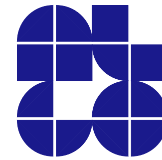


Bachelor thesis

Evolutionary k-way Node Separators

Robert Williger

Date: 6. November 2016

Supervisors: Prof. Dr. Peter Sanders
Dr. Christian Schulz
Dr. Darren Strash

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Small node separators for large graphs are used in a variety of ways, from divide-and-conquer algorithms to efficient route planning algorithms. We present a new algorithm for finding small k-way node separators on connected undirected graphs. We use an evolutionary algorithm to combine different separators in order to generate improved offsprings with locally good parts of both parent separators. Additionally, we also propose a new k-way local search to further improve already existing separators. Our experimental evaluation of our algorithm shows that we have an average improvement of 18% and 23% respectively for our two different configurations and a maximum improvement of 64% compared to the already existing algorithm in KaHIP for finding k-way node separators.

Zusammenfassung

Kleine Knotenseparatoren für große Graphen werden auf verschiedene Arten verwendet, von teile-und-herrsche Algorithmen bis zu Routenplanungsalgorithmen. Wir stellen einen neuen Algorithmus zum Finden von kleinen k-Wege Knotenseparatoren auf zusammenhängenden, ungerichteten Graphen vor. Wir benutzen einen Evolutionären Algorithmus um verschiedene Separatoren zu kombinieren, was zur Erzeugung von neuen Separatorn führt, die lokal gute Teile von beiden Anfangsseparatoren enthalten. Die experimentelle Auswertung unseres Algorithmus zeigt, dass wir eine durchschnittliche Verbesserung von 18% beziehungsweise 23% für unsere beiden Konfigurationen und eine maximale Verbesserung von 64% haben gegenüber des bereits existierenden Algorithmus in KaHIP zum finden von k-Wege Knotenseparatoren.

Contents

Abstract iii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Overview . . . 2

2 Preliminaries 3
2.1 General Definitions . . . 3
2.2 Node Separators . . . 4
2.2.1 Node Separator Problem . . . 4
2.2.2 Additional Definitions . . . 5

3 Related Work 7
3.1 Multilevel Approach . . . 7
3.2 Evolutionary Partitioning . . . 9

4 Local Search 11
4.1 Overview . . . 11
4.2 Flow-based Local Search . . . 11
4.3 k-way Local Search . . . 13
4.3.1 Preprocessing . . . 13
4.3.2 Pair Refinement . . . 14
4.4 k-way Balancing . . . 15
4.4.1 Algorithm . . . 15
4.4.2 Analysis . . . 17

5 Evolutionary Algorithm 21
5.1 Overview . . . 21
5.2 Selection . . . 22
5.3 Combination . . . 22
5.4 Mutation . . . 24
5.5 Replacement . . . 24

6 Experimental Evaluation 27
6.1 Implementation . . . 27

6.2 Experimental Setup 27
6.2.1 Environment 27
6.2.2 Tuning Parameters 27
6.2.3 Instances 29
6.3 Node Separator Evaluation 30
6.3.1 Parameter tuning 30
6.3.2 Effect of k on Runtime 33
6.3.3 KaHIP Comparison 35

7 Conclusion **47**
7.1 Future Work 47

Bibliography **49**

1 Introduction

1.1 Motivation

A frequently used algorithmic design pattern is the divide-and-conquer strategy. It is used for a multitude of problems from sorting with Mergesort or Quicksort to matrix multiplication with the Strassen algorithm [13]. The basic idea of the divide-and-conquer strategy is to divide a given problem into smaller subproblems and then solving these easier subproblems. The solutions of the subproblems are then combined to give the solution of the original problem. The divide-and-conquer strategy is often used recursively on the subproblems until the subproblems are sufficiently small to be solved by a base case algorithm. The prerequisites for this strategy are such that the decomposition of the problem instance into subproblems and the recombination of the subsolutions is fairly efficient and the subproblems should be independent and of roughly the same size [4].

The divide-and-conquer strategy can also often be applied to graph problems where a well-known approach to implement this strategy is to use node separators. Given a graph, we divide it into k blocks and an additional separator block such that no node in one block is adjacent to a vertex in any other block except for the separator. This ensures the prerequisite that the subproblems, in this case the different blocks, are independent of each other. To ensure that the subproblems are roughly the same size, the blocks have to fulfill a balance constraint which limits the size difference of the blocks. In order for the divide-and-conquer strategy to be efficient, the separator block has to be as small as possible in order to minimize the effort to recombine the solutions of the subproblems.

Our approach to computing a node separator which divides the graph into k blocks, also called a *k-way separator*, is to use a multilevel method in conjunction with an evolutionary algorithm. Since finding the best solution to the node separator problem is NP hard on general graphs [5], we use this approach in order to find equally sized blocks as well as a small node separator. The multilevel method iteratively coarsens the input graph and then computes an initial node separator on the smallest graph. We use the node separator algorithm of the KaHIP (Karlsruhe High Quality Partitioning) library [10] to compute the initial separator. We then uncoarsen the graph together with the initial separator until we reach the original input graph. In each of these uncoarsening steps we perform a novel local search refinement which reduces the size of the node separator while still fulfilling the balance constraint on the k blocks.

In order to further improve the node separator we use an evolutionary algorithm which computes multiple node separators with the multilevel method and then combines these to composite a new separator of locally "good" parts of the input separators to obtain a smaller node separator solution. Since our algorithm is not dependent on a specific implementation for computing the initial separator, it can be used to improve any previously existing separator for a given graph.

1.2 Overview

The thesis is structured as follows: In Section 2, we introduce notations and definitions which are used throughout the thesis and explain the node separator problem in detail. In Section 3 we present work which is related to the topic of this thesis and explain the basic idea of the multilevel approach we use. Our k-way local search algorithm is introduced in Section 4 where we also explain a k-way balancing method and the corresponding runtime analysis. The evolutionary algorithm and the different operations we use as part of our evolutionary algorithm are explained in Section 5. The evaluation of our algorithm in comparison to an already existing KaHIP algorithm is presented in Section 6. We summarize the results and give an outlook on future work for our algorithm in Section 7.

2 Preliminaries

In this chapter, we introduce notations and definitions which are used throughout the thesis. We first give an overview of common definitions from graph theory and then introduce more specific notations for the node separator problem.

2.1 General Definitions

Throughout this thesis $G = (V, E)$ denotes a connected undirected graph. An *undirected* graph $G = (V, E)$ is defined as a tuple of a set of vertices V and a set of edges $E \subseteq V \times V$ connecting the vertices. We denote the number of vertices in a graph by $n = |V|$ and the number of edges by $m = |E|$.

Given a graph $G = (V, E)$, we define a *weight function* $c : V \to \mathbb{R}_{\geq 0}$ which assigns each node in V a non-negative weight. This weight function can be extended to sets of vertices such that $c(V) = \sum_{v \in V} c(v)$.

