Study of Planar Graph Algorithms

—CS 597 Independent Study Report

Qixin (Stark) Zhu

Department of Computer Science
University of Illinois at Urbana Champaign

Table of Contents

1. Introduction

1.1. Planar Graph

A planar graph is a graph that can be drawn on the plane such that no edges cross each other. A plane graph can be defined as a planar graph with a mapping from every node to a point on a plane, and from every edge to a plane curve on that plane, such that the extreme points of each curve are the points mapped from its end nodes, and all curves are disjoint except on their extreme points [1]. A planar embedding of a graph G is a continuous injective function from the topological graph to the plane. A planar graph is an abstract graph that has at least one planar embedding [2].

The dual graph of a planar graph G is a graph that has a vertex for each face of G. The dual graph has an edge whenever two faces of G are separated from each other by an edge, and a self-loop when the same face appears on both sides of an edge. Each edge e of G has a corresponding dual edge, whose endpoint are the dual vertices corresponding to the faces on either side of e [3]. The correspondence between primal and dual edges easily extends to larger structures within any connected planar graph G, as summarized in the Figure 1 [1].

primal G	dual G^*	primal G	dual G^*
vertex v	face v^*	empty loop	spur
dart d	dart d^*	loop	bridge
edge e	edge e^*	cycle	bond
face f	vertex f^*	even subgraph	edge cut
$tail(d)$	$left(d^*)$	spanning tree	complement of spanning tree
$head(d)$	$right(d^*)$	$G \setminus e$	G^* / e^*
$succ$	$rev \circ succ$	G / e	$G^* \setminus e^*$
clockwise	counterclockwise	minor $G \setminus X / Y$	minor $G^* \setminus Y^* / X^*$

Figure 1. Correspondences between features of primal and dual planar graphs

1.2. Separators

"Divide and conquer" is one of the oldest and most widely used techniques for designing efficient algorithms [2]. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem [4]. This strategy can be successfully and efficiently applied to graph problems, provided we can quickly separate the graph into roughly equal subgraphs [2].

The separator theorem states that for any n-vertex planar graph G=(V, E) (n = |V|) and for any weight function w: V → $\mathbb{R}^+$, V can be partitioned into 3 sets A, B, S ⊆ V such that (1) A and B are α-balanced: w(A), w(B) ≤ $\alpha \cdot w(V)$ for some $\alpha \in (0,1)$; (2) A and B are separated: no edge joins a vertex in A with a vertex in B; (3) separator S is small: $|S| \leq f(n)$; (4) the partition can be found efficiently in linear time. Finding planar separators correctly and efficiently is important because it is a fundamental tool for many complicated tools or algorithms.

1.3. R-division

Separators are used to find a division of a graph, that is, a partition of the edge-set into two or more subsets, called regions. An r-division of G is a decomposition into O(N/r) edge-disjoint regions, each of which has vertices less than or equal to r and has $O(\sqrt{r})$ boundary vertices. The root of the separator hierarchy tree is the entire graph itself, and the two children are the roots of separator trees constructed recursively for the subgraphs A and B induced by the root-level separator S. A naïve method of constructing this r-division, by Frederickson (1986) [9], is to apply the linear-time separator when traversing the separator tree, which will take a total of $O(N \log N)$ time. A second approach is to preprocess the graph with a ρ-clustering of size $\sqrt{r}$, and then contract each piece into a single node, thus shrinking the graph into N' = $O(N/\sqrt{r})$ vertices. Doing naïve recursion on this shrunk graph takes O($N/\sqrt{r}$ logN) time, which is governed by the time of expanding pieces back and doing another $O(\log r)$ levels of recursion. So the total time cost of this approach is $O(N \log r)$. The time bound of separator hierarchy construction can be further improved to linear by using advanced data structures to perform the partitions, given by Goodrich (1995) [10]. Klein (1997) [12] uses an efficient dynamic-tree implementation of a sublinear time cycle separator algorithm, initializes the data structures once and reuses results from previous steps in subsequent recursions, and achieves linear time r-division.

1.4. Single Source Shortest Path

Single source shortest path (SSSP) problem can be solved in $O(N \log N)$ time with the classical Dijkstra's algorithm on general graphs with non-negative edges length. With a suitable r-division in $O(N \log N)$ time, Frederickson (1987) [9] improves the time complexity to $O\left(N\sqrt{logN}\right)$ on planar graphs. With the linear time r-division, Klein [12] gives a complicated linear time SSSP algorithm. The simplified version using a 3-level decomposition tree runs Dijkstra speculatively on each region with limited attention span and abandons the region until later, which gives $O(N \log\log N)$ total time complexity.

1.5. High-level Roadmap

The long-term plan of this research topic (Figure 2) includes the implementation and evaluation of all 3 above mentioned r-division methods, using them as tools to solve Multi-Source Shortest Path (MSSP) problems. Further Single-Source Shortest Path (SSSP) problem solution will have 2 approaches for study. One is using MSSP as a subroutine to find all-to-all shortest path length with in a region, then treat the region as a merged vertex and solve the SSSP in the merged graph, combine results with path lengths inside each region to reconstruct the answers for the original graph. This approach has $O(N \log\log N)$ time complexity, which is the same and worth comparison to another approach described in Klein (1997) [12]. The SSSP algorithm can be further used in the practical study of the $O(N \log N)$ Max-flow problem in the preprocessing phase [13], which is expected to be the main time-consuming step. The red parts in Figure 2 are those finished so far, and the grey parts are to be finished in the future.

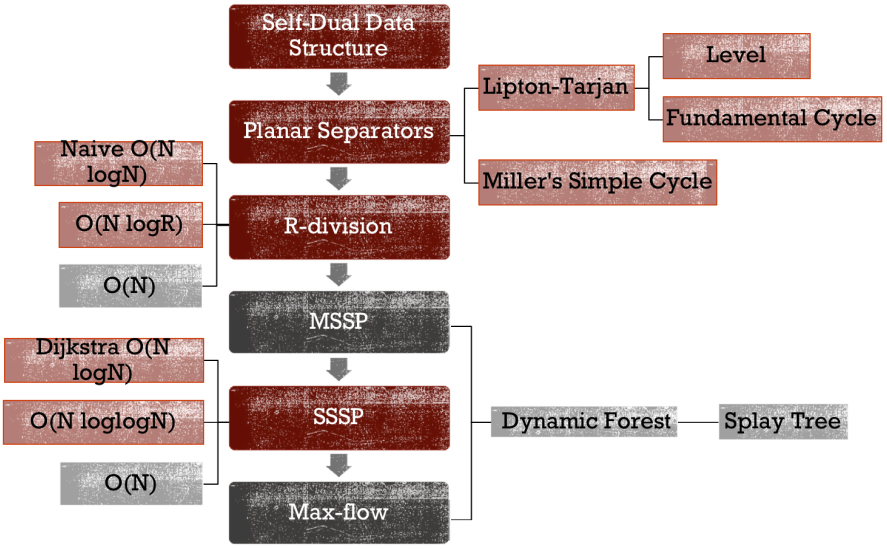


Figure 2. Road map of long-term research plan

2. Existing Research

Holzer (2009) implemented and evaluated the 3 linear-time planar separator algorithms. They presented a comprehensive experimental study of the algorithms applied to a variety of graphs scaled from 10^1 to 10^4 vertices [14]. They concluded that the Fundamental Cycle Separator (FCS) almost always outperforms other algorithms (Level Separator and Lipton-Tarjan Separator), even for graphs with large diameter. In their implementation, no dual graph is explicitly stored. To find a non-tree edge forming the cycle separator, they iterated through all non-tree edges and kept track of the vertices inside the cycle in order to return the "best" (in terms of balance and separator size) cycle found. This approach helps the algorithm to find a good cycle within linear time and thus performs well in most cases but does extra work and consumes more time than the original algorithm, which returns the first non-tree edge that forms a balanced separator.

