Evolving Multi-level Graph Partitioning Algorithms

Aaron S. Pope*†, Daniel R. Tauritz* and Alexander D. Kent†
* *Department of Computer Science,*
Missouri University of Science and Technology,
Rolla, Missouri 65409
† *Los Alamos National Laboratory*
Los Alamos, New Mexico 87545
Email: aaron.pope@mst.edu, dtauritz@acm.org and alex@lanl.gov

Abstract—**Optimal graph partitioning is a foundational problem in computer science, and appears in many different applications. Multi-level graph partitioning is a state-of-the-art method of efficiently approximating high quality graph partitions. In this work, genetic programming techniques are used to evolve new multi-level graph partitioning heuristics that are tailored to specific applications. Results are presented using these evolved partitioners on traditional random graph models as well as a real-world computer network data set. These results demonstrate an improvement in the quality of the partitions produced over current state-of-the-art methods.**

1. Introduction

The problem of graph partitioning shows up in a wide variety of application domains. Examples include organizing parallel computation workload [1], VLSI layout design [2], image processing [3], and critical infrastructure protection [4], among others. In general, optimal graph partitioning is known to be NP-hard [5]. As a result, time sensitive applications typically rely on heuristics which provide approximate partition solutions.

One of the most commonly used approaches to quickly find high-quality graph partition approximations is multi-level graph partitioning [6]. The general idea behind multi-level partitioning involves producing a smaller graph which is an approximation of the original input graph. A high-quality partition is calculated for this reduced graph and this partition is mapped back to the original graph. This is typically done over several iterations and the quality of the partition is improved at each iteration through a refinement process. Several well-known graph partition software packages implement multi-level schemes, such as METIS [7], JOSTLE [8], Scotch [9], and DiBaP [10].

Previous work has shown that partition quality can be improved by selecting specialized heuristics for classes of graphs with specific properties. For instance, superior partition heuristics have been found for graphs with power-law degree distributions [11]. If this process was repeated for a wide variety of applications, it might be possible to assemble a good set of tailored graph partition algorithms instead of relying on a general purpose solution. Selection of the appropriate algorithm for the problem at hand could even be automated, a task for which machine learning approaches have been shown to excel [12].

Unfortunately, this relies on the best solution for a problem already being available. The very nature of these custom heuristics means that the performance gains are likely to be limited to the specific class of graphs for which they were developed. To achieve the same improvements for a new application, the process of manual algorithm optimization must be repeated. Alternatively, this process of developing algorithms tailored to a specific application can be automated by searching the space of custom heuristics.

Genetic programming (GP) [13], a field of evolutionary computation, has been shown capable of automatically generating [14] and optimizing heuristics for a variety of applications [15]. Utilizing GP to optimize multi-level partitioning algorithms can provide two distinct advantages. First, the evolutionary process will consider heuristics that might have been overlooked during manual development because they are not intuitive. Second, once the framework for evolving custom heuristics for a specific application is constructed, it can be quickly applied to any number of other problems as the need arises. The work presented in this paper investigates the potential of using GP to automate the process of tailoring multi-level partitioning algorithms for specific applications, improving their performance over more generalized state-of-the-art partition methods.

2. Graph Partitioning

Given an integer $k \geq 2$ and a graph $G = (V, E, w_v, w_e)$ with the set of vertices V and the set of edges E, vertex weight vector w_v, and edge weight vector w_e, a k-way graph partition divides the vertices of V into k subsets $V_1, V_2, \ldots, V_k$, such that $V_i \cap V_j = \emptyset$ if $i \neq j$ and $V_1 \cup \cdots \cup V_k = V$. For unweighted graphs, let all the entries in w_v and w_e be one. The total weight for a set of vertices X is given by:

$$W_v(X) = \sum_{i \in X} w_v[i].$$

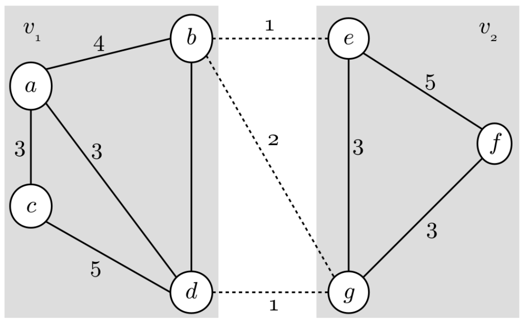


Figure 1. Example graph partition with vertex sets $v_1 = \{a, b, c, d\}$ and $v_2 = \{e, f, g\}$. Edges between partitions are indicated by dashed lines. The cut-cost of the partition is 4.