A *connected* graph $G = (V, E)$ is a graph where for every pair of vertices $u, v \in V$ there exists a path between those vertices in G.

A graph $G' = (V', E')$ is called a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. A subgraph G' of G is *induced* by a set of vertices $V' \subseteq V$ is $G' = (V', E')$ where $E' = \{\{u, v\} \in E \mid u, v \in V'\}$. A subgraph G'' of G is *induced* by a set of edges $E'' \subseteq E$ is $G'' = (V'', E'')$ where $V'' = \{u, v \in V \mid \{u, v\} \in E''\}$.

P is called a *partition* of a graph $G = (V, E)$ if P is a partition of the set of vertices V into disjoint subsets $P = \{V_1, V_2, \ldots, V_k\}$. Each V_i is called a *block*. Given such a partition, the *quotient graph* $Q = (V_Q, E_Q)$ is defined as follows. For each of the blocks $V_i \in P$ there exists one vertex v_i in V_Q. The set of edges E_Q is defined as $E_Q = \{\{v_i, v_j\} \mid \exists \{u, w\} \in E : u \in V_i, w \in V_j,\ i, j \in \{1, \ldots, k\}, i \neq j\}$. Therefore a quotient graph Q of a graph $G = (V, E)$ represents the blocks of G and if there is an edge in G connecting two nodes of two distinct blocks then there is an edge in Q connecting the respective vertices. We also call two blocks *adjacent* to each other if there exists an edge in the quotient graph between the corresponding block-vertices.

We define the *neighborhood* $N(v)$ of a node $v \in V$ to be the set of all nodes which are adjacent to v in G. Given a partition $P = \{V_1, V_2, \ldots, V_k\}$ we can divide the neighborhood of a node into multiple disjunct sets, one for each block in the partition. We define $N_i(v) := N(v) \cap V_i$ to be the subset of the neighborhood of a node v where all nodes $u \in N_i(v)$ are part of the block V_i.

2.2 Node Separators

2.2.1 Node Separator Problem

Given a graph $G = (V, E)$, a weight function $c : V \to \mathbb{R}_{\geq 0}$, a weighting parameter ϵ and $k \in \mathbb{N}_{>1}$, the node separator problem is to partition V into k subsets $V_1, V_2, \ldots, V_k$ and the node separator S such that:

(i) $V_i \cap V_j = \emptyset \;\; \forall i, j \in \{1, \ldots, k\}, i \neq j$,

(ii) $V_i \cap S \; = \emptyset \;\; \forall i \in \{1, \ldots, k\}$,

(iii) $\bigcup_{i=1}^{k} V_i \; \cup S = V$,

(iv) $\{\{u, v\} \in E \mid u \in V_i, v \in V_j, \; \forall i, j \in \{1, \ldots, k\}, i \neq j\} = \emptyset$,

(v) $c(S)$ *is minimized*,

(vi) $\forall i \in \{1, \ldots, k\} : c(V_i) \leq L_{\max} := (1 + \epsilon) \lceil c(V)/k \rceil$.

This means that we want to find disjoint sets $V_1, V_2, \ldots, V_k$ and a node separator S such that there exists no edge between any two nodes of different blocks V_i and V_j. We also want the weight of S to be minimized while the weight of the blocks V_i does not surpass a given *maximum block weight*.

If no weight function is given we define a new weight function $c(v) = 1 \;\; \forall v \in V$. This is equivalent to searching for a node separator of minimum cardinality $|S|$.

The partition $P = \{V_1, V_2, \ldots, V_k\}$ and the blocks $V_i \in P$ are also said to be *induced* by the separator S and ϵ is also called the *imbalance* or *imbalance parameter* of G.

$L_{\max}$ denotes the *maximum block weight* and it implies a *balancing constraint* for any block V_i. The separator block S has no such constraint. For small ϵ the balancing constraint implies that the weight difference of any two blocks in G is small. As ϵ increases the difference in weights of the blocks is allowed to get bigger. This also increases the number of valid solutions for the node separator problem which will likely lead to smaller node separators. Note that choosing large values for ϵ can lead to entire blocks being empty. Also note that choosing $\epsilon = 0$ does not imply that all blocks of G have the same weight. Some of the total weight of all nodes is contained in the separator S which in turn allows for a variance in block weights even though $\epsilon = 0$.

A block V_i is called *balanced* if it fulfills the balance constraint and a node separator S is called *balanced* if all blocks induced by S are themselves balanced. Conversely we call a block or separator *imbalanced* if it is not balanced.

2.2.2 Additional Definitions

We define the similarity σ of two node separators S_1 and S_2 of a graph $G = (V, E)$ as the cardinality of the symmetric difference of both separators:
$\sigma = |S_1 \Delta S_2| = |(S_1 \setminus S_2) \cup (S_2 \setminus S_1)|$. Therefore σ denotes the number of nodes contained in one separator but not in the other. If $\sigma = 0$, both separators are identical and as σ grows the separators become less similar.

In a partition P induced by a node separator all blocks $V_i \in P$ will only be adjacent to the separator block S or to no block at all by the standard definition of adjacency given above. We provide a more convenient definition for node separators: two different blocks V_i and V_j are *adjoint* to each other if there exists a separator node $s \in S$ such that $\exists u \in V_i, v \in V_j \wedge \exists \{u, s\}, \{s, v\} \in E$. Such a node s *directly separates* V_i and V_j. Note that s can directly separate more than two blocks. This definition is also equivalent to the existence of a path of length 2 from a node $u \in V_i$ over a separator node $s \in S$ to a node $v \in V_j$.

Similarly in a quotient graph $Q = (V_Q, E_Q)$ of a graph G with node separator S all nodes in V_Q will only have an edge to the node s corresponding to the separator block or to no node at all. We again provide a more convenient definition for node separators: for each block V_i there exists a vertex v_i in V_Q and for each two blocks V_i and V_j which are adjoint there exists an edge $\{v_i, v_j\}$ in E_Q.

3 Related Work

One of the most well-known contributions to research of node separators is the Planar Separator Theorem of Lipton and Tarjan [7]. It states that on planar graphs a small node separator can always be found in linear time. Planar graphs are graphs which can be embedded in the plane such that no two edges cross one another. Given a planar graph $G = (V, E)$ there therefore exists a separator S which splits the graph into two blocks A and B whose size does not exceed $2|V|/3$. The size of the separator additionally fulfills the constraint $|S| = O\left(\sqrt{|V|}\right)$.

