ORB ACADEMIC PUBLISHER

Int. J. Comput. Sci. Theor. App., 2016, vol. 5, no. 1., p. 1-11.
Available online at www.orb-academic.org

International Journal of Computer Science: Theory and Application
ISSN: 2336-0984

Linear Time Vertex Partitioning on Massive Graphs

Peter MELL[1], Richard HARANG[2], and Assane GUEYE[3]

[1] *National Institute of Standards and Technology, USA*
[2] *Army Research Laboratory, USA*
[3] *University of Maryland, USA*
Email: peter.mell@nist.gov, richard.e.harang.civ@mail.mil, agueye@umd.edu

ABSTRACT
The problem of optimally removing a set of vertices from a graph to minimize the size of the largest resultant component is known to be NP-complete. Prior work has provided near optimal heuristics with a high time complexity that function on up to hundreds of nodes and less optimal but faster techniques that function on up to thousands of nodes. In this work, we analyze how to perform vertex partitioning on massive graphs of tens of millions of nodes. We use a previously known and very simple heuristic technique: iteratively removing the node of largest degree and all of its edges. This approach has an apparent quadratic complexity since, upon removal of a node and adjoining set of edges, the node degree calculations must be updated prior to choosing the next node. However, we describe a linear time complexity solution using an array whose indices map to node degree and whose values are hash tables indicating the presence or absence of a node at that degree value. This approach also has a linear growth with respect to memory usage which is surprising since we lowered the time complexity from quadratic to linear. We empirically demonstrate linear scalability and linear memory usage on random graphs of up to 15000 nodes. We then demonstrate tractability on massive graphs through execution on a graph with 34 million nodes representing Internet wide router connectivity.

KEYWORDS
Vertex partitioning; graph cuts.

1. Introduction

The graph separator problem attempts to bisect a graph G into components containing approximately equal numbers of nodes by removing no more than k edges or nodes. This problem is NP-complete [1] [2]. In this work, we focus on a more general variant where we remove up to k vertices and attempt to minimize the size of the largest component. More formally, consider a graph G where we remove up to k vertices and let S be the set of removed nodes. We can then represent our optimization problem as follows:

$$\min_{S \subseteq V, |S| \leq k} (\max\{|C|;\ C \text{ is a cluster of } G \backslash S\}) \quad (1)$$

Since this can be used to solve the graph separator problem, it is NP-hard and it follows then that any resulting solutions will be approximate heuristics (or perform exhaustive exponential complexity searches).

Prior work has provided near optimal heuristics with a high time complexity that function up to hundreds of nodes and less optimal but faster techniques that function on up to thousands of nodes. In this work, we analyze how to perform vertex partitioning on massive graphs of tens of millions of nodes. We use a previously known and very simple heuristic technique: iteratively removing the node of largest degree and all of its edges. This approach has an apparent quadratic complexity since, upon removal of a node and adjoining set of edges, the node degree calculations must be updated prior to choosing the next node. However, we describe a linear time complexity solution using an array with embedded hash tables where the array indices map to node degree. This approach also has a linear growth with respect to memory usage which is surprising since we lowered the time complexity from quadratic to linear. We empirically demonstrate linear scalability and memory usage on random graphs of up to 15000 nodes. We then demonstrate tractability on massive graphs through execution on a graph with 34 million nodes representing Internet wide router connectivity.

Our motivation comes from applications of computer security, although the approach is generally applicable to a variety of domains. For example, a graph may represent a computer network (either physical or logical) and the removed nodes could be those hardened in order to create internal defensive barriers. Alternately, the removed nodes could represent targets of an attacker in calculations to split a network up into components. We have found such network graphs to typically have many

nodes of small degree and few nodes of high degree. These characteristics aid many of the heuristic approaches, including ours. The approach studied in this paper (which iteratively removes nodes with highest degree) has previously been proposed by Albert et. al. [3] and Reuven et. al. [4] as a method to assess the vulnerability of networks to targeted attacks and by Madar et. al. [5] as a strategy for choosing nodes to immunize in a network. This latter approach was also used by Tong et. al. [6] although a different heuristic was employed for vertex identification.

The rest of the work is organized as follows. Section 2 discusses our heuristic in more detail and present several naïve algorithms. Section 3 provides our linear time vertex partitioning algorithm. Section 4 walks through an example execution. Section 5 provides an empirical analysis, section 6 discusses prior work, and section 7 concludes.

2. Methods and Materials

The heuristic we are using to perform vertex partitioning is to iteratively extract a highest degree vertex from a graph, G. We denote G as having n vertices and m non-loop edges with the degree of each node being calculated ignoring any self-loops (since self-loops have no bearing on the connectivity of G). We iteratively remove nodes until some set number, k, of vertices are extracted with $k \leq m/2$ (were $m/2$ highest degree nodes removed, all remaining nodes would have degree 0). After extracting a vertex, we recalculate any affected node degrees (decrementing each of them). The rationale behind the approach is that removal of a highest degree node will remove the most number of edges and thus intuitively will have a better chance of disconnecting a graph compared to removing a lower degree node with fewer edges.

An obvious implementation of this heuristic is as follows. Traverse all nodes and record the node of highest degree. Remove the chosen node from the graph and repeat the procedure. This approach is simple and requires no data structure outside of G itself, however it executes in quadratic $O(n^2+m)$ time and hence is not competitive for large graphs of millions of nodes. Note that its memory usage is linear, $O(n+m)$. For the rest of this paper we will refer to this approach as the Quadratic Scan (QS) algorithm.

Given the quadratic time complexity of QS, for large graphs one might just use the initial degree values to iteratively extract nodes (without ever updating the degree of any nodes as a result of edges being removed), resulting in an $O(nlogn)$ time complexity and an $O(m)$ memory complexity solution. The algorithm would be linear except for a single operation to sort the nodes by degree. Empirically, this is a very fast operation and thus the approach operates as if it were linear even on graphs of millions of nodes. However, we show empirically that this simplified approach often produces less optimal solutions (depending upon the type of graph under analysis). For the rest of the paper, we will refer to this simplified approach as the Static Degree (SD) algorithm.

3. Linear Time and Memory Algorithm

While apparently quadratic, our vertex partitioning heuristic can be implemented as a linear algorithm in both time and memory (as we have proposed in this paper). The memory consumption is primarily in the form of $O(d)$ hash tables where d represents the maximum degree in the graph (again ignoring self-loops). The overall memory usage is thus bounded by $O(n+m+d)$. Note that, empirically, significantly few hasher tables are often used. To achieve the linear runtime, we must find the highest degree node, remove it, and update the degree values of its neighbors in constant time. Thus, for the rest of the paper we refer to our approach as the Constant Update (CU) algorithm. Note that while the processing of a removed node can be considered constant, any particular removal may include accessing up to n nodes. However, when all such updates are amortized over the life of the full algorithm we find that this operation adds at most an $O(m)$ cost to the entire algorithm. We can then vertex partition the graph using any number of nodes in $O(n+m)$ linear time.

