CS 270 Project: Graph Partitioning to Optimize Searching

Adam Chlipala
David Mandelin

April 27, 2004

1 Introduction

There are many domains where tools to help humans understand complex conceptual networks are useful. For instance, a programmer faced with the task of interfacing with a large software library would find invaluable tools for navigating the connections between components, helping him learn how to use them. It is natural to represent collections of object-oriented software with *class graphs*, where each node represents one class and edges between nodes indicate that one inherits from another, one has a method returning an instance of another, and other similar relationships.

By searching the classgraph, we can obtain a generalization of the coding assistance hints that many popular programming environments provide. For instance, the Eclipse Java IDE has a feature where all methods supported by a variable may be displayed by clicking on an instance of that variable in source code. We can extend this idea further and allow arbitrary searches through a classgraph. A programmer may find himself with an object of class A and looking for an object of class B that he knows is related to the first object, but not knowing the code that realizes the relationship. A helpful IDE could navigate the labyrinth of software library structure to suggest a few likely paths of edges, suggesting sequences of relationships that connect A and B. Likelihood can be approximated heuristically using graph-theoretic properties such as path length.

Problems arise due to the large amount of software infrastructure that may be involved with a modest software project today. It would not be surprising to find hundreds of thousands of classes that are involved in a project through inclusion of a variety of libraries. Based on the likely memory limitations of workstations on which developers would want to use a generalized hints facility, it may be prohibitively expensive to load a representation of a class graph into memory. However, this need not necessarily be a problem. Since people have quite limited memory and information processing ability, it seems likely that they will tend in practice to make simple queries. Also, query results involving complex series of relationships are unlikely to be what a user was looking for.

These observations motivate us to attempt to use automated techniques to partition classgraphs into smaller pieces. We hope to make these pieces mostly self-sufficient, in the sense that reasonable queries will tend to be satisfied with nodes from a single portion of the partition. We assume that a search facility is implemented using data on a block storage device, like a hard disk, such that the cost to read a block can be much higher than the cost to perform parts of the search. Motivated by this assumption, we study algorithms for dividing graphs into individually loadable portions, with the objective of minimizing the number of them that must be loaded for simple queries.

We survey algorithms from the literature in the areas of graph partitioning and graph clustering. Then we present some experiments with using these and other algorithms on representative samples of class graphs for real object oriented software.

2 Conventions and terminology

We will use $G = (V, E)$ to denote a graph, possibly with edge capacities c_{uv} for $uv \in E$. Let $n = |E|$ and $m = |V|$. We assume $n \leq m$. A pair (A, B) is a *cut* of G into *blocks* A and B if $A \cap B = \emptyset$ and $A \cup B = V$. The capacity of a cut for G is the sum of the edges in E that span the two blocks.

We extend the notion of a cut into that of a *partition* or *clustering* of G, where we have $(A_1, ..., A_k)$ such that $\cap A_i = \emptyset$ and $\cup A_i = V$. We will consider various metrics of the quality of a partition.

3 Algorithms for graph partitioning and clustering

3.1 Clustering techniques

In contrast to work on graph *partitioning*, graph *clustering* is generally concerned with useful presentation of information to humans rather than optimality according to agreed upon metrics. Each approach chooses some measure of similarity between nodes, such as belonging to the same biconnected component, and attempts to divide graphs into clusters of similar nodes. [6] recommends using a metric called *multiway ratio cut*, where a clustering is chosen to minimize the sum of capacities of all edges crossing clusters divided by the product of all cluster sizes.