3.1 Multilevel Approach

In general, finding a minimum weight node separator on arbitrary graphs is NP-hard [5]. Therefore, different methods and heuristics are used to compute a small node separator. One of the methods, which is also the base of thesis, is the *multilevel approach* [9], which is divided into three different phases:

Coarsening. First a hierarchy of graphs is created through multiple coarsening steps. The result of one coarsening step is a graph which has a reduced number of nodes compared to the input graph. Iterating coarsening creates a graph hierarchy where each new graph level is coarser (has fewer nodes) than the level before it.

One method of implementing a coarsening step is by constructing an edge matching. Each edge of the matching is incident to two nodes which are themselves not incident to any other edge in the matching. These edges are contracted so that the two incident nodes are combined into a new node which inherits the incident edges of both nodes.

Initial Separator Computation. After generating a hierarchy of coarser graphs, we compute an initial node separator on the coarsest graph.

Uncoarsening. The coarsening is undone by replacing each node by its two source nodes from the previous level starting at the coarsest level. The separator nodes are therefore projected from the coarser graphs to the finer ones. After each uncoarsening of a level local search can be applied to the reduce the separator size.

One iteration of coarsening, computing the initial separator and uncoarsening, is also called a *V-cycle* and iterating multiple V-cycles of a multilevel approach is called an *iterated multilevel approach* [14]. In each of these phases additional heuristics can be used to further improve the separator.

One possible way to compute the initial node separator which is used in KaHIP [10] is to construct a partition for the given graph and then transform it into the initial node separator. The transformation can be done by looking at each adjacent pair of blocks in the partition separately. For each such pair $P_i = (V_a, V_b)$ we define the edges between the blocks $E_i := \{(v,u) \in E \mid v \in V_a, u \in V_b\}$ and the borders $\partial_a P_i := \{v \in V_a \mid \exists u \in V_b : (v,u) \in E_i\}$ and $\partial_b P_i := \{v \in V_b \mid \exists u \in V_a : (v,u) \in E_i\}$. We find the smaller of both borders by weight, which we denote by $\partial_{\min} P_i$. $\partial_{\min} P_i$ separates V_a and V_b, therefore combining all these borders for all adjacent pairs into one set gives us our initial separator $S := \bigcup_{P_i} \partial_{\min} P_i$.

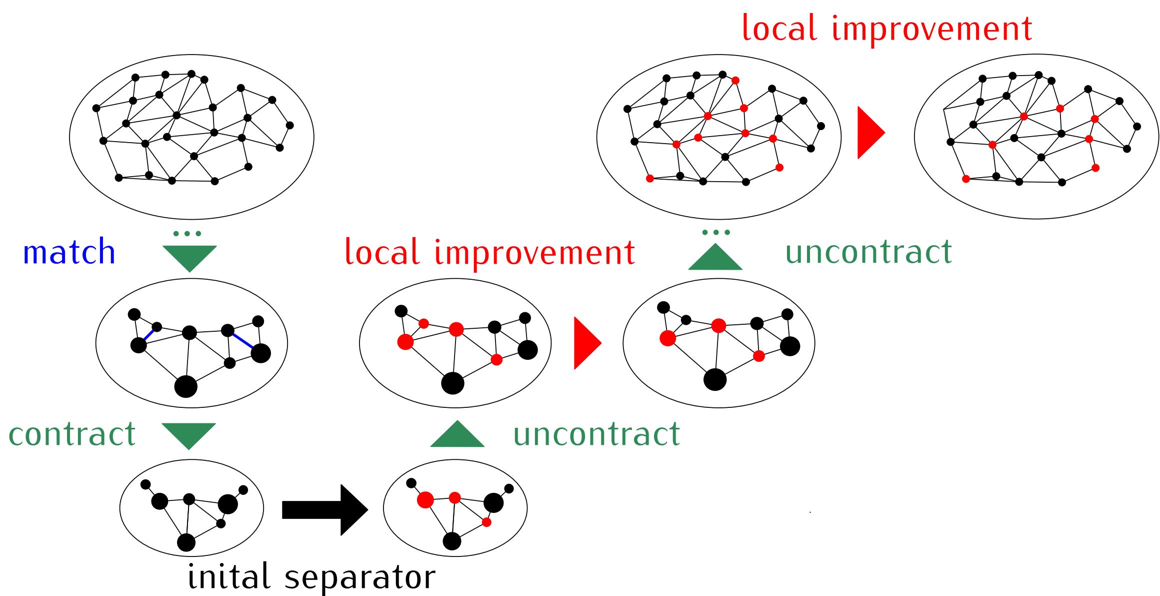


Figure 3.1: Illustration of the multilevel approach with the *coarsening phase* on the left side from top to bottom, the *initial separator computation* on the bottom on the *uncoarsening phase* on the right side from bottom to the top. The edges of the matching are depicted in blue and the separator nodes in red.

3.2 Evolutionary Partitioning

Since finding optimal solutions for optimization problems like the node separator problem or the partitioning problem is usually hard and time intensive, randomized algorithms are often used to compute solutions in a short amount of time which are close to the optimal solution. These algorithms produce different solutions of different quality depending on the initial random seed and therefore multiple iterations are performed in order to be able to chose the best solution. Sanders and Schulz [9] use an evolutionary algorithm for finding better partitions by not only performing multiple iterations but also using different, biologically inspired operations on the results of these iterations.

When performing an evolutionary algorithm, first a population of multiple individuals is produced by simply iterating an already existing partitioning algorithm. When the population is filled, two individuals from the population are *selected* and then *combined* in a specific way such that the quality of the resulting partition is at least as good as the best of both input partitions. Alternatively only one individual is selected and then *mutated* such that the result is again a partition of equal or better quality. The resulting individuals are then reinserted into the population by *replacing* an individual from the population with the new individual. These operations are iterated in order to find better solutions.

4 Local Search

Local search is used to improve an already given solution to the node separator problem. As the term implies we locally search for a better separator in the solution space around an already given separator in order to find a new locally optimal solution. Since the initial separator is included in the search space the local search operates on and the local search finds the optimum of that search space, the weight of the separator resulting from it is guaranteed to be nonincreasing.

In this chapter we first give an overview of how we use multiple local search iterations to improve a separator. We then give an outline of flow-based local search [11] which is used to improve separators where $k = 2$. We also show how to use this local search to construct a more general local search for improving arbitrary k-way separators.

4.1 Overview

As seen in Algorithm 1, we use multiple local search iterations to increase the chance of improving the separator. The first step is to preprocess and if necessary balance the separator S. Then we iteratively perform a k-way local search on S. We start with a predefined maximum value for the expansion α of the flow problem used in our local search and we reduce this value in each iteration. As we explain in Section 4.2, a larger expansion value α can lead to larger improvements in separator weight for S but it can no longer be guaranteed that S is balanced. If we find an improved separator S which is balanced then we return this separator. If after n iterations no balanced separator has been found then we perform a final k-way local search with no expansion of the flow problem which will produce a balanced separator. Since our k-way local search always produces a balanced separator if the input separator is balanced, we can guarantee that if a balanced separator S is given as input into our algorithm we will also output a balanced separator.

