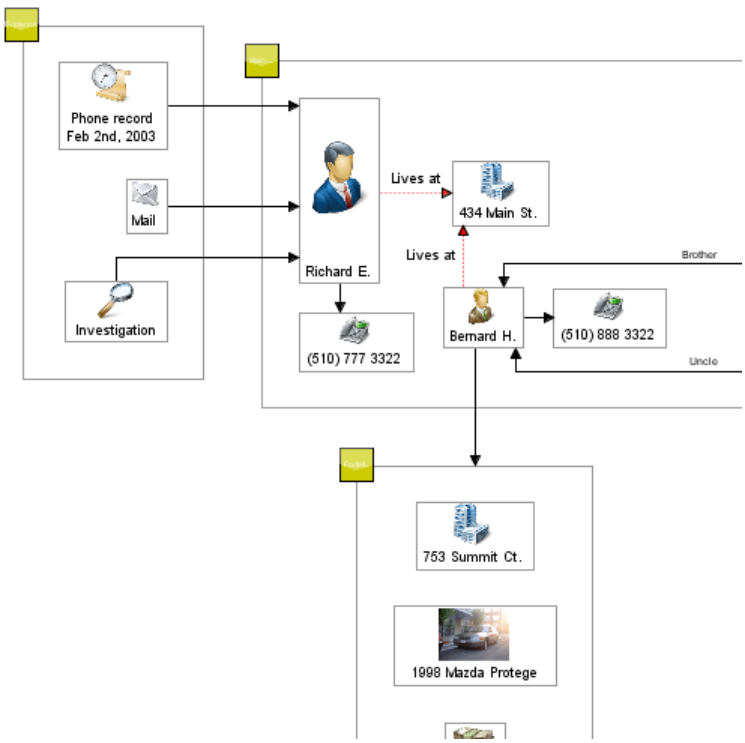

Figure 1.6: A map which can be used by intelligence agents for investigating a case (produced by [5]).

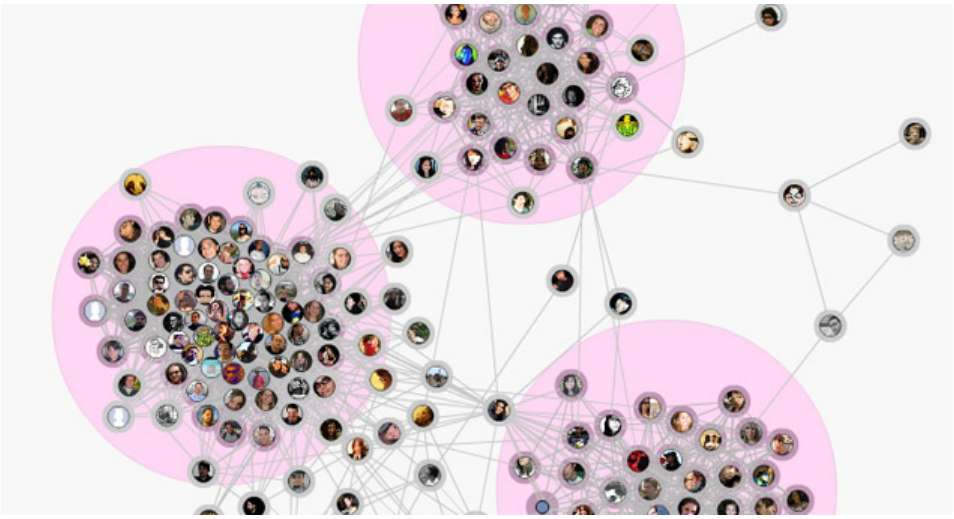

Figure 1.7: A clustered Facebook friendship map [6].

protein b, in a specific biological network?" can be visualized.

After using compound (Figure 1.4) and clustered (Figure 1.5) structures, the analyzed data aforementioned before become more readable since abstraction at different levels is revealed and cellular compartments are shown clearly. A compound graph used in intelligence assessment is shown in Figure 1.6. Another clustered graph used in social network analysis is shown in Figure 1.7.

1.2 Contribution

In this thesis, we improved the compound spring embedder (CoSE) introduced in [8]. For improving the performance of CoSE, we adapted Fruchterman and Reingold's grid variant method [7] and applied the parallel computing strategy [14]. Furthermore, we applied a multi-level scaling method [11] in order to improve the visual quality of the graphs produced by CoSE. These methods are described in sections 2.4, 2.5 and 2.6, respectively. Mentioned methods and strategies are studied, adapted to CoSE, and tested with example graphs. Obtained results are satisfactory. After the adaptation of the FR-Grid Variant method, CoSE runs 5 to 35 times faster. Applying the parallel programming strategy also makes CoSE run 1.5 to 15 times faster. Also, visual quality is significantly improved by adapting the multi-level scaling strategy. Adaptation processes, obtained results, discussions and possible future work are described in Chapter 3, 4 and 5. Additional discussions and the conclusion can be found in Chapter 6.

Chapter 2

Background and Related Work

2.1 Graphs

A graph $G = (V, E)$ is a pair of a vertex (node) set V and an edge set E [15]. An edge $e = (u, v)$ connects two vertices u, $v \in V$; thus elements of E are 2-element subsets of V. Graph G is called the owner graph of all nodes and edges; conversely, nodes $v \in V$ and edges $e \in E$ are called the members of graph G. Given an edge $e = (v, u)$; v and u are called *adjacent vertices* (or *neighbors*) and they are said to be *incident* to e. Also, two edges e and f are adjacent, if $e \neq f$ and they are connected to a common vertex. The degree of a vertex v is the number of *incident* edges to it, and denoted as $deg(v)$.

Geometry of a simple graph can vary in different applications. A vertex can be represented as a single dot, or a fixed sized disk, rectangle or triangle. For convenience, all 2-dimensional vertex representations can be thought to cover a rectangular area. Thus, for storing the geometry of a vertex, the top left point (or any reasonable reference point) and width and height of the rectangular area should be held for each vertex.

A compound graph $C = (V, E, F)$ consists of nodes V, adjacency edges E, and inclusion edges F. The inclusion graph $T = (V, F)$ is a rooted tree, where the

hierarchical structure of a compound graph is stored. It is assumed that $E \cap F$ is empty, which means a node cannot be connected to its children or parents by an adjacency edge. For the compound graph in (Figure 2.1),

$V = \{a,b,c,d,e,f,g,h,i,j\}$
$E = \{\{a,b\},\{a,g\},\{d,e\},\{d,g\},\{f,g\},\{f,h\},\{g,h\},\{i,j\}\}$
$F = \{bc,bd,be,cf,cg,ch,ei,ej\}$.

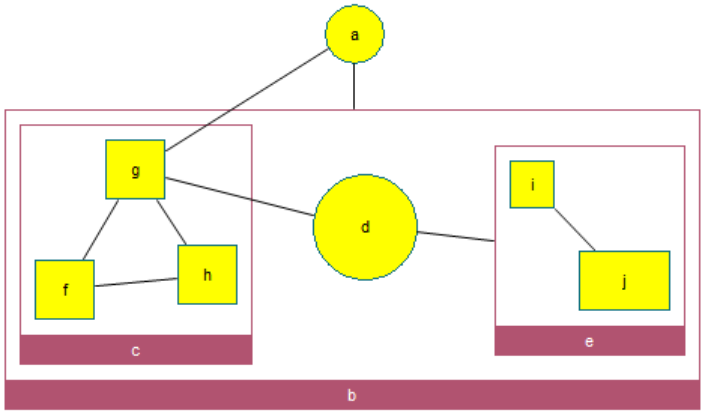

Figure 2.1: A sample compound graph.

If a node contains a graph, it is called a *compound* node; otherwise, it is called a *leaf* or *non-compound* node [8]. A graph inside a compound node is named *child graph* of that compound node. Edge $e = (u, v)$ is an *inter-graph edge* ($e \in I$), if $u \in V^{G_i}$, $v \in V^{G_j}$ and $i \neq j$. If $i = j$, then e is called *intra-graph edge*. *Root graph (G_0)* is a virtual graph that contains nodes that have no owner graph and resides at the 0^{th} level of the parent-node - child-graph hierarchy. In addition, it is assumed that *root graph* is owned by a virtual node. For the sample compound graph in (Figure 2.1),

$G_0 = \{\{a,b\},\{\{a,b\}\}\}$, $G_1 = \{\{c,d,e\},\{\{d,e\}\}\}$
$G_2 = \{\{f,g,h\},\{\{f,g\},\{f,h\},\{g,h\}\}\}$, $G_3 = \{\{i,j\},\{\{i,j\}\}$, and
$I = \{\{a,g\},\{d,g\}\}$.

In order to handle compound graph structures, a *graph manager* M =

(S, I, F) structure is introduced in [8]. S is the set of all child graphs including the root graph. I is the inter-graph edge set. And F is the inclusion graph. Notice that, definition of the set F is extended for application related reasons. In CoSE, F contains all the graphs including the root graph, and the parent-node - child-graph hierarchy excluding the root graph and the root node. For the compound graph in (Figure 2.1),

$S = \{G_0, G_1, G_2, G_3\}$, and
$F = \{G_1, G_2, G_3\}$, $\{bG_1, G_1c, G_1e, cG_2, eG_3\}$.

Introducing compound structures into graph topology makes it requisite to re-factor the graph geometry. Thus, boundary of a compound node is formed by its child graph plus a margin on all sides. Geometry of a child graph can be stored relatively (with respect to the parent compound node) or absolutely. In CoSE, nodes of child graphs are placed relatively to its owner's coordinate system. Upper left point of a compound node is defined as the origin (0,0), which is compatible with the coordinate systems of most drawing frameworks (e.g., *Java Swing* [16]).

2.2 Automated Layout Calculation

As the amount of the data grows, it gets harder to layout the graph representation of the input data in an eye pleasing way on the screen. Therefore, in order to benefit from advantages of the visual materials on analysis or research, layout should be automated.

Some existing layout algorithms treat vertices as points, and edges as straight line segments that connect the vertices. Also, in such algorithms, graph is assumed to be undirected. *Force-directed placement (layout)* algorithms are flexible, easy to understand and implement since they model a mechanical system. Eades [17] and Fruchterman and Reingold [7] use spring forces similar to the formula from the Hooke's law. In Hooke's law, a restoring force F is defined as $F = -kx$,

where k is the spring constant and x is the distance of the end of the spring from its equilibrium or ideal position. Kamada and Kawai [18] also use spring forces, but they consider the graph theoretical distance, instead of the Euclidean distance.

In Eades' algorithm [17], vertices are replaced by steel rings, and edges are replaced by springs. Steel rings repel each other, and springs apply attractive forces to the rings on both ends. The complexity of this algorithm is $O(|E|+|V|^2)$. Basically, system works as follows: initially, vertices are placed on the Euclidean space randomly. Then, the system is released, so that the spring and electrical forces exerted on the rings move the system to a minimal energy state (*global minima*) (See Figure 2.2).

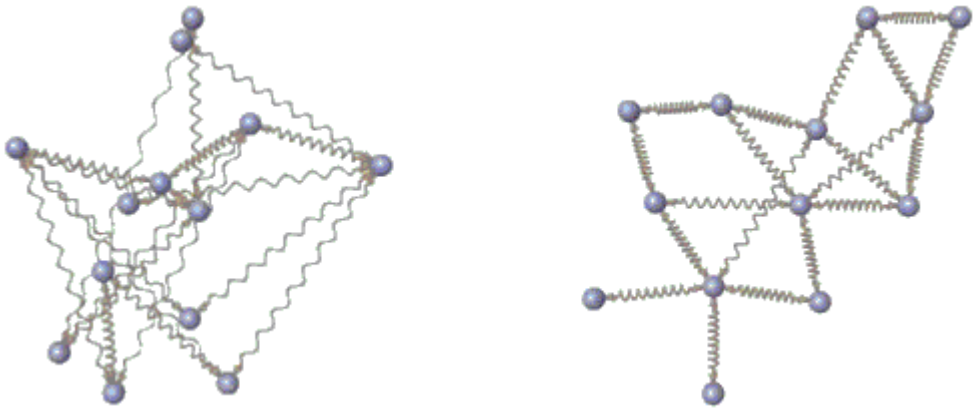

Figure 2.2: Randomly placed particles in initial layout (left), final placement of particles when the system is converged (right) [7].

Kamada and Kawai's algorithm [18] is a variant of the Eades'. Eades considers only the ideal distance (edge-length) between adjacent vertices. In addition to the Eades' method, the ideal distance between non-adjacent vertices are also considered by Kamada and Kawai. They calculate the ideal distance between any two vertices proportionally to the graph theoretical distance between them. They see the placement problem as a process of reducing the total energy of the mechanical system. The total energy of the system is reduced by solving a partial differential equation for each vertex to find a new location. Repositioning continues after the total energy becomes less than a threshold, where all vertices are placed closely at their ideal distances from each other.

Fruchterman and Reingold's spring embedder algorithm [7] is also based on the work of Eades. Vertices behave like atomic particles. Repulsive forces are calculated for each vertex pair, whereas attractive forces are calculated only between neighbor vertices. Like Kamada and Kawai, they also consider the ideal distance between any pair of vertices during the layout. Let k be the optimal distance between any vertex pair. It is calculated as follows:

$$k = C\sqrt{\frac{area}{|V|}}$$

C is a constant found experimentally, *area* is the bounding box of the graph (or simply the drawing area) and $|V|$ is the number of vertices. k gives the ideal radius of the empty circle around a vertex. Let d_p be the distance between the vertex pair $p = (u, v)$. And f_a and f_r be the attractive and repulsive forces applied on u and v. f_a and f_r for p are calculated as follows:

$$f_a(d_p) = (d_p)^2/k$$
$$f_r(d_p) = -k^2/d_p$$

Attraction force f_a between two neighbor nodes is proportional to the distance between them. On the other hand, repulsion force f_r between any node pair is inversely proportional to the square of the distance. When the distance between two vertices equals k, f_a and f_r cancel each other out (See Figure 2.3).

After finding repulsive and attractive forces, these forces are partially applied on the particles limited by the *temperature*. So that, movement of particles is limited to some maximum value which decreases over time, since the temperature of the system cools down. As the system approaches a stable state, particles move slower, and finally, stop.

Attractive force is the heart of spring-based layout algorithms. Without repulsive or other forces, a rough sketch of a graph can still be achieved. Because of that, movements caused by attractive forces are relatively greater than other forces for most of the spring embedders (This is also why they are called spring-based or spring embedder). This does not mean that, repulsive forces can be

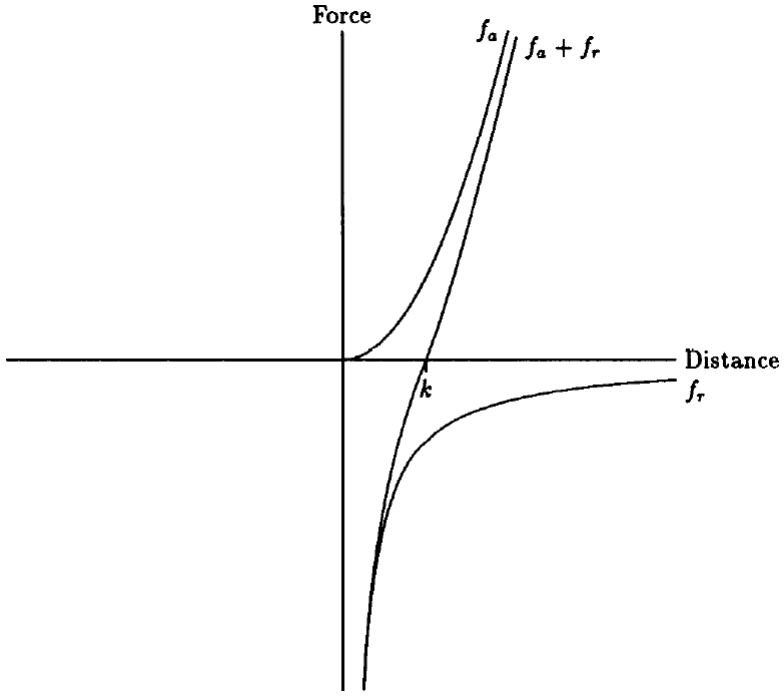

Figure 2.3: Attractive and repulsive forces versus distance [7].

ignored. They keep the nodes at acceptable distances from each other and help embedders to avoid node overlaps. But still, repulsive forces are weaker than the attractive ones.

The term of *force* is not correctly used for spring and electrical force oriented methods. *Force* induces acceleration on bodies in physics, while it is used to calculate the velocity of bodies (atomic particles) for every time quantum or each iteration. The real definition of the force leads to a dynamic equilibrium, whereas the force-directed methods seek a static equilibrium [7].

2.3 Compound Spring Embedder (CoSE)

Compound spring embedder algorithm is based on force-directed placement scheme, that handles non-uniform node sizes, inter-graph edges, and clustered and

compound graph structures [8]. CoSE extends the model proposed by Fruchterman and Reingold. It roughly simulates a mechanical system, where nodes behave like charged particles and edges behave like springs. If a spring is shorter than its desired length, it pulls the particles it connects, and repels vice versa. Electrical force between charged particles avoids overlapping nodes. Moreover, it helps to distribute non-adjacent nodes evenly onto the layout area. Additional to the attractive and repulsive forces, CoSE uses gravitational forces to keep disconnected graphs together. Isolated nodes are pulled to the barycenter in order to keep the drawing area small.

A compound node is treated as a single entity, like an elastic cart. It has its own barycenter, can move only in orthogonal directions, and shrinks or grows respectively to the bounding box of its child graph (see Figure 2.4). For the sake of simplicity and efficiency, repulsive forces are calculated for nodes which reside in the same graph. Ideal lengths of all intra-graph edges are equal to each other, and pre-defined per layout calculation. However, ideal length of an inter-graph edge is calculated proportionally to the sum of the depths of its end-nodes from their common ancestors in the parent-child hierarchy.

Most of the previously proposed layout algorithms assume the nodes as points or uniform sized. Hence, calculating the distance between any node pair is not an issue for such algorithms. In CoSE, distance between a node pair is the distance between the clipping points of the line that passes through the centers of the nodes. This calculation is costly, but still required. Despite the distance between the centers of two nodes being long, nodes may seem too close to each other, or even be overlapped (if at least one of them is really big). In such situations, it is expected for two nodes to repel each other.

CoSE layout algorithm starts with an initialization phase. Parameters (like spring or repulsion constant) are set to their initial values, and all nodes are positioned randomly.