[1] suggests using *distance k cliques* as the units of clustering, where each cluster consists of nodes with a path of length at most k between any pair. They give a heuristic algorithm for creating good clusters for this model: They construct an initial clustering by iteratively picking the highest degree node currently not assigned to a cluster and assigning it and all its unassigned neighbors to a new cluster. For each node, a subset of the clusters of its neighbors will be maintained. Each set is initialized to the complete set of a node's neighboring clusters, discounting its own cluster. The initial clustering is then refined by repeatedly merging clusters. At each stage, the node v with the most clusters represented in its neighbor set is chosen. If merging v's cluster and those of its neighbors contained in its cluster set leads to a diameter (maximum distance between nodes of a cluster) less than or equal to k, then the merge is performed. If not, the largest among v's neighbors' clusters is chosen to remove from v's cluster set. The algorithm is finished when every node's set is empty. [1] provides an implementation that runs in $O(n^3)$ worst-case time.

In general, clustering algorithms seem to be designed to work on graphs that will be analyzed visually by people. The notions of similarity of nodes may map well to notions of similarity that will guide the queries that users make in our problem domain.

3.2 Kernighan-Lin

3.2.1 The basic algorithm

The classic algorithm for graph partitioning is due to Kernighan and Lin [3]. They proposed a simple heuristic found to produce near-optimal results in practice. We will present the simplest version of it, which partitions its input into two equally sized blocks. The basic idea is to start with an initial partition of a graph (chosen randomly or through some other heuristic) and successively exchange nodes between the partitions, determining the cut size at each stage and saving the best cut found so far. After the desired number of exchanges have been performed, the saved best cut is used as the output of the algorithm. It may be refined further using various techniques.

The algorithm starts by producing an initial partition of G into A and B with $|A| = |B| = n$, perhaps randomly or perhaps using some heuristic particular to the kind of graphs in question. Throughout execution,

Algorithm 1 The Kernighan-Lin Algorithm

Choose an initial partition (A, B) with cost C
for all $v \in V$ **do**
  $D_v \leftarrow 0$
  for all $u \in V$ **do**
    if u and v in the same partition **then**
      $D_v \leftarrow D_v - c_{vu}$
    else
      $D_v \leftarrow D_v + c_{vu}$
    end if
  end for
end for
for i from 1 to n **do**
  Find $a_i \in A$ and $b_i \in B$ to maximize $\delta = D_{a_i} + D_{b_i} - 2c_{a_i b_i}$
  $C \leftarrow C - \delta$
  If C is the lowest cost found yet, save i and C as the best solution.
  $A \leftarrow A - \{a_i\}$
  $B \leftarrow B - \{b_i\}$
  for all $u \in A$ **do**
    $D_u \leftarrow D_u + 2c_{ua_i} - 2c_{ub_i}$
  end for
  for all $u \in B$ **do**
    $D_u \leftarrow D_u - 2c_{ua_i} + 2c_{ub_i}$
  end for
end for
Return the best solution found

the *gain* D_v for each $v \in V$ will be maintained. D_v indicates the difference between the costs of edges connecting v to nodes outside its partition and the corresponding costs to nodes in its own partition. The idea is that nodes with high gain are locally good choices to swap into the opposite partition.

The algorithm then divides the nodes into n pairs a_i, b_i with each pair spanning both partitions. A greedy heuristic is used to choose at each stage to swap the pair consisting of previously unswapped nodes that yields the minimum capacity cut. The decrease in the cut size for a pair a, b is given by $D_a + D_b - 2c_{ab}$. By definition D_a is the difference in the capacity of cut edges involving a before and after the swap. The analogous fact is true of D_b. A correction factor of $2c_{ab}$ must be subtracted, since c_{ab} appears positively in both D_a and D_b, although a and b will have a cut edge between them in both the old and new partitions. Therefore, the pair that maximizes this expression leads to the least cut capacity after a single swap.

After a swap, the chosen nodes are removed from consideration for further swapping, and D values are updated to reflect the swap. The terms added to them reflect the changed relative status of a_i and b_i, adding or subtracting two times the appropriate capacities to cancel out the subtraction or addition of a term and change it to an addition or subtraction, respectively.

Thus, the algorithm uses locally optimal swaps of nodes until all nodes have been swapped and returns the lowest cost partition found.