Fox-Epstein (2016) implemented the Miller's Simple Cycle Separator (SCS) and compared to Holzer's work. Fox-Epstein confirmed Holzer's finding that FCS work well in most inputs, but also noticed FCS may output large cycles depending on the root selection of the primal spanning tree. They proposed to use SCS which not only have worst-case guarantee but also output a cycle as the separator with competitive performance. They also pushed the scale of testing graphs to 40 million vertices with 12 core CPU and 48GB RAM.

To my best knowledge, there is no implementation or practical study on any r-division algorithms, nor the $O(N \log\log N)$ or $O(N)$ SSSP algorithms on planar graphs. For the r-division and SSSP algorithms, this project gives an efficient implementation and a better understanding of the performance for practical purposes.

3. Self-Dual Data Structure

A self-dual data structure is an overlay of the sorted incidence lists of primal graph G and its dual G* [2]. It consists of two maps, one for vertices of G and the other for faces of G. Each dart object stores pointers of its head vertex, tail vertex, left and right face, reverse dart, successor and predecessor dart, next and previous dart. Each record in the two maps points to an arbitrary incidental dart of the vertex or face, as shown in Figure 3. As a result, a self-dual data structure of G is also a representation of G* and all pointers are explicitly stored. There is a tradeoff between storing all pointers explicitly versus computing them on the fly. It is easier to track and debug with explicit pointers, but it also takes more space to store the information. This could be a problem for RAM when graph gets too big and is a potential place for future improvements.

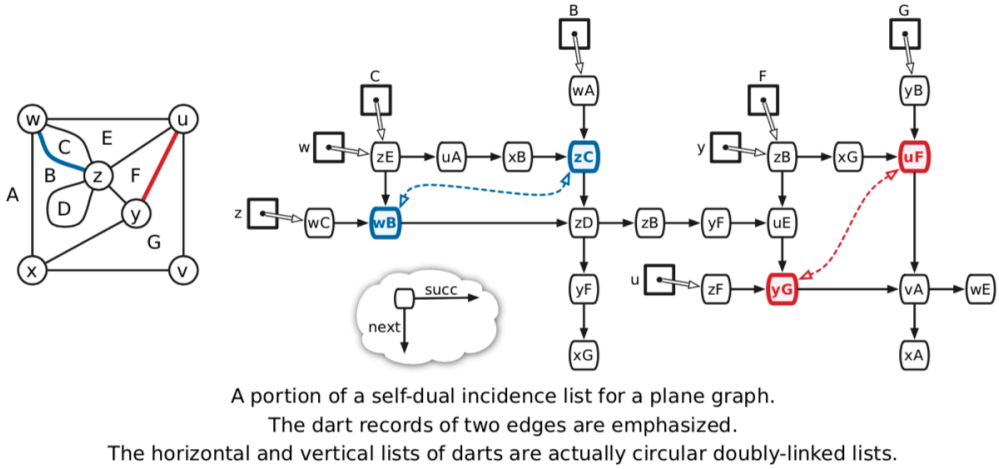


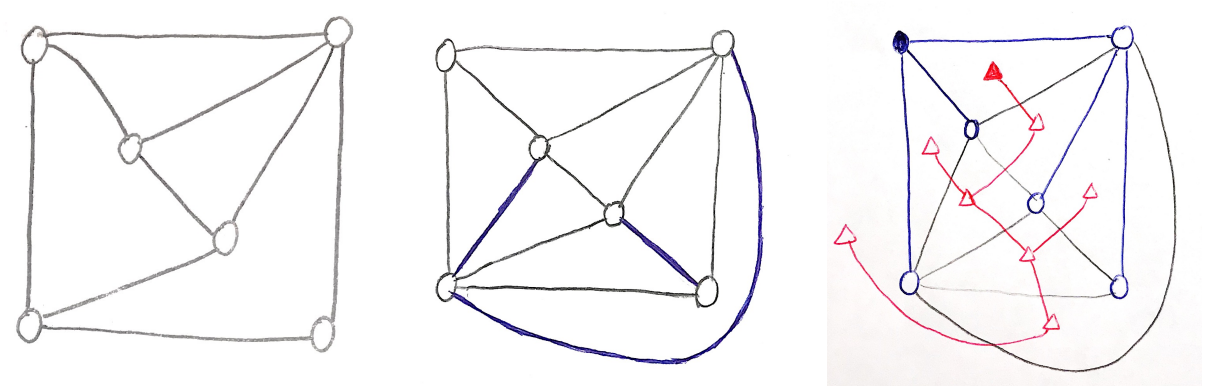
Figure 3. Partial representation of a self-dual data structure of a sample graph [2]

The Self-Dual Data Structure is implemented in this project because there is no existing data structure that performs the operations needed for updating and modifying primal-dual planar graphs. The closest existing data structure is called the half-edge data structure, in which each undirected edge is stored as a pair of half-edges for each direction. This data structure is most often used for rendering in computer graphic or meshing in finite element analysis. In both cases, the mesh is done by a separate tool which can generates a valid planar graph to be stored in the data structure. Once the mesh is generated, the vertices, edges and faces will not change. Changes can be done by re-generating a new mesh and stored in the data structure. Thus, the data structure itself does not support operations such as deleting an edge and merging its two incidental faces, adding a vertex on a face and split that face or deleting parallel edges and self-loops. Therefore, the entire Self-Dual Data Structure is implemented.

3.1. Graph Operations

Flatten

Flatten takes a planar graph, not necessary simple, checks for all parallel edges and self-loop then removes them to make the graph simple. The result on the sample graph shown in Figure 3 is shown in Figure 4(a).



(a) Flattened sample graph (b) Triangulated graph after flattened (c) Tree-coTree

Figure 4. Operations supported by self-dual data structure

Triangulate

Triangulate takes a flattened simple planar graph, examine all its faces. If a face has degree more than 3, pick an incidental vertex V and add edges between V and all its non-neighboring vertices incidental to the same face, thus splitting the face into multiple triangles. This step is needed for building a binary Dual Tree (coTree) for FCS and Lipton-Tarjan Separator. One of the many triangulation results of the previously flattened graph is shown in Figure 4(b).

Build Tree (primal) and coTree (dual)

After triangulation, the spanning tree of the primal graph (Tree) can be found using BFS, which gives the unique corresponding dual tree (coTree). The Tree and coTree is shown in Figure 4(c). The circles are primal vertices, the triangles are dual vertices (primal faces), and the root vertex is solid. By selecting the root of dual tree as the right face of an edge in the primal tree, it is ensured that the dual tree root has a degree no more than 2. This root choice further ensures the dual tree is a binary tree and guarantees the existence of an edge separating the tree by 1/3-2/3 in the worst case.

4. Dataset

The number of vertices in testing graphs is scaled from 10^1 to 10^6. For each testing graph, several trials are run with a random selected vertex as root, then average is calculated. The algorithm supports vertices and faces with different weights, but in this stage of the project, all elements are weighted equally.

Grids

A serials of grid planar graphs are generated from a fraction of a photo from NASA [17], as shown in Figure 5(a). Each vertex represents a pixel in the rectangular photo and is connected with 4 neighboring vertices (up, down, left, right) through a pair of darts.

Spheres

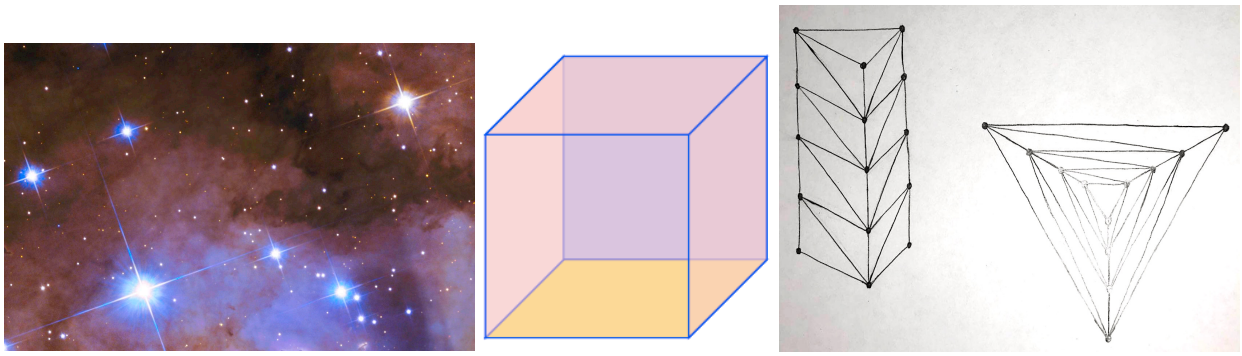
Spheres are generated from a cube or tetrahedron by iteratively adding a vertex to each face, connecting to all its incidental vertices and splitting that face, as shown in Figure 5(b). The most important characteristic of sphere type graphs is that the size of their BFS spanning tree levels grows linearly with the number of vertices.