Similarly, the total weight of a set of edges Y is:

$$W_e(Y) = \sum_{j \in Y} w_e[j].$$

For a given partition, let E' be the subset of edges from E that connect vertices in different partitions. $W_e(E')$ is the weight of this edge set, and is known as the *cost* or *cut-cost* of the partition. Typical graph partition applications require this cost be minimized. See Figure 1 for an example graph partition.

Many applications also place restrictions on the relative total weight of the partitioned vertex subsets. A *balanced* partition requires that

$$\frac{\max_{i=1\ldots k} W_v(V_i)}{\sum_{j=1\ldots k} W_v(V_j)/k} \leq 1 + \epsilon$$

for some constant imbalance factor ϵ. In other words, the ratio of the weight of the heaviest partition to the weight of the average partition cannot exceed $1 + \epsilon$.

2.1. Multi-level Graph Partitioning

Multi-level graph partitioning is one of the most widely used graph partition approximation methods. The approach generally consists of three distinct phases, typically referred to as the *coarsening*, *partition*, and *uncoarsening/refinement* stages. See Figure 2 for a visualization of the multi-level graph partitioning approach.

2.1.1. Coarsening. During the coarsening phase, a smaller approximation of the input graph is created. The coarsening process is repeated, creating a sequence of smaller and coarser graphs, until the size of the coarsest graph is sufficiently small.

The smaller approximation graphs are typically obtained by performing edge or subgraph contractions on the input graph. One common approach to selecting edges for contraction is to find a maximal matching. A maximal matching can be created in a variety of ways, but generally some simple heuristic is used to keep the complexity of the coarsening phase down. Some example heuristics that have been investigated in previous research include:

Random matching: Unmatched vertices are visited in a random order and an incident edge is randomly selected from those that do not violate the matching.

Light edge matching: Similar to random matching, but the lowest weight incident edge is selected instead of selecting randomly.

Heavy edge matching: Identical to light edge matching, except favoring heavy weight edges.

While coarsening using matching schemes has worked well for some applications [7], it has been shown that graphs with power-law degree distribution are difficult to coarsen with matchings alone. In these instances, improved performance can be achieved by contracting small, highly connected subgraphs instead [11].

2.1.2. Partition. During the partition phase, a direct partitioning approach is used to partition the coarsest graph. Due to the small size of the coarsest graph, very little time is required to get a partition of relatively decent quality. For this reason, more computationally expensive partition methods can be employed, such as spectral partitioning [16] or Kernighan-Lin (KL) [17]. Karypis et al. demonstrated that even simpler partition approaches can be used without a loss of final partition quality [7]. Some examples of these simple methods include:

Graph growing partition (GGP): A partition is grown by visiting a random vertex, then adding vertices to the partition in a breadth-first fashion until the partition contains the necessary vertex weight.

Greedy graph growing partition (GGGP): Similar to GGP, but neighboring vertices are added to the partition in an order which maximizes the decrease in the cost of the partition.

2.1.3. Uncoarsening and Refinement. During uncoarsening, the partition solution for the coarsest graph is mapped back to the next coarsest graph. The partition for the coarsest graph gives a good starting partition for the next coarsest, but the quality of the partition is then improved through a refinement step. This uncoarsening and refinement process is repeated until a refined partition is found for the original input graph. Multiple partition refinement strategies exist, and some examples include:

KL refinement: The partition to be refined is used as a starting point for the Kernighan-Lin partition algorithm, except each pass of the algorithm terminates if a configurable number of vertex swaps do not decrease the cost of the partition.

Greedy refinement: The KL refinement algorithm, limited to a single pass.

3. Evolutionary Computation

Evolutionary algorithms (EA) are a family of biologically inspired generate-and-test black-box search

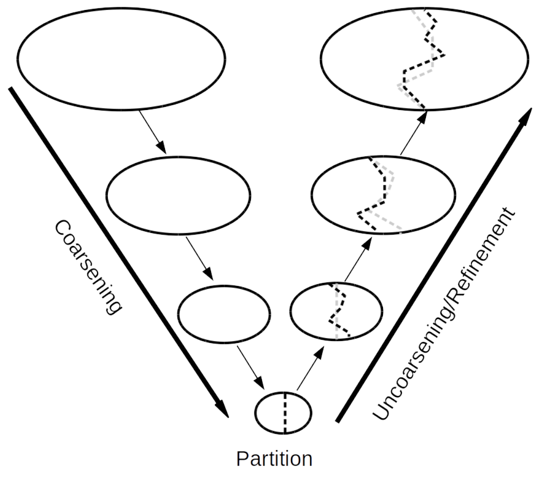


Figure 2. Multi-level graph partitioning strategy.

algorithms [18]. This process encourages solutions with higher *fitness* values, which is a measure of the solution quality, or how well it solves the problem at hand. The stages of a typical EA consist of:

Initialization: A population of solutions is randomly generated and evaluated.