After the initialization, simulation starts. Iteratively, attraction, repulsion and gravitation forces are calculated and then applied to each node. Temperature of the system is cooled down periodically which allows the physical system to reach

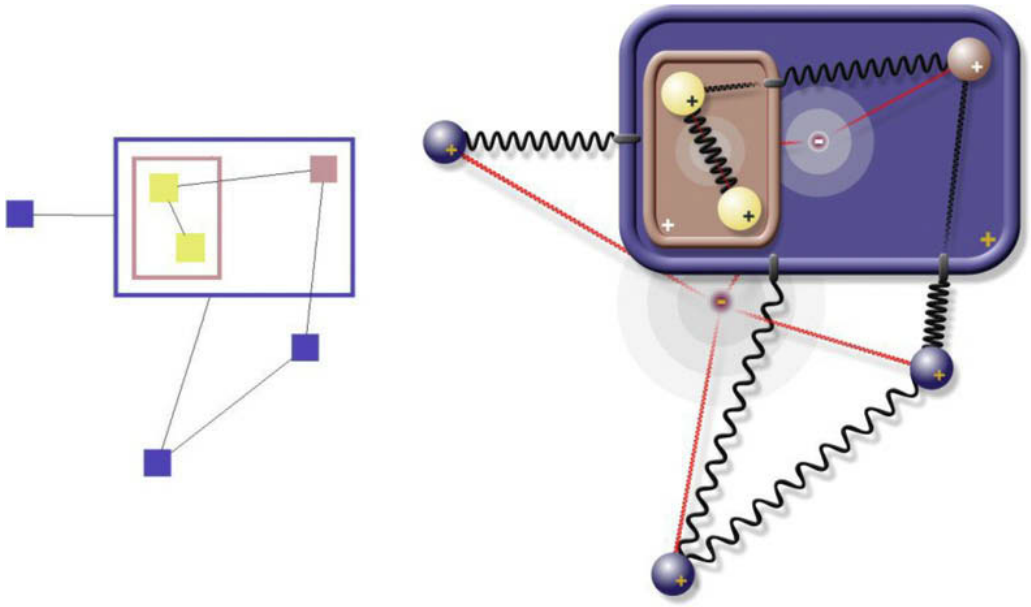

Figure 2.4: A sample compound graph (left), corresponding physical model (right). Grey circle: barycenter, red solid line: gravitational force, zigzag: regular spring force, black solid line: constant spring force [8].

a stable state. When total movement of nodes drops below a threshold value, spring embedder stops. At this point, the layout is said to *converge*.

Attractive forces are calculated for each edge, while repulsive forces are calculated for each vertex pair. This makes the complexity of one iteration of the CoSE algorithm $O(|E| + |V^2|)$. If the distance between two vertices is greater than some threshold value, then the repulsion force between those vertices is neglected. But still, distances between all vertex pairs are checked, and that makes the complexity of the repulsion force calculation quadratic in $|V|$. Maximum number of iterations is calculated to be proportional to the number of vertices; however empirical results show that number of iterations almost never increases with the graph size.

2.4 Fruchterman and Reingold's Grid Variant

Main goal of the force-directed placement algorithms is not to simulate a physical or mechanical system, but obtain pleasing layouts in a reasonable time [7]. In order to improve their force-directed algorithm, Fruchterman and Reingold focus on decreasing the complexity, which is $O(|E| + |V^2|)$. Calculating attractive forces costs $O(|E|)$, and calculating repulsive forces costs $O(|V^2|)$. So, although, repulsive forces are not considered as the heart of the embedder, calculating them increases the complexity.

The repulsive force between two particles decreases quadratically as the distance increases. So Fruchterman and Reingold ask the question "Can we neglect the contribution of the more distant vertices?", and propose the *FR grid variant algorithm*. In this method, the bounding box is divided into a grid of squares, and at each iteration, each vertex is placed in its grid square. To calculate the repulsive forces applied to a vertex v in grid s, only the vertices in the neighbor grids of s are considered. Square shape of the grid boxes causes distortion. In order to prevent the distortion, for each vertex, vertices that are distant more than one grid unit are ignored in repulsion force calculation. Formula of the repulsive force f_r is updated as below:

$$f_r(d) = \frac{k^2}{d} u(2k - d)$$

where

$$u(x) = \begin{cases} 1 & \text{if } x > 0; \\ 0 & \text{otherwise.} \end{cases}$$

One problem is: What should be the grid unit r? If r is set to a small value, then tangling between vertices may increase since the number of repulsing vertices decreases. On the other hand, if r is set to a large value, then we get eye-pleasing layouts, but we have a huge time penalty. Fruchterman and Reingold proposes $r = 2k$, where k is the desired length between any pair of vertices. For the graph in Figure 2.5, despite the fact that both vertices q and s reside on the nearby grid

cells to the vertex v, only the repulsion force between q and v will be calculated since the distance between s and v is greater than the grid unit.

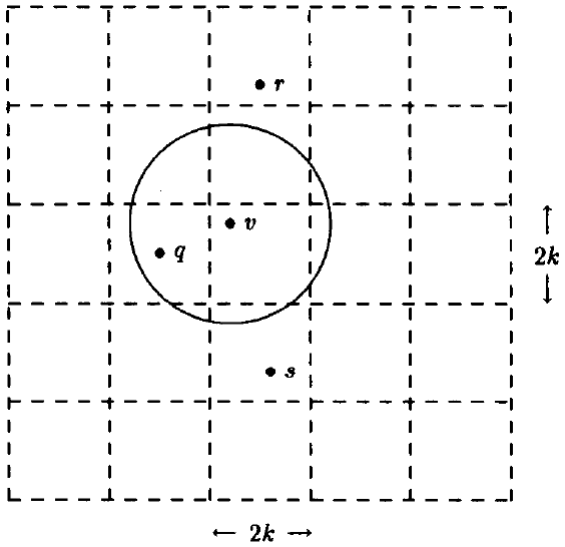

Figure 2.5: A squared bounding box of a graph [7].

Let w be the width and h be the height of the drawing box. Area a reserved for a vertex and number of grid cells will be

$a = \frac{wl}{|V|}$,
$k = \sqrt{a}$
number of grid cells$= \frac{w}{2k}\frac{l}{2k} = \frac{|V|}{4}$

In conclusion, when vertices are uniformly distributed, one grid cell contains nearly 4 vertices; therefore, approximately 35 vertices are considered for repulsion force calculation of a vertex. So, if vertices are uniformly distributed, complexity of the repulsion force calculation is asymptotically decreased to $O(|V|)$. On the other hand, for the worst case, complexity is $O(|V^2|)$, since it is still possible to check each vertex pair.

2.5 Multi-level Scaling

Force-directed placement algorithms give satisfactory results for relatively small graphs ($|V| < 100$). For larger graphs and graphs that contain a particular structure, physical systems implemented via such algorithms can be converged pre-maturely or cannot be converged although the number of iterations reaches the allowed maximum value.

Multi-level scaling methods focus on obtaining an abstract of the original graph G_0. G_0 is coarsened recursively until some conditions are satisfied (conditions may differ due to the coarsening method). Each coarser graph G_{i+1} is an abstract of the finer graph G_i. When the coarsest graph G_{k-1} is constructed, layout process starts. Final placement of G_{i+1} is used for the initial placement of G_i.

General algorithm of layouts using multi-level scaling strategy is shown in Figure 2.6.

```
method LAYOUT(Graph G)
1) i := 0
2) G_0 := G
3) while not all conditions hold do
4)   G_{i+1} := COARSENGRAPH(G_i)
5)   increase i by 1
6) decrease i by 1
7) while i ≥ 0 do
8)   CALCULATELAYOUT(G_i)
9)   if i ≥ 1 then
10)    interpolate position from (G_i) to G_{i-1}
11)  decrease i by 1
```

Figure 2.6: Layout algorithm with multi-level scaling method.

Two methods for coarsening process are studied and will be discussed in the following sub-sections.

2.5.1 Solar System

In the solar system method [9], there are 4 types of elements (vertices): In each solar system there is exactly one sun node (*s-node*). All vertices adjacent to the *s-node* are called either planet node (*p-node*) or planet with moon node (*pm-node*), and finally, there are moon nodes (*m-node*). A *p-node* does not have any moon while *pm-nodes* have exactly one moon. Directed edges are used between *m* and *pm-nodes* which indicate that current *m-node* is assigned as the moon of the current *pm-node*. (See Figure 2.7 for an example.)

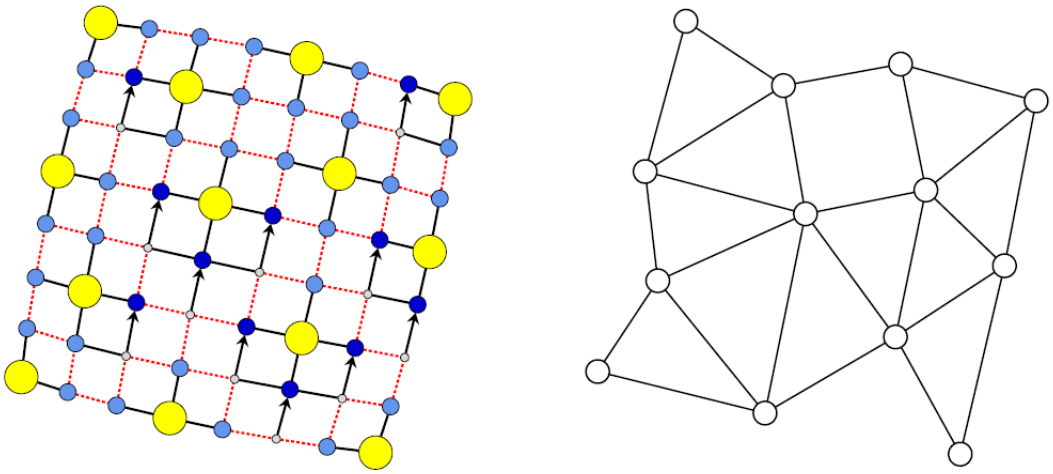

Figure 2.7: Solar system representation of an $8x8$ mesh graph G_0 (left), coarser graph G_1 of G_0 (right) [9].

Big yellow, medium light blue, medium dark blue and small grey disks are used to show *s-node*s, *p-node*s, *pm-node*s and *m-node*s, respectively. Solid black lines represent intra solar system edges, while dashed red lines represent inter solar system edges. Arrows connects *pm-node*s and *m-node*s. Coarsening algorithm proposed in [9] is shown in Figure 2.8. An example of an edge coarsening process can be found in Figure 2.9.

When the coarsest graph G_{k-1} is obtained, layout and refinement phases are started. G_{k-1} is laid out with a force-directed placement method. Final positions of vertices in G_{k-1} will be used for initial placement of G_{k-2}. This process continues until the finest graph G_0 is reached. Each vertex in a coarser graph G_{i+1}

method SOLARSYSTEMPARTITION*(Graph G)*
1) $G' := (V', E')$
2) copy V to V'
3) $i := 0$
4) **while** $|V'| > 0$ **do**
5) $s_i :=$ *randomly selected vertex from* $|V'|$
6) remove s_i and all nodes have graph theoretical distance 2 or less from s_i
7) increase i by 1
8) **for** each *s-node* s_i **do**
9) label neighbors of s_i as *p-node*
10)**for** each non-labelled node v in V **do**
11) label v as *m-node*
12) label nearest neighbor of v as *pm-node*
13)**for** each connected *solar system* pair (S_i, S_j) **do**
14) $l :=$ average ideal length of all paths
between *s-nodes* s_i and s_j (suns of S_i and S_j)
15) $e' := (s_i, s_j)$
16) $e'.idealLength := l$
17) add e' to E'
18)return G'

Figure 2.8: Solar system coarsening algorithm [9].

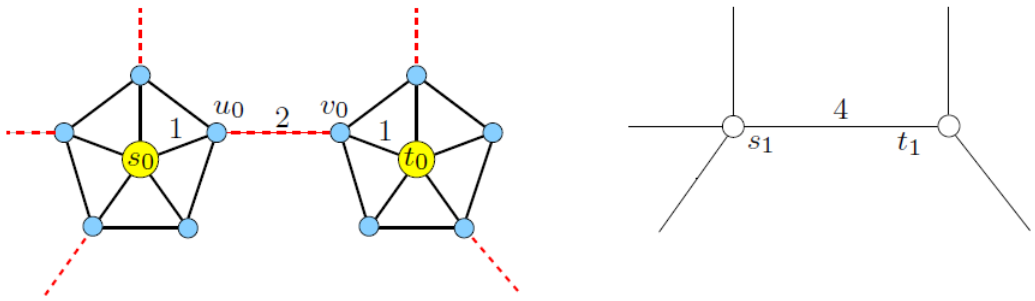

Figure 2.9: A part of G_i: 2 Solar Systems S_0 and T_0 with *s-nodes* s_0, t_0, *p-nodes* u_0, v_0 (left), corresponding part of coarser graph G_{i+1} (right) [9].

is the representation of a solar system in the finer graph G_i. Let S_i and T_i be two solar systems in G_i. Thus, *s-node*s s_{i+1} and t_{i+1}, which are members of G_{i+1}, will represent the solar systems S_i and T_i, respectively. When the layout calculation of G_{i+1} is finished and initial placement of G_i is made, s_i and t_i are put exactly to the same place where s_{i+1} and t_{i+1} stand. Planet and moon nodes in the solar system are put around the sun node. Desired edge lengths calculated before are considered during the initial placement phase (See Figure 2.10).

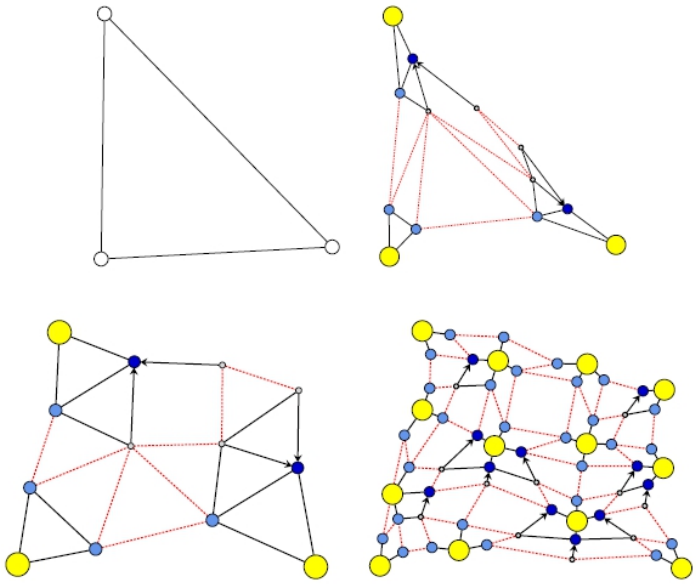

Figure 2.10: Final placement of G_2 (left-up), initial placement of G_1 (right-up), final placement of G_1 (left-bottom), initial placement of G_0 (right-bottom) [9].

2.5.2 Clustering

Walshaw [11] proposes the coarsening approach known as clustering (or matching), where vertices are matched with one of their neighbors or themselves (if there is no neighbor to match with). Thus, cluster sizes are one or two. Coarsening process ends if $|V(G_k)| < 2$.

There are several ways to find the matching of a graph. The problem of finding the optimum matching, where the number of unmatched vertices is minimized, is called "maximum cardinality matching" problem. It is a costly operation with

the complexity $O(|V^{2.5}|)$ [11]. However, Walshaw does not seek for the optimum solution, but fast and an efficient one. So, he decided to use the matching algorithm that is proposed by Hendrickson & Leland [19]. They use a randomly ordered list of the vertices and traverse this list to match each unmatched vertex with an unmatched neighboring vertex (or with itself if there is no such vertex). Matched vertices are removed from the list.

Let u_i and u_j be the vertices that are matched (contracted). Weight of the resulting vertex v will be $|v| = |u_i| + |u_j|$. If there is more than one unmatched neighbor, then the neighbor with the smallest weight is chosen for matching in order to keep the coarser graphs balanced. Note that even if G_0 is non-weighted, G_i for $i > 0$ will be weighted.

Matching all vertices with another one is the best case for the Walshaw's clustering method. In this case, number of levels $k = \log_2 |V|$. This case is guaranteed to occur for complete graphs. On the other hand, matching a star-shaped graph, where all the edges in the graph is incident with one node, is the worst case for this algorithm, since only one matching pair contains two vertices, while other pairs contain only one vertex. This makes $k = |V| - 1$. Our experiments show that, mesh-like graphs are closer to the best case; however tree-like ones are closer to the worst case.

2.6 Graphics Processor Unit (GPU) Support for Parallel Computing

Almost all proposed automated graph layout algorithms (such as [7] and [8]) consider solving the problem sequentially. In such algorithms, in order to calculate the repulsion or attraction forces that will be exerted on node v_i we have to wait for force calculations of all nodes v_j where $j < i < |V|$. This is actually unnecessary, since force calculation of a node has no dependency to or interaction with force calculation of other nodes. Thus, force calculations, the major part of the automated layout algorithms, can be parallelized.

2.6.1 Graphics Processor Unit (GPU)

GPUs are powerful for practicing parallel algorithms since they have an implementation of single instruction, multiple data (SIMD) architecture. Moreover, they contain hundreds or thousands of cores whereas central processing units (CPU) contain only a few.

*Thread*s are smallest virtual processing units in a GPU. They are organized into one, two or three dimensional *block*s. All threads run in the same block, reside at the same core. So, there is a limit for number of *thread*s per *block*, since threads in a block must share a limited memory space of a processor core. Blocks are also organized into one, two or three dimensional *grid*s (Figure 2.11). The thread-block-grid hierarchy provides a natural way to operate on elements shaped as a vector, matrix, or cube.

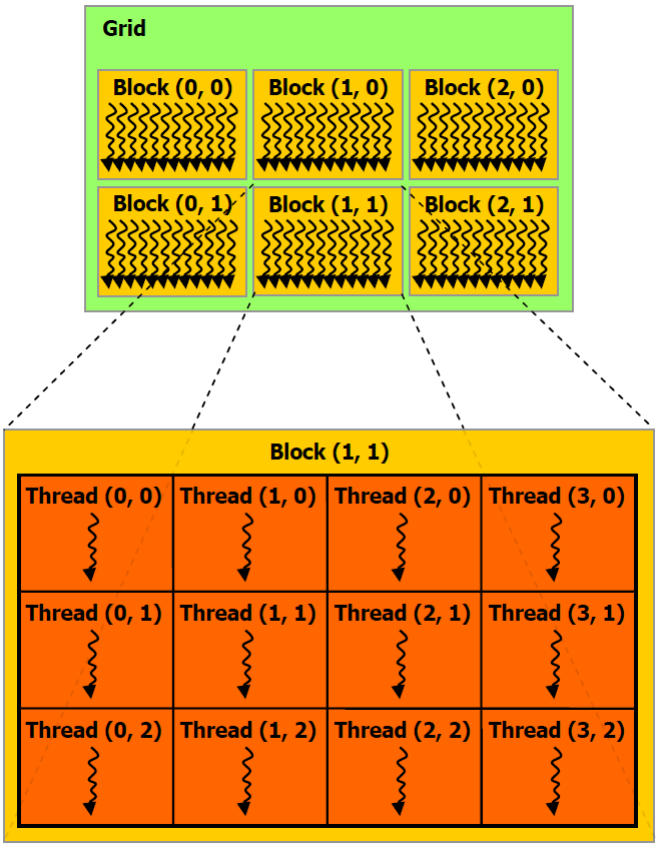

Figure 2.11: A 2×3 sized *grid* with 3×4 sized *block* [10].

A GPU contains special memory types for different programming purposes (Figure 2.12). *Global memory* is accessible from any threads, it is writeable, but the slowest memory type in the *device*. *Texture memory* is also accessible from any threads, and it is specialized for handling *CUDA array* memory. Texture memories are faster than the global one but they are read only. *Shared memory* is also faster than the global memory but is accessible for only the threads in the same block (processor core). Also, if CPU memory can be mapped to the device memory address space, then it is available for *kernel* functions. *Mapped memory page* is pinned, and is guaranteed to be used only by the kernel functions. So mapping a large amount of data will affect other programs running in CPU, since only too little memory space become available for those programs. When using *mapped memory*, instead of copying data between CPU and GPU before the kernel launch, required data will be copied from *mapped* (*pinned*) CPU memory to the global GPU memory when needed. Pinning the CPU memory is advantageous if the GPU is integrated. In integrated systems, CPU and GPU memories are physically same, so copying data between CPU and GPU will be superfluous.