Ignoring the time to generate an initial partition, we can analyze the approximate running time of Kernighan-Lin as follows: The initialization of lines 2 to 10 takes $O(n^2)$ time, considering each pair of nodes. In each pass, the a_i and b_i to swap can be chosen by first sorting the D values of nodes in A and B in descending order in $O(n \log n)$ time. The possible pairs can then be examined with an order that favors high D values' consideration early in the search. When the first pair with a D sum no better than the current best δ is found, we can rule out all pairs with both gains less than the current gains, since subtracting $2c_{a_i b_i}$ will only lead to an improvement worse than the current best. Kernighan and Lin seem to indicate that, in practice, we can assume that only a constant number of pairs will need to be considered after sorting. We can certainly set a constant cut-off and settle for inferior solutions. After a_i and b_i are chosen, we need $O(n)$ time to update D values. Thus, with their assumption, we have n runs of the loop at $O(n \log n + n)$ time each, giving total running time for the algorithm of $O(n^2 \log n)$.

Iterating Kernighan-Lin using previous results as initial partitions can lead to even better solutions. Kernighan and Lin say that in practice, a constant number of passes is sufficient to produce sufficiently optimal partitions. Unfortunately, this is highly dependent on the input graphs, and they prove no theoretical bounds. Also, as is understandable for work done in 1970, their tests are run on graphs of insignificant size by today's standards. Thus, it is unclear whether their heuristic is a good choice to apply to particular problems today.

3.2.2 The Fiduccia-Mattheyses heuristic

Fiduccia and Mattheyses [2] proposed improvements to Kernighan-Lin such that one pass of their version can be proved to run in linear time. For the case where all edges have unit capacity, they use a bucket sort-style technique based on a table with size linear in the maximum degree of a node in the input graph.

Assuming an adjacency list graph representation, Fidducia-Mattheyses can be implemented to run in $O(m)$ time. The initialization of D values is improved from $O(n^2)$ to $O(m)$ by iterating over edges instead of all pairs of nodes. Therefore, the main loop that chooses the pairs of nodes to swap is where the limiting computations occur. The non-trivial operations it uses are choosing a locally optimal pair and then updating the D values for the remaining nodes. Since all edges have unit capacity, the first operation corresponds to finding the node on each side of the current partition with the highest D value. Fiduccia-Mattheyses provides amortized constant time methods for performing each of these two operations.

The main data structures used to achieve this are arrays D^A and D^B with indices from $-d_{max}$ to d_{max}, where d_{max} is the highest degree of any node in the input. $D^A[n]$ contains a doubly linked list of exactly

those nodes $v \in A$ with $D_v = n$, and likewise for D^B. Since every D_v is the difference in sizes of two subsets of edges incident on v, d_{max} clearly bounds $|D_v|$.

Each array has associated with it a value M^A or M^B of the highest index into D^A or D^B, respectively, known to contain a nonempty list. To find the highest gain node from D^A, $D^A[M^A]$ is checked for nodes. If none is found, M^A is decremented until a nonempty bucket is found. Clearly, if no node's gain ever increases, we obtain amortized constant time to remove the highest gain node from its array, even after any sequence of gain decrements, since the total number of seeks downward is bounded by $2d_{max}$, which is $O(n)$. Any gain decrement can be performed in constant time by moving a node between linked list buckets. However, certain operations may increase the gains of nodes and force an increase of M^A. Nonetheless, as discussed below, it is possible to bound this amount of change in M^A enough to guarantee amortized constant time to look up the highest gain element of an array.

Consider the updates to node gains that start at line 17 in Algorithm 1. We can reorganize the process to only consider neighbors of a_i and b_i instead of all remaining nodes. Because only neighbors of swapped nodes need gain updating, we can associate each update with an edge in the graph. Since a node is only chosen to be swapped once per pass, we have that each edge can only have an endpoint swapped at most twice per pass. This gives an $O(m)$ bound for the number of updates the could be performed in a single pass, with each update taking constant time. From the equation for updating gains, we see that a node's gain may increase by at most 2 per pass. Thus, M^A and M^B can't go up by more than 2 per pass, so searches for highest gain nodes only use $O(n)$ time re-searching array cells with indices above previous M^A or M^B values.