Parent selection: Solutions are randomly selected from the population (typically favoring higher fitness) to participate in creating new offspring solutions.

Recombination: Offspring solutions are created using the genetic information from multiple parent solutions.

Mutation: Offspring solutions are stochastically altered to facilitate exploration of the search space.

Survival selection: The new generation of offspring is evaluated and is either added to, or replaces the current population. A subset of the population is selected to "survive" and continue on in future generations. Again, this selection process usually favors higher fitness.

Termination: The process of selecting parents, creating offspring, and selecting survivors continues until some termination criteria is met. Some example termination criteria are reaching some threshold of quality, convergence of the population, or some limit on total execution time.

3.1. Genetic Programming

Genetic programming (GP) is a field of evolutionary computation where the solutions being evolved take the form of programs or algorithms [13]. A set of primitive operations is usually constructed by observing the common and essential elements of algorithms which have been designed to solve the intended problem. This primitive operation set is used as algorithmic building blocks by the GP to piece together new candidate algorithm solutions.

Many forms of representing algorithm solutions have been developed, but one of the oldest and most common approaches represents programs as parse trees. With this representation, offspring are generated using subtree crossover, where a random node is selected in the parse trees of both parents, then the subtrees rooted at these nodes are swapped to generate two offspring. Mutation is accomplished by randomly choosing a node and replacing the subtree rooted at that node with a new, randomly generated tree. See Figure 3 for an example parse tree representing a simple partitioning algorithm.

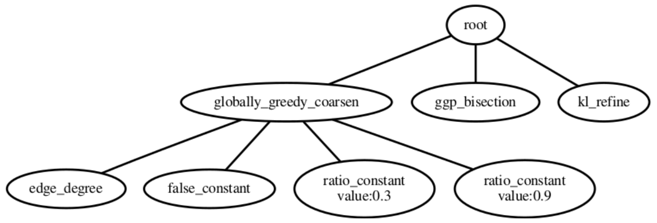


Figure 3. Simple genetic programming parse tree example composed of operations described in Section 5.1.

4. Related Work

This work was inspired by previous research that investigated the effect of the coarsening scheme used for multilevel partitioning algorithms. Abou-Rjeili et al. developed new heuristics for graph coarsening that improved the partition quality for graphs with power-law degree distributions [11]. The superior performance achieved suggests that there is potential in specializing these algorithms to specific classes of graphs. The framework developed in this research will have the added benefit of being able to quickly develop partitioning algorithms which are tailored to new problem areas simply by re-running the GP.

There are many examples of evolutionary computation techniques being used to find approximate minimum graph partitions [19]. The Karlsruhe Fast Flow Partitioner Evolutionary (KaFFPaE) leverages the inherent parallelizability of evolutionary algorithms to evolve graph partitions on a distributed system [20]. Soper et al. introduced an evolutionary search algorithm that makes use of a multilevel heuristic for crossover to generate high quality graph partitions [21]. Benlic et al. developed a multilevel memetic algorithm for the k-way graph partitioning problem [22]. While these approaches are capable of finding very low cost partition solutions, they do so at the cost of execution time. This trade off makes them suitable for applications which must infrequently find extremely high quality partitions, but inappropriate for more time-sensitive problems. This work instead aims to invest a large amount of a priori evolution time to produce algorithms that are capable of quickly finding high quality partition solutions for a specific class of graphs.

The strengths of GP have been leveraged in previous work to evolve random graph generation algorithms [23], [24]. While these works aim to solve a different problem, they still evolve graph related algorithms. Because of this similarity, there is potential overlap in the primitive operation sets used to construct candidate solution algorithms.

5. Methodology

Genetic programming is used to evolve a population of mutli-level graph bisection algorithms that minimize the cost of the partitions they produce. Note that this work is limited to bisectioning, but could be extended to more general k-way partitioning through the use of recursive bisectioning applications.
Representation: Algorithm solutions are expressed as parse trees. A strongly typed representation is employed to accommodate the three distinct phases of the partition algorithms [25]. Initial parse trees have a configurable maximum height to begin the search with simple heuristics that can grow during the course of evolution.
Initialization: The population was initialized using a ramped half-and-half method, which produces full parse trees of maximum height for half the population and variable height trees (up to the maximum) for the remainder.
Evaluation: Each candidate solution is used to partition a configurable number of graphs of the relevant type. The solution's fitness score is given by

$$Fitness = \frac{1}{|P|} \sum_{p \in P} \left[\sum_{(u,v) |\ p[u] \neq p[v]} w_e \left[(u,v) \right] \right],$$

