Abstract

Computing high-quality graph partitions is a challenging problem with numerous applications. In this paper, we present a novel meta-heuristic for the balanced graph partitioning problem. Our approach is based on integer linear programs that solve the partitioning problem to optimality. However, since those programs typically do not scale to large inputs, we adapt them to heuristically improve a given partition. We do so by defining a much smaller model that allows us to use symmetry breaking and other techniques that make the approach scalable. For example, in Walshaw’s well-known benchmark tables we are able to improve roughly half of all entries when the number of blocks is high.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms, Theory of computation → Integer programming

Keywords and phrases Graph Partitioning, Integer Linear Programming

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.4

Related Version A full version of the paper is available at https://arxiv.org/abs/1802.07144.

Funding The research leading to these results has received funding from the European Research Council under the European Community’s Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement No. 340506. Moreover, the authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (http://www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (https://www.lrz.de).

1 Introduction

Balanced graph partitioning is an important problem in computer science and engineering with an abundant amount of application domains, such as VLSI circuit design, data mining and distributed systems [38]. It is well known that this problem is NP-complete [8] and that no approximation algorithm with a constant ratio factor exists for general graphs unless $P=NP$ [8]. Still, there is a large amount of literature on methods (with worst-case exponential time) that solve the graph partitioning problem to optimality. This includes methods dedicated to the bipartitioning case [3, 4, 12, 13, 14, 15, 24, 21, 30, 39] and some methods that solve the general graph partitioning problem [16, 40]. Most of these methods rely on the branch-and-bound framework [28]. However, these methods can typically solve
only very small problems as their running time grows exponentially, or if they can solve large bipartitioning instances using a moderate amount of time \([12, 13]\), the running time highly depends on the bisection width of the graph. Methods that solve the general graph partitioning problem \([16, 40]\) have huge running times for graphs with up to a few hundred vertices. Thus in practice mostly heuristic algorithms are used.

Typically the graph partitioning problem asks for a partition of a graph into \(k\) blocks of about equal size such that there are few edges between them. Here, we focus on the case when the bounds on the size are very strict, including the case of \textit{perfect balance} when the maximal block size has to equal the average block size.

Our focus in this paper is on solution quality, i.e., minimize the number of edges that run between blocks. During the past two decades there have been numerous researchers trying to improve the best graph partitions in Walshaw’s well-known partitioning benchmark \([41, 42]\). Overall there have been more than forty different approaches that participated in this benchmark. Indeed, high solution quality is of major importance in applications such as VLSI Design \([1, 2]\) where even minor improvements in the objective can have a large impact on the production costs and quality of a chip. High-quality solutions are also favorable in applications where the graph needs to be partitioned only once and then the partition is used over and over again, implying that the running time of the graph partitioning algorithms is of a minor concern \([11, 18, 27, 29, 32, 31]\). Thirdly, high-quality solutions are even important in areas in which the running time overhead is paramount \([41]\), such as finite element computations \([37]\) or the direct solution of sparse linear systems \([20]\). Here, high-quality graph partitions can be useful for benchmarking purposes, i.e., measuring how much more running time can be saved by higher quality solutions.

In order to compute high-quality solutions, state-of-the-art local search algorithms exchange vertices between blocks of the partition trying to decrease the cut size while also maintaining balance. This highly restricts the set of possible improvements. Recently, we introduced new techniques that relax the balance constraint for vertex movements but globally maintain balance by combining multiple local searches \([36]\). This was done by reducing this combination problem to finding negative cycles in a graph. In this paper, we extend the neighborhood of the combination problem by employing integer linear programming. This enables us to find even more complex combinations and hence to further improve solutions. More precisely, our approach is based on integer linear programs that solve the partitioning problem to optimality. However, out of the box those programs typically do not scale to large inputs, in particular because the graph partitioning problem has a very large amount of symmetry – given a partition of the graph, each permutation of the block IDs gives a solution having the same objective and balance. Hence, we adapt the integer linear program to improve a given input partition. We do so by defining a much smaller graph, called \textit{model}, and solve the graph partitioning problem on the model to optimality by the integer linear program. More specifically, we select vertices close to the cut of the given input partition for potential movement and contract all remaining vertices of a block into a single vertex. A feasible partition of this model corresponds to a partition of the input graph having the same balance and objective. Moreover, this model enables us to use symmetry breaking, which allows us to scale to much larger inputs. To make the approach even faster, we combine it with initial bounds on the objective provided by the input partition, as well as providing the input partition to the integer linear program solver. Overall, we arrive at a system that is able to improve more than half of all entries in Walshaw’s benchmark when the number of blocks is high.