Thus, we have $O(m)$ time to initialize gains, $O(n) + O(n)$ time to search for highest gain nodes, and $O(m)$ time to update gains during the pass, giving overall $O(m)$ running time.

Fiduccia-Mattheyses behaves identically to the original algorithm, with no proofs of the quality of solutions generated. Nonetheless, the running time bound is improved significantly, especially for large graphs.

3.3 Multi-commodity network flow methods

A generalization of the classic max-flow problem has proven to be useful in graph partitioning. An instance of the *multi-commodity network flow* problem involves a graph $G = (V, E)$ with edge capacities and pairs of sources and sinks $s_i, t_i \in V$ between which distinct commodities should be routed. Each pair has a demand associated with it. Informally, a successful flow meets the demand for each commodity by flowing commodities along edges while keeping the total flow through every edge within its capacity.

There are variants of the problem with different ideas of varying edge capacities and demands for commodities, different metrics for maximality of a flow, and other factors. For solving graph partitioning problems, we can make use of a simple version known as the *uniform multi-commodity network flow* problem. An instance of this problem has one commodity of unit demand for every pair of nodes. The classical definition of flow values for single-commodity networks, which uses the sum of flows into the sink as the value of a flow, is extended as follows: The flow value of a flow is the highest value f such that at least f units of every commodity is conveyed from its source to its sink. This metric can be thought of as enforcing fairness: If not all demand can be satisfied, then demand should be satisfied as evenly as possible.

The value of a cut (A, B) of a multi-commodity uniform network is defined as:

$$\frac{C(A, B)}{|A||B|}$$

where $C(A, B)$ gives the sum of edge capacities going between nodes of A and B. A *min-cut* for such a network is a cut of minimal value.

For any multi-commodity network flow problem, let f denote its max-flow and c its min-cut. Like in the

single-commodity case, a simple argument shows that the min-cut bounds the max-flow from above. We have $\sum_i fD_i \leq C(A, B)$, where D_i is the demand for commodity i, with i ranging over only those demands whose sources and sinks are split by the cut. The inequality holds because every flow between these sources and sinks clearly must pass through the (A, B) cut, by definition. Since the sum of these D_i's is $D(A, B)$, the sum of all demands for commodities with split sources and sinks, we have $f \leq \frac{C(A,B)}{D(A,B)}$. As we are considering only uniform networks, $D(A, B) = |A||B|$, and we have that f is less than or equal to the cut value.

However, [5] showed that, unlike for single-commodity problems, c can be arbitrarily higher than f. In particular, it may be $\Theta(\log n)$ higher. [5] also showed that this is the worst-case difference, so $c = O(\log n)f$. This result shows that techniques for finding multi-commodity max-flows may be used as $O(\log n)$ approximation algorithms for multi-commodity min-cut.

3.3.1 Heuristics

In [4], a partitioning heuristic is proposed based on this result. The algorithm has three stages: First, an approximation of the uniform multi-commodity flow problem corresponding to the input graph is created by choosing k pairs of nodes at random as the sources and sinks for distinct commodities. Then, this simplified problem is solved using an approximate max-flow heuristic. Finally, Prim's minimum spanning tree algorithm is run several times with random root nodes, using the flow on edges in the max-flow solution as their weights. One block of the final partition is chosen to be the subtree encountered during these iterations that leads to the lowest cut size.

The max-flow heuristic depends on three parameters, d_{min}, d_{max}, and I. Let C be the set of commodities (pairs of nodes) previously chosen randomly. For $e \in E$, $f[e]$ will denote the total flow through e.