Cylinders

Cylinders are generated from a triangle by iteratively adding a parameter triangle outside existing ones then connecting and triangulating incidental triangles, as shown in Figure 5(c). The important characteristic of cylinder type graphs is that the depth of their BFS spanning trees grows linearly with the number of vertices, whereas the optimum size of separators is clearly constant 3 (one triangle can separate its inside from its outside). There are 3 sub-types of cylinder graphs, their difference is how to connect each layer of triangle. First is to connect each layer with edges of random chosen direction, second is to connect with edges of same direction forming a centrosymmetric graph, the last is to form an asymmetric graph.

Random graphs

Random graphs are generated from a triangle by iteratively adding a vertex on a randomly chosen face and triangulating that face.



(a) NASA photo for grids (b) Base cube to generate spheres (c) Cylinder and planar embedding

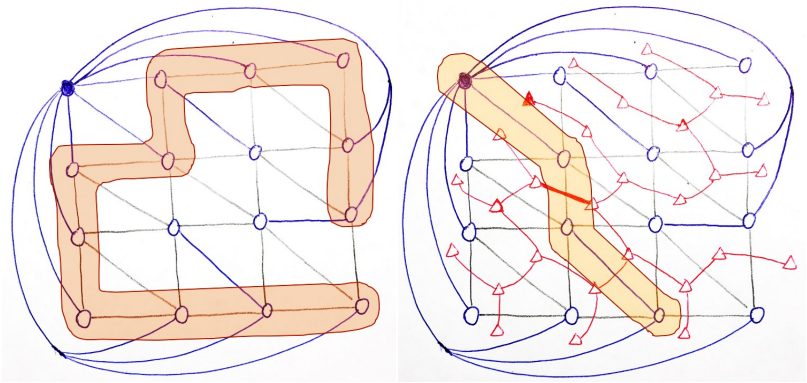
Figure 5. Datasets of planar graphs

5. Separator Algorithms

5.1. Level Separator

To find the level separator, one can pick a random vertex as root, use BFS to build the spanning tree of the primal graph and define levels of all vertices to be the distance from the root vertex. Return the median level as the separator. "median" level L_m is defined to be the level such that the number of vertices with level less than L_m is less than half, but the number of vertices with level less than or equal to L_m is greater than or equal to half. The set of vertices return as a level separator is balanced but may be big. The entire algorithm takes linear time to build and traverse the tree and count vertices number at each level.

Level separator works well for grid graphs, as shown in Figure 6(a), in the sense that when a balanced level separator is found, it will have size of $O(\sqrt{N})$. If the grid is not triangulated, pick a corner vertex as root, the algorithm will find the diagonal vertices as the median level, which contains exactly $\sqrt{N}$ vertices. If the grid is triangulated (in the not so elegant way as shown), the algorithm may find a rectangular shape cycle of size at most $4\sqrt{N}$ which separates the grid as "inside" and "outside".



(a) Level Separator (b) Fundamental Cycle Separator

Figure 6. Separators for sample grid graph

5.2. Fundamental Cycle Separator

Similar as in level separator, the algorithm picks a random vertex as root, uses BFS to find the primal spanning tree. The corresponding coTree is unique and can be built easily in linear time. Traverse the coTree and return the edge that separates it into 2 balanced sub-trees of size between 1/3 and 2/3. The cycle formed by adding that edge into the primal spanning tree is returned as the separator. The entire algorithm takes linear time to build Tree-coTree and linear time to find the edge from coTree to form the cycle.

The fundamental cycle separator may give separators of large size due to the large depth of the primal tree. This means it is dependent on the structure of the input graph as well as the selection of the root vertex and the building process of the spanning tree. The result is shown for the sample grid graph as in Figure 6(b).

5.3. Modified FCS

The original FCS returns the first dart found to form a balanced cycle, which may be as large as twice the depth of spanning tree. The modified FCS (MFCS) uses heuristics to select the "best" dart that can not only give a balanced cycle but also minimize the cycle size. Different constant time heuristics, including the distance to leaf, distance to root and minimum of those two, are studied and compared with the exact least common ancestor (LCA) algorithm, which is not useful in practice due to its linear runtime.

5.4. Lipton-Tarjan Separator

Lipton and Tarjan (1979) [5] combined the ideas from the naïve separators and constructed a balanced separator whose sized can be limited to $O(\sqrt{N})$. First, construct the BFS spanning tree and find the median level, same as the level separator method. Then find a level above and below the median level L_m that contains $\sqrt{N}$ vertices, named as L_a and L_z, as shown in Figure 7. Since all levels in between have size more than $\sqrt{N}$, there are at most $\sqrt{N}$ such levels. The algorithm then constructs a new graph by replacing all vertices above level L_z with one super vertex as root. It also deletes all vertices below level L_z. Triangulate the newly constructed graph and applies FCS to it. The cycle C found in this modified graph is guaranteed to have size at most $2\sqrt{N}$ because the tree depth is at most $\sqrt{N}$. The union of L_a, L_z and C can be returned as the Lipton-Tarjan Separator. The original graph is separated into 4 parts, which can be combined to form 2 subgraphs satisfying the balance requirement. In this approach, the size of the final separator is guaranteed to be no more than $4\sqrt{N}$. The entire algorithm takes linear time to perform operations similar to level separator and FCS, and the extra time to build a new graph is also linear.

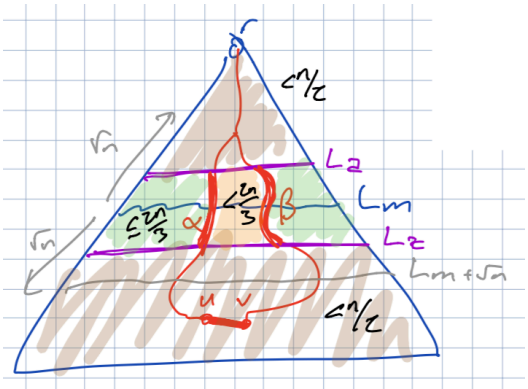


Figure 7. Sketch for Lipton-Tarjan algorithm from reference [2]

The Lipton-Tarjan algorithm is slightly modified in this project. Instead of building a new graph and re-triangulate it, the original coTree is re-assigned with new weight such that dual vertices (primal faces) outside the central zone (between L_a and L_z) is assigned zero weight. In this way, the FCS is actually finding a cycle balanced in the number of faces/vertices inside the central zone, which is equivalent to applying FCS on the new modified graph. This change improves the constant factor of the algorithm's performance by avoiding build new graph and do triangulation.

5.5. Simple Cycle Separator

The advantage of simple cycle separator (SCS) over all previous separators is that SCS output a simple cycle as the separator and the 2 separated parts are both connected, assuming the original graph is connected and triangulated. This simple cycle property is useful in some algorithms such as the $O(N \log N)$ multiple source shortest path algorithm in [11] when all sources are on the same face (incidental to the cycle separator). The connectivity of separated parts makes it easier to run the same algorithm recursively on them for further division, which is useful in r-division algorithms.

The first algorithm to find SCS is given by Miller (1986) [7], which is then implemented and studied by Fox-Epstein [15]. The implementation is rather sophisticated. Instead, I implement the SCS using a possibly simpler algorithm given by Har-Peled and Nayyeri [16]. At high level, the algorithm finds a possibly long cycle separator, uses the root-most node as a new root and rebuild a spanning tree that keeps the previously found cycle. It then computes the BFS levels of all vertices and identify the boundary cycle for each level, which forms a nested sequence of disjoint short cycles. If no level cycle is short, the algorithm finds 2 levels to divides the graph into 4 balanced parts in a way similar to Lipton-Tarjan separator and finally output a balanced combination.