where P is the set of partitions produced by the evolved solution and w_e is the vector of edge weights as described in Section 2. If a solution algorithm takes an excessive amount of time to compute a partition, or produces partitions that violate the balance constraint described in Section 2, a penalized fitness value is assigned, which is given by

$$Penalized\ Fitness = \sum_{(u,v) \in E} w_e \left[(u,v) \right],$$

where E is the complete set of edges in the input graph. In other words, the penalized fitness is the cost of a partition which removes every edge from the graph. Lower fitness values are considered superior, which encourages algorithms that quickly produce low cost partitions while respecting balance.
Parent selection: Parents are selected using binary tournaments, which randomly select two solutions from the population, then return the highest fitness of the two. The low tournament size lowers selection pressure to counteract the elitism introduced by survival selection.
Recombination: 95% of the offspring are created using subtree crossover from two donor parent solutions as described in Section 3.1.
Mutation: The remaining 5% of the offspring are created by performing subtree replacement mutation on a single donor parent as described in Section 3.1. Because subtree crossover and subtree replacement both have the potential to dramatically alter a solution, only one method is applied to each offspring.

TABLE 1. GP PARAMETER VALUES

Parameter	Value
Population size and offspring per generation	60
Partitions per evaluation	10
Minimum parse tree depth	2
Maximum parse tree depth	5
Mutation probability	25%

Survival selection: Truncation selection is used for survival, simply selecting the fittest individuals. This approach is very elitist, and encourages exploitation of currently known high fitness solutions.
Termination: Evolution is terminated when the best fitness seen has not improved for thirty consecutive generations.

The values for the parameters of the GP can be seen in Table 1. These parameters were tuned using a random restart hill climbing search.

5.1. Primitive Operation Set

The individuals in the population of the GP are constructed from the following set of operations.

5.1.1. Root Node. All solutions use the same operation for the root node of their parse tree. This node has three child nodes, which correspond to the three phases of the multi-level partition approach. The first child node takes a graph as input and returns a coarsened graph. This process is repeated, storing the sequence of coarsened graphs, until the coarsest graph contains at most fifty vertices. The second child node takes the coarsest graph as input and returns an initial partition assignment of the vertices. Finally, the third child takes two consecutive graphs from the sequence of coarsened graphs, along with a partition assignment, and returns a refined partition assignment for the less coarse graph. The uncoarsening and refining step is repeated, working from the coarsest graph back to the original graph, until the partition assignment for the original input graph is obtained, which is returned as the final result of the algorithm.

5.1.2. Graph Coarsen Nodes. The first set of coarsening nodes are inspired by traditional multi-level partitioning approaches.
Random matching coarsen: Coarsens the input graph by contracting the edges of a random maximal matching.
Heavy edge matching coarsen: Contracts the edges of a heavy edge maximal matching, as described in Section 2.1.1.
Light edge matching coarsen: Contracts the edges of a light edge maximal matching, as described in Section 2.1.1.

The remaining nodes are inspired by the coarsening schemes developed by Abou-Rjeili et al. [11].
Globally greedy coarsen: This node takes input from four child nodes. The first provides a formula which evaluates

an edge in the graph and returns a metric value. The second returns a boolean that determines if the preceding metric is to be maximized or minimized. The third returns the *maximum vertex weight ratio*, which is the portion of the entire graph's total vertex weight that an individual contracted vertex cannot exceed. The fourth returns the *maximum contraction ratio*, which determines the percentage of the vertices that can be contracted during a single coarsening phase. This operation sorts all of the edges in the graph using the metric formula and attempts to contract them in order, skipping any edge contraction that would violate the maximum vertex weight restriction. The process terminates when the maximum contraction ratio is reached, or all edges are considered, whichever occurs first.
Locally greedy, globally random coarsen: Identical to the globally greedy coarsening strategy, except for the procedure used to generate the list of edges for contraction. Instead of ranking the graph's entire set of edges, the list of edges is built by randomly visiting vertices in the graph and using the metric to select one incident edge using the edge metric input.

It is worth noting that Abou-Rjeili et al. fixed the values of the maximum vertex weight ratio and the maximum contraction ratio to 0.05 and 0.5, respectively. This work instead chooses to allow evolution to attempt to optimize the values for these parameters.

5.1.3. Edge Metric Nodes. The following metrics were chosen because they can be calculated without increasing the overall complexity of the multi-level partitioning algorithm. Note that these values can be combined and manipulated using the operations listed in Section 5.1.4.
Edge degree: Returns the sum of the degrees of the vertices incident to the edge.
Edge weight: The weight of the edge.
Edge node weight: The sum of the weights of the vertices incident to the edge.
Edge core number: The sum of the core numbers of the vertices incident to the edge. For a description of node core numbers, see [26].