The rest of the paper is organized as follows. We begin in Section 2 by introducing basic concepts. After presenting some related work in Section 3 we outline the integer linear
A. Henzinger, A. Noe, and C. Schulz

program as well as our novel local search algorithm in Section 4. Here, we start by explaining
the very basic idea that allows us to find combinations of simple vertex movements. We
then explain our strategies to improve the running time of the solver and strategies to
select vertices for movement. A summary of extensive experiments done to evaluate the
performance of our algorithms is presented in Section 5. Finally, we conclude in Section 6.

2 Preliminaries

2.1 Basic concepts

Let \( G = (V = \{0, \ldots, n-1\}, E) \) be an undirected graph. We consider positive, real-valued
degree and vertex weight functions \( \omega \) resp. \( c \) and extend them to sets, i.e., \( \omega(E) := \sum_{x \in E} \omega(x) \)
and \( c(V') := \sum_{x \in V'} c(x) \). Let \( N(v) := \{ u : \{v, u\} \in E \} \) denote the neighbors of \( v \). The
degree of a vertex \( v \) is \( d(v) := |N(v)| \). A vertex is a boundary vertex if it is incident to at
least one vertex in a different block. We are looking for disjoint blocks of vertices \( V_1, \ldots, V_k \)
that partition \( V \); i.e., \( V_1 \cup \cdots \cup V_k = V \). The balancing constraint demands that each block
has weight \( c(V_i) \leq (1 + \epsilon) \lfloor \frac{c(V)}{k} \rfloor =: L_{\text{max}} \) for some imbalance parameter \( \epsilon \). We call a block \( V_i \)
overloaded if its weight exceeds \( L_{\text{max}} \). The objective of the problem is to minimize the total
\( \omega(E \cap \bigcup_{i<j} V_i \times V_j) \) subject to the balancing constraints. We define the gain of a vertex
as the maximum decrease in the cut value when moving it to a different block.

3 Related Work

There has been a huge amount of research on graph partitioning and we refer the reader to the surveys given in [6, 9, 37, 43] for most of the material. Here, we focus on issues closely related to
our main contributions. All general-purpose methods that are able to obtain good partitions
for large real-world graphs are based on the multi-level principle. Well-known software
packages based on this approach include Jostle [43], KaHIP [34], Metis [25] and Scotch [33].

Chris Walshaw’s well-known benchmark archive has been established in 2001 [41, 42].
Overall it contains 816 instances (34 graphs, 4 values of imbalance, and 6 values of \( k \)).
In this benchmark, the running time of the participating algorithms is not measured or
reported. Submitted partitions will be validated and added to the archive if they improve on
a particular result. This can either be an improvement in the number of cut edges or, if they
match the current best cut size, an improvement in the weight of the largest block. Most
entries in the benchmark have as of Feb. 2018 been obtained by Galinier et al. [19] (more
precisely an implementation of that approach by Frank Schneider), Hein and Seitzer [22] and
the Karlsruhe High-Quality Graph Partitioning (KaHIP) framework [36]. More precisely,
Galinier et al. [19] use a memetic algorithm that is combined with tabu search to compute
solutions and Hein and Seitzer [22] solve the graph partitioning problem by providing tight
relaxations of a semi-definite program into a continuous problem.

The Karlsruhe High-Quality Graph Partitioning (KaHIP) framework implements many
different algorithms, for example flow-based methods and more-localized local searches, as
well as several coarse-grained parallel and sequential meta-heuristics. KaBaPE [36] is a
coarse-grained parallel evolutionary algorithm, i.e. each processor has its own population
(set of partitions) and a copy of the graph. After initially creating the local population, each
processor performs multi-level combine and mutation operations on the local population.
This is combined with a meta-heuristic that combines local searches that individually violate
the balance constraint into a more global feasible improvement. For more details, we refer
the reader to [36].
Local Search based on Integer Linear Programming

We now explain our algorithm that combines integer linear programming and local search. We start by explaining the integer linear program that can solve the graph partitioning problem to optimality. However, out-of-the-box this program does not scale to large inputs, in particular because the graph partitioning problem has a very large amount of symmetry. Thus, we reduce the size of the graph by first computing a partition using an existing heuristic and based on it collapsing parts of the graph. Roughly speaking, we compute a small graph, called model, in which we only keep a small number of selected vertices for potential movement and perform graph contractions on the remaining ones. A partition of the model corresponds to a partition of the input network having the same objective and balance. The computed model is then solved to optimality using the integer linear program. As we will see this process enables us to use symmetry breaking in the linear program, which in turn drastically speeds up computation times.