3.1 Narrative Description

To implement our approach, we create an array of hash tables where the array indices refer to node degree cardinality. More specifically, we create an array L where the indices of L map to degree values. Thus, $L[5]$ maps to the set of nodes of degree 5. Since no node can have a degree greater than $n-1$ (we ignore self-loops), the length of L is $n-1$. At a particular array index (e.g., $L[23]$), we store the names of all the nodes with that degree using a hash table. Thus, we have one hash table per active node index (where we define an 'active' index to be one where the index maps to at least one node of that degree). Inactive indices in L are not given a hash table and are simply assigned a default value of 0 (optimizing the memory usage). To enter a node into a hash table, we use the node name as the key and use a default Boolean 'True' for the value. Thus, the purpose of the hash table is not to look up values for keys (since the values are always just 'True') but instead to enable one to find a node entry in constant time if one already knows the node's degree.

From a given graph G, we can create L in linear time. We first create an array L of size $n-1$ and set the value of all the fields to 0. We then traverse the set of nodes in G. At each node x, we extract the degree value, d_x, and use that as the appropriate index in L. If the value of $L[d_x]$ is 0, we replace it with an empty hash table. We then enter the node name into the hash table in constant time.

We next traverse L sequentially from the highest index to the lowest. At each index i, we check to see if a hash table exists. If one exists, then we extract the set of keys in constant time. These keys will be the names of the highest degree nodes. We arbitrary choose one node, x. We remove it from G and then update L to reflect the new state of G in constant time (this procedure is described below). We iteratively continue this process of identifying nodes to remove until we have removed k nodes, at which point the algorithm terminates. Note that

between node removals, the set of keys designating the nodes of highest degree must be regenerated since the set of nodes of highest degree can be altered by the removal of a node.

The last thing that remains is to show how to update L in constant time to reflect the removal of a node x from G. For each neighbor of x with degree d, we need to remove it from the hash table at $L[d]$ and add it to the hash table at $L[d-1]$. If a hash table does not exist at $L[d-1]$ we first create an empty one. If removal of the neighbor from the $L[d]$ hash table empties that table, we delete it and replace it with a 0 (i.e., in this case $L[d]$=0 which signifies that no nodes of degree d currently exist in G). Lastly, we remove the node x from G and any associated edges. A single removal of a node x from G then appears to take $O(n)$ time because a node may have up to $n-1$ neighbors. However, the number of neighbors over the life of the algorithm cannot exceed the number of edges, m. Thus, this contributes an additive $O(m)$ term to the overall complexity and the updates for a set of neighbors for any particular x can be considered a constant.

Reviewing the entire execution cost, initial creation of L takes $O(n)$ time. Traversing L from the highest index to lowest index while identifying nodes of highest degree takes $O(n)$. Removing a node from G while updating L to reflect the new state of G takes constant time with respect to evaluating n but adds an amortized cost of $O(m)$ to the entire algorithm. This makes the runtime linear, bounded by $O(n+m)$.

3.2 Pseudo-code Presentation

We now provide pseudo-code (heavily based on Python) to specify the algorithm's execution. For space considerations and to simplify the presentation, we omit the code that optimizes memory usage (i.e., when to create and delete hash tables). Thus, this pseudo-code creates one hash table per index in L for the life of the algorithm. Our actual Python code only creates hash tables for indices that correspond to the actual degree values of nodes and then it dynamically creates and deletes hash tables as the algorithm removes nodes and adjusts node degree values.

The pseudo-code is provided in Figure 1. The input parameters G and k represent the initial graph and the desired number of removed nodes respectively. L is our primary data structure as described in section 4.1 that is an array with embedded hash tables. S is a list of the names of the removed nodes. We use [ ] to denote the creation of an empty array and { } to denote the creation of an empty hash table. 'current' is the monotonically decrementing variable denoting the array index in L that we are currently processing.

3.3 Example Linear Execution

We now trace through the execution of our linear algorithm on a small toy graph using a k value of 3. In tracing through our walk through, keep in mind how the algorithm allows each node removal step (finding node of highest degree, removing it and associated edges, and updating the data structure to reflect the new graph) to be done in constant time (noting that the full algorithm amortized analysis adds an additive $O(m)$ factor).

```
def Linear_Time_Vertex_Partitioning(G,k):
1   L=[]
2   S=[]
3   for each node in G: L.append({})
4   for each node in G:
5       deg=degree(G.node)
6       L[deg][node]=True
7   current=G.number_of_nodes()-1
8   while len(S)<k:
9       while len(L[current].keys())==0: current-=1
10      keys=L[current].keys()
11      while len(keys)>0:
12          choice=keys[0]
13          S.append(choice)
14          neighbors=G.neighbors(choice)
15          for neigh in neighbors:
16              ndegree=degree(G,neigh)
17              delete(L[ndegree][neigh])
18              L[ndegree-1][neigh]=True
19          G.remove_node(choice)
20          del(L[current][choice])
21          if len(S)==k: break out of while loop
22          keys=L[current].keys()
23  return S
```

Figure 1. *Python Based Pseudo-code for Linear Time Vertex Partitioning Algorithm.*

Figure 2 shows the initial graph and our construction of the starting data structure (the array L with embedded hash tables). The check mark indicates which degree index in L is being evaluated next to choose a node to remove. The column of numbers from 0 to 4 are the array indices of L. The letters followed by the '$->$' sign pointing to the value 'True' represent the hash tables embedded in L (at most one per array index).

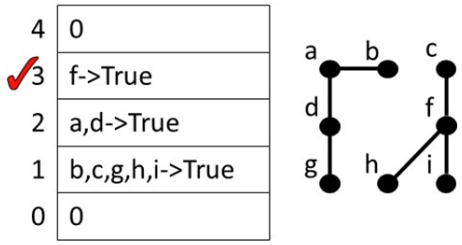

Figure 2. *Initial toy graph and data structure construction.*

Based on Figure 2, our algorithm chooses node e to remove first. The resulting graph and updated L are shown in Figure 3. Note how the removal of e caused c, d, g, and h to be decremented to a lower index in L.

Based on Figure 3, our algorithm next chooses node f to remove because no other nodes with degree 3 exist (and this is the degree index we are currently processing). This action decrements c, h, and i and results in the updated graph and array L as shown in Figure 4. Finally, our algorithm evaluates index 2 in L and finds two nodes of that degree. For the final node removal it arbitrarily chooses one (say d) causing the final state as shown in Figure 5.