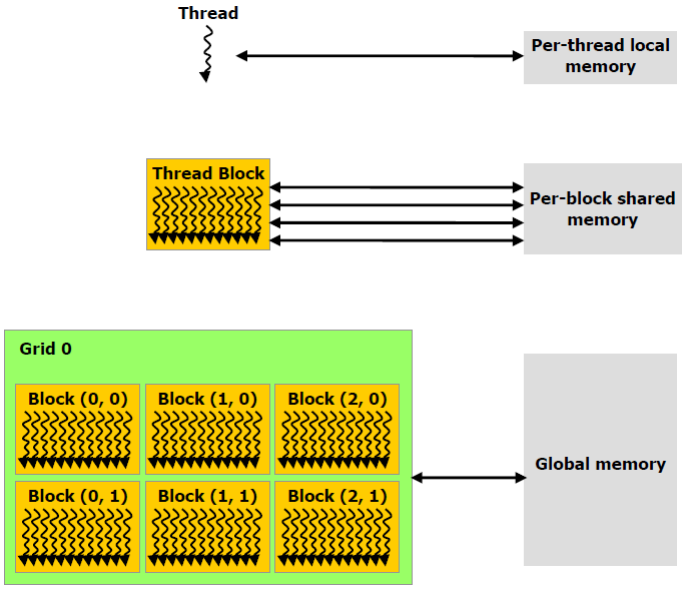

Figure 2.12: Memory hierarchy of GPU [10].

2.6.2 CUDA Framework

CUDA [20] is NVDIA parallel computing architecture. Using CUDA-enabled GPUs increases the performance of parallel programs dramatically. CUDA provides a subset of C programming language for implementing kernel executions which are run on GPU devices, and C and C++ are available for implementing the *host* executions which are run on the CPU. Beside the CUDA C and C++, there are other ways to take advantage of the power of GPU computing, such as OpenCL, DirectCompute, and CUDA Fortran.

One of the key abstractions of CUDA is a hierarchy of thread groups which allows the programmers to partition the problem into sub-problems and solve these sub-problems in parallel by blocks of threads. CUDA C extends C and allows programmers to implement C functions called kernels. When a kernel is called, it is executed n times in parallel by n different threads. In order to make kernel calls, programmers have to specify number of threads per block and number of blocks per grid. All threads in a block reside on the same processor. Grids are abstract structures for organizing blocks for different programming goals.

Chapter 3

Improving by Fruchterman and Reingold's Grid Variant

Fruchterman and Reingold proposed the grid-variant method in order to decrease the running time of the repulsive force calculations, which is $O(|V^2|)$. Complexity is quadratic since all vertex pairs are processed for repulsive force calculation. However the repulsive force between two distant vertices can be neglected, because it decreases quadratically as the distance between the vertices increases.

CoSE algorithm has exactly the same problem aforementioned above. If the distance between two nodes is greater than some threshold value, then the repulsion force between these nodes is not calculated, still, the distance control is done for all node pairs, and this causes the complexity of the repulsion force calculation to be $O(|V^2|)$.

In order to decrease the execution time of our compound spring embedder asymptotically, we decided to adapt the Fruchterman and Reingold's (FR) grid variant method.

Basically, in FR grid-variant method, drawing area is divided into a grid of squares, and for a vertex, only the vertices on neighbor squares are considered for repulsive force calculation. A distance check is done in order to prevent the

distortion. (See Figure 2.5)

3.1 Adaptation

Drawing area is the bounding rectangle (box) of the root graph in CoSE. Size of the length of a square is proposed as $2k$ in FR grid-variant method, where k is the ideal distance between any pair of vertices (or radius of the ideal empty field around a vertex). In CoSE, only the ideal length l of an edge, in other words the ideal distance between neighbor vertices, is defined. However, this ideal edge length can be used for the size of a square of the grid.

Finding the optimum grid square edge size or the repulsion range is not easy. As the repulsion range increases, the ideal empty area around a node increases too. On the other hand, increasing the repulsion range also increases the average number of nodes that apply a repulsive force on a specific node, so using wider grid squares grants a better looking layout but decreases the performance. Although, using narrow squares will increase the performance but, it produces worse looking layouts.

When using the multi-level scaling method during CoSE calculations, we can change the repulsion range for obtaining better looking layouts with a slight performance penalty. Since the coarser graphs contain fewer nodes, we can enlarge the grid squares proportional to the level of the graph. Let r be the repulsion range and l be the ideal edge length, then;

$$r = 2 \times (level + 1) \times l$$

For the input graph, where the level equals to zero, r becomes $2l$.

Bounding box of the root graph is virtually squared and divided into a grid of squares where the length of one side of the square is $2l$. Each grid square contains a collection of nodes. Since CoSE handles variable-sized nodes, one node can occur in more than one collection, unlike in the original FR grid-variant. This

can affect the performance negatively, if the input graph contains huge nodes. In such cases, a large number of grid squares will be checked for such nodes. This also makes it difficult to scale the performance of the adapted method.

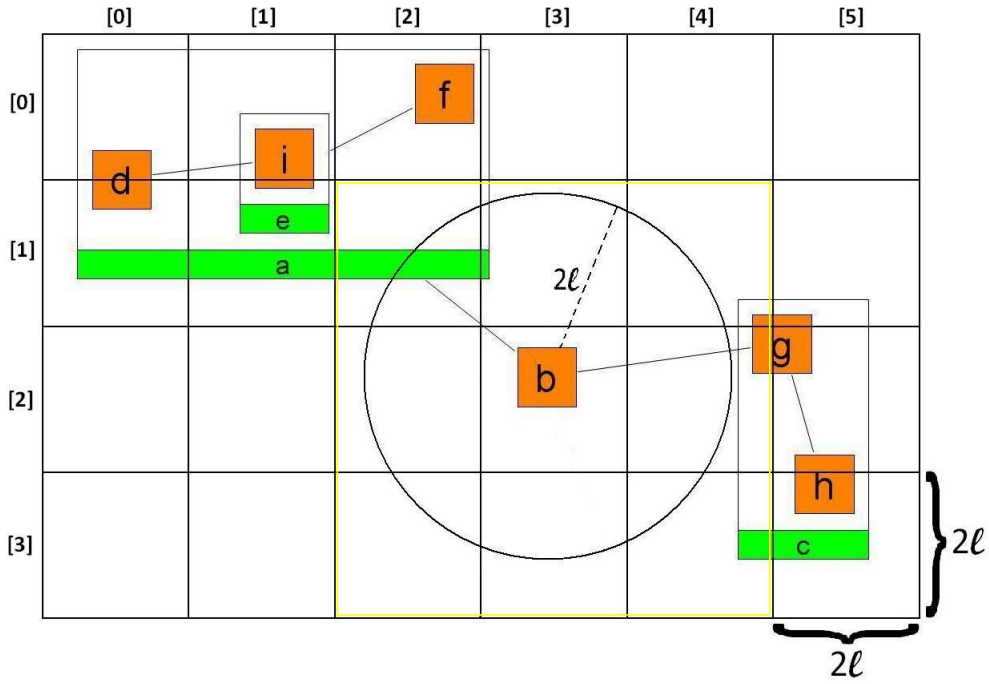

Figure 3.1: A gridded CoSE Graph. The circle around node b indicates the *repulsion circle* of b.

In order to fill the grid squares, the node list is traversed. Grid coordinates of each node are calculated, and each node is put to its corresponding grid cell(s) by simply adding the node in question to the corresponding collection(s). To easily access the grid cells occupied by a node, *start* and *finish* grid coordinates on both x and y axes are maintained for each node. In the sample graph in Figure 3.1, for compound node c, *start* and *finish* coordinates on x axis are 4 and 5, and *start* and *finish* coordinates on y axis are 1 and 3, respectively. For node b, both *start* and *finish* coordinates on x axis is 3.

Nodes that reside in the circular area of the radius $2l$ around each node are found and stored in a list named *surrounding*. For obtaining the *surrounding* list, all occupied cells with their neighbor cells are traversed. Since start and finish coordinates of each node are stored, CoSE will not need to traverse the whole

grid matrix for finding the occupied cells by a node. For instance, for calculating the surrounding of node b in Figure 3.1, all nodes in the highlighted matrix (cells 2 to 4 in x axis and 1 to 3 in y axis) are checked. The nodes in the *repulsion circle* of b (which is only a for the sample graph) are put to the *surrounding* list of b. Repulsion circle of b is the circle with a radius of $2l$ and b is in the center.

In order to achieve the most accurate spring embedder simulation, CoSE needs to update the grid cells before calculating the repulsive forces in each iteration. However, since we are looking for an improvement at the performance, and simulating a mechanical system just roughly, updating grid cells periodically is a good solution. Currently, CoSE re-calculates the grid cells for each node once in every ten iteration.

Once *surrounding* lists of all nodes are constructed, these lists are traversed for each node, and repulsive forces between a node and its surrounding can be calculated cumulatively as in the CoSE.

For nodes v and u, where the distance between them is less than the *repulsion range*, v is added to the *surrounding list* of u, only if u is not included in the *surrounding* list of v. This condition is checked in order not to calculate the repulsive force between a node pair twice. This violates the term of *surrounding* list, but saves great time in application.

```
method CalcRepulsionForcesWithFRGridVariant(VertexSet V)
1)  if number of iterations is multiple of 10 then
2)    construct empty grid cells
3)    for all nodes v_i ∈ V do
4)      calculate start and finish coordinates of v_i
5)      put v_i to corresponding grid cell(s)
6)    for all nodes v_i ∈ V do
7)      for all nodes u_i inside the neighbor grid cells of v_i do
8)        if distance between v_i and u_i is less than 2l
9)        and owner graph of v_i and u_i are same then
10)         add u_i to surrounding list of v_i
11)for all nodes v_i ∈ V do
12)   calculate the repulsion forces between v_i and its surrounding
```

Figure 3.2: Repulsive force calculations after the adaptation of FR-Grid Variant method.

3.2 Complexity Analysis

Variable sized nodes not only make it hard for the performance to scale but also the complexity analysis difficult for adapted FR-grid variant method. For convenience, analysis will be done for only fixed-sized nodes with default values 40×40 *pixels* for $level = 0$ and $l = 50$ (which is the default value for ideal edge length in CoSE).

Repulsion range r equals 100 with the default values above, which means a node with a 40×40 size, can occupy 4 grid cells in the worst case. Thus, number of grid cells to be checked for such a node is 16. As it is described in section 2.4, the number of grid cells is found as $\frac{|V|}{4}$, if nodes are evenly distributed, such that one grid cell contains about 4 nodes. In conclusion, approximately 64 nodes are considered for repulsion force calculation of a node, on average, so that complexity of the repulsion force calculation is $O(|V|)$ with a large constant.

Grid cells are refreshed in every ten iteration. This operation has two phases: constructing the grid, and calculating the surroundings of all nodes. Deciding the size of the grid takes constant time. Traversing the node list and putting the nodes in the correct grid cells take $O(|V|)$ time. Thus, total time to construct the grid is $O(|V|)$. In order to calculate the surrounding of each node, neighbor cells are traversed. As we make the analysis for fixed-sized nodes and assume that nodes are distributed uniformly, there will be 16 grid cells to traverse and approximately 64 nodes to add as neighbors. Therefore, total time to calculate surrounding of all nodes is also $O(|V|)$. In conclusion, preparations for FR-grid variant calculation in one iteration costs $O(|V|)$ but since the grid cells are refreshed in every ten iteration, the cost is reasonable on the average.

As a result, the complexity of repulsive force calculation is asymptotically decreased from $O(|V^2|)$ to $O(|V|)$.

3.3 Results

Mesh-like, tree-like and compound graphs are generated randomly for testing the execution time of CoSE after adapting the FR-grid variant. Generated meshes are neither dense nor sparse with edge-vertex ratio ($\frac{|E|}{|V|}$) approximately 1.80. For generated trees, probability of a node having children is 0.5, a node can have 1 to 4 children and edge-vertex ratio approximately equals to 1. For generated compound graphs, inter-graph edges are a quarter of all the edges. Number of siblings in a compound node is 4, and compound depth is 2 with a pruning probability of 0.4 for the inclusion tree. Edge-vertex ratio of compound graphs is approximately 0.5. Tests are run on an ordinary PC with Intel(R) Core(TM)2 Quad CPU Q8400 2.67 GHz processor, 3.25 GB usable RAM, and Windows 7 Professional 32-bit operating system. Comparisons of execution times before and after integrating the FR-grid variant to our compound spring embedder are shown in Figure 3.3, Figure 3.4 and Figure 3.5.

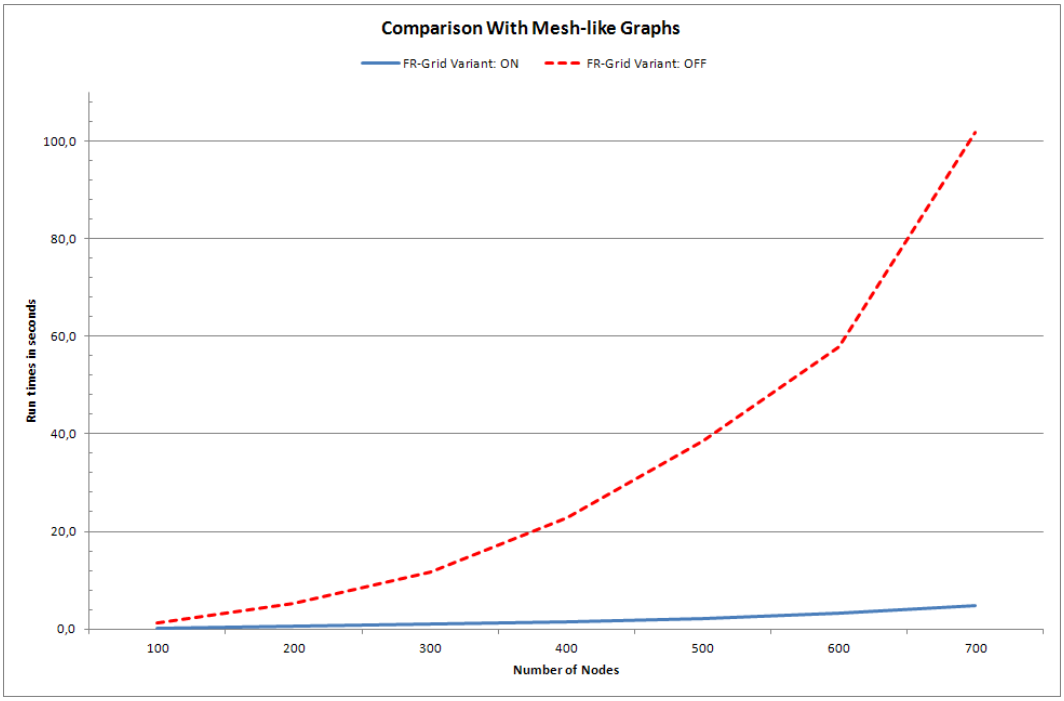

Figure 3.3: Execution time comparison with mesh-like graphs (Using FR-grid variant method).

Results are quite satisfactory. We obtained a dramatic decrease in running

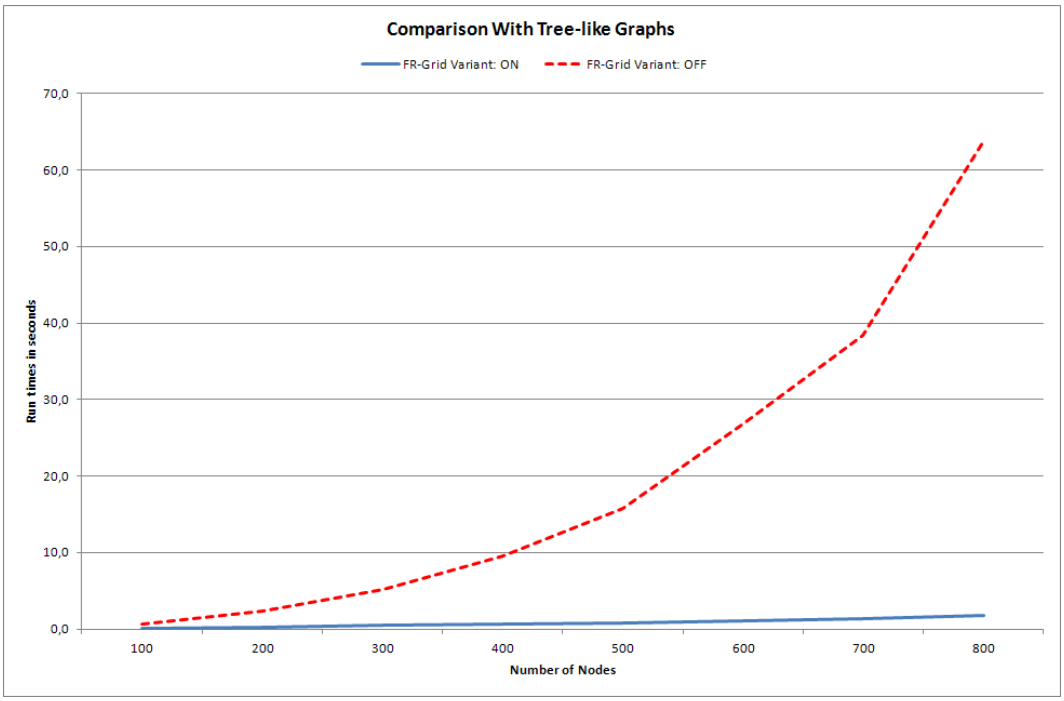

Figure 3.4: Execution time comparison with tree-like graphs (Using FR-grid variant method).

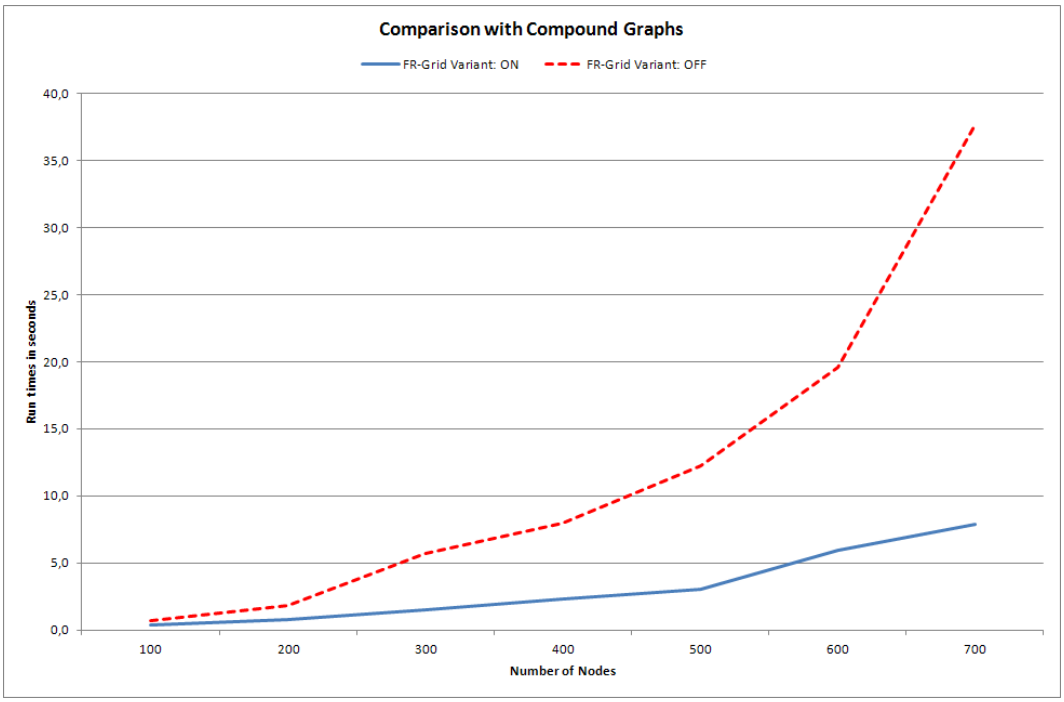

Figure 3.5: Execution time comparison with compound graphs (Using FR-grid variant method).

times after applying the FR-grid variant method. As the size of the input graph increases, difference between the running times increases greatly in line with our theoretical analysis.