4.2 Flow-based Local Search

The local search we apply uses a flow-based technique [11] in order to find the smallest separator from a given subgraph where $k = 2$. The idea is to expand the separator S into S' by performing two breadth first searches (BFS) into both blocks V_1 and V_2 and then find the best separator in S' by solving a node-capacitated flow problem F induced by S'.

Algorithm 1: Separator refinement algorithm which uses k-way local search

```
Input: graph G and separator S
1  preprocess S
2  if S imbalanced then
3      balance S
4  α = maximum expansion value                        // parameter for expansion of flow problem
5  while less than n iterations do                     // do at most n iterations
6      perform k-way local search on S with maximum imbalance of (1 + α)L_max
7      if S balanced then
8          break
9      else
10         α = α/2                                     // decrease expansion value
11         reset S to last balanced state
12 if S imbalanced then
13     perform k-way local search on S with maximum imbalance of L_max
14 return separator S
```

A *border* of $A \subset V$ is defined as $\partial A := \{u \in A \mid \exists\{u, v\} \in E : v \notin A\}$ as well as the *left border* $\partial_1 A := \partial A \cap V_1$ and the *right border* $\partial_2 A := \partial A \cap V_2$.

The subgraph is selected by doing two BFSs. Each BFS is initialized with S. The first BFS only traverses nodes in V_1 and any touched nodes are added to a vertex set S_1. Similarly the second BFS only traverses V_2 and adds its nodes to S_2. The first BFS stops as soon as the weight of S_1 would exceed $L_{\max} - c(V_1) - c(S)$ and the second BFS stops if the weight of S_2 would exceed $L_{\max} - c(V_2) - c(S)$. These stopping criteria ensure that each possible separator which can be found in $S' = S_1 \cup S_2$ fulfills the balance constraint.

The set of vertices S' induces a node-capacitated flow problem $F = (V_F, E_F)$. F is a directed graph containing all nodes of S' and additionally the nodes s and t where s is the source of the flow problem and t the sink. For each undirected edge $\{u, v\}$ in the subgraph induced by S' both directed edges (u, v) and (v, u) are contained in E_F. E_F also contains all directed edges from s to $\partial_1 S'$ as well as all directed edges from $\partial_2 S'$ to t. The edge-capacities of all edges are ∞ and the node-capacities are the weights of the nodes in the original graph G.

By solving the maximum flow problem stated above we obtain the smallest separator that can be found in S'. The nodes with maxed capacities directly translate to a balanced separator in G.

In order to find even smaller node separators, it is possible to expand S' even further by defining a new stopping criterion with a parameter α: $(1 + \alpha)\, L_{\max} - c(V_i) - c(S)$. Here, a bigger value for α leads to a greater depth of the BFSs and therefore a larger search space for the flow problem. Note that by setting α to zero we obtain the original criterion.

4.3 k-way Local Search

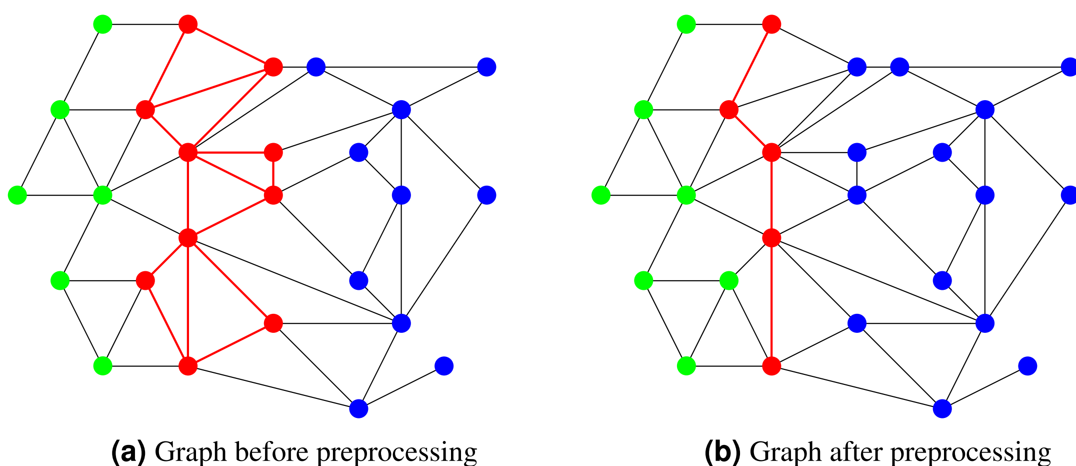
Our k-way local search builds on top of the flow-based search which is intended for improving a separator with $k = 2$. The idea is to find pairs of adjoint blocks and then perform the local search on the subgraph induced by these block pairs.

4.3.1 Preprocessing

In order to find pairs of adjoint blocks, we look at separator nodes which *directly separate* two different blocks, meaning these separator nodes are adjacent to nodes from at least two different blocks not including the separator block. We denote these pairs by $P = \{\{V_i, V_j\} \mid \exists s \in S \textit{ such that } s \textit{ directly separates } V_i \textit{ and } V_j\}$.

In general these separator nodes do not have to exist which means that although a separator divides two blocks, the shortest distance from one block to the other block through the separator is greater than two. In this case, we move nodes from the separator to an adjacent block to narrow the separator. We do this by first moving all separator nodes into a queue and then iterating through all nodes in the queue until it is empty. For each node in the queue, we look at the neighboring nodes and move it into an adjacent block if possible or discard the node if it cannot be moved. Let s be the next separator node in the queue.

If s only has neighbors from one block V_i in addition to the separator block, we can directly move this node to block V_i. If there are two or more neighboring non-separator nodes belonging to different blocks, we found a separator node which cannot be moved to another block because it already directly separates at least two blocks. We mark s and then discard it.



(a) Graph before preprocessing   **(b)** Graph after preprocessing

Figure 4.1: Illustration of the preprocessing algorithm for a graph with blocks V_1 (green), V_2 (blue) and separator S (red).

The final case which can occur is that s only has other separator nodes as neighbors. Here, we differentiate between two subcases. If all neighboring nodes are already marked, then none of these nodes are eligible to move to another block and s can be discarded. Otherwise at least one node is still eligible to move and we enqueue s again.

A problem that can occur is that a group of non-marked separator nodes can be surrounded by marked separator nodes in which case the non marked nodes will be repeatedly enqueued resulting in an infinite loop. To circumvent this problem we use two separate queues, one for dequeuing nodes and one for enqueuing nodes. After emptying the dequeuing queue we check if we moved any separator nodes to other blocks while dequeuing. If this is not the case then either the enqueuing queue is empty or it contains the aforementioned circumscribed separator nodes which either way indicates the end of the separator node moving algorithm.