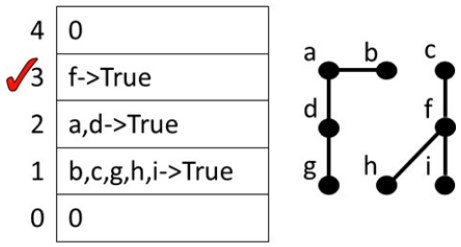

Figure 3. *Updated graph and data structure with e removed.*

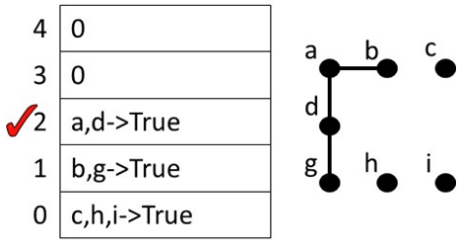

Figure 4. *Updated graph and data structure with e and f removed.*

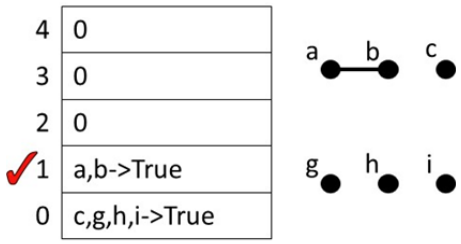

Figure 5. *Final graph and data structure state with d, e, and f removed.*

Not demonstrated in this example is how the algorithm will skip over an array index if there is no associated hash table. Also not shown is how the algorithm may process a particular index multiple times if there are several candidate nodes for removal of the same degree. After each node removal at a particular degree, some of the candidate nodes may drop off the candidate list if they have an edge to one of the previously removed nodes.

4. Results and Discussion

In the following sections, we examine our proposed approach from several angles. For this, we evaluate all three algorithmic approaches: our linear time constant update (CU) approach, the naïve quadratic scan (QS) implementation, and the O(nlogn) time static degree (SD) method. We also include a betweenness algorithm from [7] (henceforth, the 'BT' algorithm) that is highly effective, however, we show that its runtime increases at a large quadratic rate.

First, we examine the effectiveness of our partitioning scheme on a number of random graph models, examining graphs of up to 15000 nodes (scaling both the graph size and vertex partitioning cutset size) using both our CU method and the related SD method. We show that the CU method reliably produces better partitions. We next verify the improvement in speed of our linear time implementation of CU over the naïve QS method (which produces identical partitions). We also include the SD, QS, and betweeness algorithms for comparison. We follow this with an analysis of CU, SD, and QS memory usage on graphs up to 1 million nodes. Finally, we apply our method to an extremely large real-world graph of 34 million nodes representing Internet wide router connectivity. We show that the presence of high number of leaf nodes can damage the performance of our CU approach relative to the SD method, however a simple modification to the CU approach, which ignores leaf nodes, restores the advantage of the CU approach while also improving the overall quality of the cut.

4.1 Algorithm Partitioning Effectiveness

Here we examine the quality of the cuts produced by the CU algorithm over a range of graph sizes and types, also comparing to those produced by the SD algorithm. We show that, across several classes of random graphs, for a range of graph sizes and separator set sizes, the CU algorithm reliably outperforms the SD algorithm, in many cases by a substantial margin, at a relatively modest cost in time. Note that graphs of the size we consider here may be tractable with other existing tools but we are focusing here only on approaches that operate empirically at linear time complexity (i.e., those that massively scale).

We considered the following classes of random graphs:

- Barabasi-Albert preferential attachment model with $m = 1$ (tree) and $m = 3$.
- Erdos-Renyi graph with link probability $\frac{(1+0.05)logn}{n}$, such as to produce connected graphs with high probability.
- Newman-Watts-Strogatz small-world graph with 2 neighbors and link probability 0.3

All experiments were conducted by drawing 50 random graphs of the specified size and type (using the NetworkX library [7]), applying the CU and SD methods to each of them, and recording the timing results as well as the size of the largest connected component. For each graph type, we show the size of the largest connected components for a range of graph sizes when 25% of the nodes are removed. Also for each graph type, we show a graph of 15000 nodes when from 1% to 100% of the nodes are removed. The points indicate the mean across all samples.

Figure 6 and Figure 7 show the results for random trees (Barabasi-Albert preferential attachment model with $m = 1$). In Figure 6, as we increase the cut size for a tree of 15000 nodes, the CU algorithm provides smaller largest-connected-component sizes. When the cutset size exceeds 4500 nodes, the results of the two algorithms converge. In Figure 7, as the graph size is increased while removing 25% of the nodes, the CU algorithm shows a consistent advantage.

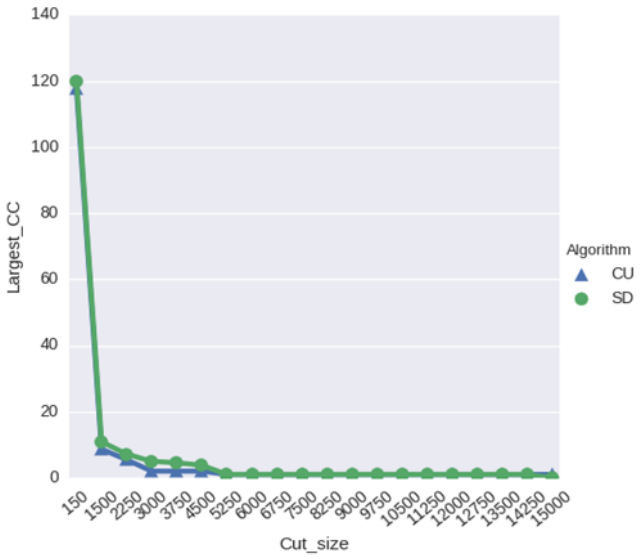

Figure 6. *Largest Connected Components for Variable Cutset Sizes, Barabasi-Albert m=1 Graphs.*

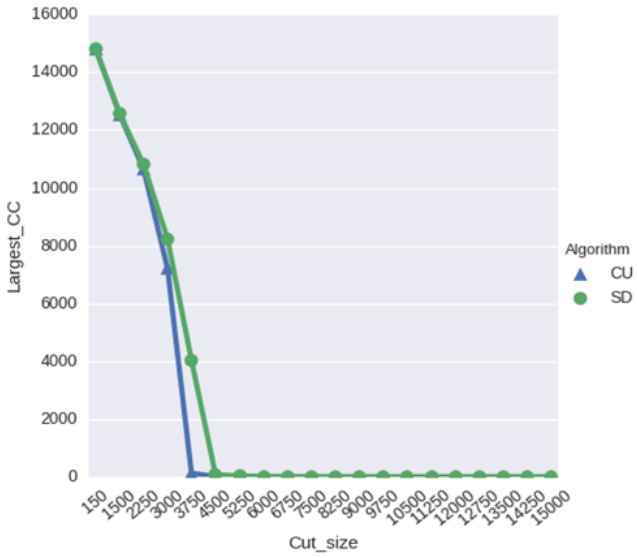

Figure 8. *Largest Connected Components for Variable Cutset Sizes, Barabasi-Albert m=3 Graphs.*