5.6. Result and Interpretation

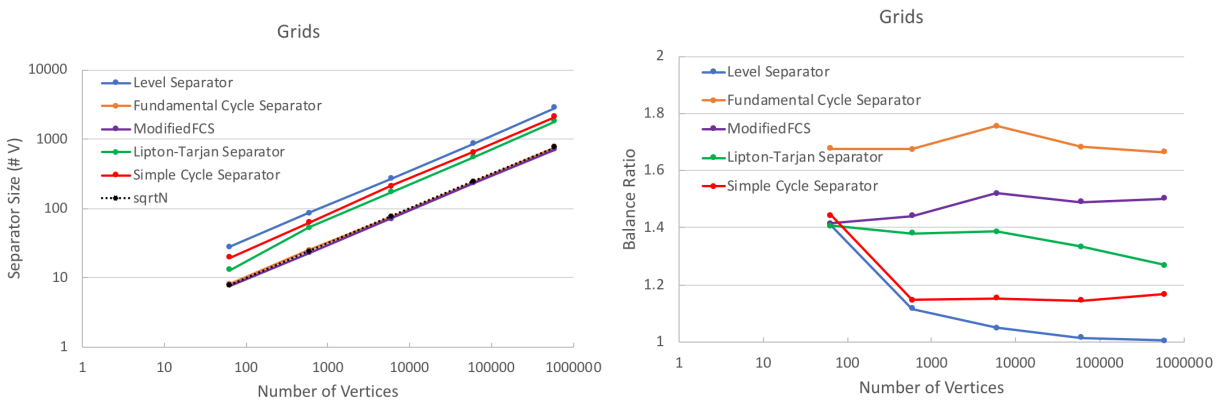
5.6.1. Metrics

Separator Size is defined to be the number of vertices in the separator returned by the above algorithms. The smaller separator size is preferred.
Balance Ratio is defined to be the ratio between the size of the two subgraphs separated by the separator. Note that the size of one subgraph is the number of vertices in the subgraph, which also includes the vertices in the separator. The "perfect" balance is equivalent to balance ratio 1.0, and the "worst" is 2.0 since both Tree and coTree are binary, in which case a 1/3-2/3 separation is guaranteed.
Runtime is defined to be the clock time of invoking find-separator method. The runtime is then plotted versus the size of graphs, to verify that the time complexity of algorithms are linear.

5.6.2. Grids

Separators have $O(\sqrt{N})$ in size as shown in Figure 8(a). Five types separator algorithms differ in terms of constant factor, among which FCS has constant factor almost 1 and level separator has constant close to 4. It is noticed that although the worst-case theoretical size bound is $4\sqrt{N}$ for Lipton-Tarjan, in grids the constant factor is about 2, less than the worst-case.

Balance ratios are between 1.0 and 2.0 as expected. It is noted that FCS gives the worst balance ratio on average, because the FCS algorithm returns the first edge it finds that can split coTree into 1/3-2/3, when there are edges that provide much better-balanced splits This problem is improved by using heuristics for all candidate cycles in MFCS.

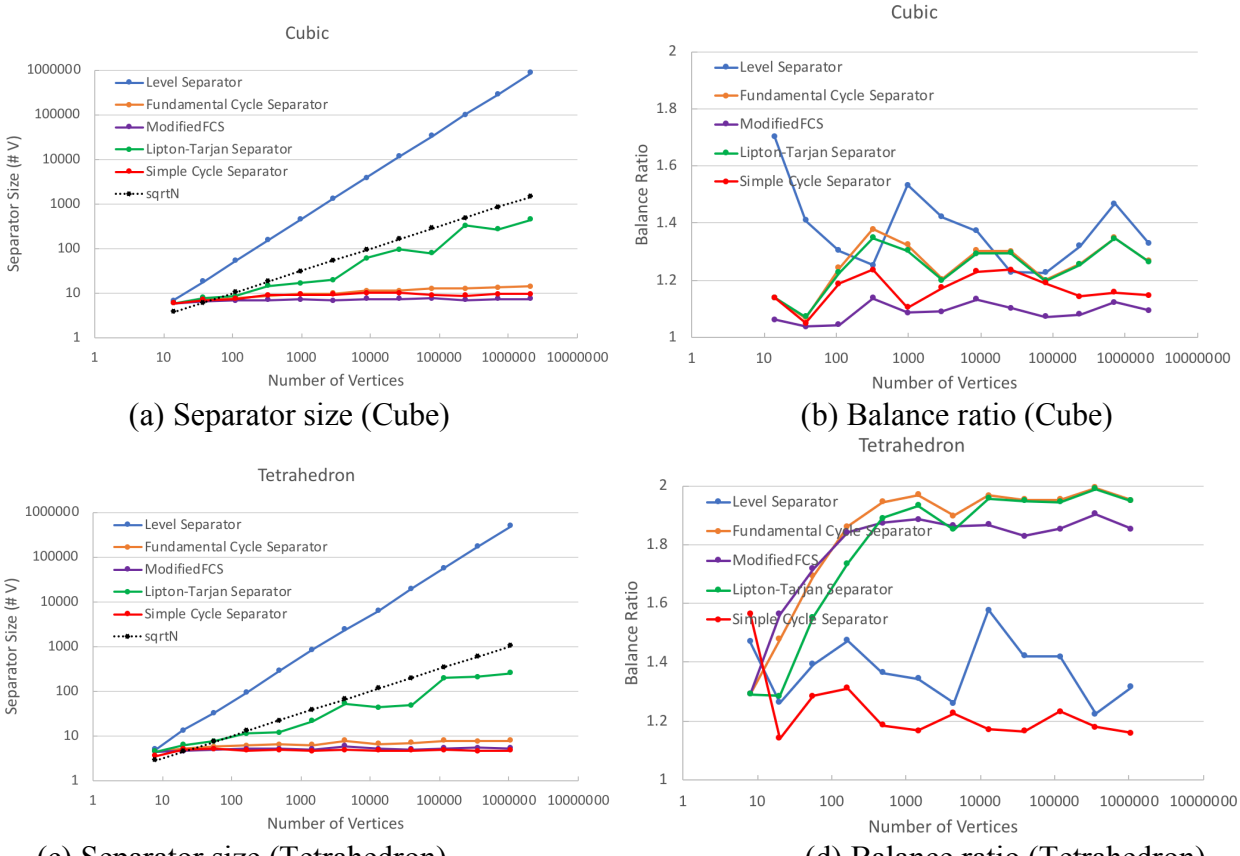


(a) Separator size (b) Balance ratio

Figure 8. Results for grids

5.6.3. Spheres

Due to the large middle levels of spheres, the size of level separators grows linearly with the number of vertices in the sphere graphs. Lipton-Tarjan still follows the $O(\sqrt{N})$ trend, with some variance, as guaranteed by the theoretical proof [5]. FCS performs best in sphere graphs with an almost O(logN) growth, since it only needs to find a parameter of the sphere to cut it into 2 halves.