5.1.4. Math Operators. Basic addition, subtraction, multiplication, division, modulus, exponentiation, additive and multiplicative inverse. Some of these operators require special attention due to the stochastic nature of the process. For example, if division would produce a division by zero exception, it instead divides by a value very close to zero.

5.1.5. Numerical Constants. These nodes return a constant value that is randomly chosen once during initialization.
Ratio constant: Randomly selected value from $\{0.1, 0.2, \ldots, 1.0\}$.
Probability constant: Randomly selected value from $\{0.001, 0.01, 0.02, 0.025, 0.05, 0.1, 0.2, 0.25, 0.5, 1.0\}$.
The possible values for these nodes were chosen to allow the GP to recreate and expand upon the functionality of existing heuristics.

5.1.6. Boolean Nodes. True and false constant nodes, as well as a node that randomly returns true according to an input probability.

5.1.7. Partition Nodes. These nodes take a graph as input and return an assignment of vertices into partitions.
Random bisection: Randomly assigns vertices into two partitions. The order of the vertices is randomized and then iterated through. Vertices are added to the first partition until the partition exceeds half the total vertex weight. The remaining vertices are assigned to the second partition.
GGP bisection: Graph is partitioned using graph growth partitioning, as described in Section 2.1.2.
GGGP bisection: Graph is partitioned using greedy graph growth partitioning, as described in Section 2.1.2.
Spectral bisection: Spectral partitioning is used to bisect the graph (see [16]).
KL bisection: Graph is bisected using the Kernighan-Lin algorithm (see [17]).

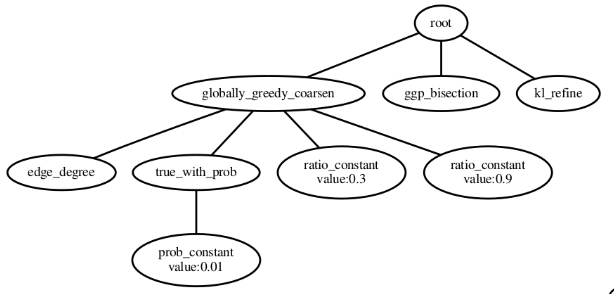
5.1.8. Uncoarsening Nodes. Two uncoarsening nodes are used, which only differ in the refinement method they employ.
KL refinement: Kernighan-Lin refinement, as described in Section 2.1.3.
Greedy refinement: Greedy partition refinement, as described in Section 2.1.3.

6. Experiment

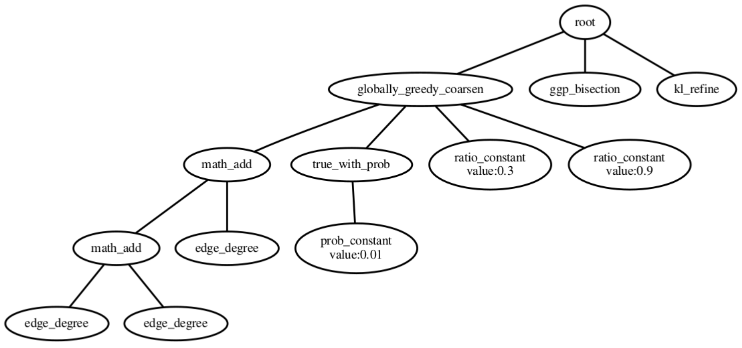
The GP approach is used to evolve multi-level partition algorithms for three types of graphs. The first two applications are targeted at partitioning graphs from two specific random graph models: Erdös-Rényi [27], and Barabási-Albert [28]. These models, which are known to have different degree distributions, were selected to illustrate the effectiveness of algorithm specialization. In order to demonstrate real-world applicability, the third application targets graphs created from actual network data released by Los Alamos National Laboratory (LANL) [29]. One month of the network data set was modeled as a bipartite graph with 9,924 user vertices, 14,822 computer vertices, and 106,693 authentication edges. Subgraphs were created by inducing the set of vertices visited by a random walk of the total graph.

During solution evaluation, a set of the application specific random graphs are generated, each with 100 vertices. The size of the graphs are kept small because a large number of these graphs will need to be generated during the full course of evolution. The candidate solution being evaluated is used to partition each graph in the set, and the solution's quality is determined by the average cost of the partitions produced. By using multiple randomly generated graphs for each evaluation, evolution encourages solutions which are good at partitioning that class of graphs instead of overspecializing on a small, fixed set of specific graphs.