Algorithm 2 Multi-commodity flow heuristic

1: Initialize all flows to 0.
2: **for all** $(s, t) \in C$ **do**
3: Route one unit of flow from s to t along a shortest path determined with edge weights $f[e]^{d_{min}}$.
4: **end for**
5: **for** d from d_{min} to d_{max} **do**
6: $maxflow \leftarrow \max_{e \in E} f[e]$
7: $E' \leftarrow \{e \in E : f[e] = maxflow\}$
8: **for all** $e \in E'$ **do**
9: Pick a random commodity (s, t) currently routed through e and re-route it through a shortest path from s to t, using weights $f[e]^d$.
10: **end for**
11: **end for**

The initialization in 1 takes $O(km)$ time. Throughout the algorithm, the $f[e]$ values may be updated in constant time whenever the flow of a particular commodity through an edge changes. The initial routing of 2 to 4 can use k runs of Dijkstra's shortest path algorithm, at $O((n + m) \log n)$ time each. The loop of 5 to 11 requires $d_{max} - d_{min} + 1$ iterations, each with $O(m)$ work to find $maxflow$ and E', with a run of Dijkstra's algorithm for each, giving $O(m(n + m) \log n)$ time per iteration. Therefore, the overall worst-case running time is $O(km + (d_{max} - d_{min})m(n + m) \log n)$. Of course, in practice we can expect each E' to contain significantly fewer edges than E, leading to faster expected running times.

[4] presented results showing that this heuristic has superior performance to variants of the Fiduccia-Mattheyses algorithm on graphs involving two-dimensional points with edges present between exactly those nodes within a fixed distance of each other.

4 A partitioning heuristic designed to optimize graph searches

We have developed a simple heuristic for bisecting an unweighted graph, based on intuitions from our problem domain. Since we are interested in partitioning a graph to minimize the number of blocks that must be loaded to perform a shortest path query, we will use a random selection of plausible queries to choose the partition. Algorithm 3 depends on parameters k, the number of random searches to perform; and r, the maximum shortest path distance for a query.

Algorithm 3 The Random Queries partitioning heuristic

1: Set all edge weights to 0.
2: **for** i from 1 to k **do**
3: Choose a node $v \in V$ at random.
4: Use BFS to find all nodes within distance r of v.
5: Increment the weight of every edge in the BFS tree.
6: **end for**
7: Run Kernighan-Lin or any other partitioning algorithm using the edge weights that have been computed.

The time overhead our heuristic adds beyond that of the underlying partitioning algorithm used comes from running k BFS's. Each BFS takes $O(m)$ time, so the additional time overhead is $O(km)$. In practice, since we may use a small r, for many kinds of graphs the additional overhead may be effectively constant for fixed k and r.

We tested empirically the effectiveness of this and other techniques.

5 Experimental setup

5.1 Algorithms used

As a control, we tested an algorithm that we call Random, which uses a "balls and bins" style of cluster assignment. The number of clusters to form is set ahead of time, and each node is assigned to one of these clusters with uniform probability. A random assignment is like what we would expect from a haphazard layout of a graph on disk, with no attention paid to limiting the number of blocks needed to load per query.

We tried what we will call the Simple clustering algorithm. It involves repeatedly choosing a node u not yet assigned to a cluster and adding u and all unassigned nodes within a fixed distance r of it to a new cluster.

The last algorithm that we tried not previously mentioned is what we will call the BFS algorithm. It picks a root u at random and generates a breadth-first search tree for the graph. It then tries all the possible cuts between depths d in the tree, such that the partition puts all nodes closer than d to u in the BFS tree in one block and those at least d from u in the other. The partition formed this way with the best cut value is chosen as the final partition.

To see if standard partitioning algorithms provided partitions useful for query optimization, we tried the two main varieties of these algorithms. We tested Kernighan-Lin starting from random partitions, as a representative of local partitioning techniques; and the heuristic of [4] (which we will call Flow), as a representative of multi-commodity flow techniques.