(a) Separator size (Cube) (b) Balance ratio (Cube)

(c) Separator size (Tetrahedron) (d) Balance ratio (Tetrahedron)

Figure 9. Results for spheres

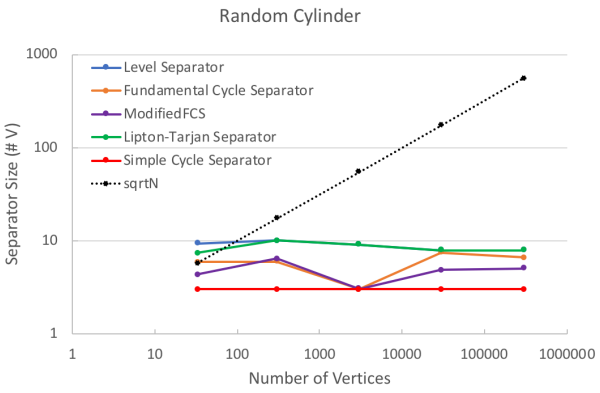
5.6.4. Cylinders

The characteristic of large spanning tree depth of cylinders leads to very different results from previous. Since the size of FCS is highly dependent on the tree depth, it grows linearly in the case of cylinders. The balance ratio is also close to 2.0 as explained before. For asymmetric cylinder graphs, it is noticed that 1 out of 32 trials of FCS did find an optimum separator of size 3. The FCS needs to be lucky "twice" to find an optimum separator: once in selecting the random root luckily need the middle of the cylinder, once in finding the optimum non-tree edge at the first time traversing coTree. Both Lipton-Tarjan and Level separator can find the optimum separators of constant size for cylinders, much less than the $O(\sqrt{N})$ theoretical bound. They also have almost 1.0 balance ratio, which means they both find the separator near the middle of the cylinder regardless where the random root is, which is an impressive property.

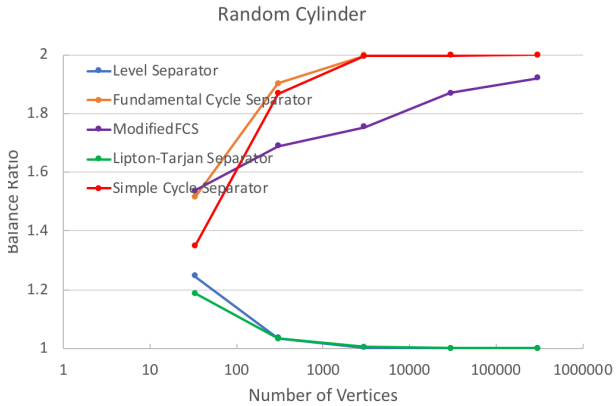
There are 2 observations different from previous researches:

(1) The observation regarding the balance ratio is different from Holzer (2009) [14] due the different implementation choices. In their paper, since no explicit dual graph stored, and no dual tree built, they examine all non-tree edges of the primal spanning tree and keep track of the inside of the cycle formed by that non-tree edge. They return the edge that gives a good balance ratio and generates the smallest cycle, which is the best option after the spanning tree is determined. This approach yields a better result while maintain the linear running time. In this project, FCS returns the first edge forming a balanced cycle, but may not be the best choice. Therefore, Holzer concluded FCS performs well in most cases, but the results shown here indicates that FCS performs not so well on large diameter graphs.

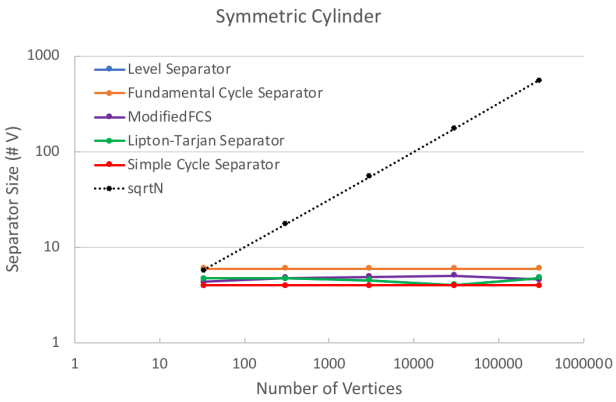
(2) The observation in this project shows that FCS and MFCS performs well only in symmetric and random cylinder graphs, which in consistent with conclusions from Holzer [14] and Fox-Epstein [15]. However, neither FCS nor MFCS can perform well in asymmetric cylinder. They both produces separators whose sizes grow linearly with the size of the graph. Therefore, I claim that Holzer's conclusion "FCS almost always outperforms other algorithms even for graphs with large diameter" is not valid.



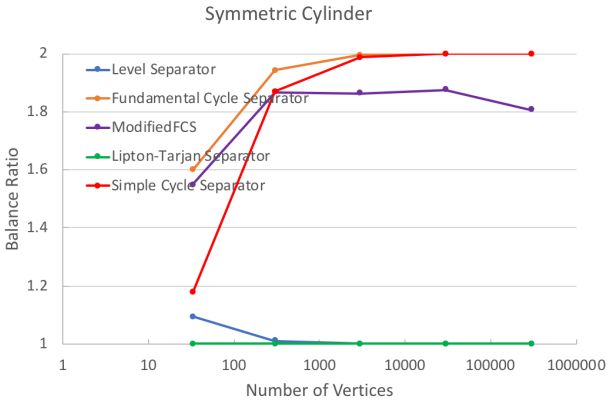
(a) Separator size (random)



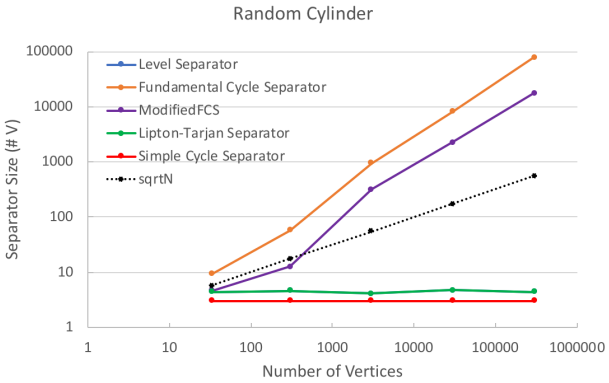
(b) Balance ratio (random)



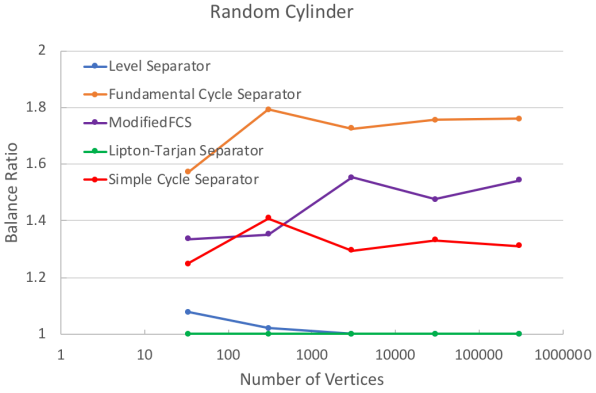
(c) Separator size (symmetric)



(d) Balance ratio (symmetric)



(e) Separator size (asymmetric)

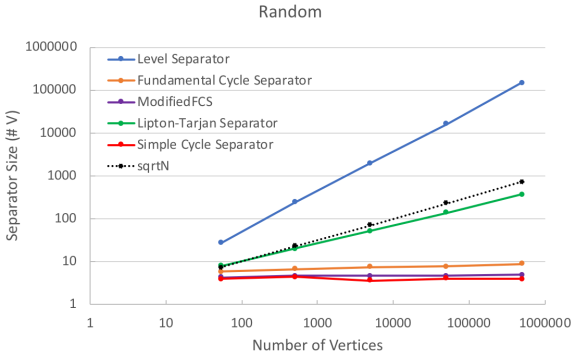


(f) Balance ratio (asymmetric)

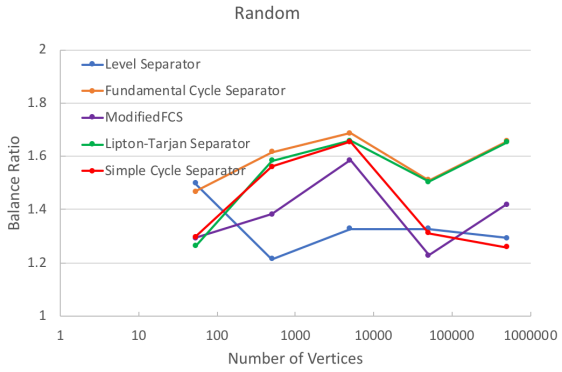
Figure 10. Results for cylinders

5.6.5. Random graphs

The trend of separator sizes is similar to the sphere graphs, discussion is skipped here.