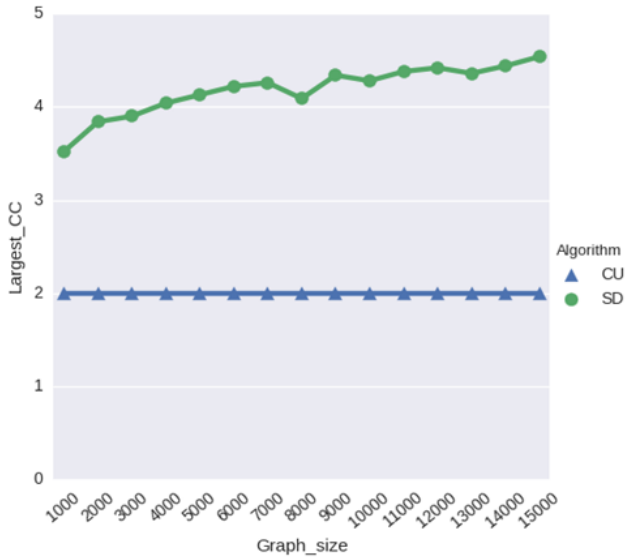

Figure 7. *Largest Connected Components for Variable Graph Sizes, Barabasi-Albert m=1 Graphs.*

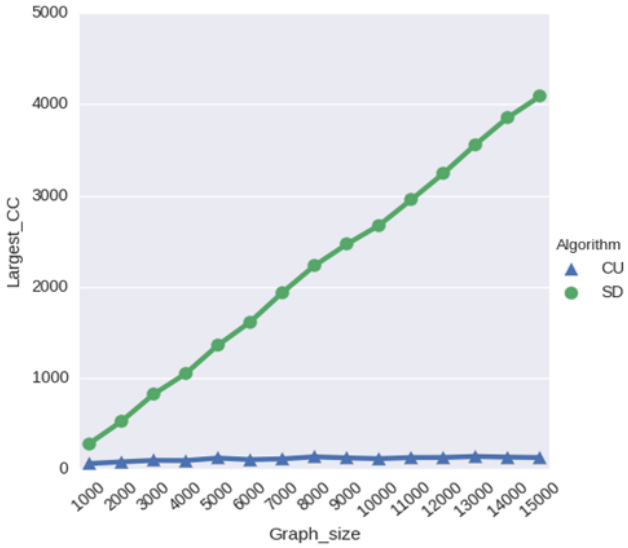

Figure 9. *Largest Connected Components for Variable Graph Sizes, Barabasi-Albert m=3 Graphs.*

Figure 8 and Figure 9 show the results for random scale free graphs (Barabasi-Albert preferential attachment model with $m = 3$). In Figure 8, we see similar results as with the random trees with the CU algorithm having an advantage at smaller cutset sizes and with the results of the two heuristics converging at higher cutset sizes (above 8250). In Figure 9, we see the CU method performing extremely well regardless of graph size while the SD method has increasingly worse performance as the graph size increases.

Figure 10 and Figure 11 show the results for random Erdos-Renyi graphs. In Figure 10, we see both heuristics performing similarly at both small and large cutset sizes with the CU approach showing a distinct advantage for large swath of mid-range cutsizes (cutsizes from 3000 to 14000). The difference between the two algorithms is greatest between cutsizes of 7000 to 12000 nodes. In Figure 11, we see the CU and SD algorithms demonstrating almost identical performance. This is because the cutset size of 25% was just within the region where the algorithms performed similarly. If the cutset size percentage is increased, the CU algorithm will outperform the SD algorithm

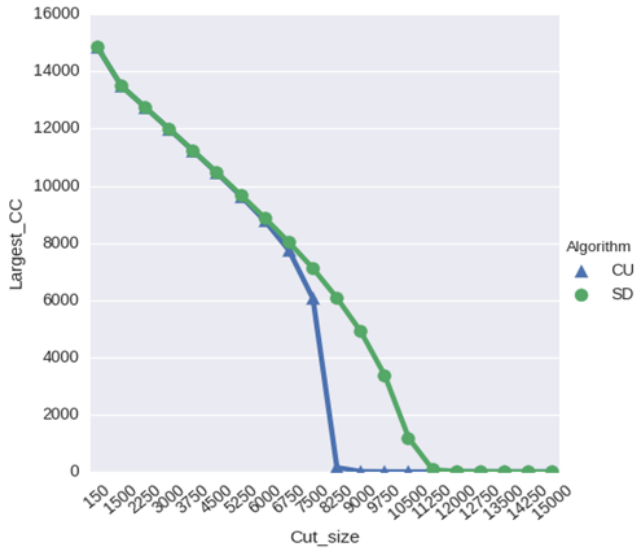

Figure 10. *Largest Connected Components for Variable Cutset Sizes, Erdos-Renyi Graphs.*

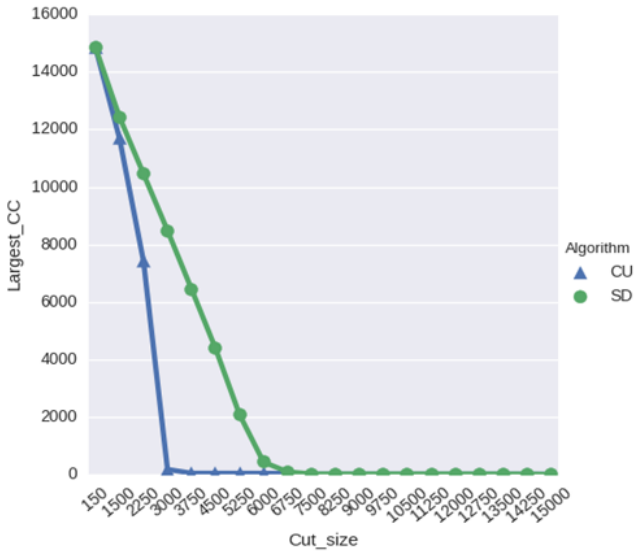

Figure 12. *Largest Connected Components for Variable Cutset Sizes, Newman-Watts-Strogatz Graphs.*