A separate set of thirty verification graphs are generated to evaluate the performance of partition algorithm solutions



(a) Erdös-Rényi



(b) Barabási-Albert

Figure 4. Example evolved partition algorithms for both random graph model applications.

from the final population of each GP run. For comparison, the verification graphs were also partitioned using standard spectral partitioning as well as the k-way partitioning function of the METIS software library. To examine the extent to which the evolved solutions are specialized for their target graph type, they are also used to partition the other graph types and their relative performance is compared. The cost of the partitions produced by each method are compared pairwise using Wilcoxon rank-sum tests at a 95% significance. Finally, evolved solutions are used to partition graphs of various sizes to demonstrate their scalability compared to the general purpose partition solutions.

TABLE 2. Relative average partition cost

Method	$\mathbf{E_{ER}}$	$\mathbf{E_{BA}}$	$\mathbf{E_{LANL}}$	METIS	SP
$\mathbf{E_{ER}}$	0.0	-0.06	-0.38	-0.06	-4.55
$\mathbf{E_{BA}}$	-0.53	0.0	-0.12	-0.72	-0.92
$\mathbf{E_{LANL}}$	-0.48	-0.10	0.0	-2.97	-3.90

Each value is the average cost of partitions produced by the method evolved for that application, minus the average cost of partitions produced by the partitioner listed at the top of the column. A negative value indicates that the evolved solution produces a lower average partition cost, with shaded cells indicating the difference is statistically significant at the $\alpha = 0.05$ level.

7. Results

The parse tree representation of two sample evolved solutions can be seen in Figure 4, one for each of the random graph model applications. The evolved solutions for the LANL network application tend to be far more complex, and as a result, are too large to be included.

See Table 2 for the relative performance comparison of the evolved partitioning methods as well as METIS and spectral partitioning (SP). $\mathbf{E_{ER}}$, $\mathbf{E_{BA}}$, and $\mathbf{E_{LANL}}$, refer to the solutions evolved to target the Erdös-Rényi, Barabási-Albert, and LANL network graph sets, respectively. Each row compares the average partition cost of the method evolved for that application against each of the other partition algorithms. A negative value indicates that the evolved solution produces a lower average partition cost than the method indicated for that column. A shaded cell indicates the difference is statistically significant at the $\alpha = 0.05$ level.

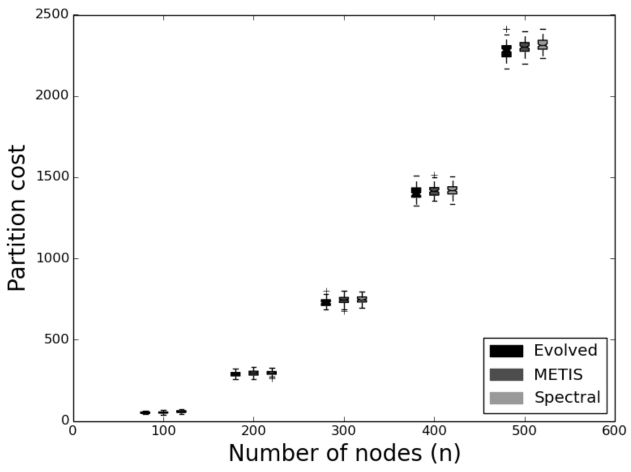
It is encouraging to see that for each graph type, the partitioner evolved for that type produces the lowest average cost, even if these differences are not always statistically significant. The evolved methods consistently outperform the traditional spectral partition method. Compared to the off-the-shelf METIS software, the evolved solutions for the Barabási-Albert and LANL network graphs are also statistically superior. A notable exception is the evolved partition algorithm for the Erdös-Rényi application. The inability to statistically outperform METIS might be a result of the high randomness inherent to the Erdös-Rényi model, which might not consistently produce any graph characteristics that can be exploited during evolution.

The comparisons between the evolved solutions do suggest some amount of specialization has taken place, but the performance difference is not always statistically significant. $\mathbf{E_{BA}}$ and $\mathbf{E_{LANL}}$ do significantly outperform the $\mathbf{E_{ER}}$ on their targeted graph types. However, $\mathbf{E_{BA}}$ and $\mathbf{E_{LANL}}$ perform very similarly when interchanged. This could indicate that the graphs created from the LANL network data resemble graphs generated by the Barabási-Albert model; both evolved solutions might be taking advantage of characteristics common in both graphs.