3.3.1 Implementation Issues

Currently, CoSE updates the grid cells once in every ten iteration. In the early iterations, the spring embedder is not stable since temperature is high. Because of that, nodes are more likely to make dramatic movements and oscillations, so at the beginning of the spring embedder iterations, occupied grid cells by a node are more likely to change. Moreover, as the system is closer to convergence state, nodes move slightly and less likely to change occupied cells. Thus, in order to obtain more realistic simulations without a huge time penalty, updating grid cells frequently in early stages and increasing the re-calculation period proportional to the number of total iterations passed can be useful.

Chapter 4

Improving by Multi-level Scaling Strategy

Most of the spring embedder algorithms remain insufficient in laying out large graphs that are especially symmetric or contain a particular structure or a pattern. There are several multi-level methods aiming to resolve this problem by recursively coarsening the input graph. When the coarsest graph is constructed, layout process is started. Final positions of the nodes in each laid out coarser graph are interpolated to the finer graph.

We have surveyed through several methods and researched two methods in detail earlier in section 2.5. Because we need a fast, easy, and efficient method, we decided to implement the Walshaw's clustering method for adapting the multi-level scaling strategy to the CoSE algorithm.

4.1 Adaptation

The most challenging problem of adapting a multi-level scaling method to CoSE is handling compound nodes during the coarsening process. If compound nodes of graph G_i are treated as leaf nodes while G_i is being coarsened, it means all

child nodes of each compound node will be ignored, and only the first hierarchic level of the input graph will be considered during the coarsening. For instance, if the input graph G_0 has thousands nodes, but in the root graph, there are only a few compound nodes that own all other nodes, then G_0 would have a few coarsening levels. Thus, CoSE could not benefit from the advantages of the multi-level scaling method.

We investigated the approach of handling compound nodes during the coarsening process by coarsening each level of input graph separately. This would give fine layouts inside each compound node, but would create another problem. Separate hierarchical levels should be put together in order to obtain the final layout. In hierarchical level l, if a child graph exceeds the boundary of its parent node v or shrinks too much, then v should be resized. So the layout of level $l-1$ should be computed before level l. This results in a contradictory situation to the inter-graph edges supported by CoSE. The reason is simply that we want to put neighbors close to each other, and so CoSE needs to calculate the attractive forces between nodes connected with an edge but reside on different levels. Due to the reasons mentioned, we had to give up this idea and decided to consider only non-compound nodes during the coarsening process.

A special graph structure named *coarsening graph* is introduced in order to ease and modularize the coarsening process. In the *coarsening graph*, only leaf nodes and the *intra-graph* edges that are between these nodes are included.

4.1.1 Coarsening Algorithm

There is a one-to-one mapping between *coarsening graph*s and *compound graph*s. We use M_l notation for the *compound graph* in coarsening level l, and G_l for the *coarsening graph* which contains only the leaf nodes of M_l and the intra-graph edges between these nodes. Thus, a *compound graph* M_l indicates the original graph will be laid out in level l, whereas the *coarsening graph* G_l is used for only coarsening purpose. We defined *reference* pointers in order to maintain the one-to-one mapping from nodes in the *coarsening graph* to the nodes in the *compound*

graph.

For coarsening level zero ($l = 0$), G_0 is constructed with no nodes or edges initially. By a recursive traversal, leaf nodes of M_0 are gathered, and for each leaf node, a corresponding *coarsening node* is created, and added to G_0. In order to obtain a one-to-one mapping between M_0 and G_0, leaf nodes of M_0 are mapped with all nodes in G_0. After creating and mapping all the nodes in G_0, edge list of M_0 is traversed. Only the edges that connects the leaf nodes (excluding the inter-graph edges) are added to G_0. *Weight*s of all nodes are set to one. By this way, construction of G_0 is completed.

G_0 is coarsened recursively, until the size of G_l becomes 1, or size of last two coarsening graphs G_l and G_{l-1} are equal to each other. If $|V^{G_l}| = |V^{G_{l-1}}|$, it means all nodes in both G_l and G_{l-1} are *isolated*, and no coarser graphs can be generated any more.

For a coarsening level l, all nodes in G_l are flagged as *unmatched*. Nodes of a coarsening graph are stored in a list instead of a set, in order to assure a deterministic search for unmatched nodes. If there is a node in G_l, then the first node v in the node list is checked, whether it is matched or not. If v is unmatched, then v and if exists, its matching node u are contracted and merged into node t. Unmatched neighbors of v in the same graph are traversed, and the node with the minimum weight is selected for matching. Weight of t is set to the sum of weights of v and u, and t is flagged as *matched*. Neighbors of v and u are connected to t. Finally, t is added to the end of the node list of G_l, while v and u are removed (See the Figure 4.1). Coarsening steps of a compound graph are shown in the Figure 4.2 and the Figure 4.3.

Two pointers; *previous* and *next*, are used for tracing the coarsening processes and interpolating the final positions of finer graphs to the coarser ones. Each node has two *previous* pointers, and one *next* pointer. For instance, let us assume that $t \in M_l$. Nodes $v, u \in M_{l-1}$, which are merged and contracted to the node t are pointed by *previous* pointers of t (see the 4^{th} and 6^{th} screenshots in Figure 4.1). For this instance, *next* pointers of v and u point to the t.

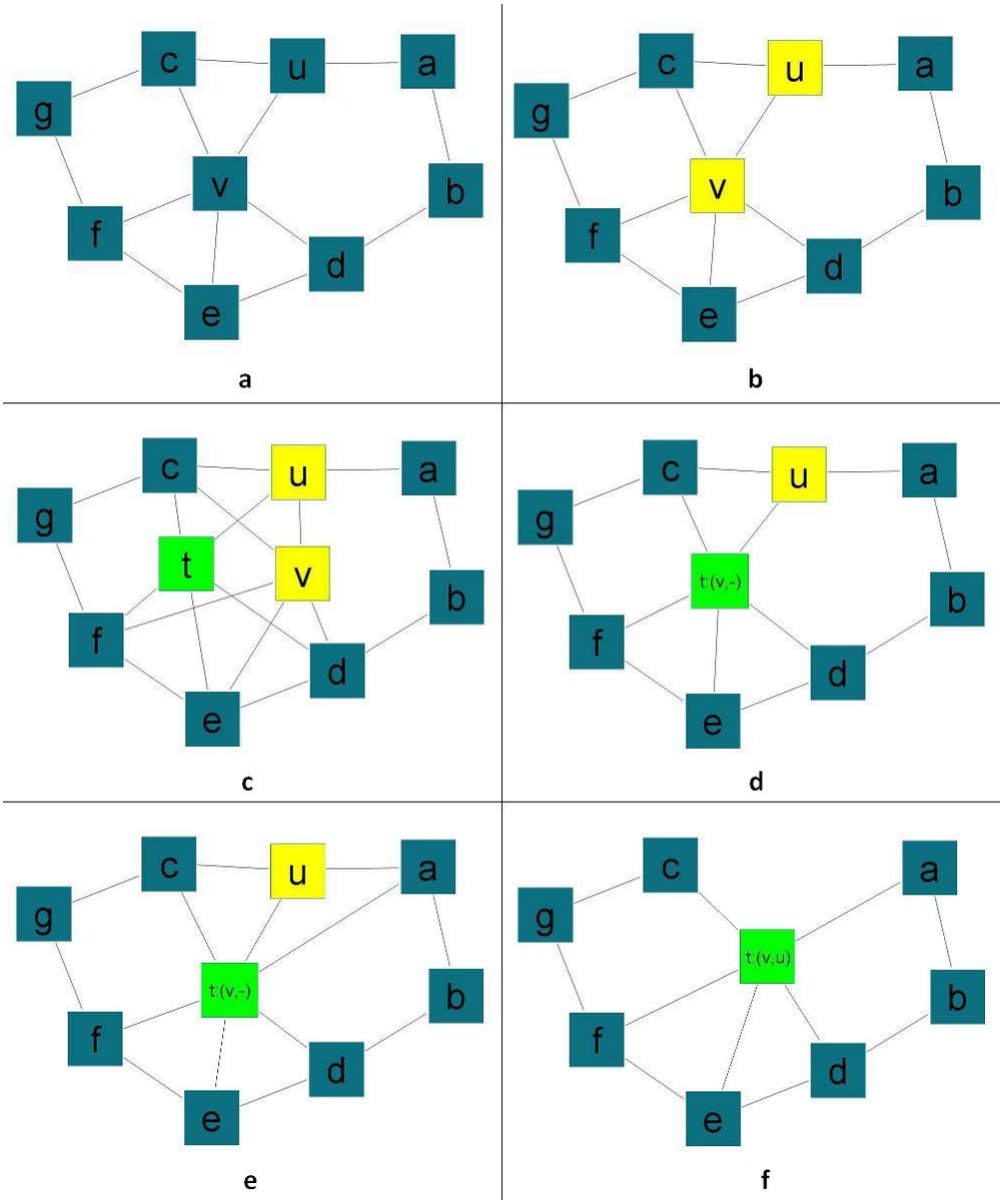

Figure 4.1: A sample contraction process. a) The initial phase of the coarsening graph, b) v and u are chosen to be matched, c) Node t is created and neighbors of v is connected to the t, d) v is removed from the coarsening graph e) Neighbors of u is connected to the t, f) u is removed from the coarsening graph.

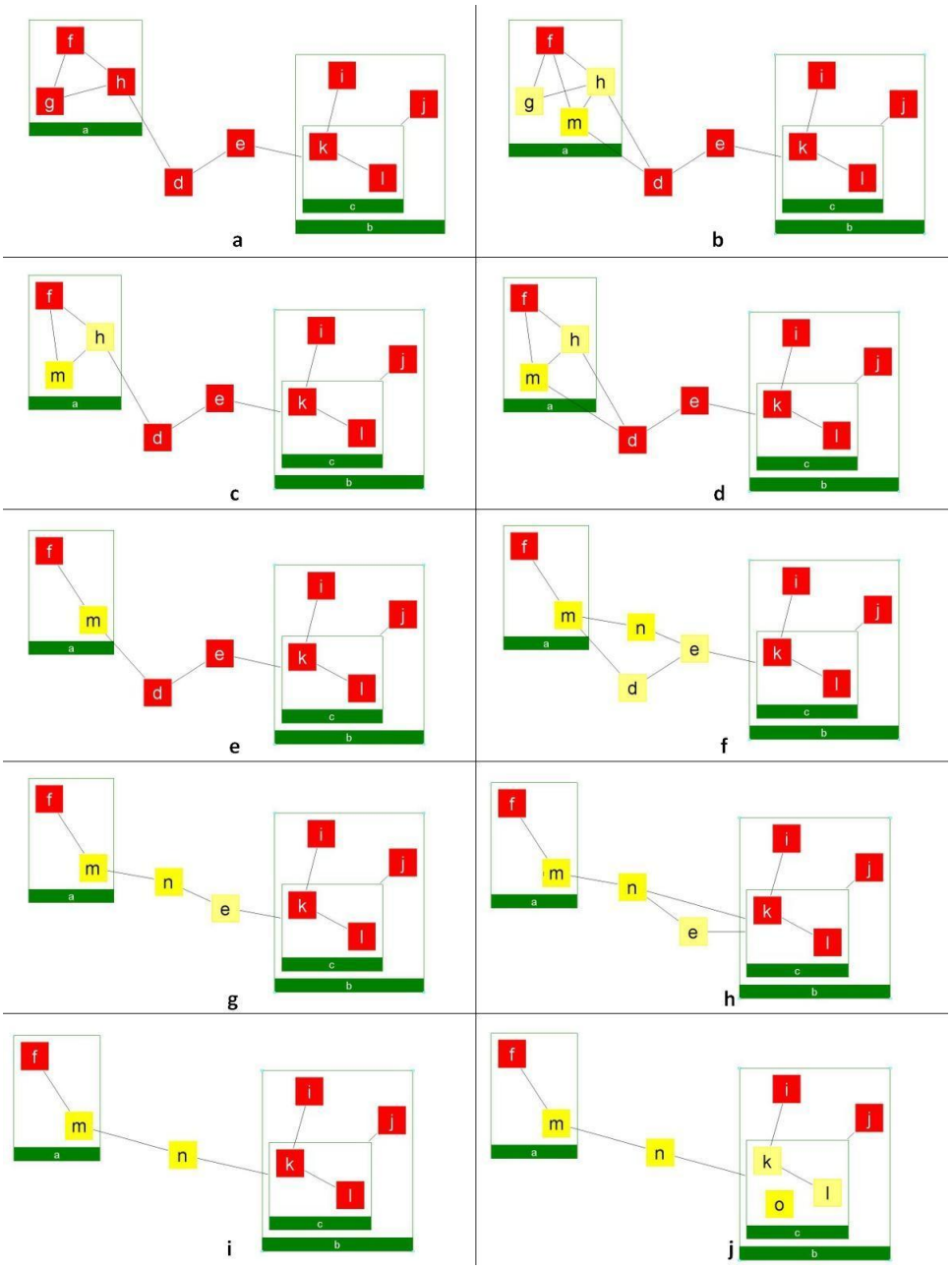

Figure 4.2: Coarsening steps of a compound node. a) The input graph M_0, b) g and h are matched, a new node $m \in M_1$ is created, neighbors of g are connected to m, c) g is removed, d) Neighbors of h are connected to m, e) h is removed, f) d and e are matched, a new node $n \in M_1$ is created, neighbors of d are connected to n, g) d is removed, h) Neighbors of e are connected to n, i) e is removed, j) k and l are matched, a new node $o \in M_1$ is created.

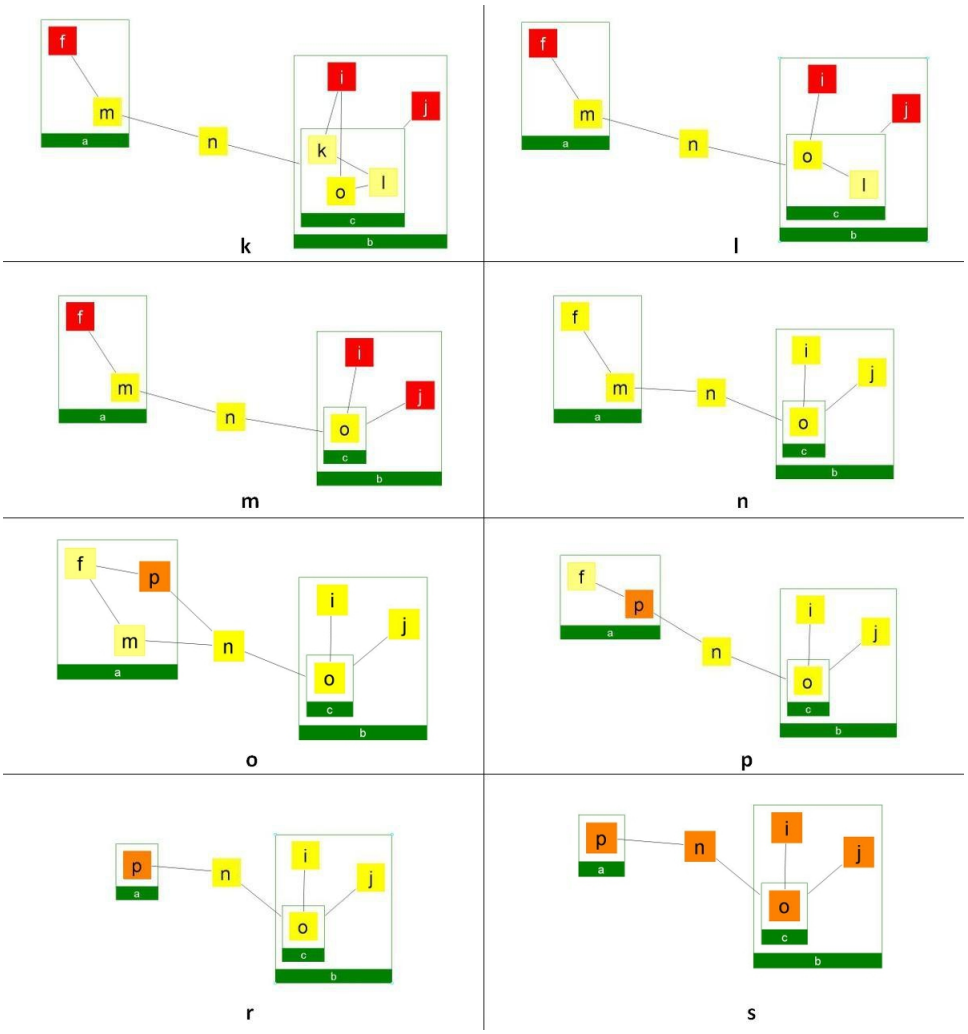

Figure 4.3: Coarsening steps of a compound node (Continues from the Figure 4.2). k) Neighbors of k are connected to o, l) k is removed, m) l is removed, n) M_1, o) f and m are matched, a new node $p \in M_2$ is created, neighbors of m are connected to p, p) m is removed r) f is removed s) M_2.

```
method COARSEN(CoarseningGraph G=(V,E))
1) for i ≤ V.size do
2)   V[i].matched:= false
3) while V[0] is not matched do
4)   CoarseningNode v := V[0]
5)   CoarseningNode u := lowest weighted neighbor of v
6)   CONTRACT(G, v, u)
7) for i ≤ V.size do
8)   CoSENode newNode:= create a new node
9)   newNode.previous1:= V[i].node1.reference
10)  V[i].node1.reference.next:=newNode
11)  if V[i].node2 is not null then
12)    newNode.previous2:= V[i].node2.reference
13)    V[i].node2.reference.next:=newNode
```

Figure 4.4: Coarsening method.

```
method CONTRACT(CoarseningGraph G=(V,E), CoarseningNode v,
                CoarseningNode u)
1) CoarseningNode t := create a new node
2) add t to the end of V
3) t.node1:= v
4) for i ≤ v.neighbors.size do
5)   if v.neighbors[i] is not equal to t then
6)     CoarseningEdge e := (t, v.neighbors[i])
7)     add e to E
8) t.weight:= v.weight
9) remove v from V
10)if u is not null then
11)  t.node2:= u
12)  for i ≤ u.neighbors.size do
13)    if u.neighbors[i] is not equal to t then
14)      CoarseningEdge f := (t, u.neighbors[i])
15)      add f to E
16)  t.weight:= t.weight+u.weight
17)  remove u from V
18)t.matched:= true
```

Figure 4.5: Contraction method.