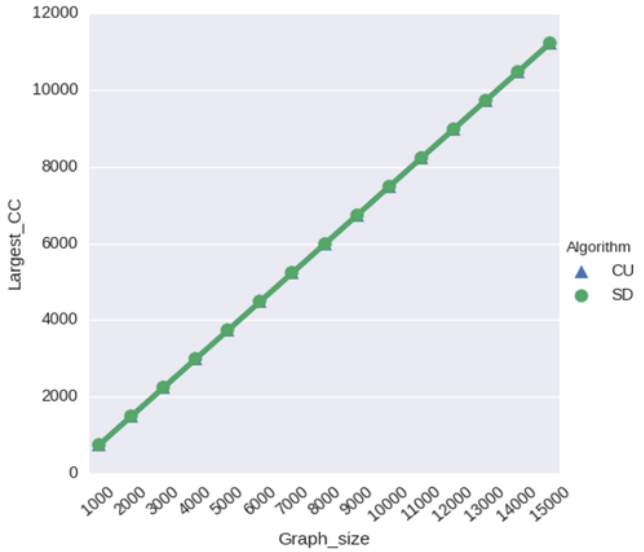

Figure 11. *Largest Connected Components for Variable Graph Sizes, Erdos Renyi Graphs.*

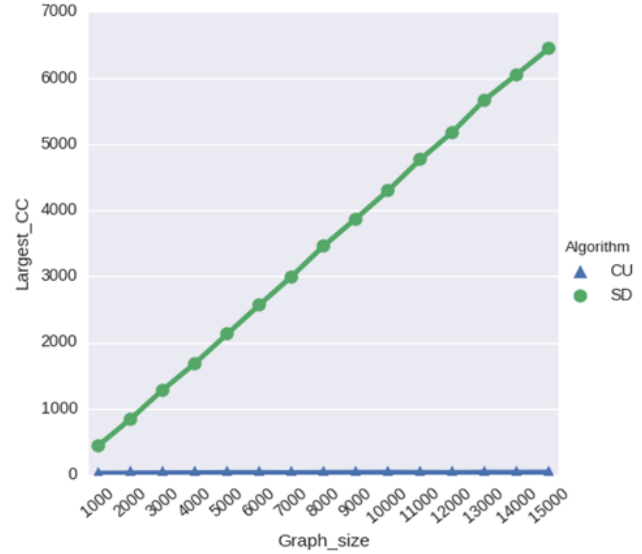

Figure 13. *Largest Connected Components for Variable Graph Sizes, Newman-Watts-Strogatz Graphs.*

(not shown). Note that the equal performance of both algorithms here shows that use of the CU approach does not guarantee better results but, in our experiments on synthetic data, it always performed at least as well as the SD approach.

Figure 12 and Figure 13 show the results for random Newman-Watts-Strogatz graphs. The results here were very similar to those of the Barabasi-Albert ($m = 3$) graphs. However, in Figure 12 the advantage of the CU approach at lower cutsizes is more pronounced. Likewise, in Figure 13 the disadvantage of the SD approach is also more pronounced.

In summary, in the case of small world and power-law type graphs, the CU method produces a separator that is substantially better than the SD method, while in the case of the Erdos-Renyi graph the advantage is less pronounced (but it depends on the cutset size used).

4.2 Execution Time Growth Rates

In this section we examine the growth rate of the execution time for SD, CU, QS, and BT algorithms. Of particular interest is the improvement from the QS algorithm to our CU algorithm

(which produce the exact same answers). SD and BT were included for comparative purposes.

To test execution time, we performed two experiments (shown in Figure 14 and Figure 15). Figure 14 examines the Barabasi-Albert graphs with m=3 from 1000 to 15000 nodes, finding the best separator using 25% of the nodes. Timing results for the other graph types were substantially similar (not shown). Figure 15 performs the same analysis, but on a much larger scale for just the linear time complexity algorithms; it covers graphs from 100000 to 1000000 nodes.

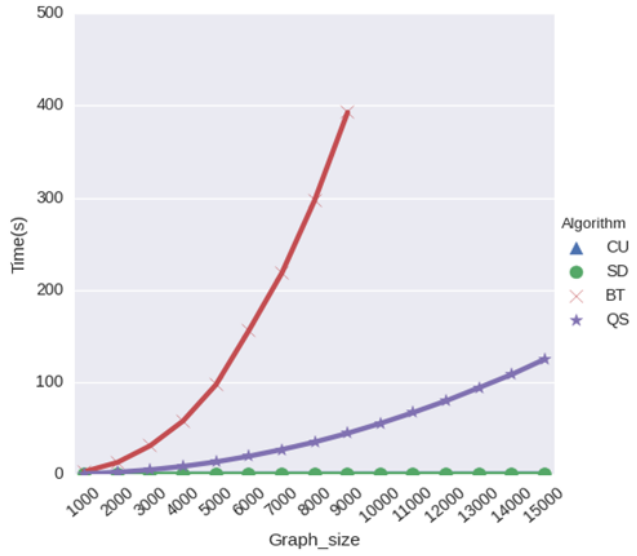

Figure 14. *Execution Times for Variable Graph Sizes, Barabasi-Albert m=3 Graphs.*

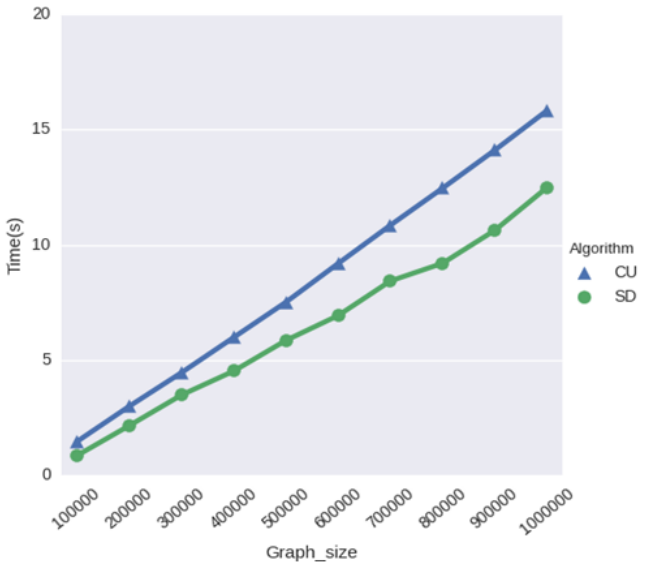

Figure 15. *Linear Complexity Algorithm Execution Times for Variable Graph Sizes, Very Large Barabasi-Albert m=3 Graphs.*

Figure 14 clearly shows the quadratic nature of both the QS and BT algorithms. The BT algorithm took 393 seconds for just 9000 nodes while the QS algorithm took 125 seconds for just 15000 nodes. The CU and SD algorithms appear almost constant here, taking less than .2 seconds for all data points. Note that the CU data line hugs the x-axis, underneath the SD data line, and is not visible.