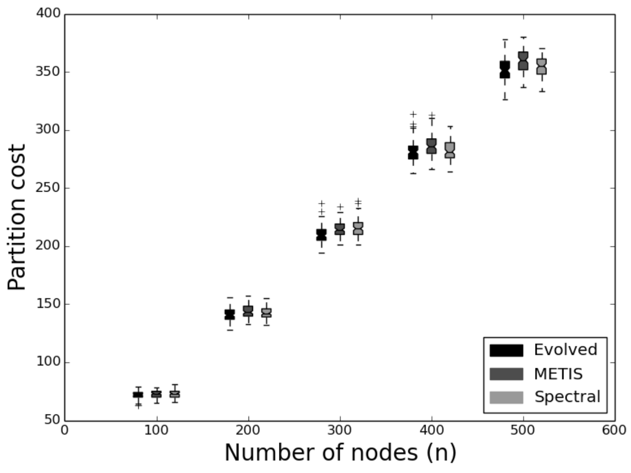
Figure 5 shows the relative cost of partitioning graphs using the evolved partition solutions as well as METIS and spectral partitioning as the size of the graph grows. For each plot, the "Evolved" label refers to the partitioner that was evolved specifically for that graph type. Despite the fact that the solutions are evolved to target graphs with 100 vertices, the evolved partition algorithms still consistently outperform METIS and spectral partitioning as the size of the graphs increase.

8. Conclusion

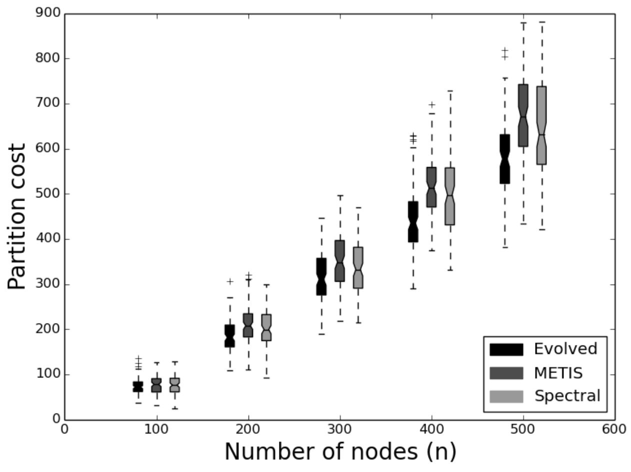
Graph partitioning is a fundamental computer science problem with applications in many domain areas. Multi-



(a) Erdös-Rényi



(b) Barabási-Albert



(c) LANL network

Figure 5. Cost of partitioning graphs of various sizes for each graph type.

level partitioning is a widely used state-of-the-art approach to efficiently approximate optimal partitioning. Although there are a variety of multi-level partitioning algorithms available, most are intended to serve as general purpose solutions. Some work has been done attempting to exploit common graph characteristics through the manual development of tailored solutions, but this tedious process must be repeated for each application. Even if a good set of specialized partition algorithms were available, it might not contain an adequate solution to an entirely new graph application.

This work addresses this limitation by employing genetic programming to automatically generate novel multi-level graph partitioning algorithms tailored to each application. The potential of this approach is demonstrated by evolving a set of algorithms, each tailored to perform well on graphs from a different source: two traditional random graph models and real world computer network subsets. These specialized solutions outperform traditional partitioning methods on their target graph types, and continue to do well as the size of the graphs increases. The platform implemented in this work can be quickly reapplied to any new application domains as they arise instead of relying on general purpose, off-the-shelf solutions.

9. Future Work

An obvious continuation of this work would be to include the consideration of execution time. Instead of focusing on reducing the cost of the partitions produced, evolved partitioners could instead aim to reduce the time needed to partition certain types of graphs for time-sensitive applications. This could be accomplished by using a multi-objective optimizer that presents the end user with a set of solutions with various trade-off values in terms of partition quality and execution time. Evaluating execution time as an objective will require the implementation of automated code generation of evolved algorithms, since currently the solutions can only be applied within the GP. This limitation adds significant computational overhead and makes a direct efficiency comparison difficult.

The functionality of the current implementation is limited to evolving balanced bisection algorithms. Although the imbalance threshold is configurable, new features would have to be incorporated to allow for specifying additional partition weight restrictions. While any number of partitions can be achieved using repeated applications of a bisection algorithm, partition heuristics evolved for a specific number of partitions might improve upon the efficiency of this process.

This work demonstrates the potential for employing GP to improve graph algorithm performance. A similar process could be used to evolve novel solutions to other problems in the graph theory domain, such as community detection [30] or efficient calculation of vertex centrality values in dynamic graphs [31].

Previous work has demonstrated that increasing the granularity of the set of primitive operations can increase the

GP evolution time required to reach convergence, but also improve the overall final solution quality [32]. The primitive set granularity in this work could be increased by relaxing the assumed structure of the multi-level partition algorithm. This basic structure is currently implemented in the root and coarsen nodes. These primitives could be disassembled into a set of lower-level primitives, allowing for more diverse and expressive algorithm representation.