4.1 Integer Linear Program for the Graph Partitioning Problem

We now introduce a generalization of an integer linear program formulation for balanced bipartitioning [7] to the general graph partitioning problem. First, we introduce binary decision variables for all edges and vertices of the graph. More precisely, for each edge $e = \{u, v\} \in E$, we introduce the variable $e_{uv} \in \{0, 1\}$ which is one if $e$ is a cut edge and zero otherwise. Moreover, for each $v \in V$ and block $k$, we introduce the variable $x_{v,k} \in \{0, 1\}$ which is one if $v$ is in block $k$ and zero otherwise. Hence, we have a total of $|E| + k|V|$ variables. We use the following constraints to ensure that the result is a valid $k$-partition:

\begin{align*}
\forall \{u, v\} \in E, \forall k : e_{uv} &\geq x_{u,k} - x_{v,k} \quad (1) \\
\forall \{u, v\} \in E, \forall k : e_{uv} &\geq x_{v,k} - x_{u,k} \quad (2) \\
\forall k : \sum_{v \in V} x_{v,k} c(v) &\leq L_{\text{max}} \quad (3) \\
\forall v \in V : \sum_{k} x_{v,k} &= 1 \quad (4)
\end{align*}

The first two constraints ensure that $e_{uv}$ is set to one if the vertices $u$ and $v$ are in different blocks. For an edge $\{u, v\} \in E$ and a block $k$, the right-hand side in this equation is one if one of the vertices $u$ and $v$ is in block $k$ and the other one is not. If both vertices are in the same block then the right-hand side is zero for all values of $k$. Hence, the variable can either be zero or one in this case. However, since the variable participates in the objective function and the problem is a minimization problem, it will be zero in an optimum solution.

The third constraint ensures that the balance constraint is satisfied for each partition. And finally, the last constraint ensures that each vertex is assigned to exactly one block. To sum up, our program has $2k|E| + k + |V|$ constraints and $k \cdot (6|E| + 2|V|)$ non-zeros. Since we want to minimize the weight of cut edges, the objective function of our program is written as:

$$\min \sum_{\{u, v\} \in E} e_{uv} \cdot \omega(\{u, v\})$$

(5)
4.2 Local Search

The graph partitioning problem has a large amount of symmetry – each permutation of the block IDs gives a solution with equal objective and balance. Hence, the integer linear program described above will scan many branches that contain essentially the same solutions so that the program does not scale to large instances. Moreover, it is not immediately clear how to improve the scalability of the program by using symmetry breaking or other techniques.

Our goal in this section is to develop a local search algorithm using the integer linear program above. Our main idea is to contract vertices “that are far away” from the cut of the partition. In other words, we want to keep vertices close to the cut and contract all remaining vertices into one vertex for each block of the input partition. This ensures that a partition of the contracted graph yields a partition of the input graph with the same objective and balance. Hence, we apply the integer linear program to the model and solve the partitioning problem on it to optimality. Note, however, that due to the performed contractions this does not imply an optimal solution on the input graph.

We now outline the details of the algorithm. Our local algorithm has two inputs, a graph $G$ and a partition $V_1, \ldots, V_k$ of its vertices. For now assume that we have a set of vertices $K \subset V$ which we want to keep in the coarse model, i.e. a set of vertices which we do not want to contract. We outline in Section 4.4 which strategies we have to select the vertices $K$. For the purpose of contraction we define $k$ sets $V_i := V_i \setminus K$. We obtain our coarse model by contracting each of these vertex sets. The contraction of a vertex set $V_i$ works as follows: the set of vertices is contracted into a single vertex $\mu_i$. The weight of $\mu_i$ is set to the sum of the weight of all vertices in the set that is contracted. There is an edge between two vertices $\mu_i$ and $v$ in the contracted graph if there is an edge between a vertex of the set and $v$ in the original graph $G$. The weight of an edge $(\mu_i, v)$ is set to the sum of the weight of edges that run between the vertices of the set and $v$. After all contractions have been performed the coarse model contains $k + |K|$ vertices, and potentially much less edges than the input graph. Figure 1 gives an abstract example of our model.

There are two things that are important to see: first, due to the way we perform contraction, the given partition of the input network yields a partition of our coarse model that has the same objective and balance simply by putting $\mu_i$ into block $i$ and keeping the block of the input for the vertices in $K$. Moreover, if we compute a new partition of our coarse model, we can build a partition in the original graph with the same properties by putting the vertices $V_i$ into the block of their coarse representative $\mu_i$ together with the vertices of $K$ that are in this block. Hence, we can solve the integer linear program on the coarse model to compute a partition for the input graph. After the solver terminates, i.e. found an optimum solution of our mode or has reached a predefined time limit $\mathcal{T}$, we transfer the best solution to the original graph. Note that the latter is possible since an integer linear program solver typically computes intermediate solutions that may not be optimal.