Figure 15 enables us to better see the execution time growth rate of the CU and SD algorithms. As expected from the prior theoretical analysis, the data appears linear with the CU algorithm using more time than the SD algorithm by a constant factor. Note that the quadratic execution time of the QS and BT algorithms prohibited us from executing them on these larger graph. We tried executing the BT algorithm with sampling pairs of nodes on which to calculate betweenness in order to improve the runtime, but this doesn't eliminate the quadratic nature of the algorithm and causes a reduction in effectiveness (not shown).

4.3 Memory Growth Rates

We now analyze the memory usage of the two algorithms that scale to large graphs: CU and SD. For this, we execute both algorithms on graphs from 100000 to 1000000 nodes using the Barabasi-Albert $m = 1$, Barabasi-Albert $m = 3$, and the Watts-Strogatz graphs. We didn't include the Erdos-Renyi graphs because, for graphs of this size, the random graph generator required excessive resources.

Figure 16 provides the results, showing empirically the expected linear memory growth. Surprisingly, for all graph types the CU algorithm uses almost the same amount of memory as the SD approach. This means that the data structures used by CU are empirically very efficient resulting in the majority of the memory usage simply being the graph itself.

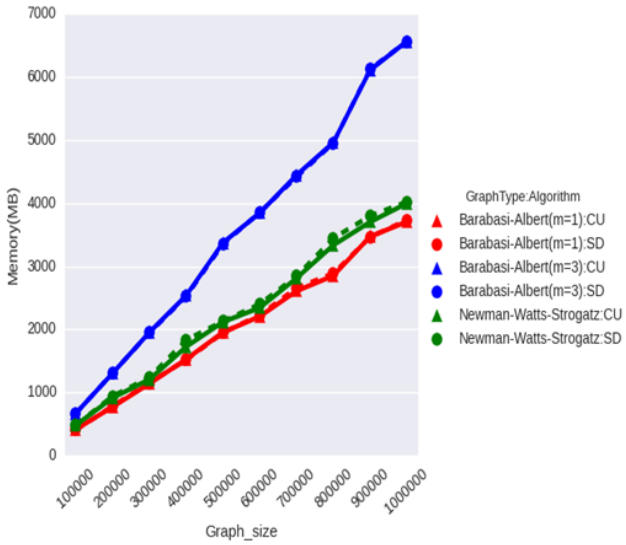

Figure 16. *Memory Usage for Linear Time Complexity Algorithms.*

4.4 Execution on Massive Graphs

To evaluate our approach on massive graphs, we obtained router interconnectivity data from the Center for Applied Internet Data Analysis (CAIDA) consortium [8] for 2007, and constructed an undirected graph depicting the approximate routing structure of the internet. The resulting graph had 34386931 nodes and 35799372 edges. The primary purpose of this experiment is to check the runtime of the CU and SD algorithms at scale. We also report on partitioning effectiveness.

This graph has the unusual feature of having a large number of very large star subgraphs with a majority of the nodes being leaves (both within each star subgraph and for the graph at large). This may explain why the SD algorithm outperforms the CU algorithm here when identifying large cutsets (e.g., 160000 nodes). The stars are problematic for the CU algorithm because, instead of choosing nodes that will break apart the infrastructure of the graph, it is biased towards choosing large stars. When a star subgraph is chosen and processed, this removes edges to important infrastructure nodes (the edges that connected the star to the rest of the graph). This removal of 'infrastructure' edges then biases the CU algorithm against choosing infrastructure nodes and towards choosing more star subgraphs. To test our theory that the star subgraphs limit CU performance with large cutsets, in this experiment we also use a variant on the CU algorithm that counts only non-leaf nodes when choosing the next node for the cutset. We call this variant LCU.

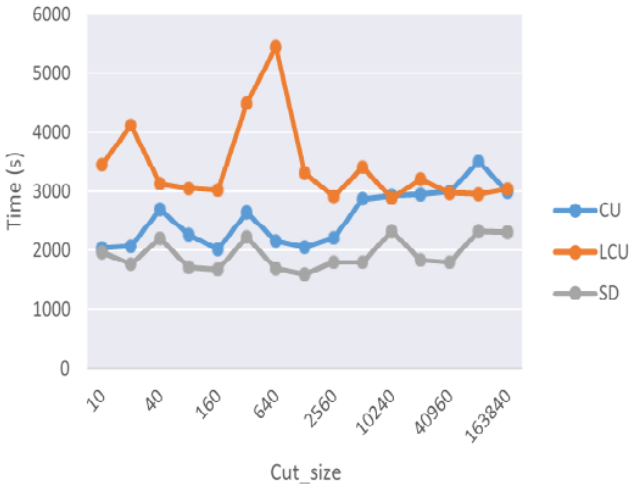

Figure 17. *Timing Results on CAIDA Graph.*

Figure 17 shows the results of our timing tests. The significant crests and troughs are a result of this being an experiment on a single large graph (each datapoint represents a single measurement and not a mean of many measurements). Despite the jitter and the single large outlier, one can see that regardless of the separator size, processing time is approximately constant within the two methods. This is due to the fact that traversing the graph and building the associated data structures to enable analysis consumes most of the processing time for both heuristics. Thus, varying the cutsize measured does not significantly change the execution time. In comparing the two heuristics, the SD method continues to appear to enjoy a slight and consistent advantage with respect to speed. From a theoretical point of view this is surprising, as the SD algorithm is $O(nlogn)$ and the CU algorithm is $O(n)$. Apparently, the *nlogn* cost associated with the SD algorithm sorting the degree values of the 34 million nodes is not sufficient to enable the CU algorithm to demonstrate a smaller execution time.

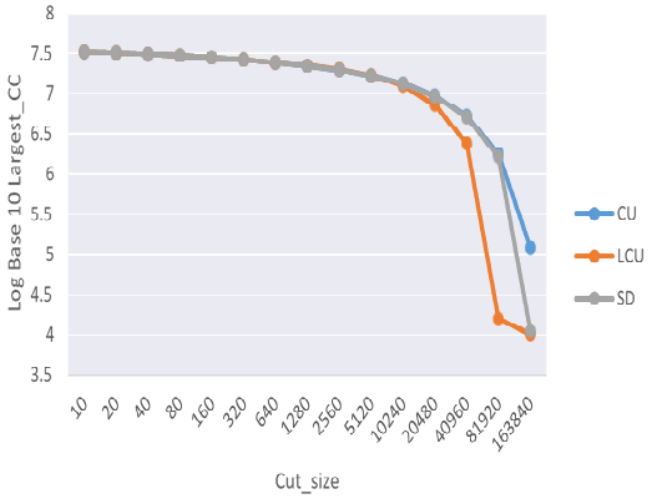

Figure 18. *Largest Connected Components for CAIDA Graph.*

Figure 18 shows the effectiveness of CU, SD, and LCU on the CAIDA Internet topology graph. The results are virtually identical up to a cut size of 10240 at which point the LCU algorithm begins to show a distinct advantage. However, the SD algorithm converges with the LCU algorithm at a cutsize of 163840. The CU algorithm lags in performs for high cutsizes, likely due to the prevalence of very large stars in the graph as discussed previously. This result is not surprising as most approximation algorithms work with variable effectiveness given the nature of the graph to be analyzed. No one heuristic will guarantee the best result on all input data.