After constructing G_{l+1} from G_l, coarser *compound graph* M_{l+1} will be generated from G_{l+1}. A new root for M_{l+1} is created. Then, nodes of M_l are traversed recursively (where the initial call is made with the root graphs of M_l and M_{l+1}) as follows: If current node v is a compound one, then a new compound node y is created with an empty child graph and added to the graph which is passed as parameter. *Next* pointer of v is set to y and one of the *previous* pointers of y is set to v. Then, a recursive call is invoked with the child graphs of v and y. If v is not a compound node, then *next* pointer of v is added to the graph which is passed as parameter. After generating nodes of coarser graph M_{l+1}, edge list of M_l is traversed, and edges of M_{l+1} are generated.

method GENERATECOARSERCOSEGRAPH*(GraphManager M_l)*
1) Create M_{l+1}
2) Create a root for M_{l+1}
3) GENERATENODES*(M_l.root, M_{l+1}.root)*
4) GENERATEEDGES*(M_l.root, M_{l+1}.root)*

Figure 4.6: Generation of coarser *CoSE* graph from *CoarseningGraph*.

method GENERATENODES*(CoSEGraph $G_l = (V_l, E_l)$,*
CoSEGraph $G_{l+1} = (V_{l+1}, E_{l+1})$)
1) **for** all nodes of G_l **do**
2) **if** a node (v_l) is *compound* **then**
3) create v_{l+1} with an empty *child* graph
4) v_l.*next*:= v_{l+1}
5) v_{l+1}.*previous1*:= v_l
6) GENERATENODES*(v_l.child v_{l+1}.child)*
7) **otherwise**
8) add v_l.*next* to G_{l+1}
9) copy geometry of v_l to v_{l+1}

Figure 4.7: Generation of nodes of coarser graph.

method GENERATEEDGES*(CoSEGraph $G_l = (V_l, E_l)$,*
CoSEGraph $G_{l+1} = (V_{l+1}, E_{l+1})$)
1) **for** all edges of G_l **do**
2) create e_l with no *source* or *target*
3) e_l.*source*:= e_l.*source.next*
4) e_l.*target*:= e_l.*target.next*

Figure 4.8: Generation of edges of coarser graph.

All graphs from the finest to the coarsest are held in a list. Layout phase begins with the coarsest graph M_{k-1}, where k is the number of levels. M_{k-1} is laid out with our compound spring embedder. When layout calculations are finished, final positions of nodes in M_{k-1} are used for initial positioning of M_{k-2}. As aforementioned, nodes, that are contracted in order to generate a node in a coarser graph, can be accessed via *previous* pointers. Assume that $v \in M_{k-1}$ and previous pointers of v points to the node u and w, where u, $w \in M_{k-2}$. When interpolating the positions, node u, which is pointed by the first *previous* pointer is placed to the exactly same place with v. Node w, the second *previous* pointer, is placed to the lower right of the first node, to a distance of *ideal edge length* from both x and y axes. Refinement continues until the input graph M_0 is reached. Whole layout calculation is finished when M_0 is laid out.

In the Figure 4.9 and Figure 4.10, laid out levels of a randomly generated mesh-like graph and a compound graph with a particular structure are shown. It is observable that, each coarser graph is an abstraction of the finer one.

4.2 Complexity Analysis

Making the complexity analysis of a multi-level scaling method is not easy, since the number of abstraction levels heavily depends on the structure of the input graph. For dense graphs like complete meshes, it is more probable to find a matching for a node. For the best case, each node is matched with another one which makes number of abstraction level $k = \log_2 |V|$. On the other hand, for sparse graphs like trees, more than half of the clusters may contain only one node. For the worst case, only one node is matched with another one, but other nodes match with themselves so that number of abstraction level $k = |V| - 1$.

Let $T_{single}(|V|, |E|)$ be the running time of a single level force-directed placement algorithm and $T_{multi}(|V|, |E|)$ be the running time of the same algorithm after applying Walshaw's clustering method [11], [9]. $T_{multi}(|V|, |E|) = \sum_{i=0}^{k-1}(T_{single}(|V_i|, |E_i|) + T_{refine}(|V_i|, |E_i|) + T_{init}(|V_i|, |E_i|) + T_{coarsen}(|V_i|, |E_i|))$.

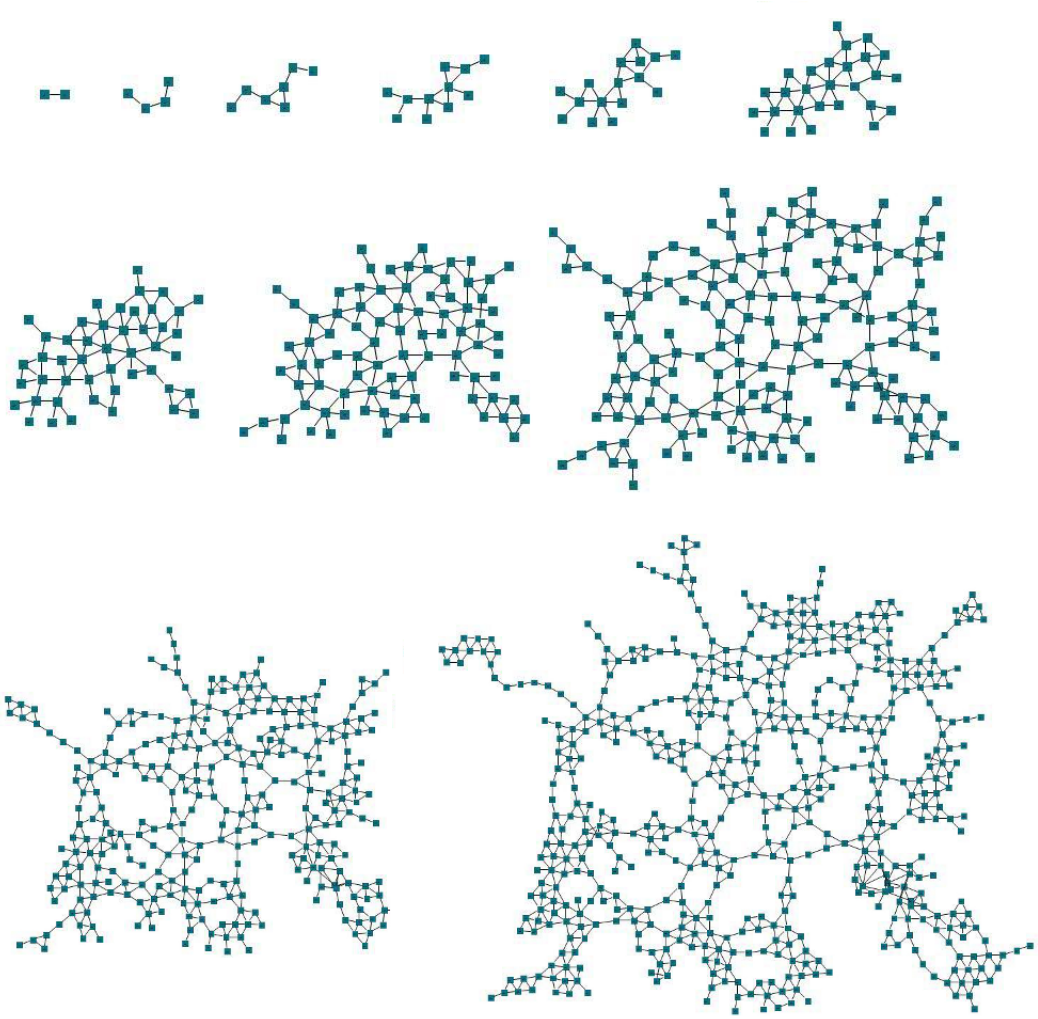

Figure 4.9: A random mesh-like graph M, coarsened in 11 steps. Levels are laid out via CoSE [8] after adapting the Walshaw's clustering method [11]. On the left-top, there is M_{10}, the coarsest graph, with 2 nodes. On the right-bottom, there is the final layout of $M = M_0$.

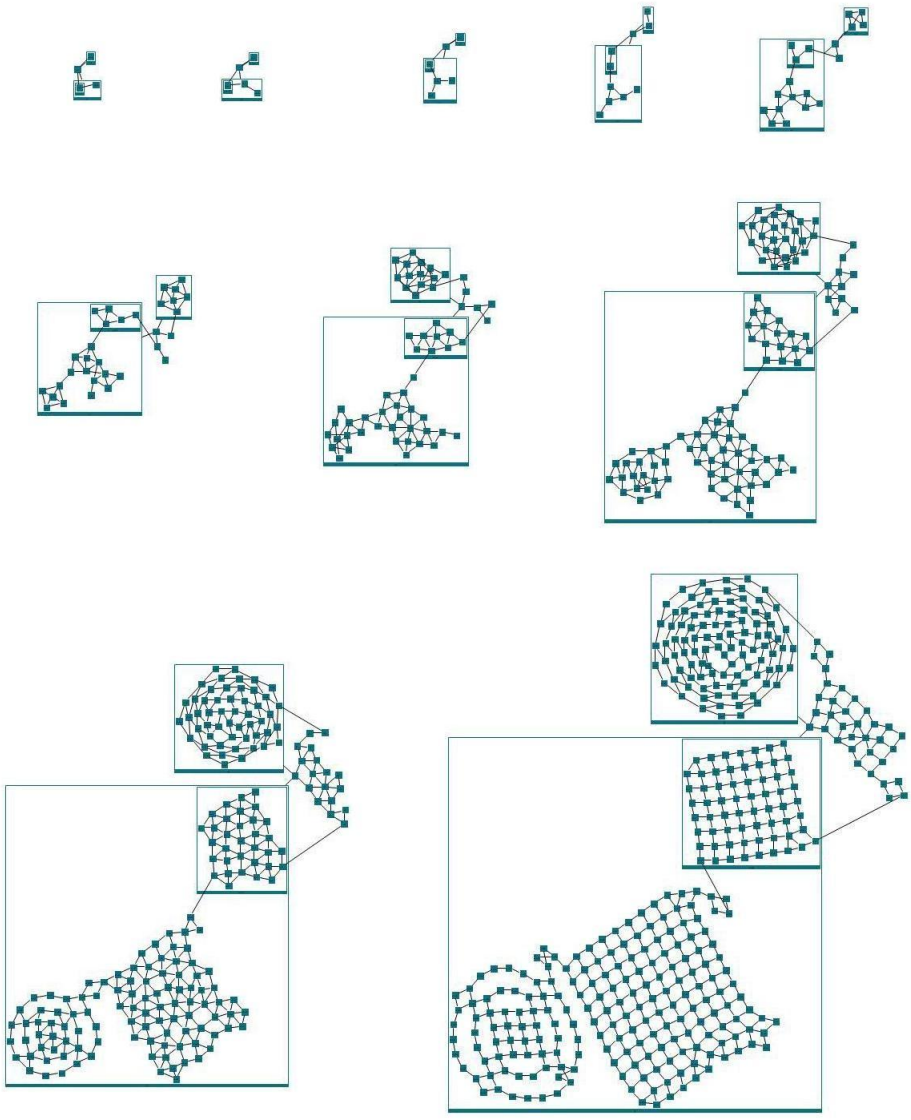

Figure 4.10: A compound graph N, with 10 levels. Levels are laid out via CoSE [8] after adapting the Walshaw's clustering method [11]. On the left-top, there is N_9, the coarsest graph, with 3 nodes in the root graph, and 7 nodes in total. On the right-bottom, there is the final layout of $N = N_0$.

$T_{refine}(|V_i|, |E_i|)$ is the time to refine graph G_i from G_{i-1}, $T_{init}(|V_i|, |E_i|)$ is the time to make initial placement of G_i and $T_{coarsen}(|V_i|, |E_i|)$ is the time to coarsen graph G_i to G_{i+1}.

For the best case, $T_{refine}(|V_i|, |E_i|)$ and $T_{coarsen}(|V_i|, |E_i|)$ are $(|V_i| + |E_i|)/2$, because the traversal of half of the node list is enough to refine or coarsen the graph G_i. For the worst case, $T_{refine}(|V_i|, |E_i|)$ and $T_{coarsen}(|V_i|, |E_i|)$ are $|V_i| - 1$, because whole list has to be traversed to refine or coarsen the graph G_i. $T_{init}(|V_i|, |E_i|)$ is $|V_i|$ for any case. In conclusion $\sum_{i=0}^{k-1}(T_{refine}(|V_i|, |E_i|) + T_{init}(|V_i|, |E_i|) + T_{coarsen}(|V_i|, |E_i|))$ is asymptotically linear in $|V|$ and $|E|$.

For the best case, each cluster contains two nodes, so $|V_{i+1}| = |V_i|/2$. Also for graphs where edge node ratio $|E|/|V|$ is greater than 1, $|E_{i+1}| \leq |E_i|/2$ since number of nodes in coarser graph is decreased by the factor of 1/2. On the other hand, for the worst case, all clusters but one contain one node, so $|V_{i+1}| = |V_i| - 1$ and $|E_{i+1}| \leq |E_i| - 1$. Therefore, $\sum_{i=0}^{k-1} T_{single}(\frac{|V|}{2^i}, \frac{|E|}{2^i}) \leq \sum_{i=0}^{k-1} T_{multi}(|V_i|, |E_i|) \leq \sum_{i=0}^{k-1} T_{single}(|V| - i, |E| - i)$. Lower bound of the inequality is less than $2 \times T_{single}(|V|, |E|)$ and upper bound is less than $k \times T_{single}(|V|, |E|)$. In conclusion, the inequality above becomes $2 \times T_{single}(|V|, |E|) \leq \sum_{i=0}^{k-1} T_{multi}(|V_i|, |E_i|) \leq |V| \times T_{single}(|V|, |E|)$.

As a result, as the density increases or the structure of the graph gets closer to the best case, complexity of the multi-level scaling methods is closer to the lower bound, which is asymptotically the same as the single-level methods. However, as the density decreases or the structure of the graph becomes closer to the worst case, complexity converges to the upper bound, which multiplies the complexity of the single-level methods by $|V|$ in theory. But in practice, layout in each level except the first one is incremental, so the system should converge a lot faster than in theory.

4.3 Results

In order to test the execution time and resulting layouts produced by CoSE with Walshaw's multi-level strategy, mesh-like, tree-like, and compound graphs are generated randomly. For performance comparisons, tests are run with the same graphs and the same system/machine configuration that are used for testing the FR-grid variant. In addition, graphs with a specific structure are manually created for testing the visual quality. Comparisons of execution times before and after adapting the multi-level scaling strategy to our compound spring embedder are shown in Figure 4.11, Figure 4.12 and Figure 4.13.

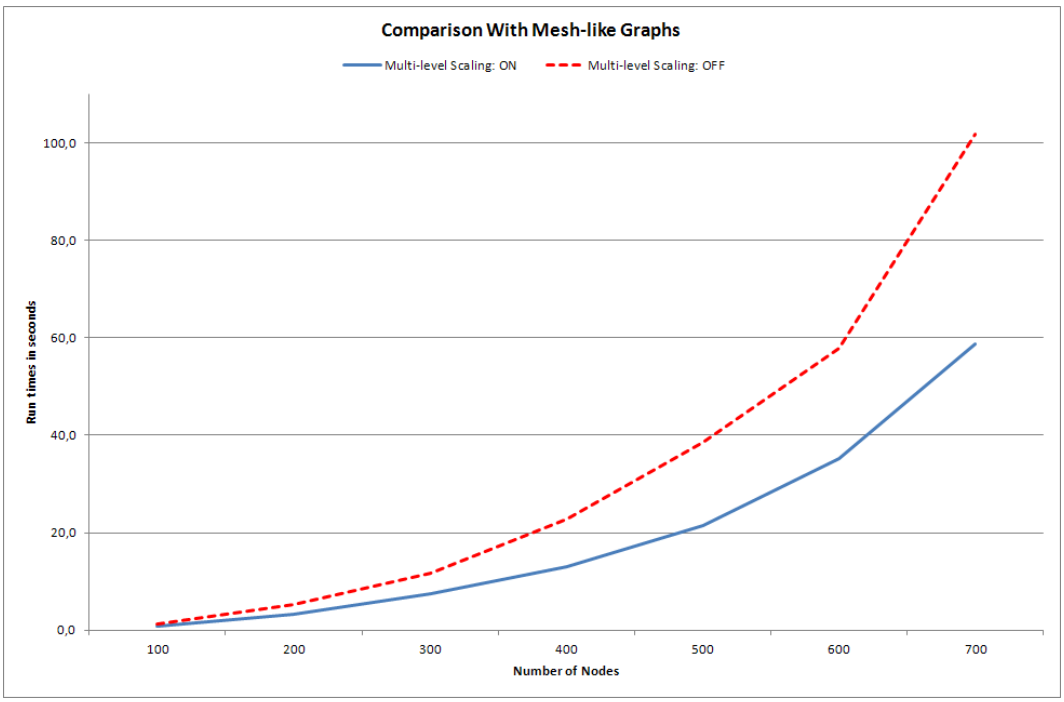

Figure 4.11: Execution time comparison with mesh-like graphs (Using multi-level scaling strategy).

Walshaw's clustering method gives better results for mesh-like graphs. On the other hand, it is out-performed by our original spring embedder for tree-like and especially compound graphs. Since the edge-vertex ratio of the tested compound nodes is less than 1, and for trees it is highly probable that more than half of the clusters have only one node, we obtained a worse performance for trees and compound graphs. However, meshes are denser and structured more similar to

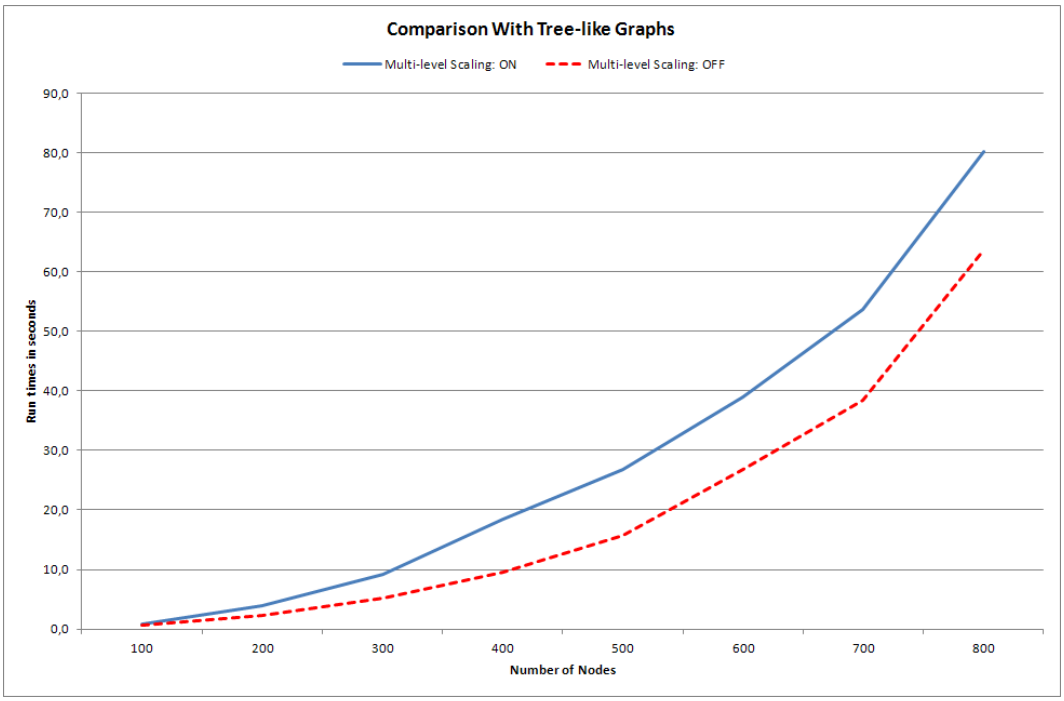

Figure 4.12: Execution time comparison with tree-like graphs (Using multi-level scaling strategy).