4.3 Optimizations

Independent of the vertices $K$ that are selected to be kept in the coarse model, the approach above allows us to define optimizations to solve our integer linear program faster. We apply four strategies: (i) symmetry breaking, (ii) providing a start solution to the solver, (iii) add the objective of the input as a constraint as well as (iv) using the parallel solving facilities of the underlying solver. We outline the first three strategies in greater detail:
**Symmetry Breaking.** If the set $\mathcal{K}$ is small, then the solver will find a solution much faster. Typically, our algorithms selects the vertices $\mathcal{K}$ such that $c(\mu_i) + c(\mu_j) > L_{\text{max}}$. In other words, no two contracted vertices can be clustered in one block. We can use this to break symmetry in our integer linear programming by adding constraints that fix the block of $\mu_i$ to block $i$, i.e. we set $x_{\mu_i,i} = 1$ and $x_{\mu_i,j} = 0$ for $i \neq j$. Moreover, for those vertices we can remove the constraint which ensures that the vertex is assigned to a single unique block – since we assigned those vertices to a block using the new additional constraints.

**Providing a Start Solution to the Solver.** The integer linear program performs a significant amount of work in branches which correspond to solutions that are worse than the input partitioning. Only very few - if any - solutions are better than the given partition. However, we already know a fairly good partition (the given partition from the input) and give this partition to the solver by setting according initial values for all variables. This ensures that the integer linear program solver can omit many branches and hence speeds up the time needed to solve the integer linear program.

**Solution Quality as a Constraint.** Since we are only interested in improved partitions, we can add an additional constraint that disallows solutions which have a worse objective than the input partition. Indeed, the objective function of the linear program is linear, and hence the additional constraint is also linear. Depending on the objective value, this reduces the number of branches that the linear program solver needs to look at. However, note that this comes at the cost of an additional constraint that needs to be evaluated.

### 4.4 Vertex Selection Strategies

The algorithm above works for different vertex sets $\mathcal{K}$ that should be kept in the coarse model. There is an obvious trade-off: on the one hand, the set $\mathcal{K}$ should not be too large, otherwise the coarse model would be large and hence the linear programming solver needs a large amount of time to find a solution. On the other hand, the set should also not be too small, since this restricts the amount of possible vertex movements, and hence the approach is unlikely to find an improved solution. We now explain different strategies to select the vertex set $\mathcal{K}$. In any case, while we add vertices to the set $\mathcal{K}$, we compute the number of non-zeros in the corresponding ILP. We stop to add vertices when the number of non-zeros in the corresponding ILP is larger than a parameter $N$.

**Vertices Close to Input Cut.** The intuition of the first strategy, Boundary, is that changes or improvements of the partition will occur reasonable close to the input partition. In this simple strategy our algorithm tries to use all boundary vertices as the set $\mathcal{K}$. In order to adhere to the constraint on the number of non-zeros in the ILP, we add the vertices
of the boundary uniformly at random and stop if the number of non-zeros \( N \) is reached. If the algorithm managed to add all boundary vertices whilst not exceeding the specified number of non-zeros, we do the following extension: we perform a breadth-first search that is initialized with a random permutation of the boundary vertices. All additional vertices that are reached by the BFS are added to \( K \). As soon as the number of non-zeros \( N \) is reached, the algorithm stops.

**Start at Promising Vertices.** Especially for high values of \( k \) the boundary contains many vertices. The \textit{Boundary} strategy quickly adds a lot of random vertices while ignoring vertices that have high gain. However, note that even in good partitions it is possible that vertices with positive gain exist but cannot be moved due to the balance constraint.

Hence, our second strategy, \textit{Gain}_\( \rho \), tries to fix this issue by starting a breadth-first search initialized with only high gain vertices. More precisely, we initialize the BFS with each vertex having gain \( \geq \rho \) where \( \rho \) is a tuning parameter. Our last strategy, \textit{TopVertices}_\( \delta \), starts by sorting the boundary vertices by their gain. We break ties uniformly at random. Vertices are then traversed in decreasing order (highest gain vertices first) and for each start vertex \( v \) our algorithm adds all vertices with distance \( \leq \delta \) to the model. The algorithm stops as soon as the number of non-zeros exceeds \( N \).

Early gain-based local search heuristics for the \( \epsilon \)-balanced graph partitioning problem searched for pairwise swaps with positive gain [17, 26]. More recent algorithms generalized this idea to also search for cycles or paths with positive total gain [36]. An important advantage of our new approach is that we solve the combination problem to optimality, i.e. our algorithm finds the best combination of vertex movements of the vertices in \( K \) w.r.t. to the input partition of the original graph. Therefore we can also find more complex optimizations that cannot be reduced to positive gain cycles and paths.