When testing Flow, in all trials we used $d_{min} = 1$, $d_{max} = 3$, $I = k$ (where I is the number of rerouting iterations and k is the number of commodities), and $N = 3$ (where N is the number of cuts generated). Initially, we found that for our graph, Flow usually found cuts where the smaller partition contained only one vertex, so we modified the procedure for generating cuts to produce only cuts where each partition contains

between one fourth and three fourths of the vertices. We varied the number of commodities in different trials, as noted in the results (Section 6).

We also tested our Random Queries heuristic.

5.2 Metrics

We evaluated the results of the algorithms using two kinds of metrics.

First, we have the traditional metrics used for clustering and partitioning problems. These metrics are meant to indicate good quality for solutions to problems like VLSI layout, where a circuit must be decomposed into limited-sized pieces in a way that minimizes the need for communication between pieces. We have the simple measure of *cut size*, which is the number of edges in the graph that span different blocks of the partition. There is the ratio cost metric used in [6] and [5], where the cost of a partition is its cut size divided by the product of block sizes. Finally, we measured the *quotient costs* of partitions as defined in [4], where the cost of a partition is its cut size divided by the minimum of its block sizes instead of the product.

We propose a new set of metrics for measuring the effectiveness of partitioning algorithms in speeding up graph search. These metrics are all based on a distribution of search queries, where a search query is a pair (s, t) and we want to find the shortest path from s to t. The distribution may be either a standard probability distribution or an empirical distribution.

The first new metric is *single-partition probability*, which is the probability that a random query from the distribution can be answered by looking at only one partition. This gives some idea how well the partitioning matches the query distribution, but it tells us little about the effect on query processing performance. For that, we need *expected cost* metrics, which are simply the expected cost of processing a random query from the distribution. In our Java IDE problem, the main cost of processing a query is loading the graph. A sensible cost model is that each partition is loaded only if the search touches one of its vertices, each partition is loaded at most once, there is a fixed overhead cost for loading a partition, and a per-vertex cost for loading each partition. The overhead cost is necessary because otherwise putting each node in its own partition would be optimal, which is both uninteresting and not true in practice. We can also consider simpler cost models, such as the *all-or-nothing model*, where if the query can be answered looking at only one partition, the cost is the size of the partition, otherwise the cost is the size of the entire graph.

5.3 The Class Graph

Our class graph is extracted from a Java class library. The set of vertices is the set of Java classes, including all array types referred to in the library. The set of edges is the set of pairs of classes (u, v) such that (i) u is a subclass of v, (ii) u has an instance field of type v, (iii) u has an instance method that has only primitive type (e.g., `int`) parameters, or (iv) any class has a static method where one parameter is an instance of u, all other parameters are primitive types, and the return value is of type v. Intuitively, if there is an atomic Java expression that transforms an instance of type u to an instance of type v, there is an edge from u to v. Using this definition, for any code sequence that transforms an object from one type to another repeatedly, there is a path in the graph. Thus, we can generate such code sequences from paths in the graph. Not every path corresponds to valid code, but solving that problem is beyond the scope of this paper.

The class graph used in this paper was extracted from the J2SE 1.4 runtime, Eclipse 3.0, and Apache Common Collections library 3.0. Those libraries produce a graph of 27,037 vertices and 164,281 edges. This graph has several features that make partitioning especially difficult. For example, every Java object supports the method `getClass()`, which returns an instance of type `java.lang.Class`, so vertex `java.lang.Class` is connected to every other vertex. To make the partitioning problem easier, we modified the graph by removing 13 vertices of very high degree and extracting the largest connected component. The resulting

graph has 13,467 vertices and 78,280 edges. The median distance between a pair of vertices is 10. 90 percent of pairs of vertices are within 19 edges of each other. The maximum distance between any pair is 56.

6 Results

Table 1 shows the results of bisecting the graph using four different partitioning algorithms. All of the real algorithms substantially outperformed the random baseline, as expected. Kernighan-Lin performed the best. Flow improved substantially when using 10,000 commodities instead of 1,000, so Flow may outperform Kernighan-Lin when run with a larger number of commodities. Our test implementation is of low quality and runs slowly, so we did not have enough time to run a larger trial.