By moving the separator nodes, we have ensured that we can find adjoint pairs of blocks through looking only at the separator nodes which directly separate two or more different blocks. A drawback to this preprocessing step is that we can no longer guarantee that the balance constraint is met. We remedy this with a separate balance operation which is discussed in detail in Section 4.4.

4.3.2 Pair Refinement

Subsequent to the preprocessing step we can now identify the set of all adjoint block pairs P by simply iterating through all separator nodes and their adjacent blocks. We iterate through all pairs $p_i = (V_{i_1}, V_{i_2}) \in P$ and build the respective subgraph G_i. G_i is induced by the set of nodes consisting of all nodes in V_{i_1} and in V_{i_2} as well as all separator nodes which *directly separate* V_{i_1} and V_{i_2}. After building G_i, we run local search designed for 2-way separators on this subgraph as shown in Section 4.2.

In order to increase the total improvement and because the graph changes each time we run local search, we repeatedly run local searches on the pairs of blocks. We do this by iterating over the set P of all pairs and performing local search on all pairs $p \in P$ until P is empty. We refer to the number of iterations as *maximum pair refinement steps*. After each local search on a pair p we record the improvement of the local search. If it does not improve the separator then we refrain from doing another local search on p and remove p from P. If it does improve the separator but we have already performed the specified maximum number of local searches we also remove p from P. In all other cases we move p to the back of P and perform another local search on p after all other pairs in P have been visited.

4.4 k-way Balancing

4.4.1 Algorithm

To guarantee that the balance constraint is fulfilled by the resulting separator of our node separator refinement, we need a balanced separator before calling our k-way local search. We now present a balance operation which given an arbitrary separator of a graph $G = (V, E)$, will produce a balanced separator for G. The idea is to move nodes from the imbalanced and therefore heavy blocks towards the lightest block until all blocks are balanced again as is illustrated in Figure 4.2. As long as there are imbalanced blocks in the G we iteratively repeat the following steps, each iteration of which will improve the balance in the graph.

Finding the path

First we identify the currently heaviest block $V_{\max}$ and lightest block $V_{\min}$. Then we compute the quotient graph Q from G and search a path P from $V_{\max}$ to $V_{\min}$ in Q. We can always find such a path because G is a connected graph which implies that the quotient graph of G is also connected. We represent P as a list of directed edges in the form $P = (\,(A_1, A_2), (A_2, A_3), \ldots, (A_{p-1}, A_p)\,)$ where $A_i \in \{V_1, \ldots, V_k\}$, $A_1 = V_{\max}$ and $A_p = V_{\min}$.

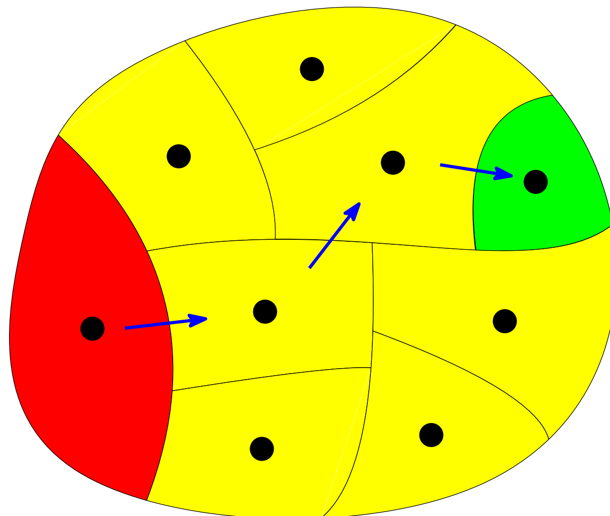


Figure 4.2: Illustration of the quotient graph and the balancing path (blue) from an imbalanced block (red) to the lightest block (green).

Moving nodes

Next we iterate through the edges $(A_i, A_{i+1}) \in P$ and push nodes from A_i into A_{i+1} until A_i is balanced. This is done by creating a priority queue of separator nodes where the priority represents the expected gain of moving a node from the separator into A_{i+1}. Note that when moving a node s from the separator S to A_{i+1} we have to insert all adjacent nodes $N_i(s)$ of s which lie in A_i into the separator otherwise S would no longer be a valid separator. We therefore define the *gain* $g(s)$ of a separator node to be $g(s) := c(s) - \sum_{v \in N_i(s)} c(v)$ which means $g(s)$ is equal to the change of the separator weight if we move s into A_{i+1}.

Other definitions for the gain of a node are also conceivable. We use this definition because our primary target is to minimize the increase of the separator size.

We fill the priority queue with all separator nodes which are adjacent to both A_i and A_{i+1}. Then we dequeue nodes from the queue until A_i is balanced. We move each dequeued node s to A_{i+1} and move its neighbors $N_i(s)$ from A_i to the separator. We also check for any other nodes in S which can be moved to A_{i+1} without moving new nodes into the separator and move them too. This is done so we don't leave any nodes in the separator which touch only one block and therefore retain the state of the separator which was created by the preprocessing step in Section 4.3.1.

After moving the nodes A_i will be balanced and because we started from the beginning of the path all $A_j, j \leq i$ along the path are now balanced. Should A_{i+1} also be balanced, then we don't have to move any more nodes along the path and can continue with balancing the next imbalanced block in G. If however A_{i+1} is imbalanced we iterate further through the path and move nodes along the next pair in P from A_{i+1} to A_{i+2}.

In the case where A_{i+1} is imbalanced and $A_{i+1} = V_{\min}$, meaning there is no pair left in P, we simply push the excess weight into the separator. This is done to ensure termination of the algorithm. Since most of the excess weight from the A_1 has been distributed along the path and also because practical tests have shown that this scenario is unlikely, the impact on the separator size is minor.

Another way of handling this scenario would be to simply add A_{i+1} to the set of imbalanced partitions. This will prevent unnecessary increase of the separator size but the termination of the algorithm could theoretically no longer be guaranteed.

4.4.2 Analysis

Termination

We now show that this procedure will balance all blocks in G and terminate in a finite number of steps.

By always selecting the heaviest and lightest blocks to build our path along which we move nodes, we end the movement of the nodes in one of two possible cases:

Case 1: We stop moving nodes because both blocks of the current edge (A_i, A_{i+1}) in the path are balanced. Obviously this means that we now have at least one less imbalanced block. At the beginning of the node movement phase we had at least one imbalanced block ($A_1 = V_{\max}$) along the path until A_{i+1} and at the end all blocks up to and including A_{i+1} are balanced. Additionally if there still exist other imbalanced blocks then $V_{\max}$ will no longer be the heaviest block, instead one of the other imbalanced blocks will now be the new heaviest block.