5 Experiments

5.1 Experimental Setup and Methodology

We implemented the algorithms using C++-17 and compiled all codes using g++-7.2.0 with full optimization (-O3). We use Gurobi 7.5.2 as an ILP solver and always use its parallel version. All of our experiments were conducted on a machine with two Haswell Xeon E5-2697 v3 processors. The machine has 28 cores at 2.6GHz as well as 64GB of main memory and runs the SUSE Linux Enterprise Server (SLES) operating system. Unless otherwise mentioned, our approach uses the shared-memory parallel variant of Gurobi using all 28 cores. In general, we perform five repetitions per instance and report the average running time as well as cut. Unless otherwise mentioned, we use a time limit for the integer linear program. When the time limit is passed, the integer linear program solver outputs the best solution that has currently been discovered. This solution does not have to be optimal. Note that we do not perform experiments with Metis [25] and Scotch [33] in here, since previous papers, e.g. [34, 35], have already shown that solution quality obtained is much worse than results achieved in the Walshaw benchmark. When averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the final score.

**Performance Plots.** These plots relate the fastest running time to the running time of each other ILP-based local search algorithm on a per-instance basis. For each algorithm, these ratios are sorted in increasing order. The plots show the ratio \( t_{\text{best}} / t_{\text{algorithm}} \) on the y-axis to highlight the instances in which each algorithm performs badly. For plots in which we measure solution quality, the y-axis shows the ratio \( \text{cut}_{\text{best}} / \text{cut}_{\text{algorithm}} \). A point close to
Table 1 Basic properties of the benchmark instances.

<table>
<thead>
<tr>
<th>Graph</th>
<th>$n$</th>
<th>$m$</th>
<th>Graph</th>
<th>$n$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walshaw Graphs (Set B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add20</td>
<td>2395</td>
<td>7462</td>
<td>wing</td>
<td>62032</td>
<td>$\approx$ 121K</td>
</tr>
<tr>
<td>data</td>
<td>2851</td>
<td>15093</td>
<td>brack2</td>
<td>62631</td>
<td>$\approx$ 366K</td>
</tr>
<tr>
<td>3dlt</td>
<td>4720</td>
<td>13722</td>
<td>finan512</td>
<td>74752</td>
<td>$\approx$ 261K</td>
</tr>
<tr>
<td>uk</td>
<td>4824</td>
<td>6837</td>
<td>fe_tooth</td>
<td>78136</td>
<td>$\approx$ 452K</td>
</tr>
<tr>
<td>add32</td>
<td>4960</td>
<td>9462</td>
<td>fe_rotor</td>
<td>99617</td>
<td>$\approx$ 662K</td>
</tr>
<tr>
<td>bcsstk33</td>
<td>8738</td>
<td>$\approx$ 291K</td>
<td>598a</td>
<td>110971</td>
<td>$\approx$ 741K</td>
</tr>
<tr>
<td>whitaker3</td>
<td>9800</td>
<td>28989</td>
<td>fe_ocean</td>
<td>143437</td>
<td>$\approx$ 409K</td>
</tr>
<tr>
<td>crack</td>
<td>10240</td>
<td>30380</td>
<td>144</td>
<td>144649</td>
<td>$\approx$ 1.1M</td>
</tr>
<tr>
<td>wing_nodal</td>
<td>10937</td>
<td>75488</td>
<td>wave</td>
<td>156317</td>
<td>$\approx$ 1.1M</td>
</tr>
<tr>
<td>fe_4elt2</td>
<td>11143</td>
<td>32818</td>
<td>m14b</td>
<td>214765</td>
<td>$\approx$ 1.7M</td>
</tr>
<tr>
<td>vibrobox</td>
<td>12328</td>
<td>$\approx$ 165K</td>
<td>auto</td>
<td>448695</td>
<td>$\approx$ 3.3M</td>
</tr>
<tr>
<td>bcsstk29</td>
<td>13992</td>
<td>$\approx$ 302K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4elt</td>
<td>15606</td>
<td>45878</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fe_sphere</td>
<td>16386</td>
<td>49152</td>
<td>delaunay_n15</td>
<td>32768</td>
<td>$\approx$ 98.274</td>
</tr>
<tr>
<td>cti</td>
<td>16840</td>
<td>48232</td>
<td>rgg_15</td>
<td>32768</td>
<td>$\approx$ 160K</td>
</tr>
<tr>
<td>memplus</td>
<td>17758</td>
<td>54196</td>
<td>2cubes_sphere</td>
<td>101492</td>
<td>$\approx$ 772K</td>
</tr>
<tr>
<td>cs4</td>
<td>22499</td>
<td>43858</td>
<td>cfd2</td>
<td>123440</td>
<td>$\approx$ 1.5M</td>
</tr>
<tr>
<td>bcsstk30</td>
<td>28924</td>
<td>$\approx$ 10.0M</td>
<td>boneS01</td>
<td>127224</td>
<td>$\approx$ 3.3M</td>
</tr>
<tr>
<td>bcsstk31</td>
<td>35588</td>
<td>$\approx$ 572K</td>
<td>Dabcova3</td>
<td>146809</td>
<td>$\approx$ 1.7M</td>
</tr>
<tr>
<td>fe_pwt</td>
<td>36519</td>
<td>$\approx$ 144K</td>
<td>G2_circuit</td>
<td>150102</td>
<td>$\approx$ 288K</td>
</tr>
<tr>
<td>bcsstk32</td>
<td>44609</td>
<td>$\approx$ 985K</td>
<td>thermal2</td>
<td>1227087</td>
<td>$\approx$ 3.7M</td>
</tr>
<tr>
<td>fe_body</td>
<td>45087</td>
<td>$\approx$ 163K</td>
<td>as365</td>
<td>3799275</td>
<td>$\approx$ 11.4M</td>
</tr>
<tr>
<td>t60k</td>
<td>60005</td>
<td>89440</td>
<td>adaptive</td>
<td>6815744</td>
<td>$\approx$ 13.6M</td>
</tr>
</tbody>
</table>