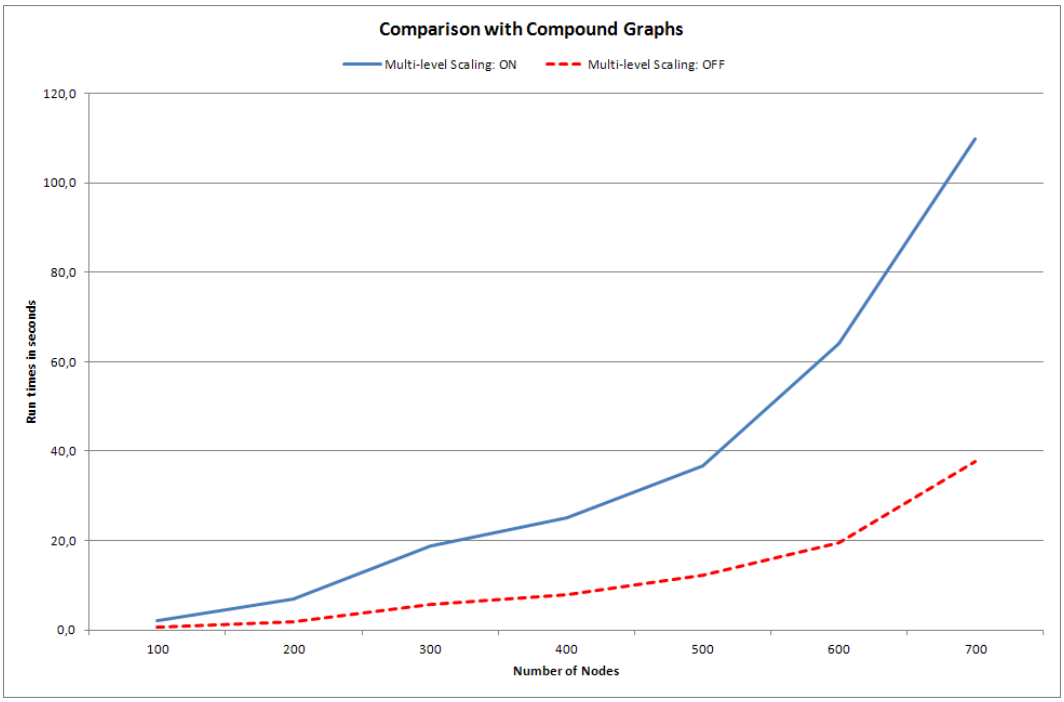

Figure 4.13: Execution time comparison with compound graphs (Using multi-level scaling strategy).

the complete graphs. So that, execution time of layout calculation is decreased for meshes after adapting the multi-level strategy.

Main goal of adapting a multi-level scaling method is to obtain better looking layouts for especially large graphs. Without this adaptation, many times CoSE fails to solve the structure of large graphs. After implementing Walshaw's multi-level scaling method, we observed that number of edge crossings decreased dramatically, and we started to obtain much more eye pleasing layouts. There are screen shots taken via *Chied* [21] with and without the multi-level strategy in Figure 4.14 and Figure 4.15. In conclusion, after the adaptation of multi-level scaling strategy, CoSE is slower only by a small factor, but produces higher quality layouts.

4.4 Future Work

Because of the difficult nature of compound structures, we simply ignored the compound nodes during the coarsening calculations. Eventually, this decision effects the visual quality of compound graph's layouts. Especially, compound graphs that are nested deeply, and contain more compound nodes than leaf ones cannot take advantage of the multi-level scaling method. On the other hand, after implementing Walshaw's clustering method, CoSE produces better looking layouts for graphs that have a few levels and contain compound nodes mostly own leaf nodes. For simple graphs that contain no compound nodes, results are really satisfactory.

Because we have been seeking for an easy and efficient method, we have chosen the Walshaw's clustering algorithm for implementing the multi-level scaling. In the future, Sonar system method that mentioned in Section 2.5.1, or another multi-level scaling method can be considered for adapting to CoSE.

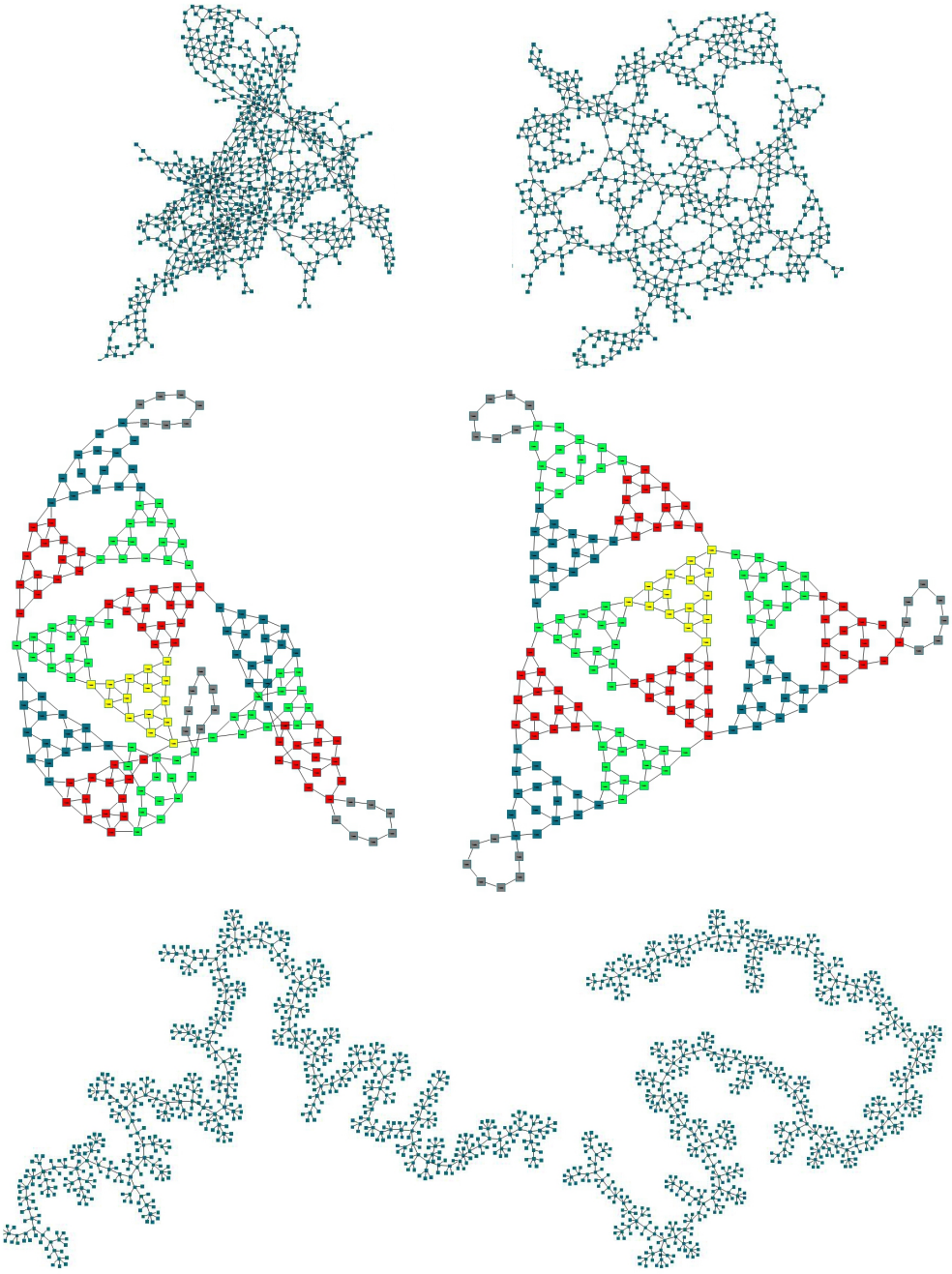

Figure 4.14: Graphs which are laid out via CoSE, before adapting the multi-level scaling strategy (left); same graphs after adapting the multi-level scaling strategy (right).

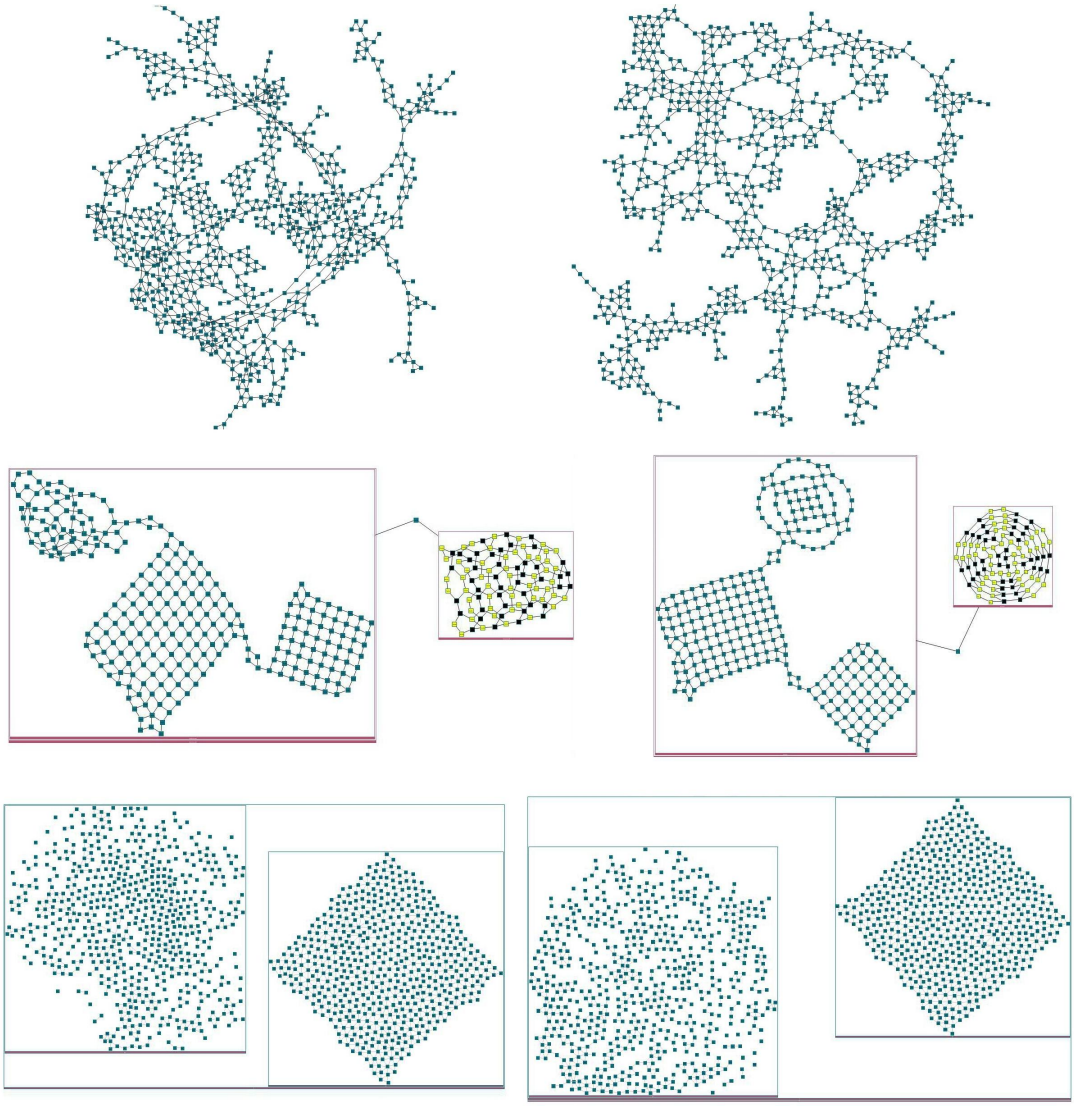

Figure 4.15: Graphs which are laid out via CoSE, before adapting the multi-level scaling strategy (left); same graphs after adapting the multi-level scaling strategy (right).

Chapter 5

Improving by GPU Parallelism

For huge graphs, where $|V| >> 1000$, execution times of spring embedders increase dramatically. But hopefully, it is possible to parallelize the force calculations, since force calculations of a node have no dependency to or interaction with force calculations of other nodes. In [14], the force-directed placement algorithm, which uses GPU support during the calculation of forces, outperforms the same algorithm which uses only CPU. Depending on graph structure, GPU supported algorithm runs 20 to 60 times faster than the CPU implementation of the same algorithm. Therefore, we were inspired by the work in [14] and decided to adapt their parallel programming strategy to CoSE.

5.1 Device Memory

GPUs provide a powerful environment for implementing parallel algorithms, because they are an implementation of the SIMD architecture. Moreover, they give developers the chance to achieve real parallelism via virtual threads. With GPUs that contain thousands of processors, it is possible to run hundreds of thousands of threads in parallel.

There are specific memory types for specific purposes in a GPU. For an

integrated system, where CPU and GPU memories are physically same, using *mapped-pinned memory* is very advantageous. However, most of the new systems are not integrated, and for very large graphs (with high memory requirements), using mapped-pinned memory reduces the performance of other programs currently running and thus the system in general.

Shared memory is faster than the *global memory* and available for both reading and writing. Each block has its own shared memory, and a shared memory is only available for the owner *block*. Optimum size of a block (maximum number of threads in the block) depends on the size of its shared memory. Generally, maximum number of threads that can run in parallel in a block is 512 or 1024. Therefore, for a graph that contains more than 512 or 1024 nodes, we have to split the graph. However even this will not work, since it may be desired to access all nodes within a kernel launch, which, in this case, is not possible since a block cannot access the shared memory of other blocks.

Texture memory is a cache memory type (which is also called *texture cache*), where the global memory can be accessed through. It is read-only, and optimized for 2-dimensional spatial locality. Texture cache is accessed via texture fetching, which costs one read from memory only on a cache miss. Otherwise, it costs just one read from the texture cache.

In order to maximize the memory throughput, it is recommended to copy the host data to the device in one piece. Copying data in many small pieces performs worse because of the overhead associated with each data transfer [10].

CUDA provides two types of memory allocations: *Linear* and *CUDA array*. Linear memory type is used to allocate a linear block of the GPU memory. Linear memory blocks are useful for allocating *primitive type*s, *struct*s and *array*s. Linearly allocating 2D or 3D arrays is bad for performance, since accessing such arrays costs more than one read from the memory. *CUDA array* type is specialized for memory handling of 2D and 3D arrays to avoid additional costs to access such arrays on the device side.

5.2 Adaptation

Calculating the forces to be applied to a node has no dependency or interaction with force calculation of other nodes. So, one can use one thread per node for calculating repulsion, attraction and gravitational forces in parallel. Thus, each thread t_i will be responsible for calculation of the total force applied to a node v_i. There is also alternative approach to parallel computation: Each thread can be assigned for calculating a repulsion or attraction force between only a node pair (v, u). However, such an approach has disadvantages. More than one thread might access the same field for writing, which needs synchronization on device side resulting in loss of parallelism. Let us say we have n nodes, and $n \times n$ threads for calculating repulsion force between each node pair. When we synchronize, the device will wait n times in total to access to a field in order to write the resulting repulsion force. This solution is worse in comparison with the previous one (thread per node). In addition, thread per force calculation method is opposed to the principle that "All processors should be busy at all time" [10].

5.2.1 Device Side

Data Structures

Each node occupies a 2-dimensional rectangular area. For storing the geometry, the top-left point and width and height of the rectangular area are used. Additionally, since repulsion force is only calculated for nodes in the same compound node (including the root), index of the owner node for each node should be stored. For repulsion force calculation, *top*, *left*, *width*, *height*, and *owner index* attributes will be sufficient.

Geometry and hierarchy of a graph can be stored in a 2D integer array. Each row stores information about only one node, and each column stores an attribute of node. For the sample graph in Figure 5.1, graph geometry matrix is given in Table 5.1. This 2D array will be allocated as CUDA array memory because of

	top	left	width	height
a	10	173	252	177
b	350	111	165	182
c	441	213	40	40
d	202	183	40	40
e	455	121	40	40
f	109	226	40	40
g	350	10	40	40
h	20	270	40	40
i	360	138	40	40
j	267	69	40	40
root	0	0	525	360

Table 5.1: Graph geometry matrix for the graph in Figure 5.1.

the performance issues mentioned above. After the allocation, graph data are needed to be copied from the host (CPU) to the device (GPU). And finally, a texture memory space will be bound for the graph data.

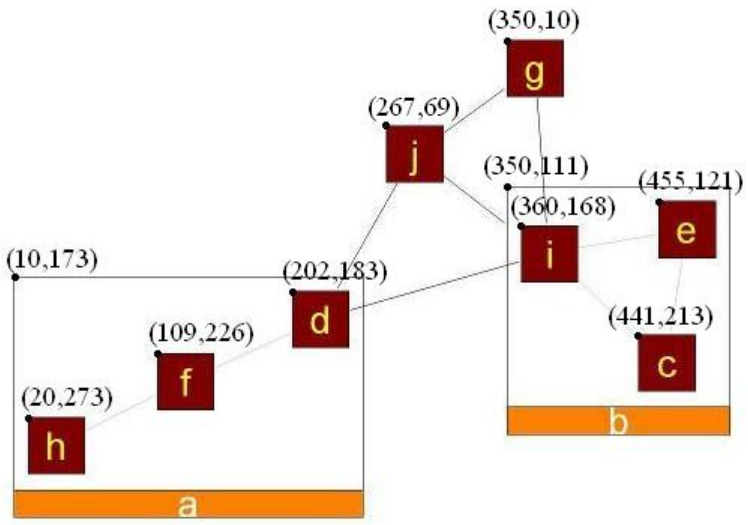

Figure 5.1: A sample graph. (Left-top coordinates of nodes are indicated)

Resulting forces exerted to each node in x and y directions will be written to the global memory. For each axis and force type, there will be an array for storing the resulting forces. Thus, there will be six arrays stored in global device memory. Length of each array is $|V|$. Since each thread is assigned to one node, it is guaranteed that, each thread will only access one field of the resulting force

arrays exclusively for writing. Thus, the device is not needed to be synchronized explicitly. Explicit synchronization in GPU should be avoided because it has a negative effect on parallelism.

On the GPU side, geometry and hierarchy of the input graph and its edges with their ideal lengths are stored in *texture memory* in order to maintain the layout process on the device side. Some of these attributes will not be changed during the layout calculations. Thus, storing these data in the fastest read-only memory, the *texture cache* or simply the *texture* memory in the GPU, will be the best solution.

For storing the edges, the data structures defined in [14] are used. Two arrays are stored; one array named *edge-value* is used to store the adjacency of the nodes, and the other one named *edge-index* is used to keep the start and end indices of the adjacency lists in *edge-value* array (Figure 5.2). Because of the performance issues, *edge-value* array is stored in the texture cache, while *edge-index* array is stored in the global memory. Reason of storing these two arrays in different memory spaces is the difference in the access frequencies. In a thread i, *edge-index* array is accessed two times, whereas *edge-value* array is accessed as many as the number of neighbors of node v_i times. Size of the *edge-value* array is $2 \times |E|$, while the size of the *edge-index* array is $|E| + 1$.