5. Prior Work

The general problem of removing either edges or vertices to disconnect a graph (for different purposes) has largely been studied in the graph theory literature under different names: graph partitioning [9], vertex/edges separation [10], [11], [12], vertex/edge cut [13], graph bisection [14], p-separator problem [15], and p-way vertex cut [16]. A common observation is that the problem is NP-hard [1] [2] in general and most of the effort has been devoted to finding good heuristic algorithms.

Our formulation of the problem (find a subset of vertices of size at most k whose removal minimizes the size the maximum component in the remaining graph) can be related to the p-separator problem [15] which can be stated as follow: *find a*

minimum-size subset (i.e. separator) of vertices whose removal leads to a graph where the size of largest connected component is less than or equal to p. The p-separator problem has been shown to be NP-hard in general [2]. Ben-Ameur et al. [15] have shown that it can be solved in polynomial time for a certain class of graphs. They also presented approximation algorithms.

By adding to the p-separator problem the additional constraint that the size of the separator has to be at most k and varying the bound p over its possible values, one can solve our *min-max component problem*. On the other hand, by solving the *min-max component problem* for different value of the size of the separator (k), on can find a solution to the p-separator problem. As a consequence, the *min-max component problem* is also NP-hard in general.

In a recent paper, Chen and Hero [17] consider the same *min-max component problem*. They relate the optimization problem to the spectrum of the graph by showing that minimizing the largest component of a graph (by removing a set of nodes) is equivalent to finding a set of sparse orthogonal vectors that span the nullspace of the associated graph Laplacian matrix. Although a greedy algorithm exists to compute such a sparse orthogonal set, finding the best set is of combinatorial order $\binom{n}{k}$. Chen and Hero propose an approximation algorithm that iteratively chooses a node to remove by first finding a Fiedler (edge) cut [18] of the largest component and then chooses a vertex incident to the cut whose removal minimizes the size of the resulting largest component.

Unfortunately, for networks with a large number of nodes, such as the ones in this paper, Chen-Hero's algorithm has a considerable execution time. This is not surprising as the algorithm requires an iterative computation of the Fiedler vector [19] of the largest component. The Fiedler vector is the eigenvector associated with the second smallest eigenvalue of the Laplacian matrix (also known as the algebraic connectivity) of the graph. Its computation involves solving a large eigenvalue problem which is $O(n^3)$ [20] and can be computationally very expensive for large graphs. Several algorithms have been proposed for computing the Fiedler vector in a parallel setting [21], or finding an approximation of it [22]. However, the problem remains complex.

Chen and Hero's [17] algorithm belongs to a family of algorithms that uses features of the graph spectrum and are sometimes referred as *spectral-based approaches*. Another spectral-based algorithm is proposed by Estrada and Hatano [23]. Here, the proposed heuristic is based on the observation that nodes that correspond to entries of the Fiedler vector with a value close to zero do not strongly belong to any connected component; they are located between the components. As such, by iteratively removing these nodes from the largest component, one can derive a vertex separation of the graph.

Many other spectral-based algorithms have been proposed to solve closely related problems such as graph partitioning [18] [24], [25], [26]. In general, the advantage of these algorithms is that they provide a performance guarantee [26], and work relatively well for small size graphs and graphs with special properties [26]. However, for large graphs, the task of computing the eigenvalues and eigenvectors is often too costly and not competitive [27].

The vertex separator problem [10], [11], [12] is another closely related problem that has received some interest in the literature. Here, the aim is to remove a minimum-size subset of vertices such that each connected component in the remaining graph has a size less than $\alpha|V|$ (for a fixed $\alpha < 1$). It is a special case of the p-separator problem where the graph is to be partitioned in three subgraphs: the separator, and two subgraphs, each of size less than $\alpha|V|$.

An exact solution of the vertex separator problem can implemented by using the algorithm by Chen and Liang [28] that enumerates all minimal separators of the graph. A minimal separator is one that does not contain another separator. Once all minimal separators are enumerated, we just need to choose one that satisfies the constraints. If none of the minimum separators satisfy the constraints, one can search for an approximate solution by taking unions. This approach is clearly not scalable because the number of minimal separators is potentially exponential.

Another exact algorithm for solving the vertex separator problem can be found in Didi Biha and Meurs [29]. Their solution is based on the polyhedral formulation by de Souza and Balas [10] who cast the problem as a mixed integer program (MIP). They then relax the MIP to obtain a linear program (LP) whose constraints define a polyhedron. In a companion paper [11], de Souza and Balas use this LP relaxation to propose a branch-and-cut algorithm by introducing cutting plan for the polyhedron. Didi Biha and Meurs [29] extend this work by finding a class of new inequalities that form the facets of the polyhedron. Then, using a natural lower bound, they are able to compute a solution of the LP in small amount of time without using branch-and-cut methods. Unfortunately, this approach does not give any guarantee and the worst case complexity for solving an LP is exponential [30].

Several other heuristics can be found in the literature that attempt to separate a graph using different criterion. Tong et al. [6] study the problem of removing a set of nodes to minimize the largest eigenvalue of the resulting graph. They propose a greedy approach that iteratively selects the current node with highest eigen-drop as the one to be removed next. Not only does this algorithm suffer from the limitations of spectral-based methods (it is one), a quick test has shown that it performs poorly in our framework. The heuristic proposed by Schneider et al. [31] separates the graph by iteratively removing the node with highest betweenness centrality. The betweenness centrality of a node is defined as fraction of shortest paths between all pairs of nodes going through it. A fast algorithm has been proposed by Brandes [32] that runs in $O(nm)$ and requires storage space of order $O(n+m)$. Our tests on smaller graphs have shown that in general this algorithm perform less optimally compared to the heuristic employed in this paper.

6. Conclusion

The NP-complete problem of finding an optimal vertex based graph separator appears in a wide range of contexts. A number of excellent approximations to this problem exist, but do not scale well past graphs of several thousand nodes. For larger graphs, the heuristic of removing the highest-degree node has been previously proposed elsewhere, and our results show that for several classes of graphs it obtains good cuts, however the quadratic scaling of the naïve implementation renders it difficult to use for very large graphs. One may approximate this high-degree heuristic by not recalculating the node degrees following vertex removal (the "static degree" or SD method), however we show that this often, but not always, significantly damages the quality of the cut. In this work, we present an algorithm with both linear time and memory complexity for iteratively removing the highest-degree node from a graph that obtains the separator quality provided by the full quadratic implementation. We may thus obtain good separators on graphs of tens of millions of nodes quickly and efficiently.