Case 2: We stop moving nodes because we are at the end of the path and $V_{\min}$ is now imbalanced. Because we now push the excess weight of $V_{\min}$ into the separator, which is always possible, $V_{\min}$ is now balanced too. By similar reasoning as above we now have at least one less imbalanced block.

Summarizing, in each case the number of imbalanced blocks is reduced. Therefore this algorithm will terminate after at most as many iterations as there are imbalanced blocks and it will result in a graph where all blocks are balanced.

Runtime

We now determine an upper bound on the runtime of this algorithm. Let B be the set of all imbalanced blocks. For each imbalanced block $b \in B$ we calculate the shortest path to the lightest block via BFS and balance the blocks along this path. To simplify the runtime analysis we use the following insights:

(i) The path from the imbalanced block to the lightest block contains at most k pairs of blocks, else there would be loops in this path and it would not be the shortest path.

(ii) If there are blocks along the path which are balanced then they can potentially store some of the excess weight which we push along the path. We can disregard this fact in our estimation since it would only lead to performing fewer operations when balancing the other pairs in the path because we have less excess weight to move.

(iii) If there are blocks along the path P which are imbalanced then this means from this point on we push the excess weight of more than one block along the path. However, at the end of balancing all remaining pairs in P all excess weight of all imbalanced partitions along the path has been distributed to other blocks and the separator. In other words we performed the balancing of the paths of all theses imbalanced

blocks on P together. We can consider each of these paths individually since the total amount of weight we push along the paths and therefore the number of operations has not changed.

With these insights we can write out the runtime of the balancing algorithm as:

$$O\left(\sum_{b\in B} \textit{balance } b\right) = O\left(\sum_{b\in B} \textit{balance path } P_b\right).$$

The runtime of finding the shortest path in the quotient graph $Q = (V_Q, E_Q)$ is equal to the runtime of the BFS which is in $O(|V_Q| + |E_Q|) = O(|E_Q|)$.

The first step in each balance procedure for a pair $p \in P_b$ is to initialize the queue with all separator nodes which separate the two blocks of the pair and to calculate their gain value. Since these separator nodes are mostly different for each pair p in the path we can estimate the total amount of separator nodes which are initially moved into the queue to be in $O(|S|)$ for each path P_b. The calculation of the gain value for a separator node s is done by looking at all its neighbors $N(s)$. We use the fact that the number of adjacent nodes $|N(n)|$ of any node $n \in V$ will always be less than or equal to the maximum node degree Δ. Therefore, we have a runtime of $O(\Delta \cdot |S|)$ for initializing all queues for a given path P_b.

This yields the following runtime for the whole balancing algorithm:

$$\begin{aligned} O\left(\sum_{b\in B} \textit{balance path } P_b\right) &= O\left(\sum_{b\in B}\left(\textit{BFS } + \textit{initialize queues } + \sum_{p\in P_b} \textit{balance pair } p\right)\right) \\ &= O\left(\sum_{b\in B}\left(|E_Q| + \Delta \cdot |S| + \sum_{p\in P_b} \textit{balance pair } p\right)\right). \end{aligned}$$

In each balancing procedure for a pair $p = (A_i, A_{i+1})$, $p \in P_b$ we have to push the excess weight of the imbalanced block b from A_i to A_{i+1}. Let $C_b := c(b) - L_{\max}$ be the excess weight of $b \in B$ and $C_B = \sum_{b\in B} C_b$ the total excess weight of all imbalanced blocks. Let S_i be the set of all nodes $s \in S$ which will be moved from the separator S to A_{i+1}. As described in Section 4.4.1, for each such node s we also look at all its neighbors $N(s)$. We make the estimation that for each node s we move into A_{i+1} we will also move on average one node $n \in A_i$ form A_i into S. Let M_i be the set of all such nodes which we move into S. We again have to look at all neighbors.

We will look at the worst case where each node $n \in V$ has weight $c(n) = 1$ meaning we have to move C_b nodes from A_i to A_{i+1}. This also means by our prior estimation that $|S_i| = |M_i| = C_b$. By using the prior upper bound Δ for $|N(n)|$ we get the following estimated runtime for a balancing procedure of a pair on the path P_b:

$$
\begin{aligned}
O\left(\textit{balance pair } p = (A_i, A_{i+1})\right) = O\left(\sum_{s\in S_i} N(s) + \sum_{m\in M_i} N(m)\right) &= O\left(\Delta \cdot (|S_i| + |M_i|)\right) \\
= O\left(2 \cdot \Delta \cdot C_b\right) \qquad\qquad\qquad &= O\left(\Delta \cdot C_b\right).
\end{aligned}
$$

We can now combine this with our first runtime approximation:

$$
\begin{aligned}
O\left(\sum_{b\in B}\left(|E_Q| + \Delta \cdot |S| + \sum_{p\in P_b} \textit{balance pair } p\right)\right) &= O\left(\sum_{b\in B}\left(|E_Q| + \Delta \cdot |S| + \sum_{p\in P_b} \Delta \cdot C_b\right)\right) \\
&= O\left(\sum_{b\in B}\left(|E_Q| + \Delta \cdot |S| + k \cdot \Delta \cdot C_b\right)\right) \\
&= O\left(\sum_{b\in B}\left(|E_Q| + \Delta \cdot |S|\right) + \sum_{b\in B}\left(k \cdot \Delta \cdot C_b\right)\right) \\
&= O\left(\sum_{b\in B}\left(|E_Q| + \Delta \cdot |S|\right) + k \cdot \Delta \cdot C_B\right) \\
&= O\left(|B| \cdot (|E_Q| + \Delta \cdot |S|) + k \cdot \Delta \cdot C_B\right).
\end{aligned}
$$

Therefore our final runtime of the whole k-way balancing algorithm is:

$$
O\left(\sum_{b\in B} \textit{balance } b\right) = O\left(|B| \cdot (|E_Q| + \Delta \cdot |S|) + k \cdot \Delta \cdot C_B\right).
$$

This clearly shows that the runtime is divided into two parts. The preparation part which includes the calculation of the shortest path and the initialization of the queues has runtime $O\left(|B| \cdot (|E_Q| + \Delta \cdot |S|)\right)$. The runtime of the actual movement of the nodes has runtime $O\left(k \cdot \Delta \cdot C_B\right)$ and is thus only proportional to the number of blocks k, the maximum node degree Δ and the total excess weight C_B of all imbalanced blocks. We can now simplify our runtime by using the following upper bound approximations: $O(|B|) = O(k)$, $O(|E_Q|) = O(k^2)$, $O(|S|) = O(n)$ and $O(C_B) = O(n)$. Our simplified runtime is therefore:

$$
\begin{aligned}
O\left(k \cdot \left(k^2 + \Delta \cdot n\right) + k \cdot \Delta \cdot n\right) &= O\left(k^3 + k \cdot \Delta \cdot n + k \cdot \Delta \cdot n\right) \\
&= O\left(k^3 + k \cdot \Delta \cdot n\right).
\end{aligned}
$$

For planar graphs the size of the separator can be approximated with $O(|S|) = O(\sqrt{n})$ yielding a simplified runtime of $O\left(k^3 + k \cdot \Delta \cdot \sqrt{n}\right)$.