Instances. We perform experiments on two sets of instances. Set $A$ is used to determine the performance of the integer linear programming optimizations and to tune the algorithm. We obtained these instances from the Florida Sparse Matrix collection [10] and the 10th DIMACS Implementation Challenge [5] to test our algorithm. Set $B$ are all graphs from ChrisWalshaw’s graph partitioning benchmark archive [41, 42]. This archive is a collection of instances from finite-element applications, VLSI design and is one of the default benchmarking sets for graph partitioning.

Table 1 gives basic properties of the graphs from both benchmark sets. We ran the unoptimized integer linear program that solves the graph partitioning problem to optimality from Section 4.1 on the five smallest instances from the Walshaw benchmark set. With a time limit of 30 minutes, the solver has only been able to compute a solution for the graphs uk and add32 with $k = 2$. For higher values of $k$ the solver was unable to find any solution in the time limit. Even giving a starting solution does not increase the number of ILPs solved. Hence, we omit further experiments in which we run an ILP solver on the full graph.

5.2 Impact of Optimizations

We now evaluate the impact of the optimization strategies for the ILP that we presented in Section 4.3. In this section, we use the variant of our local search algorithm in which $K$ is...
Figure 2 Left: performance plot for five variants of our algorithm: Basic does not contain any optimizations; BasicSym enables symmetry breaking; BasicSymSSol additionally gives the input partitioning to the ILP solver. The two variants BSSSConst= and BSSSConst< are the same as BasicSymSSol with additional constraints: BSSSConst= has the additional constraint that the objective has to be smaller or equal to the start solution, BSSSConst< has the constraint that the solution must be better than the start solution. Right: performance of the slowest (Basic) and fastest ILPs (BasicSymSSol) depending on the number of non-zeros in the ILP.

obtained by starting depth-one breadth-first search at the 25 highest gain vertices, and set the limit on the non-zeros in the ILP to \( N = \infty \). However, due to preliminary experiments we expect the results in terms of speedup to be similar for different vertex selection strategies. To evaluate the ILP performance, we run KaFFPa using the strong preconfiguration on each of the graphs from set \( A \) using \( \epsilon = 0 \) and \( k \in \{2, 4, 8, 16, 32, 64\} \) and then use the computed partition as input to each ILP (with the different optimizations). As the optimizations do not change the objective value achieved in the ILP, we only report running times of our different approaches. We set the time limit of the ILP solver to 30 minutes.

We use five variants of our algorithm: Basic does not contain any optimizations; BasicSym enables symmetry breaking; BasicSymSSol additionally gives the input partitioning to the ILP solver. The two variants BSSSConst= and BSSSConst< are the same as BasicSymSSol with additional constraints: BSSSConst= has the additional constraint that the objective has to be smaller or equal to the start solution, BSSSConst< has the constraint that the objective value of a solution must be better than the objective value of the start solution. Figure 2 summarises the results.