References

[1] AFUKUYAMA, Junichiro. NP-completeness of the Planar Separator Problems. *J. Graph Algorithms Appl.*, 2006, vol. 10, no 2, p. 317-328.

[2] KORTE, B. et VYGEN, J. Combinatorial Optimization Theory and Algorithms, Springer-Verlag Berlin Heidelberg , 2012.

[3] ALBERT, Réka, JEONG, Hawoong, et BARABÁSI, Albert-László. Error and attack tolerance of complex networks. *nature*, 2000, vol. 406, no 6794, p. 378-382.

[4] COHEN, Reuven, EREZ, Keren, BEN-AVRAHAM, Daniel, et al. Breakdown of the Internet under intentional attack. *Physical review letters*, 2001, vol. 86, no 16, p. 3682.

[5] MADAR, Nilly, KALISKY, Tomer, COHEN, Reuven, et al. Immunization and epidemic dynamics in complex networks. *The European physical journal b-condensed matter and complex systems*, 2004, vol. 38, no 2, p. 269-276.

[6] TONG, Hanghang, PRAKASH, B. Aditya, TSOURAKAKIS, Charalampos, et al. On the vulnerability of large graphs. In : *Data Mining (ICDM), 2010 IEEE 10th International Conference on. IEEE*, 2010. p. 1091-1096.

[7] "NetworkX: High-productivity software for complex networks," [Online]. Available: https://networkx.github.io/.

[8] "CAIDA: Center for Applied Internet Data Analysis," [Online]. Available: http://www.caida.org/home/.

[9] BULUC, Aydin, MEYERHENKE, Henning, SAFRO, Ilya, et al. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[10] BALAS, Egon et DE SOUZA, Cid C. The vertex separator problem: a polyhedral investigation. *Mathematical Programming*, 2005, vol. 103, no 3, p. 583-608.

[11] DE SOUZA, Cid et BALAS, Egon. The vertex separator problem: algorithms and computations. *Mathematical Programming*, 2005, vol. 103, no 3, p. 609-631.

[12] BENLIC, Una et HAO, Jin-Kao. Breakout Local Search for the Vertex Separator Problem. In : *23th Intl. Joint Conference on Artificial Intelligence (IJCAI-13)*, Beijing, 2013.

[13] BONDY, John Adrian et MURTY, Uppaluri Siva Ramachandra. *Graph theory with applications*. London : Macmillan, 1976.

[14] BUI, Thang Nguyen, CHAUDHURI, Soma, LEIGHTON, Frank Thomson, et al. Graph bisection algorithms with good average case behavior. *Combinatorica*, 1987, vol. 7, no 2, p. 171-191.

[15] BEN-AMEUR, Walid, MOHAMED-SIDI, Mohamed-Ahmed, et NETO, José. The k-separator problem. In : *19th International Conference*, COCOON , Hangzhou, China, 2013.

[16] BERGER, André, GRIGORIEV, Alexander, et ZWAAN, Ruben. Complexity and approximability of the k-way vertex cut. *Networks*, 2014, vol. 63, no 2, p. 170-178.

[17] CHEN, Pin-Yu et HERO, Alfred O. Node removal vulnerability of the largest component of a network. In : *Global Conference on Signal and Information Processing (GlobalSIP)*, 2013 IEEE. IEEE, 2013. p. 587-590.

[18] SPIELMAN, Daniel A. et TENG, Shang-Hua. Spectral partitioning works: Planar graphs and finite element meshes. *Linear Algebra and its Applications*, 2007, vol. 421, no 2, p. 284-305.

[19] FIEDLER, Miroslav. Algebraic connectivity of graphs. *Czechoslovak mathematical journal*, 1973, vol. 23, no 2, p. 298-305.

[20] PAN, Victor Y. et CHEN, Zhao Q. The complexity of the matrix eigenproblem. In : *Proceedings of the thirty-first annual ACM symposium on Theory of computing*. ACM, 1999. p. 507-516.

[21] MANGUOGLU, Murat, COX, Eric, SAIED, Faisal, et al. TRACEMIN-fiedler: A parallel algorithm for computing the Fiedler vector. In : *High Performance Computing for Computational Science–VECPAR 2010*. Springer Berlin Heidelberg, 2010. p. 449-455.

[22] SRINIVASAN, Ashok et MASCAGNI, Michael. Monte Carlo techniques for estimating the Fiedler vector in graph applications. In : *Computational Science—ICCS 2002*. Springer Berlin Heidelberg, 2002. p. 635-645.

[23] ESTRADA, Ernesto et HATANO, Naomichi. Resistance distance, information centrality, node vulnerability and vibrations in complex networks. In : *Network science*. Springer London, 2010. p. 13-29.

[24] POTHEN, Alex, SIMON, Horst D., et LIOU, Kang-Pu. Partitioning sparse matrices with eigenvectors of graphs. *SIAM journal on matrix analysis and applications*, 1990, vol. 11, no 3, p. 430-452.

[25] CHUNG, Fan. Four cheeger-type inequalities for graph partitioning algorithms. In : *ICCM*. 2007. p. 1-4.

[26] KELNER, Jonathan A. Spectral partitioning, eigenvalue bounds, and circle packings for graphs of bounded genus. *SIAM Journal on Computing*, 2006, vol. 35, no 4, p. 882-902.

[27] CHUNG, F. A Local Graph Partitioning Algorithm Using Heat Kernel Pagerank. In : *6th International Workshop*, Barcelona, 2009.

[28] KLOKS, Ton et KRATSCH, Dieter. Listing all minimal separators of a graph. *SIAM Journal on Computing*, 1998, vol. 27, no 3, p. 605-613.

[29] BIHA, Mohamed Didi et MEURS, Marie-Jean. An exact algorithm for solving the vertex separator problem. *Journal of Global Optimization*, 2011, vol. 49, no 3, p. 425-434.

[30] BOYD, Stephen et VANDENBERGHE, Lieven. *Convex optimization*. Cambridge university press, 2004.

[31] SCHNEIDER, Christian M., MIHALJEV, Tamara, HAVLIN, Shlomo, et al. Suppressing epidemics with a limited amount of immunization units. *Physical Review E*, 2011, vol. 84, no 6, p. 061911.

[32] BRANDES, Ulrik. A faster algorithm for betweenness centrality*. *Journal of mathematical sociology*, 2001, vol. 25, no 2, p. 163-177.