5 Evolutionary Algorithm

In this chapter, we explain our new approach to compute small node separators by using an evolutionary algorithm. The intuition of this method is that different separators are combined by compositing a new separator of locally "good" parts of the input separators to obtain a smaller node separator solution. The different operations with which we realize in our evolutionary algorithm are inspired by the techniques of Sanders and Schulz [9].

5.1 Overview

The basis of an evolutionary algorithm is a population of multiple individuals and a fitness function to assess the quality of each individual. In our case, the population is composed of different node separators and the fitness function is the size of the separator. The evolutionary algorithm then uses the biological concepts of recombination, mutation and selection on individuals from the population to iteratively produce new individuals of better solution quality.

One iteration consists of selecting two individuals from the population and combining them to generate an improved offspring. An individual of the population is then chosen to be replaced by the new offspring. To increase the diversity of the population an iteration can alternatively consist of selecting a single individual and mutating it to increase diversity in the population, therefore preventing the individuals from getting too similar.

Since our evolutionary algorithm generates only one offspring per iteration round and inserts it into the population by evicting another individual, the size of the population remains constant. This algorithm is therefore also called *steady-state* [6].

Algorithm 2: A general evolutionary algorithm

```
1 create initial population P
2 while stopping criterion not fulfilled do
3     select parents p1, p2 from P
4     combine p1 with p2 to create offspring o
5     mutate offspring o
6     replace an individual from the population with o
7 return the fittest individual in the population
```

5.2 Selection

The selection operation is the method of choosing individuals from the population which are then used to generate a new offspring. The easiest method of choosing two parent individuals is just taking two random individuals out of the population. This ensures that every individual has the same chance of generating an offspring.

Another method of choosing the individuals is to choose the first one randomly and then select the individual which is least similar to the first one. The idea behind this method is that it could produce better offspring (given the right combination operation described in Section 5.3) because the diversity of the parents increases the search space for the offspring. The obvious drawback is the increased run time since the similarity between the first parent and every other individual in the population has to be calculated.

Our approach. We use another selection rule which aims to improve the fitness of the offspring, the tournament selection rule [8]. It selects two individuals at random from the population and then chooses the fittest of these two for the first parent individual. This is repeated for the second parent individual, although if the fittest of these two random individuals is the same as the first chosen parent individual, the second random individual is chosen as the second parent individual.

5.3 Combination

The combination operation takes two individuals and creates a new offspring with traits from both parent individuals which is illustrated in Figure 5.1. In our case, we ensure that the separator size of the offspring is at most the lowest of both parents (i.e. the fitness of the offspring improves compared to the parents).

Our combination operation combines the parent node separators S_1 and S_2 by using the iterated multilevel approach as described in Section 3.1. Let $B_1 = \{V_1^1, \ldots, V_1^k\}$ be the blocks induced by the parent separator S_1. We define $C_1 = \{\{v, s\} \in E_1 \mid v \in V_1^i \wedge s \in S_1,\ i \in \{1, \ldots, k\}\}$ to be the set of separating edges which connect a non-separator node of a block in B_1 node to a separator node in S_1. We similarly define C_2 for S_2. During the coarsening phase of the multilevel method, no edges in the set of all such separating edges $C = C_1 \cup C_2$ are matched or contracted.

At the end of the coarsening phase, we select the parent separator with the smaller weight and apply it to the graph at the coarsest level. This also enables us to skip the next phase where the initial node separator is normally calculated. Since the contraction of all separating edges was blocked, only edges which connect nodes of the same block will be contracted. Therefore the remaining nodes in the coarsest graph represent sets of nodes which belong to the same block and it is possible to directly apply the parent separator to the coarsest graph. Since the following uncoarsening and refinement steps are guaran-

teed to not increase the separator weight and the so far found solution is the weight of the applied separator, we can guarantee the aforementioned improvement of the separator size. Blocking the separating edges of one separator from contracting effectively contracts all adjacent nodes of the same block into single nodes at the coarsest level. Additionally blocking the separating edges of another separator will split these nodes with regards to the second separator. Hence the following refinement step can easily exchange these sizable parts by just moving a few nodes at the coarsest level and can thus bring together good parts of both parent separators.