(a) Separator size

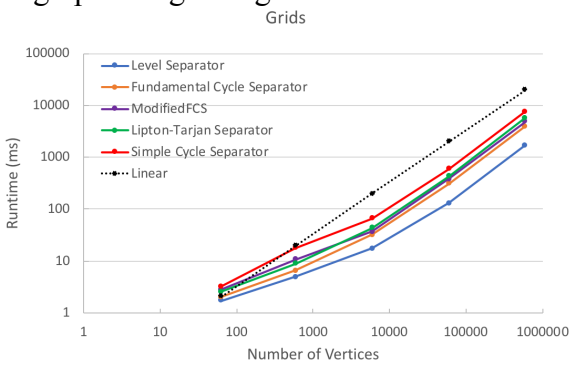


(b) Balance ratio

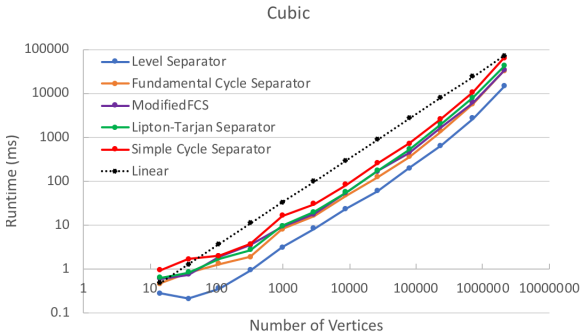
Figure 11. Results for random graphs

5.6.6. Runtime

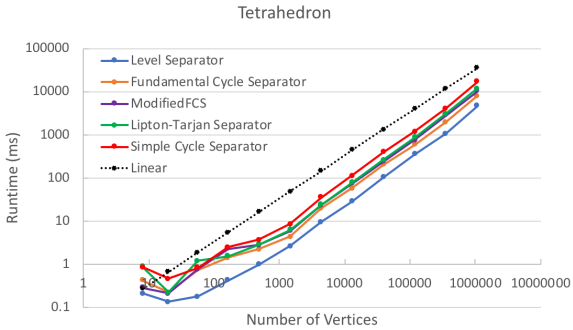
Figure 12 (a)-(g) clearly shows that all separator algorithms run in linear time on all kinds of graphs. There may be some non-trivial overhead to do sanity check for small graphs, which is negligible when graph is big enough.



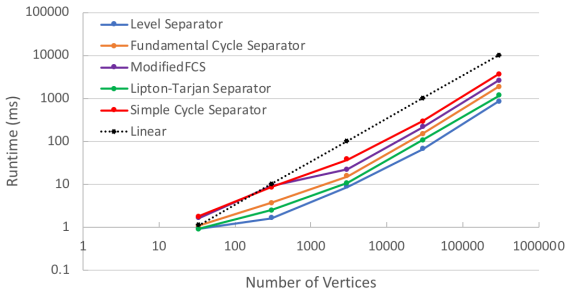
(a) Grids



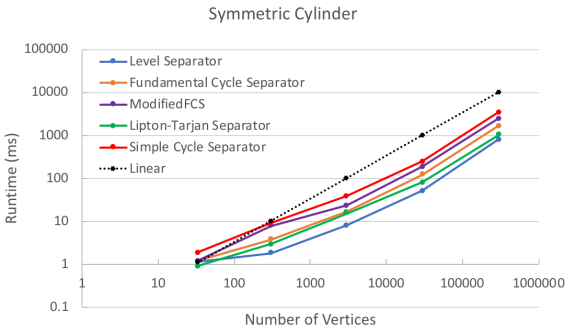
(b) Spheres (Cube)



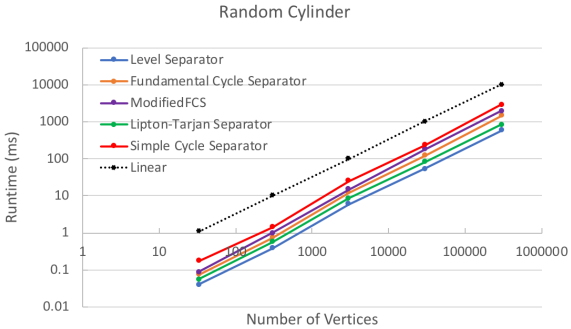
(c) Spheres (Tetrahedron)



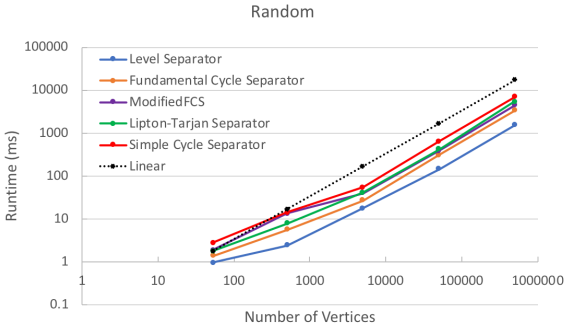
(d) Cylinders (Random)



(e) Cylinders (Symmetric)



(f) Cylinders (Asymmetric)



(g) Random

Figure 12. Runtime for separator algorithms on different graphs

5.6.7. Heuristic Study for FCS

This section tests 3 constant time heuristics for finding the smallest FCS and compares the outputs with the exact least common ancestor (LCA) algorithm, which will find the smallest cycle in linear time. The separator size ratio is defined to be the ratio between the output separator size and the smallest cycle separator size, the latter is found by checking all candidate darts (can form balanced cycle) and selecting the one with closest LCA.

As shown in Figure 13, the heuristic that uses the vertex's distance to the leaf of spanning tree gives the worst output on all kinds of graphs. In comparison, using vertex's distance to root or combine the 2 methods both give good ratios that are very close to 1.0 in most graphs. The cylinder graph with randomly directed connections between triangle layers is difficult for existing heuristics to predict the cycle size. It remains a question whether LCA of two vertices can be found exactly in constant time in spanning trees of planar graphs.

Graph #	Graph Info
1	Grids
2	Sphere Cube
3	Sphere Tetrahedron
4	Cylinder Random
5	Cylinder Symmetric
6	Cylinder Asymmetric
7	Random

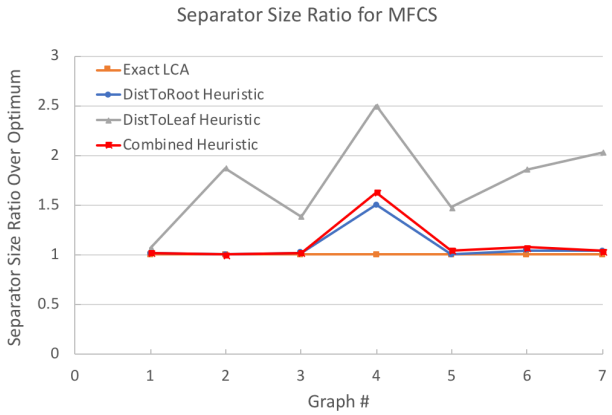


Figure 13. Separator size ratio over optimum for different heuristics

6. R-Division

6.1. Recursive Division

This simple idea first comes from Frederickson [9]. To get a r-division of a graph, we can simply apply the linear time separator algorithm recursively on the separated parts until no part has greater then r vertices. At each level of recursion, the total work done is linear in terms of the number of vertices. It takes roughly O(logN) levels of recursion, thus the entire r-division algorithm takes $O(N \log N)$ time.

6.2. Clustered Division

The high-level idea of clustered division for graph G is as follow: (1) use DFS to find a ρ-clustering ($\rho = r^{0.5}$) in linear time, which groups connected vertices into clusters; (2) contract edges in each cluster until only 1 vertex remains, the clustered graph has N' = N/ρ vertices; (3) use recursive division in 6.1 to do r-division in the clustered graph G', the time complexity is $O(N' \log N')$; (4) after finish, expand each clustered vertex into its original group, each cluster of G' now has at most $r^{1.5}$ vertices; (5) for any cluster more than r vertices, do another r-division on it. Upon finish, each group will have no more than r vertices.