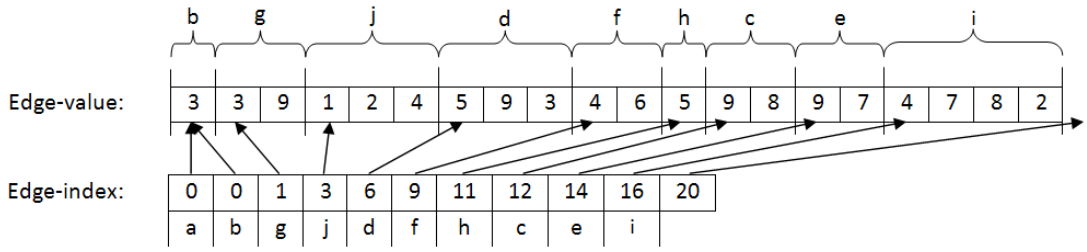

Figure 5.2: Edge-value and edge-index arrays for the graph in Figure 5.1. For instance, node b has one neighbor which is the 3^{rd} node j, node g has 2 neighbors which are the 3^{rd} and 9^{th} nodes (j and i), and so on.

Edges in a compound graph can have various ideal lengths depending on their source and target node's placement in the graph hierarchy. Ideal edge lengths between nodes in the same owner graph and nodes in different levels or different compound nodes differ greatly. Aim of applying the spring force is to reach a

stable state, where the lengths of the edges are very close to the ideal lengths that are calculated beforehand. Thus, storing the ideal edge lengths is essential. Although ideal lengths can be computed on the device side, we decided to store the *ideal-edge-lengths* in an array on texture memory, since these calculations require recursive calls, which should be avoided on the device side [10]. Size of the *ideal-edge-lengths* array is $|E|$.

As aforementioned, a 2D CUDA array is fetched to the texture cache in order to store the geometry of the graph. In CoSE, nodes lay over a rectangular area in 2D Euclidean space. Y coordinate of the top line and x coordinate of the left line are needed to locate a rectangular area. Additionally, width and height of the rectangular area should be stored in order to obtain the full geometric information about a node. Each row in *graph-geometry* matrix is reserved for a node. It has four columns corresponding to x, y, width, and height attributes. Finally, geometry of the *root graph* is also required for calculations. Therefore the size of the *graph-geometry* matrix is $4 \times (|V| + 1)$ (see Table 5.1).

In order to calculate the gravitational force that will be applied on a node, firstly, we should know whether or not the node in question is isolated. In other words, do we need to calculate the gravitational force for a node, or not? Secondly, gravitational forces in compound nodes depend on the current sizes of the compound nodes. So, the current sizes of the compound nodes should also be stored. Finally, owner node of each node should be known to be able to know the owner node's current size. Owner node of a node can be stored via an integer that holds the row index of the node in the *graph-geometry* matrix. For the graphs which are owned directly by the root graph, -1 can be used. Owner node index is also necessary for calculating the surrounding nodes in the device side, because the repulsion force is calculated between only the nodes with the same owner. In conclusion, we use a 2D CUDA array named *owner-data* to store the data mentioned above. Each row in the *owner-data* matrix is reserved for a node. First column stores the index of the owner node, second one stores the estimated size of the child graph of each compound node (for leaf nodes, value of the second column will be -1), third column is used for checking whether gravitational force will be calculated or not (0 or 1). Size of the *owner-graph* matrix is $3 \times |V|$.

A *surrounding-matrix* is stored in the global memory for handling FR-grid variant method in the device efficiently. At the i^{th} row of the matrix, indices of geometric neighbors of i^{th} node (v_i) are listed. These neighbors are the nodes that apply a repulsion force on v_i. For the graph in Figure 3.1, in the second row of the *surrounding-matrix*, which is allocated for node b, only the index of node a is stored. Since all nodes in the graph (except v_i) can be neighbors of v_i, there should be at least $|V|-1$ columns in the *surrounding-matrix*. Moreover, we need a delimiter index, like -1, so the size of the matrix is $|V| \times |V|$.

Resulting forces in x and y directions are stored separately. Since there are three kinds of forces (attraction, repulsion and gravitational), there will be six arrays named *spring-x*, *spring-y*, *rep-x*, *rep-y*, *grav-x* and *grav-y*. These arrays are located in the global memory, because they will be accessed for both reading and writing. Size of each array is $|V|$.

Beside the data structures mentioned above, there are also constants and layout dependent parameters. Constant parameters like *spring constant*, *repulsion constant*, *fr-grid calculation check period* are hard-coded in the header file of the *kernel* CUDA C source. Layout dependent variables like *repulsion range*, *number of nodes*, and *number of current iterations* are stored in global memory, and are accessible from all threads. In addition, pointers of the data structures that are located on the global memory and layout dependent variables are passed as parameters from CPU (via JCuda framework [22]) to the *global* function in the kernel. A global function is like a *main* function for a CPU executable.

Algorithms

Total number of blocks (*blocks per grid* $\times$ *number of grids*) equals to and cannot exceed the total number of processors in the GPU. Moreover, number of threads in a CUDA block is limited due to the local memory of a processor. Our experiments show that, using the full capacity of blocks gives better performance results in comparison to using all the processors first. So, we decided to use *maximum number of threads per block* ($thread_{MAX}$). Hence, there will be $\frac{|V|}{thread_{MAX}}+1$ blocks.

In each and every iteration of the layout calculation, each thread is assigned to calculate the attractive, repulsive and gravitational forces to be applied on each node. However, *number-of-blocks* $\times$ $thread_{MAX} \geq |V|$, and one block will have idle threads if $|V|$ is not a multiple of $thread_{MAX}$. Threads from 0^{th} to $|V| - 1^{st}$ indices are dedicated to nodes from 0^{th} to $|V| - 1^{st}$ indices, respectively. In other words, if index of a thread is less than the number of nodes, then force calculations are handled in this thread.

Firstly, all forces in both x and y axis are set to zero and owner node id is obtained from *owner-data* matrix. Starting from the first iteration, *surrounding-matrix* is re-calculated periodically in every ten iteration.

During the calculation of surrounding nodes, grid cells occupied by v_i are calculated at the beginning. Then all nodes in the neighbor cells are traversed. If a node v_j is in a neighbor cell of v_i, and distance between v_i and v_j is less than or equal to the length of the grid unit, and owner nodes of v_i and v_j are the same, then index of v_j is added to i^{th} row of the *surrounding-matrix*.

In order to calculate attractive forces applied on v_i, neighbors of v_i in graph topology need to be known. Indices of the neighbors of v_i are held in the *edge-value* array, from *edge-index*$[i]$ to *edge-index*$[i + 1]$ (see Figure 5.2). Attractive forces between v_i and its neighbors are calculated in the device as it is in the sequential version. The method that calculates the attraction force between two nodes is ported from Java to CUDA C. All attractive forces applied on v_i are summed up and stored in *spring-x*$[i]$ and *spring-y*$[i]$.

In the effort to calculate repulsive forces applied on v_i, geometric neighbors of v_i in the graph should be known. Indices of surrounding nodes are stored in the i^{th} row of *surrounding-matrix*. Hence, this row is traversed until reaching the delimiter index -1, or reaching the end of row. Repulsive forces between v_i and its surrounding are calculated in the device as it is in the sequential version. The method that calculates the repulsion force between two nodes is ported from Java to CUDA C. All repulsive forces applied on v_i are summed up and stored in *rep-x*$[i]$ and *rep-y*$[i]$.

If the third column of i^{th} row in *owner-data* matrix equals to 1, this means node v_i is isolated and a gravitational force should be applied on it. So, the gravitational force applied on v_i is calculated and stored in *grav-x*$[i]$ and *grav-y*$[i]$. Calculation of gravitational force applied on a node in the device is implemented as it is in the sequential version. The method that calculates the gravitational force on a node is ported from Java to CUDA C.

The overall algorithm that is implemented on the device side is shown in Figure 5.3.

method CALCULATEALLFORCESONANODE*(int numberOfNodes)*
1) i :=*threadIndex*
2) **if** i <*numberOfNodes* **then**
3) obtain owner graph index of the i^{th} node
4) set forces to be exerted on the i^{th} node to zero
5) calculate the total spring force applied on the i^{th} node
6) **if** current iteration number is multiple of 10 **then**
7) construct the surrounding list of the i^{th} node
8) calculate the total repulsion force applied on the i^{th} node
9) **if** a gravitational force should be applied on the i^{th} node **then**
10) calculate the gravitational force applied on the i^{th} node

Figure 5.3: General force calculation algorithm on the kernel. This algorithm runs simultaneously on each thread.

5.2.2 Host Side

Data Structures

One of the weaknesses of CUDA C framework is that it does not allow allocation or de-allocation of the GPU memory dynamically on the device side. On the other hand, CUDA C provides memory handling functions in the host side. Thus, parallel computations are started just after allocating required spaces for variables that will be used in the kernel.

Because CoSE layout algorithm is implemented in Java language, we prefer to use Java for implementing the host side. So, use the JCuda framework [22], which

wraps the CUDA functionality (including driver methods) and lets developers use a Java interface for calling CUDA methods.

All of the data structures mentioned in Section 5.2.1 are allocated from CPU via JCuda framework. Data structures like *edge-value*, *edge-index*, *ideal-edge-lengths*, *surrounding-matrix*, *owner-data* and *graph-geometry* are allocated before the layout calculation is started.

Edge-value, *edge-index*, *ideal-edge-lengths* arrays and *owner-data* matrix are constructed and copied from the host to the device only once and used during the whole layout process, because these structures are dependent to the graph topology and cannot be changed during the layout calculations. On the other hand, *graph-geometry* matrix is constructed on the CPU for each iteration, and copied to the GPU texture memory. Memory allocations of output arrays (*spring-x*, *spring-y*, *rep-x*, *rep-y*, *grav-x* and *grav-y*) in GPU global memory are handled before making the kernel call in each iteration. During the construction of data structures that will be copied to the device memory, Java arrays are used regardless of whether or not the constructed data is one or two dimensional.

Algorithms

If a CUDA enabled GPU is plugged to the motherboard, and parallel programming option is enabled by the user, then parallel computation is initiated via *JCuda*. In order to make kernel calls, we need to create a file that can be loaded and executed using the *CUDA Driver API*. For creating this file, kernel source code has to be compiled by the *NVCC* compiler [23]. We have two options for compilation of the kernel source. One is creating a *PTX* file, which is a human-readable (but hardly human-understandable) file containing a form of *assembler* source code. The other option is creating a *CUBIN* file, which is a *CUDA binary* and contains the compiled code that can be directly loaded and executed by the GPU. *CUBIN* files are specific to the capability of the GPU, which is kind of a version number for the GPU hardware. *CUBIN* files that have been generated for a *compute capability* cannot be loaded on a GPU with a lower capability. On the

other hand, *PTX* files are generated at runtime for the GPU of the target machine which makes it *write-once, compile-anywhere*. Thus, we prefer to generate *PTX* files [23].

First, the device is initiated. Created *PTX assembler* file is loaded to the initiated device. This *PTX assembler* file is known as the *module* in CUDA terminology. After loading the *module*, a function pointer to the kernel main function is obtained.

All data structures except the outputs are allocated before running the spring embedder. Structures that are related with the graph topology and cannot be changed during the layout process are constructed and copied to the device. For arrays and matrices which will be stored in a *texture cache*, a texture reference is set in addition.

In each iteration of the spring embedder, *graph-geometry* matrix is constructed, and copied to the *texture* memory of the device. Output arrays are allocated from the *global* memory of the device. After that, kernel main function is called via the obtained function pointer. Pointer to the output arrays are passed as argument to the kernel main function. For preventing a miscalculation, each kernel call should be synchronized, so that, execution in the host side is paused until the device becomes idle. When force calculations in the device are completed, output forces are copied back to the host side. Then, device memory allocated for this iteration is de-allocated. Finally, attractive, repulsive, and gravitational forces are applied to the nodes on the CPU side. When the spring embedder finishes the layout computation, device memory used for this layout is de-allocated.

The general algorithm that is implemented on the host side is shown in Figure 5.4.

method ParallelLayout*(Graph G)*
1) generate binary file
2) initialize the device
3) load the binary file
4) obtain the function pointer to the kernel main function
5) **while** G is not converged **or** max. number of iterations is not reached **do**
6) construct graph geometry matrix
7) prepare device memory fields
8) call the kernel main function
9) copy resulting force values to host
10) clean-up device memory
11) apply forces

Figure 5.4: General layout algorithm on the host side after implementing the parallel computing approach.

5.3 Complexity Analysis

Let us first focus on the execution time of the force calculations and then move onto the whole spring embedder. Since the execution time is mostly dependent on the structure of the input graph, we will make some assumptions for simplifying the complexity analysis.

Applying FR-grid variant method to CoSE, puts the average execution time of calculation of total force to be exerted on a single node in the order of $2|E|/|V|+C$. Total time to calculate all the attractive forces is $2|E|$, because the attractive force that will be applied on two nodes are calculated in two threads. Total time to calculate the repulsive and the gravitational force to be applied on a single node is constant and indicated with C. Assuming edges are uniformly distributed to nodes, sequentially calculating the total forces for all nodes makes the average complexity of one spring embedder iteration $O(|E| + |V|)$ with a large constant factor.

FR-grid variant method is implemented for kernel side and is currently being used by default. Therefore, we can assume that vertices are laid out evenly on the drawing area as we did in the analysis of the FR-grid variant. In this case, there will be at most 64 vertices to be checked for repulsive force calculation as described

in Section 3.2. Secondly, we assume that edges are uniformly distributed between nodes. So that $deg(v)$ becomes approximately the same for all $v \in V$. Note that, $deg(v) \simeq \frac{2|E|}{|V|}$ and number of edges can be indicated as a multiple of the number of vertices. The maximum value of $\frac{2|E|}{|V|}$ is $|V|$ for complete graphs, but for convenience, $\frac{2|E|}{|V|}$ can be assumed to be constant. Finally, calculating the gravitational force to be applied on a vertex takes constant time.

To sum up, calculation of all forces takes constant time on the average. As we know, for $|V| < |T|$, where $|T| = \textit{number of processors} \times \textit{maximum threads per block}$, kernel runs for each one of the vertices is handled in parallel. Thus, on the average case, force calculations are executed in constant time for $|V| < |T|$.

In order to implement the force calculations on the GPU, the device should be initialized and topology of the graph should be copied to the kernel side once for each layout. On the other hand, geometry of the graph should be updated for each iteration. So, CoSE spends $O(|V|)$ per iteration for copying the geometry of the input graph. Because of the memory copying costs, running time of the CUDA supported CoSE becomes $O(|V|)$ with a less constant factor compared to the plain Java implementation, assuming $|V| < |T|$. Meanwhile, Java function overheads for device initialization takes relatively long time but luckily, this process is done only once for each layout calculation. In conclusion, for graphs containing more nodes than a threshold value, parallel CoSE will eventually yield better results in comparison with the sequential one.

However, input graphs with structure far different than the one described above will yield different results. For instance, in complete graphs, there are $\frac{n(n-1)}{2}$ edges. In such a case, we cannot assume $\frac{2|E|}{|V|}$ as a constant value. Also for a star-shaped graph, there is a *god vertex* that is incident with all the edges, while other vertices in the graph are incident with only one edge. Thus, the thread that calculates the attractive forces on the *god node* spends $O(|V|)$ time. Because the kernel is waited to be idle before passing to the next iteration, force calculations will cost $O(|V|)$.

5.4 Results

For testing the running time of the parallel implementation of the CoSE, three types of graphs: mesh-like, tree-like and compound graphs are generated randomly. For performance comparisons, tests are run with graphs that contain same structural properties with the graphs that are used for testing the previous methods. Also, tests are run on the same system/machine configuration with the previous methods. Different from testing of previous methods, we used larger graphs ranging from 500 to 2500 nodes. For each graph size, we run sequential and parallel CoSE ten times and take the average execution times. Comparisons of execution times before and after adapting the parallel computing approach to our compound spring embedder are shown in Figures 5.5 through 5.10.

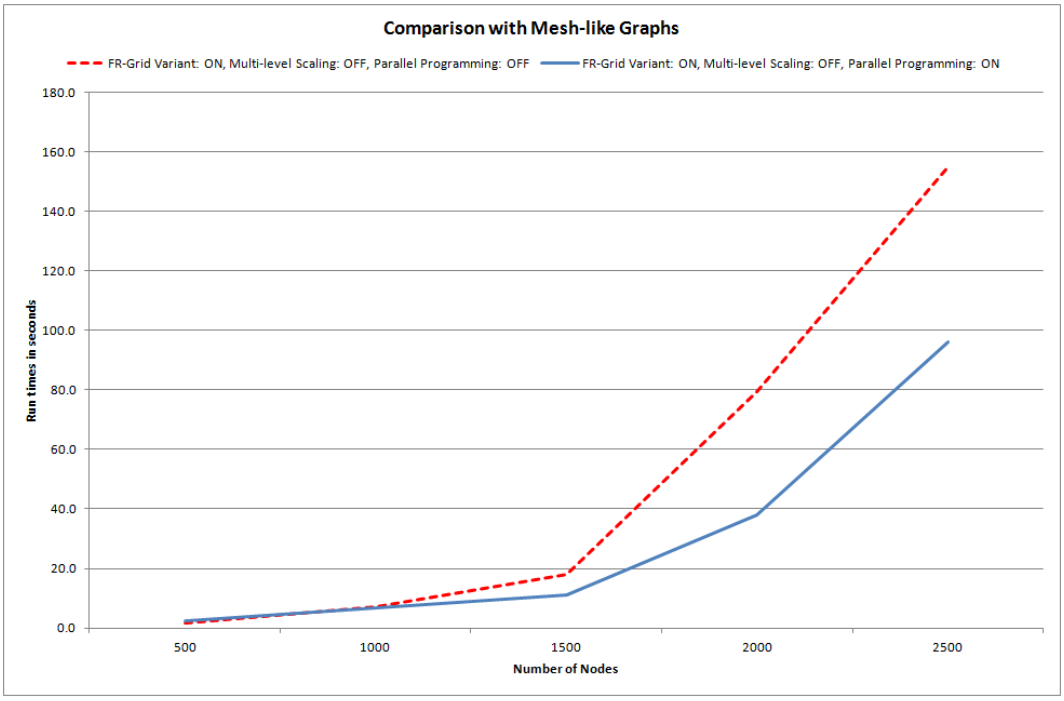

Figure 5.5: Parallel CoSE vs. Sequential CoSE with mesh-like graphs (Only FR-grid variant method is applied).