Overall, the variations in performance among the algorithms were small. Since Flow and Kernighan-Lin are so different, the similar performance suggests these algorithms are finding near-optimal partitions for these metrics.

Algorithm	Partition Sizes	Cut Capacity	Quotient Cost	Ratio Cost ($\times 10^{-5}$)
Random	6758/6709	38798	5.8	85.5
BFS	6803/6664	9488	1.4	2.1
Kernighan-Lin	6734/6733	6562	1.0	1.4
Flow (1000 commodities)	9639/3828	8238	2.2	2.2
Flow (10000 commodities)	6456/7011	8367	1.3	1.8

Table 1: Bisection Results

Table 2 shows the single-partition probability (the probability a random query can be answered using only one partition – see Section 5.2) for four partitioning algorithms. The table shows the fraction of queries that can be resolved by looking at only one partition. Each trial used 500 shortest-path queries from a source vertex chosen uniformly at random to a destination vertex chosen uniformly at random from all vertices within 6 edges of the source (but not including the source).

Our implementation of Flow produced very asymmetrically sized partitions when used to make 32 or 64 partitions, so those results are omitted. Also, because the Random Queries heuristic was so ineffective applied to two partitions, we did not continue trials with it.

Again, the real algorithms greatly outperformed Random. This time, Flow, even with a relatively small number of commodities, performed the best. We speculate that the way Flow routes commodities along shortest path between random vertex pairs is in some way related to the idea of finding shortest paths between random pairs of query nodes. One problem is that in these trials, Flow often produced unequally-sized partitions, which tends to increase single-partition probability even though it does not improve query performance (to see why, consider a trivial partition that places all vertices in one partition and leaves the rest empty). Trials with a larger number of commodities may solve this problem, but we did not have time to run them.

Surprisingly, Random Queries performed no better than Random. It may be that the heuristic is ineffective. It may also be that the number of random queries used in the trial, 100,000, was too few to accurately approximate the edge weights given by the true random distribution.

We were able to collect only limited data on expected query processing costs, so we do not report it in a detailed table. For the all-or-nothing cost model, none of the algorithms improved on the baseline (always loading the entire graph) by more than 20%. This suggests that the corresponding implementation is not of practical interest. The more detailed cost model, which allows loading of any subset of the partitions, may yield better results, but we did not have time to test it.

	Number of Partitions					
Algorithm	2	4	8	16	32	64
Random	.078	.032	.016	.012	.008	.002
BFS	.264	.160	.100	.134	.102	.082
Kernighan-Lin	.308	.220	.200	.184	.104	.102
Random Queries	.082					
Flow (1000 commodities)	.408	.274	.236	.222		

Table 2: Single-Partition Probability

7 Conclusion

We described a set of graph partitioning algorithms, and how graph partitioning can be used to speed up graph searches. We also described a new heuristic intended to make the algorithms produce partitions specifically to speed up graph searches. We tested the algorithms on a specific graph of interest in software engineering, and found that the classic algorithms were effective according to several metrics.

References

[1] Jubin Edachery, Arunabha Sen, and Franz-Josef Brandenburg. Graph clustering using distance-k cliques. In *Graph Drawing*, pages 98–106, 1999.

[2] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the nineteenth design automation conference*, pages 175–181. IEEE Press, 1982.

[3] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Tech J.*, 49(2):291–307, 1970.

[4] Kevin Lang and Satish Rao. Finding near-optimal cuts: an empirical evaluation. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 212–221. Society for Industrial and Applied Mathematics, 1993.

[5] T. Leighton and S. Rao. An approximate max-flow mincut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proc. 29th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 422–431, 1988.

[6] Tom Roxborough and Arunabha Sen. Graph clustering using multiway ratio cut. In Giuseppe Di Battista, editor, *Proc. 5th Int. Symp. Graph Drawing, GD*, number 1353, pages 291–296. Springer-Verlag, 18–20 1997.