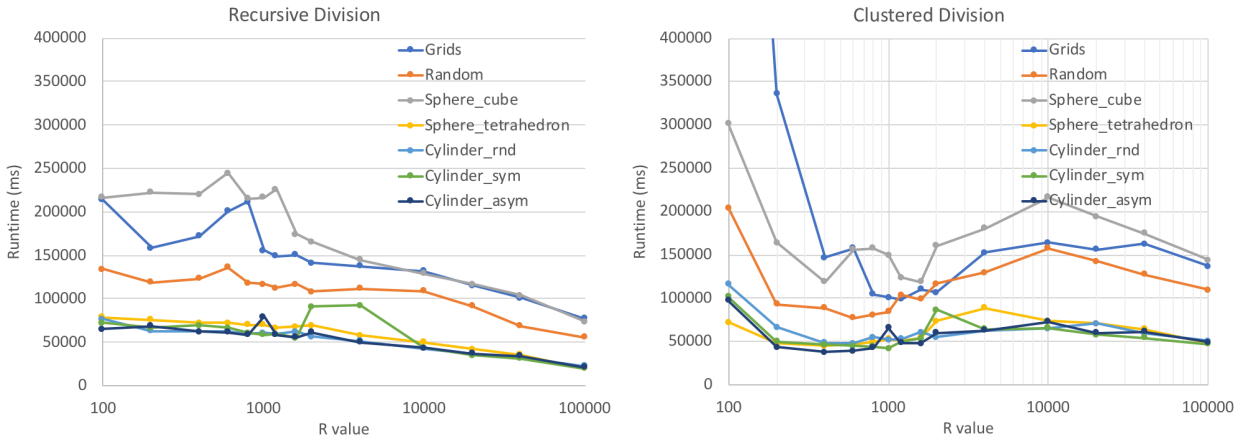
The time complexity for step (1) (2) (4) are linear. Step (3) takes $O(N' log N') = O\left(\frac{N}{\sqrt{r}} \log \frac{N}{\sqrt{r}}\right)$ time and step (5) takes $O(N \log r)$ time. The total time complexity for the algorithm is $O\left(\frac{N}{\sqrt{r}} \log \frac{N}{\sqrt{r}} + N \log r\right)$. If we set $r = (\log N)^2$, then the time complexity is $O(N \log \log N)$. Note that to achieve this, we must have at least r vertices left after contraction (otherwise the R-division will not do anything on G'), therefore $N \geq r^{1.5}$. This constraint implies that the graph G has to have at least 500 vertices to be able to show the time complexity difference between the clustered division and the recursive division algorithm.

One thing worth mention is the expansion of clustered vertices in step (4): previous research did not give details in how to handle the expansion of a boundary vertex in clustered graph G'. The naïve solution is to include them in multiple expanded regions, resulting in many faces and vertices contained in multiple regions. This may cause the total boundary size to be large and may lead to problems in algorithms that uses a decomposition tree, because many edges and vertices will have multiple parents in the tree. The implementation done in this project is to assign each face in G to exactly 1 expanded region. A face F is assigned to region R if and only if all incidental vertices of F are in region R and F has not been assigned to any other region yet. Then for each face F contained in region R, add F's incidental vertices to R. The boundary vertices may still be contained in multiple regions, but not faces. One more detail is that the clustered vertices in G' may be connected through artificial edges (added by triangulation), so the corresponding expanded region may not be connected as those artificial edges do not exist in G. To ensure the connectivity of each region, each connected component is considered to be an individual region.

6.3. Result and Interpretation

Figure 14 shows the trend of runtime for the above two R-division algorithms. Different types of graphs are used in evaluation and the number of vertices varies from 300k to 600k. The major parameter studied is the region size r.

For recursive division, it is obvious that a larger r gives shorter runtime because the graph is divided into less regions and thus can stop earlier. For clustered division, the trend is more complicated and is not so obvious since the runtime is composed of two parts. The first part is $O\left(\frac{N}{\sqrt{r}}\log\frac{N}{\sqrt{r}}\right)$ from step (3). Larger r leaves less vertices in clustered graph G', thus the recursive division runs faster on G' resulting in shorter runtime. The second part is $O(N\log r)$ from step (5). Larger r gives more vertices in each expanded region, so the recursive division runs slower on expanded regions. Though setting $r = (\log N)^2$ on a graph with more than 500 vertices guarantee the algorithm to run in $O(N\log\log N)$ time, the speed in practice is unknow and thus studied here.



(a) Recursive Division (b) Clustered Division

Figure 14. R-division algorithm runtime for different R value

From Figure 14, we can see that, as expected, the runtime for recursive division (RD) decreases with the increase of r value. The runtime for clustered division (CD) is smaller than RD's $O(N\log N)$ when r value is in some proper range. This range varies for different types of graphs but mostly lies in [200, 4000]. When r value is very small, the runtime for CD increase rapidly, mostly because the overhead of contraction (step (1) and (2)) and expansion (step (4)) is not helping in saving runtime. When r value is very large, the clustered graph G' may have less than r vertices thus the RD does nothing on G' and simply returns 1 region. Then the runtime is governed by step (5) and is about the overhead plus the runtime of RD.

7. Single Source Shortest Path

7.1. Dijkstra Baseline

Classical Dijkstra's algorithm for SSSP is implemented as a baseline for comparison. The algorithm is implemented using priority queue to store the vertices to be relaxed, resulting in $O(E \log V)$ time complexity.

7.2. Regional Speculative Dijkstra

This algorithm is described by Klein [12] and is named to be Regional Speculative Dijkstra (RSD) due to the key idea in the algorithm. At high-level, RSD uses r-division to partition the graph into small regions and maintains priority queues for each of the regions. It then performs relaxation similar to Dijkstra's algorithm on regions speculatively but moves between the queues of different regions. Since many edges need to be relaxed several times, RSD has a limited "attention span": it chooses a region, performs a number of relaxation steps then abandons that region until later and skips around between regions. The runtime improvement mainly comes from the faster queue operation for small regions due to less elements.

RSD first call r-division with a r value, which is a tunable parameter studied in the following section. A 3-level decomposition tree is constructed, whose root is the entire graph and leaves are atomic regions consisting a single edge. It initializes the distance of source vertex s to be 0 and all other vertices to have infinite distance and key. For each outgoing edge (s, w), it activates all regions R that contain (s, w) and their parents by updating key(sw) = 0 in Queue(R). Process the root region while the min-key of its queue is not infinite (there exist an active child region). The detailed pseudo code is given in [12] hence not repeated here.

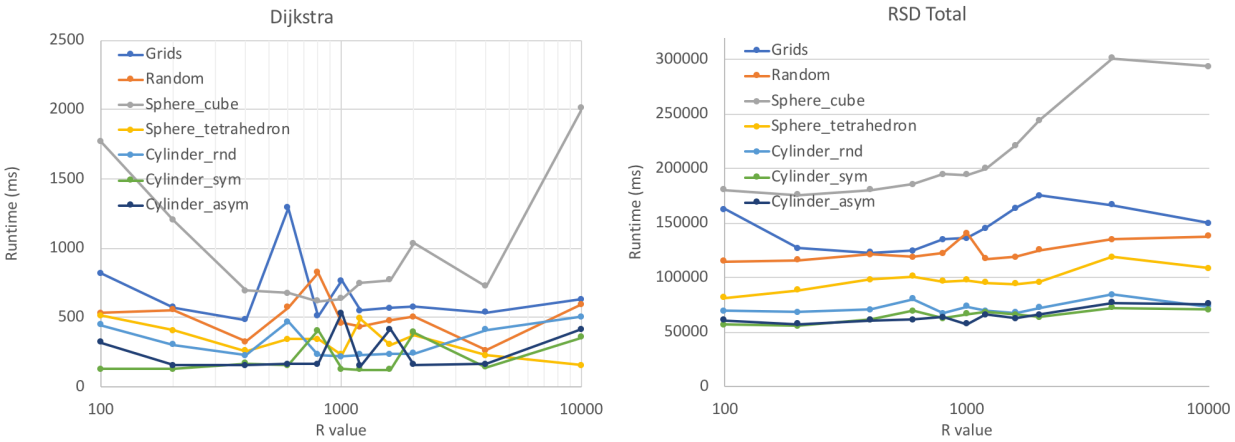
7.3. Result and Interpretation

The number of vertices in different testing graphs varies from 300k to 600k. The R value is a parameter to be studied in these experiments. Since in section 6, we have concluded that the proper range of r value is [200, 4000] for the clustered division to work efficiently, we set the study range to be [100, 10000] in this section. As shown in Figure 15, the total runtime of RSD is about 150 times of Dijkstra's algorithm in practice, even though RSD has better theoretical bound. There are 3 major reasons for this:

(1) The R-division step is taking 90% of the runtime in RSD. From section 6, we have found out that the R-division for graphs of roughly 500k vertices takes 50~150 seconds. Therefore, the RSD, which uses R-division as a subroutine, must pay at least the same amount of time cost as R-division.