In our experiments, the basic configuration reaches the time limit in 95 out of the 300 runs. Overall, enabling symmetry breaking drastically speeds up computations. On all of the instances which the Basic configuration could solve within the time limit, each other configuration is faster than the Basic configuration. Symmetry breaking speeds up computations by a factor of 41 in the geometric mean on those instances. The largest obtained speedup on those instances was a factor of 5663 on the graph adaptive for \( k = 32 \). The configuration solves all but the two instances (boneS01, \( k = 32 \)) and (Dubcov3, \( k = 16 \)) within the time limit. Additionally providing the start solution (BasicSymSSol) gives an addition speedup of 22% on average. Over the Basic configuration, the average speedup is 50 with the largest speedup being 6495 and the smallest speedup being 47%. This configuration can solve all instances within the time limit except the instance boneS01 for \( k = 32 \). Providing the objective function as a constraint (or strictly smaller constraint) does not further reduce the running time of the solver. Instead, the additional constraints even increase the running time. We attribute this to the fact that the solver has to do additional work to evaluate the constraint. We conclude that BasicSymSSol is the fastest configuration of the ILP. Hence, we use this configuration in all the following experiments. Moreover, from Figure 2 we can see that this configuration can solve most of the instance within the time
Table 2 From top to bottom: Number of improvements found by different vertex selection rules relative to the total number of instances, average running time of the strategy on the subset of instances (graph, k) in which all strategies finished within the time limit, and the relative number of instances in which the strategy computed the lowest cut. Best values are highlighted in bold.

<table>
<thead>
<tr>
<th>k</th>
<th>ρ = 0</th>
<th>ρ = −1</th>
<th>ρ = −2</th>
<th>δ = 1</th>
<th>δ = 2</th>
<th>δ = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>70%</td>
<td>70%</td>
<td>70%</td>
<td>50%</td>
<td>70%</td>
<td>70%</td>
</tr>
<tr>
<td>4</td>
<td>50%</td>
<td>60%</td>
<td>80%</td>
<td>70%</td>
<td>70%</td>
<td>70%</td>
</tr>
<tr>
<td>8</td>
<td>50%</td>
<td>60%</td>
<td>78%</td>
<td>60%</td>
<td>60%</td>
<td>60%</td>
</tr>
<tr>
<td>16</td>
<td>30%</td>
<td>50%</td>
<td>70%</td>
<td>40%</td>
<td>30%</td>
<td>30%</td>
</tr>
<tr>
<td>32</td>
<td>60%</td>
<td>60%</td>
<td>46%</td>
<td>50%</td>
<td>50%</td>
<td>20%</td>
</tr>
<tr>
<td>64</td>
<td>70%</td>
<td>70%</td>
<td>50%</td>
<td>30%</td>
<td>20%</td>
<td>20%</td>
</tr>
</tbody>
</table>

Average Running Time

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>189.943s</td>
<td>292.573s</td>
<td>357.145s</td>
<td>34.045s</td>
<td>61.152s</td>
<td>92.452s</td>
</tr>
<tr>
<td>4</td>
<td>996.934s</td>
<td>628.950s</td>
<td>428.353s</td>
<td>87.357s</td>
<td>255.223s</td>
<td>588.578s</td>
</tr>
<tr>
<td>8</td>
<td>552.183s</td>
<td>244.470s</td>
<td>244.046s</td>
<td>105.737s</td>
<td>167.164s</td>
<td>340.900s</td>
</tr>
<tr>
<td>16</td>
<td>118.532s</td>
<td>52.547s</td>
<td>90.363s</td>
<td>53.385s</td>
<td>141.814s</td>
<td>243.957s</td>
</tr>
<tr>
<td>32</td>
<td>40.300s</td>
<td>24.607s</td>
<td>94.146s</td>
<td>27.156s</td>
<td>80.252s</td>
<td>167.164s</td>
</tr>
<tr>
<td>64</td>
<td>15.866s</td>
<td>21.908s</td>
<td>24.253s</td>
<td>14.627s</td>
<td>30.558s</td>
<td>44.813s</td>
</tr>
</tbody>
</table>

Relative Number Best Algorithm

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20%</td>
<td>60%</td>
<td>50%</td>
<td>10%</td>
<td>10%</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>10%</td>
<td>0%</td>
<td>50%</td>
<td>10%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>8</td>
<td>0%</td>
<td>20%</td>
<td>30%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>16</td>
<td>0%</td>
<td>10%</td>
<td>54%</td>
<td>10%</td>
<td>0%</td>
<td>10%</td>
</tr>
<tr>
<td>32</td>
<td>0%</td>
<td>8%</td>
<td>38%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>64</td>
<td>0%</td>
<td>16%</td>
<td>36%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

5.3 Vertex Selection Rules

We now evaluate the vertex selection strategies to find the set of vertices K that model the ILP. We look at all strategies described in Section 4.4, i.e. Boundary, Gainρ with the parameter ρ ∈ {−2, −1, 0} as well as TopVerticesδ for δ ∈ {1, 2, 3}. To evaluate the different selection strategies, we use the best of five runs of KaFFPa strong on each of the graphs from set A using ε = 0 and k ∈ {2, 4, 8, 16, 32, 64} and then use the computed partition as input to the ILP (with different sets K). Table 2 summarizes the results of the experiment, i.e. the number of cases in which our algorithm was able to improve the result, the average running time in seconds for these selection strategies as well as the number of cases in which the strategy computed the best result (the partition having the lowest cut). We set the time limit to 2 days to be able to finish almost all runs without running into timeout. For the average running time we exclude all graphs in which at least one algorithm did not finish in 2 days (rgg_15 k = 16, delaunay_n15 k = 4, G2_circuit k = 4, 8). If multiple runs share the best result, they are all counted. However, when no algorithm improves the input partition on a graph, we do not count them.