References

[1] H. Meyerhenke, "Shape optimizing load balancing for MPI-parallel adaptive numerical simulations," *Proceedings of the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering*, pp. 67–82, 2013.

[2] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer Science & Business Media, 2011.

[3] B. Peng, L. Zhang, and D. Zhang, "A survey of graph theoretical approaches to image segmentation," *Pattern Recognition*, vol. 46, no. 3, pp. 1020–1038, 2013.

[4] H. Li, G. W. Rosenwald, J. Jung, and C.-C. Liu, "Strategic Power Infrastructure Defense," *Proceedings of the IEEE*, vol. 93, no. 5, pp. 918–933, 2005.

[5] A. E. Feldmann, "Fast Balanced Partitioning is Hard Even on Grids and Trees," in *Proceedings of the 37th International Conference on Mathematical Foundations of Computer Science*, ser. MFCS'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 372–382. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32589-2_34

[6] I. Safro, P. Sanders, and C. Schulz, "Advanced Coarsening Schemes for Graph Partitioning," *J. Exp. Algorithmics*, vol. 19, pp. 2.2:1–2.2:24, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2670338

[7] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997

[8] C. Walshaw and M. Cross, "JOSTLE: Parallel Multilevel Graph-Partitioning Software–An Overview," *Mesh partitioning techniques and domain decomposition techniques*, pp. 27–58, 2007.

[9] C. Chevalier and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering," *CoRR*, vol. abs/0907.1375, 2009. [Online]. Available: http://arxiv.org/abs/0907.1375

[10] H. Meyerhenke, B. Monien, and T. Sauerwald, "A New Diffusion-based Multilevel Algorithm for Computing Graph Partitions of Very High Quality," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–13.

[11] A. Abou-Rjeili and G. Karypis, "Multilevel Algorithms for Partitioning Power-law Graphs," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 10–pp.

[12] P. D. Hough and P. J. Williams, "Modern Machine Learning for Automatic Optimization Algorithm Selection," in *Proceedings of the INFORMS Artificial Intelligence and Data Mining Workshop*, 2006, pp. 1–6.

[13] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT press, 1992, vol. 1.

[14] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, 2013.

[15] H. H. Hoos, *Automated Algorithm Configuration and Parameter Tuning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 37–71. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21434-9_3

[16] W. E. Donath and A. J. Hoffman, "Algorithms for Partitioning of Graphs and Computer Logic Based on Eigenvectors of Connections Matrices," *IBM Technical Disclosure Bulletin*, vol. 15, 1972.

[17] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[18] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer, 2003, vol. 53.

[19] J. Kim, I. Hwang, Y.-H. Kim, and B.-R. Moon, "Genetic Approaches for Graph Partitioning: A Survey," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2011, pp. 473–480.

[20] P. Sanders and C. Schulz, "High Quality Graph Partitioning," *Graph Partitioning and Graph Clustering*, vol. 588, no. 1, 2012.

[21] A. J. Soper, C. Walshaw, and M. Cross, "A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning," *Journal of Global Optimization*, vol. 29, no. 2, pp. 225–241, 2004.

[22] U. Benlic and J.-K. Hao, "A Multilevel Memetic Approach for Improving Graph k-Partitions," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 5, pp. 624–642, Oct. 2011.

[23] A. Bailey, M. Ventresca, and B. Ombuki-Berman, "Genetic Programming for the Automatic Inference of Graph Models for Complex Networks," *Evolutionary Computation, IEEE Transactions on*, vol. 18, no. 3, pp. 405–419, 2014.

[24] K. R. Harrison, "Network Similarity Measures and Automatic Construction of Graph Models using Genetic Programming," 2014.

[25] D. J. Montana, "Strongly Typed Genetic Programming," *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, Jun. 1995. [Online]. Available: http://dx.doi.org/10.1162/evco.1995.3.2.199

[26] S. B. Seidman, "Network Structure and Minimum Degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[27] P. Erdös and A. Rényi, "On Random Graphs, I," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[28] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[29] A. Hagberg, A. Kent, N. Lemons, and J. Neil, "Credential Hopping in Authentication Graphs," in *2014 International Conference on Signal-Image Technology Internet-Based Systems (SITIS)*. IEEE Computer Society, Nov. 2014.

[30] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009. [Online]. Available: http://dx.doi.org/10.1080/15427951.2009.10129177

[31] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung, "QUBE: A Quick Algorithm for Updating Betweenness Centrality," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 351–360. [Online]. Available: http://doi.acm.org/10.1145/2187836.2187884

[32] M. A. Martin and D. R. Tauritz, "Hyper-Heuristics: A Study On Increasing Primitive-Space," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO Companion '15, 2015, pp. 1051–1058.