(2) RSD itself, without r-division time, is slower than Dijkstra due to large constant factor. As shown in Figure 16, just the core part of RSD algorithm is still 10~15 times of Dijkstra's algorithm. On one hand, in RSD, each region may be activated more than once, since the initial visit of a vertex may not be from an optimum path. Even with theoretical guarantee that the total number of visits will not be large for a vertex, the constant may still be significantly larger than 1. On the other hand, each vertex is only pulled from the priority queue once in Dijkstra's algorithm. Moreover, not all vertices stay in the priority queues all the time, only those encountered by the search frontier, resulting a much smaller queue in practice.

(3) The number of vertices need to be at least 600 million for RSD to outperform Dijkstra's algorithm. The RSD algorithm requires $r = (\log N)^4$ in r-division. As we have concluded in section 6, for clustered division to achieve $O(N \log\log N)$ runtime and to outperform recursive division, the number of vertices in graph is at least $r^{1.5}$. As a result, we have $N \geq (\log N)^6$ and N is about 600 million. Within the limitation of this project, the largest graph tested has 600k vertices, hence I did not observe the RSD outperforming Dijkstra's algorithm. For practical purpose, graphs of 600 million vertices are very rare.



(a) Dijkstra' algorithm (b) RSD

Figure 15. Runtime comparison for Dijkstra algorithm and total RSD

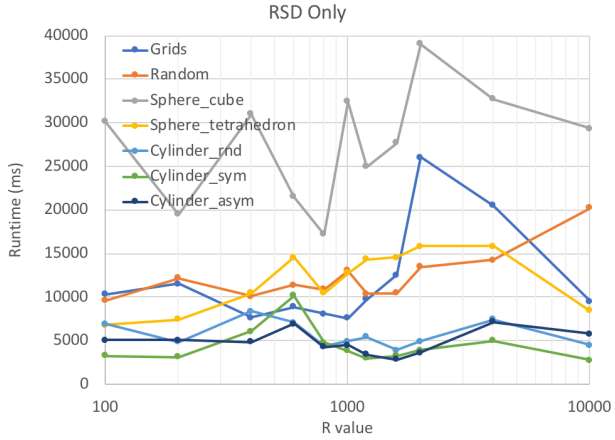


Figure 16. Runtime for core RSD

8. Future work

1. More types of testing graphs can be used in performance evaluation. One type is to duplicate an existing planar graph and connect both parts with a few added vertices. This type of twin graphs has an explicit know minimum size separator. The other type is to use real world road maps as planar graphs, which may need some preprocessing to ensure the connectivity and planarity.
2. Lipton-Tarjan implementation can be optimized. The current algorithm returns the union of entire level L_a, L_z and the cycle. It is possible to only select the vertices that are effective separators. For example, if the central orange region in Figure 7 is one subgraph and the rest forms the other subgraph, the separator only needs to include the central part of level L_a, L_z and the cycle. This potential improvement will not change the theoretical size bound of Lipton-Tarjan separator, but may be helpful in practice, without increase the time complexity.
3. More advanced planar graph algorithms are to be implemented. There is a linear time r-division algorithm [12] that constructs a decomposition tree, which can then be used in linear time SSSP algorithm. Another way of solving SSSP is to use an $O(N \log N)$ multi-source shortest path (MSSP) algorithm [11] as subroutine, which is also worth exploring. The implementation of linear SSSP and MSSP might take a while since there are a lot advanced data structures needed. The final step is to use all these as tools to implement the $O(N \log N)$ max-flow algorithm [13].
4. Due to the limitation of graph size in this project, the maximum number of vertices in a test graph is 3 million, which may not be big enough to tell the difference between $O(N \log\log N)$ algorithms, like recursive division and RSD, and $O(N \log N)$ ones like Dijkstra's algorithm. It remains a question whether the $O(N \log\log N)$ RSD algorithm can outperform Dijkstra's algorithm when the graphs are big enough. This is also a question for other linear time SSSP algorithms, since the constant factor may be large.

9. Conclusions

Runtime of all 5 planar separator algorithms (Level separator, FCS, MFCS, Lipton-Tarjan separator and SCS) are linear with respect to the number of vertices in the planar graphs. The size of separators found by FCS grows linearly when the graph has large diameter. The size of level separators grow linearly in sphere type graphs. Lipton-Tarjan separator and SCS yield separators of much more stable sizes, which is consistent with the theoretical guaranteed worst-case $O(\sqrt{N})$ bound.

Clustered division outperforms recursive division when the region size r is in a proper range. Though the theoretical lower limit of R is $O(\log^2 N)$, in practice, it may require experiments to determine the proper range of R value for different type of graphs.

For practical purpose, Dijkstra's algorithm is more efficient on planar graphs even though its theoretical time bound is worse than the $O(N \log\log N)$ algorithm by Klein [12]. It worth exploring if this is still truth when the number of vertices in the graph is large enough. The same question holds for other $O(N \log\log N)$ or $O(N)$ SSSP algorithms and advanced algorithms that uses them.

All source codes in this project is saved in public repository on GitHub [18].

10. References

[1] https://en.wikipedia.org/wiki/Planar_graph
[2] Jeff's notes
[3] https://en.wikipedia.org/wiki/Dual_graph
[4] https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm
[5] Lipton, Tarjan, A separator theorem for planar graphs, SIAM Journal on Applied Mathematics, 36 (2): 177–189, 1997
[6] Djidjev, On the problem of partitioning planar graphs, SIAM Journal on Algebraic and Discrete Methods, 3 (2): 229–240, 1982.
[7] Miller, Finding small simple cycle separators for 2-connected planar graphs, Journal of Computer and System Sciences, 32 (3): 265–279, 1986
[8] Djidjev, Venkatesan, Reduced constants for simple cycle graph separation, Acta Informatica, 34 (3): 231–243, 1997.
[9] Frederickson, Fast algorithms for shortest paths in planar graphs, with applications, SIAM J. Computing, 1004-1022, 1987.
[10] Goodrich, Planar separators and parallel polygon triangulation, J. Comput. System Sci. 51, 374–389, 1995.
[11] Klein, Multiple-source shortest paths in planar graphs, Proceedings of 16th ACM-SIAM Symposium on Discrete Algorithms, 146-155, 2005
[12] Klein, Rao, Rauch, Subramanian, Faster shortest-path algorithms for planar graphs, Journal of Computer and System Sciences, 55 (1): 3–23, 1997
[13] Erickson. Maximum Flows and Parametric Shortest Paths in Planar Graphs. Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms, 794-804, 2010.
[14] Martin Holzer, Frank Schulz, Dorothea Wagner, Grigorios Prasinos, and Christos D. Zaroliagis. Engineering planar separator algorithms. ACM Journal of Experimental Algorithmics 14 (2009), 5:1.5–5:1.31
[15] Eli Fox-Epstein, Shay Mozes, Phitchaya Mangpo Phothilimthana, and Christian Sommer. 2016. Short and simple cycle separators in planar graphs. J. Exp. Algorithmics 21, 2, Article 2.2
[16] Sariel Har-Peled, Amir Nayyeri. A simple algorithm for computing a cycle separator, arXiv:1709.08122
[17] https://www.spacetelescope.org/images/heic1509a/
[18] https://github.com/StarkZhu/PlanarGraphAlgo