Looking at the number of improvements, the Boundary strategy is able to improve the input for small values of k, but with increasing number of blocks k improvements decrease to no improvement in all runs with k = 64. Because of the limit on the number of non-zeros, the ILP contains only random boundary vertices for large values of k in this case. Hence,

limit if the number of non-zeros in the ILP is below 10^6. Hence, we set the parameter N to 10^6 in the following section.
there are not sufficiently many high gain vertices in the model and fewer improvements for large values of $k$ are expected. For small values of $k \in \{2, 4\}$, the Boundary strategy can improve as many as the $\text{Gain}_{\rho=-2}$ strategy but the average running times are higher.

For $k = \{2, 4, 8, 16\}$, the strategy $\text{Gain}_{\rho=-2}$ has the highest number of improvements, for $k = \{32, 64\}$ it is surpassed by the strategy $\text{Gain}_{\rho=-1}$. However, the strategy $\text{Gain}_{\rho=-2}$ finds the best cuts in most cases among all tested strategies. Due to the way these strategies are designed, they are able to put a lot of high gain vertices into the model as well as vertices that can be used to balance vertex movements. The TopVertices strategies are overall also able to find a large number of improvements. However, the found improvements are typically smaller than the Gain strategies. This is due to the fact that the TopVertices strategies grow BFS balls with a predefined depth around high gain vertices first, and later on are not able to include vertices that could be used to balance their movement. Hence, there are less potential vertex movements that could yield an improvement.

For almost all strategies, we can see that the average running time decreases as the number of blocks $k$ increases. This happens because we limit the number of non-zeros $N$ in our ILP. As the number of non-zeros grows linearly with the underlying model size, the models are far smaller for higher values of $k$. Using symmetry breaking, we already fixed the block of the $k$ vertices $\mu_i$ which represent the vertices not part of $K$. Thus the ILP solver can quickly prune branches which would place vertices connected heavily to one of these vertices in a different block. Additionally, our data indicate that a large number of small areas in our model results faster in solve times than when the model contains few large areas. The performance plot in Figure 3 shows that the strategies Boundary, TopVertices$_{\delta=1}$ and $\text{Gain}_{\rho=-2}$ have lower running times than other strategies. These strategies all select a large number of vertices to initialize the breadth-first search. Therefore they output a vertex set $K$ that is the union of many small areas around these vertices. Variants that initialize the breadth-first search with fewer vertices have fewer areas, however each of the areas is larger.

5.4 Walshaw Benchmark

In this section, we present the results when running our best configuration on all graphs from Walshaw’s benchmark archive. Note that the rules of the benchmark imply that running time is not an issue, but algorithms should achieve the smallest possible cut value while satisfying the balance constraint. We run our algorithm in the following setting: We take existing partitions from the archive and use those as input to our algorithm. As indicated by the experiments in Section 5.3, the vertex selection strategies $\text{Gain}_{\rho \in \{-1, -2\}}$ perform best.

![Figure 3](image-url) **Left:** performance plot for all vertex selection strategies **Right:** cut value of vertex selection strategies in comparison to the best result given by any strategy.
We presented a novel meta-heuristic for the balanced graph partitioning problem. Our approach is based on an integer linear program that solves a model to combine unconstrained vertex movements into a global feasible improvement. Through a given input partition, we were able to use symmetry breaking and other techniques that make the approach scale to large inputs. In Walshaw’s benchmark, we were able to improve a large number of partitions. We plan to further improve our implementation and integrate it into the KaHIP framework. We would like to look at other objective functions as long as they can be modelled linearly. Moreover, we want to investigate whether this kind of contractions can be useful for other ILPs. It may be interesting to find cores for contraction by using the information provided an evolutionary algorithm like KaFFPaE [35], i.e. if many of the individuals of the population of the evolutionary algorithm agree that two vertices should be put together in a block then those should be contracted in our model. Lastly, besides using other exact techniques like branch-and-bound to solve the model, it may also be worthwhile to use a heuristic algorithm instead.
References


ILP-based Local Search for Graph Partitioning