The major part of the compound spring embedder is the force calculations. Thus, calculating the applied forces in parallel should cause a drastic decrease in the execution time. However, because of the Java function overheads and memory copying costs, our parallel implementation will be advantageous for graphs

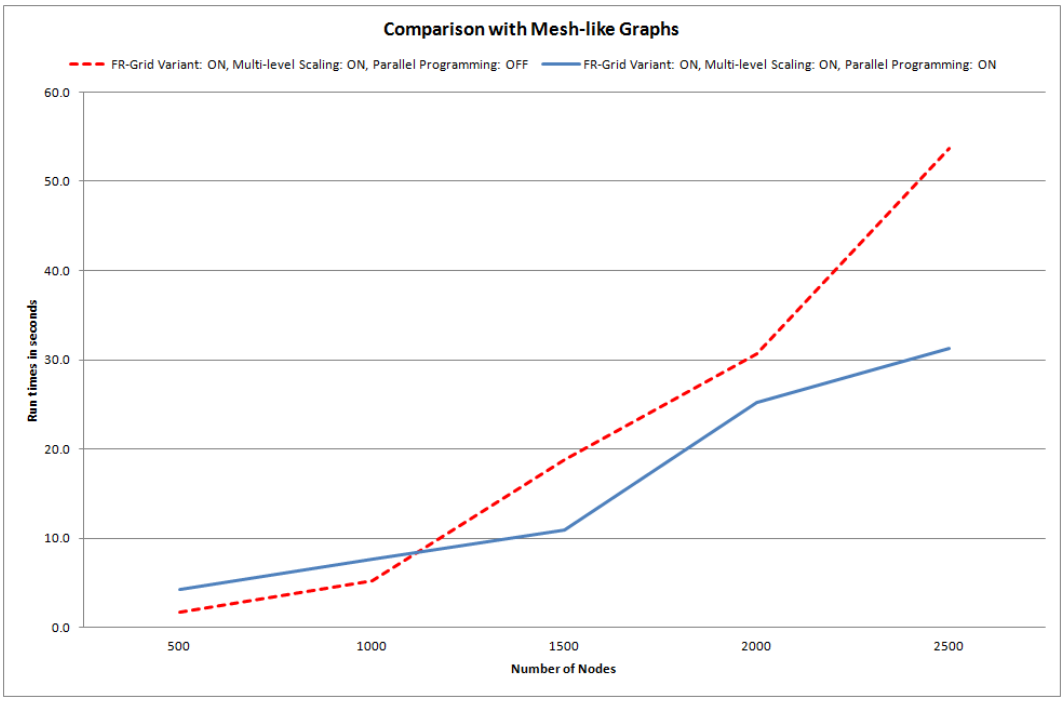

Figure 5.6: Parallel CoSE vs. Sequential CoSE with mesh-like graphs (Both FR-grid variant and multi-level scaling methods are applied).

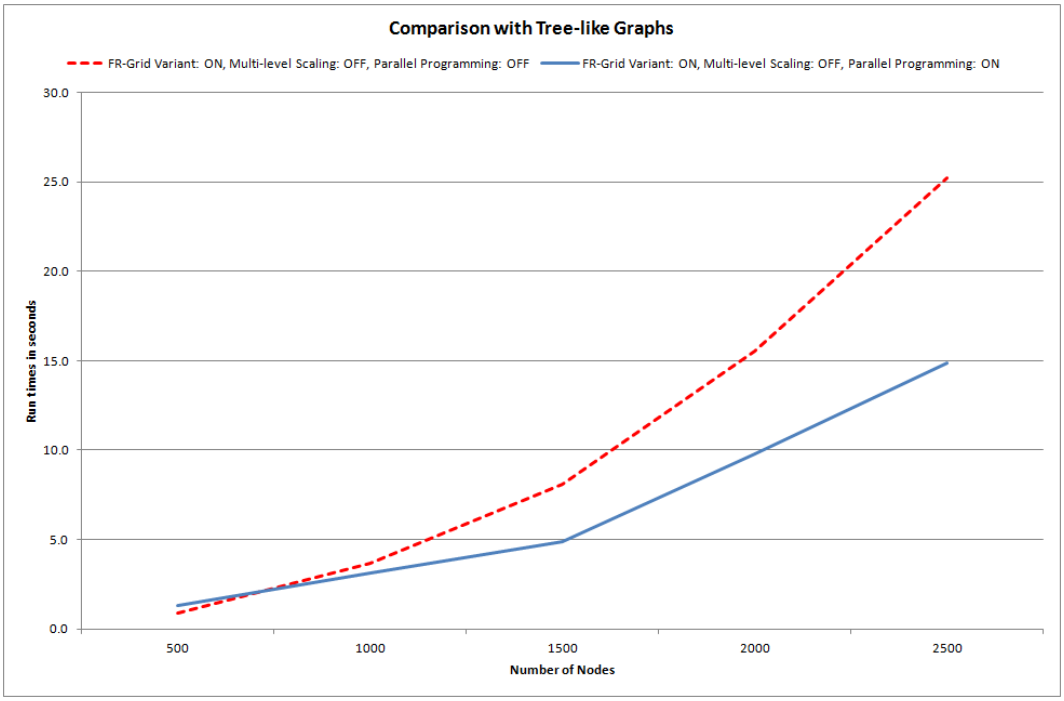

Figure 5.7: Parallel CoSE vs. Sequential CoSE with tree-like graphs (Only FR-grid variant method is applied).

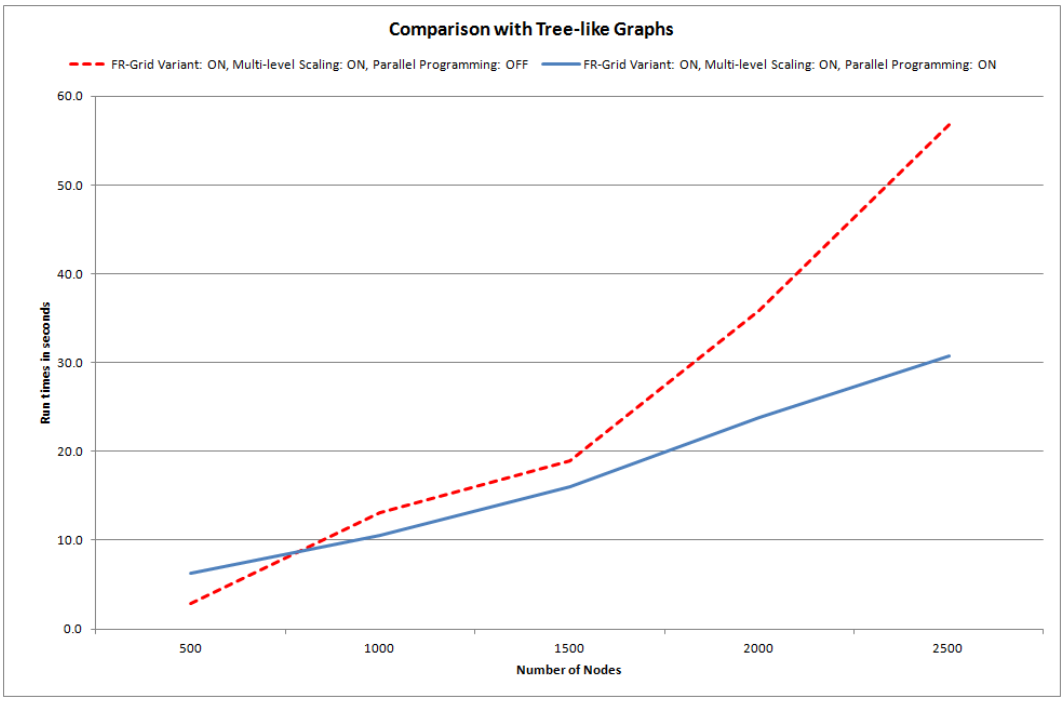

Figure 5.8: Parallel CoSE vs. Sequential CoSE with tree-like graphs (Both FR-grid variant and multi-level scaling methods are applied).

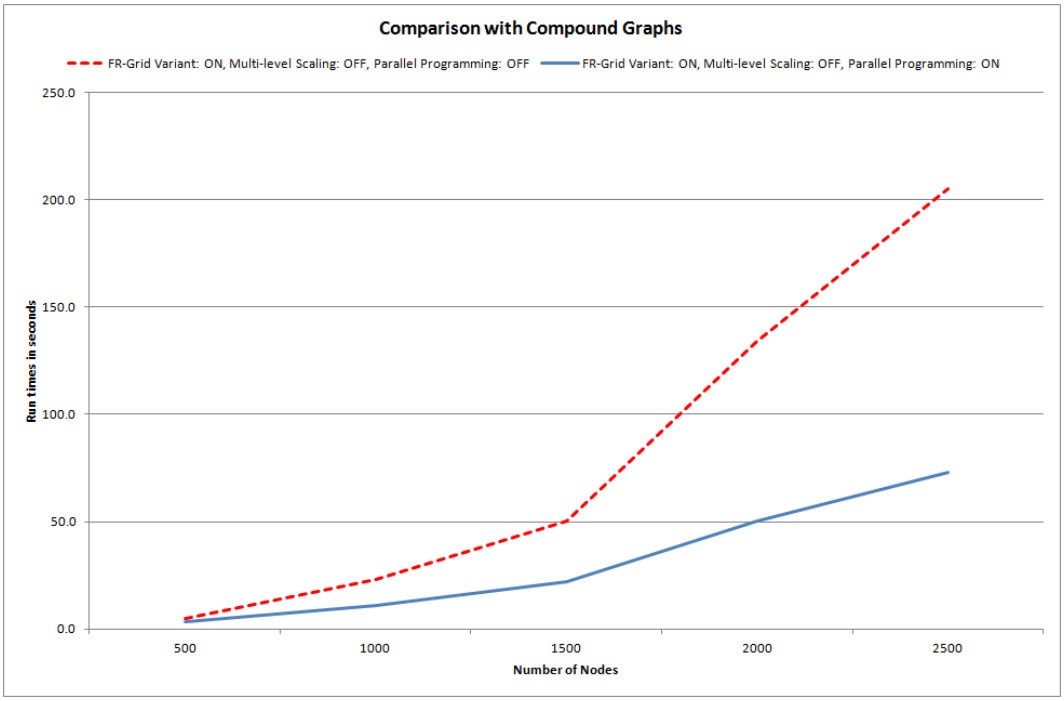

Figure 5.9: Parallel CoSE vs. Sequential CoSE with compound graphs (Only FR-grid variant method is applied).

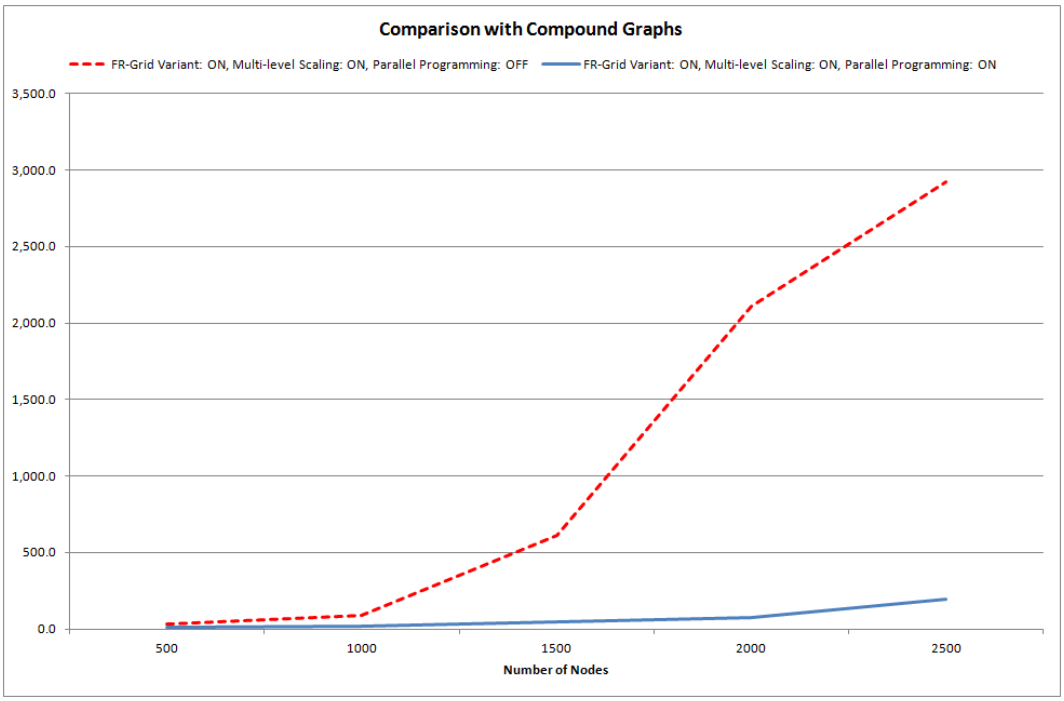

Figure 5.10: Parallel CoSE vs. Sequential CoSE with compound graphs (Both FR-grid variant and multi-level scaling methods are applied).

containing more than a certain number of vertices. This is the reason why we used graphs in greater sizes.

Sequential and parallel implementations are compared for observing the performance improvement. FR-grid variant method is used by default during all test runs. We also compared the run times by applying our multi-level scaling method. As expected, for huge graphs where $|V| > 1000$ we obtained a substantial decrease in run times and our GPU supported CoSE implementation is 1.5 to 15 times faster than the CPU-only implementation. Approximately, the threshold sizes for mesh, tree and compound graphs where the parallel implementation starts to outperform the sequential one are 1500, 1000, and 500, respectively.

5.5 Future Work

Currently, only the force calculation process is run in parallel. The reason is CoSE calculates the boundaries of the compound nodes recursively and recursive invocations are avoided in GPU computing because of the SIMD structure. In the future, whole embedder may be implemented in parallel by avoiding recurrence. Moreover, drawing the final placement using GPU programming can be a good improvement on performance.

Chapter 6

Conclusion

In order to decrease the asymptotic running time complexity from quadratic to linear, we adapted the grid variant of Fruchterman and Reingold's spring embedder algorithm [7] to CoSE. We observed that adapting FR-Grid variant makes CoSE to run 5 to 35 times faster for graphs ranging from 100 to 700 nodes.

For graphs with more than 100 nodes, and graphs that contain a particular structure, CoSE converged pre-maturely or could not converge until number of iterations hits to its maximum value. For obtaining better looking placements, we adapted Walshaw's clustering method [11], which is an implementation of multi-level scaling. As expected, visual quality is significantly improved after implementing the Walshaw's method.

In CoSE, attractive, repulsive and gravitational forces exerted on node v_i had been calculated after the force calculations of all v_j's are finished, where $j < i < |V|$ and. This sequential process is not the only way to calculate forces, since force calculation of a node has no dependency to or interaction with the force calculation of other nodes. CoSE can calculate the applied forces in parallel.

We were inspired by the work in [14] during the adaptation of the parallel computation principles to CoSE. Applying parallel programming strategy decreased

the high constant in the complexity of CoSE (with the FR-grid variant applied). However, due to the copying costs, improvement in running time was observable for very large graphs with more than a thousand nodes. After the adaptation, CoSE runs 1.5 to 15 times faster.

6.1 Discussion

When we obtain the results of performance comparisons between sequential and parallel CoSE, we notice that, if multi-level scaling strategy is applied, the difference between run times increases slightly faster. The reason may be the sizes of the graphs. Force-directed algorithms are efficient for placement of small graphs, where $|V| < 100$. For greater graphs, force-directed methods can converge prematurely, or oppositely, can reach the maximum number of iterations without laying out the graph properly.

6.1.1 Parameter Tuning

Tuning of the parameters like *spring constant*, *repulsion constant*, *cooling factor*, etc. should be considered, as it affects the behavior of the mechanical system. For example, as the springs get stronger, final placements get better in visual quality but with a late convergence. However, a strong spring (with a high constant value) might cause high degree nodes to oscillate. Oscillations in early iterations provide a reasonable empty area around a node. These oscillations result relatively better looking layouts, but system reaches the stable state lately.

Increasing the repulsion constant, improves the visual quality. Since the nodes repel each other more strongly and the empty area around the nodes increases. Changing the repulsion constant does not affect in the same proportion as changing the spring constant, since the impact of an attractive force is greater than a repulsive force in a spring embedder.

In addition, if the initial placement of the graph is not far from desired,

CoSE is run incrementally and no random placement is done. For incremental layouts, *maximum displacement threshold* for one iteration can be smaller, since it is assumed that, the graph is close to a stable state and nodes should move slowly.

In conclusion, we ignore the performance penalty and increase the spring and repulsion constants slightly, because the performance of the CoSE is improved by adopted methods.

6.2 Availability

All of the improvements mentioned in this thesis are implemented under *Chisio Layout* (*ChiLay*) [24] project which is stored in sourceforge [25], [26].

After implementing and testing FR-grid variant and multi-level scaling methods to CoSE, we published ChiLay version 2.0 [24]. It contains several layout styles including the improved CoSE. In order to use this layout package, Chisio Editor Version 2.0 [21] can be downloaded and installed. For our GPU supported compound spring embedder, we created a new branch under the sourceforge repository [26] and named it *improved-CoSE-CUDA*. Source of the fully improved CoSE can be accessed from the repository [27].

Bibliography

[1] "Phenixvision." http://www.hospitals-management.com/products_services/communications_infotechnology/phenix_vision.html, Accessed in August 2012.

[2] "Pathway commons." http://www.pathwaycommons.org, Accessed in August 2012.

[3] "Cytoscape." http://www.cytoscape.org/, Accessed in August 2012.

[4] A. Dilek, M. E. Belviranli, and U. Dogrusoz, "Visibioweb: visualization and layout services for biopax pathway models," *Nucleic Acids Research. 38(Suppl), pp. W150-154*, 2010.

[5] "Tom Sowyer Software." http://www.tomsawyer.com/, Accessed in August 2012.

[6] "Facebook Social Graph." http://www.mihswat.com/labs/app/facebook-social-graph/, Accessed in August 2012.

[7] T. Fruchterman and E. Reingold, "Graph drawing by force-directed placement," *Software Practice and Experience 21 (11) (1991) 11291164*, 1991.

[8] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir, "A layout algorithm for undirected compound graphs," *Information Sciences, 179, pp. 980994*, 2009.

[9] S. Hachul and M. Jnger, "Large-graph layout with the fast multipole multilevel method," *Technical report, Zentrm fr Angewandthe Informatik Kln*, 2005.

[10] *NVIDIA CUDA C Programming Guide*. NVIDIA CUDA, 2011.

[11] C. Walshaw, "A multilevel algorithm for force-directed graph drawing," *(Proc. Graph Drawing) LNCS, 1984:171-182*, 2001.

[12] "BioPAX - Biological Pathway Exchange." `http://sbml.org/Main_Page/`, Accessed in August 2012.

[13] "GraphML - A File Format for Graphs." `http://graphml.graphdrawing.org/`, Accessed in August 2012.

[14] A. Godiyal, J. Hoberock, M. Garland, and J. C. Hart, "Rapid multipole graph drawing on the GPU," *GD 2008, LNCS 5417, pp. 90101*, 2009.

[15] *Graduate Texts in Mathematics, Graph Theory*. Springer, 2000.

[16] "Java Swing." `http://www.javaswing.org/`, Accessed in August 2012.

[17] P. Eades, "A heuristic for graph drawing," *Congressus Nutnerantiunt, 42, 149160*, 1984.

[18] T. Kamada and S. Kawai, "Automatic display of network structures for human understanding," *Technical Report 88-007, Department of Information Science, Tokyo University*, 1988.

[19] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," *S. Karin, editor, Proc. Supercomputing '95. ACM Press*, 1995.

[20] "CUDA: Parallel Programming and Computing Platform." `http://www.nvidia.com/object/cuda_home_new.html`, Accessed in August 2012.

[21] "Chied: Chisio editor." `http://cs.bilkent.edu.tr/~ivis/chied.html`, Accessed in August 2012.

[22] Y. Yan, M. Grossman, and V. Sarkar, "Jcuda: A programmer-friendly interface for accelerating java programs with cuda," *Lecture Notes in Computer Science, 2009, Volume 5704/2009, 887-899*, 2009.

[23] "JCuda Tutorial." `http://www.jcuda.de/tutorial/TutorialIndex.html`, Accessed in August 2012.

[24] "Chilay: Chisio layout." http://cs.bilkent.edu.tr/~ivis/chilay.html, Accessed in August 2012.

[25] "Sourceforge." http://sourceforge.net/, Accessed in August 2012.

[26] "Source of chilay." https://chilay.svn.sourceforge.net/svnroot/chilay, Accessed in August 2012.

[27] "Fully improved cose." https://chilay.svn.sourceforge.net/svnroot/chilay/chilay2x-improved-CoSE-CUDA, Accessed in August 2012.