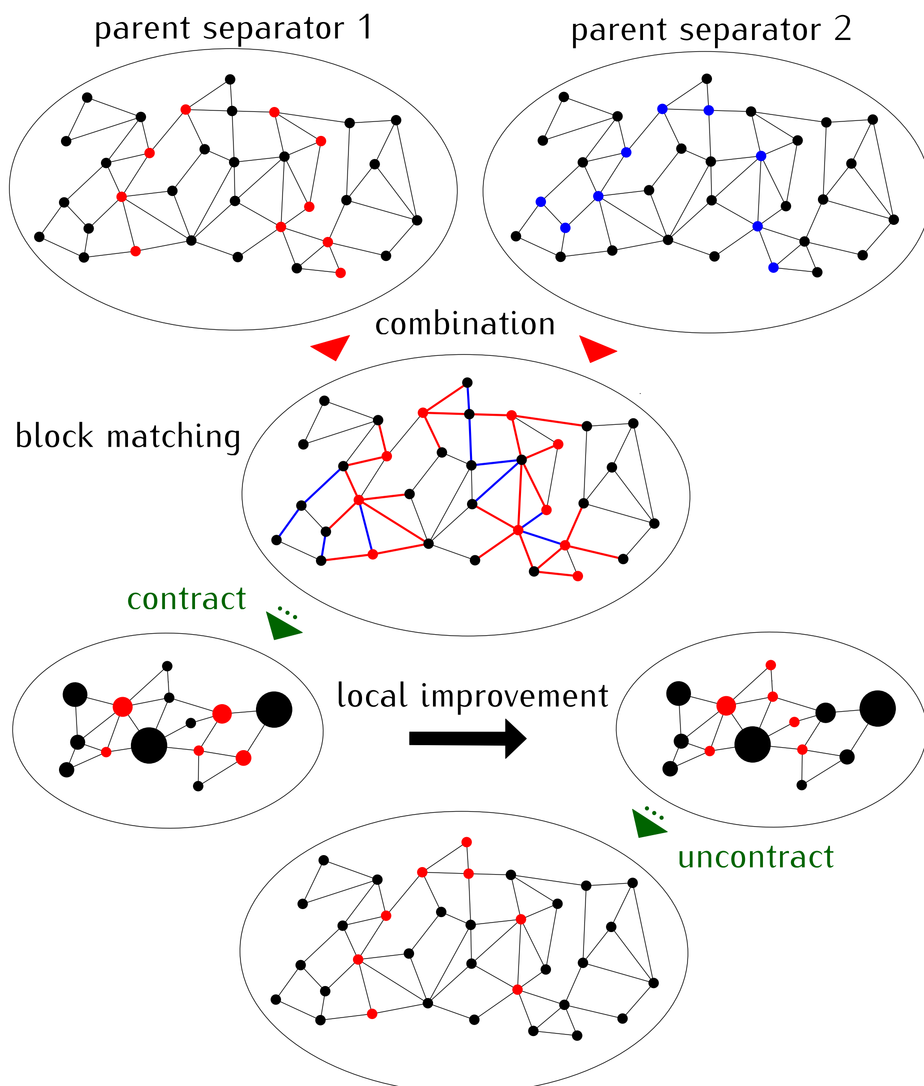


Figure 5.1: Illustration of the combination operation where both parent separators prevent edges from being matched and contracted. The blocked edges of separator 1 are depicted in red and the additional blocked edges of separator 2 in blue.

5.4 Mutation

The mutation operation is used to modify a given individual in order to decrease the similarity to its parent individuals and to the rest of the population. This increase in diversity and the resulting expansion of the search space helps the algorithm avoid converging to local optima and therefore finds a better global solution [3].

Usually the mutation operation is used on the offspring generated by the combination operation as seen in Algorithm 2. In our implementation we either perform a combination operation or we mutate a random individual from the population. The ratio between these operations is predefined.

To mutate a given separator, we use a strategy similar to the combination operation. We first coarsen the graph while restricting the contraction of edges between nodes of different blocks induced by the separator. We then apply the separator to the graph on the coarsest level and perform refinement and uncoarsening until we reach the topmost level. Applying the separator at the coarsest level ensures nondecreasing quality of the separator. Since our internal algorithms are randomized, this increases the chance of reducing the similarity of the new separator to the given input separator. A mutation operation can potentially be iterated multiple times to further reduce the similarity between the input and output individuals and to increase the fitness even more.

5.5 Replacement

A replacement operation takes place whenever an individual is inserted into the population. In order for the population to maintain a fixed size, an individual must be replaced by the newer one. A naive approach would be to simply select a random individual from the population and replace it. This is a less favorable method since it could lead to evicting the fittest or one of the fittest individuals worsening the overall solution quality of the population.

We therefore choose to only replace individuals with low fitness by ones with higher fitness. This ensures that the average fitness of the population successively improves. This leads to an other simple approach where we select the individual with the worst fitness for replacement guaranteeing that average fitness in the population increases.

Our Approach. The approach we use also incorporates the similarity of the new individual to the others in the population. We select the individual which is most similar to the new individual out of the subset of the population which is eligible for replacement meaning those individuals with lower fitness. This ensures that the diversity of the population remains high and at the same time the overall quality of the population increases.

Using this approach also has the benefit that if there is already an individual identical to the new individual in the population, it will be selected for replacement because it is obviously the most similar. This means that if there are no two identical individuals in the initial population, then this will also be the case for all future generations of the population.

6 Experimental Evaluation

6.1 Implementation

We implemented all parts of our algorithm in the graph partitioning framework KaHIP [10]. We use the multilevel approach, described in Section 3.1, to compute a node separator. The node separators on the coarsest level in our V-cycles are computed by constructing a partition with KaFFPa (Karlsruhe Fast Flow Partitioner), a partitioning algorithm already implemented in KaHIP. This partition is then transformed into a node separator according to the method for creating an initial node separator described in Section 3.1. Afterwards we apply our refinement algorithm (Algorithm 1) which uses k-way local search on the initial node separator as well as after each uncoarsening step to improve the separator size.

The node separators generated by this method are then used as individuals in the initial population in our evolutionary algorithm in Chapter 5. Each combination and mutation operation we perform within the evolutionary algorithm also uses the multilevel approach and therefore we also apply local search after each uncoarsening step.

6.2 Experimental Setup

6.2.1 Environment

All our experiments were done on a system equipped with 64 processors (Intel Xeon CPU E5-2670 v3) which run at a clock speed of 2.3 GHz and with 512 GB of total main memory. Our algorithms are written in C++ and compiled with gcc 4.8.5 with optimization level -O3. The experiments were run in parallel, where each instance ran on its own core.

6.2.2 Tuning Parameters

We give an overview of different tuning parameters of our implementation. These parameters affect the quality of the solution as well as the runtime of our whole algorithm.

Evolutionary Algorithm

The size of the initial population determines how many individuals will be stored in the population at any point. A bigger population obviously increases the time it takes to compute the initial population for the evolutionary algorithm but it can also increase the diversity in the population.

The selection operation can be realized in multiple ways which is explained in detail in Section 5.2 which means the choice of a specific selection operation is a tuning parameter of our algorithm. Similarly the choice of the replacement operation detailed in Section 5.5 is another tuning parameter.

In each iteration of the evolutionary algorithm we either perform a combination or a mutation operation. The ration between these operations is a tuning parameter which affects the diversity in the population and the final solution quality. The mutation operation will generally increase the diversity of the population where as the combination operation will find new node separators of better solution quality based on the diversity in the population.

We also have a parameter for the maximum runtime of our whole algorithm. The *time factor* parameter specifies how long combination and mutation operations should be performed in terms of multiples of the average runtime of the computation of one individual. This average runtime is calculated by taking the average of all individual computations from the initial population.

Local Search

Our k-way local search algorithm uses the tuning parameter *maximum pair refinement steps* as described in Section 4.3.2 which determines how many times each adjoint pair of blocks should be refined. Increasing this parameter leads to improved solution quality but also to a large increase in run time.

Since we iterate our local search algorithm with different expansion values within one call to our separator refinement algorithm as shown in Algorithm 1, the *maximum BFS expansion steps*, which is the number of iterations for different expansion values, is also a tuning parameters of our local search.

The final tuning parameter of our local search algorithm is the *maximum local search repetitions* which determines how often we perform the refinement algorithm per uncoarsening step.

Graph	$\lvert V_{cc}\rvert$	$\lvert E_{cc}\rvert$	$\lvert V\rvert$	$\lvert E\rvert$
Walshaw's Benchmark Archive				
wave	156 317	2 118 662	156 317	2 118 662
G2_circuit	150 102	576 572	150 102	576 572
2cubes_sphere	101 492	1 545 772	101 492	1 545 772
auto	48 695	6 629 222	448 695	6 629 222
fe_tooth	78 136	905 182	78 136	905 182
fe_body	30 581	226 848	45 087	327 468
vibrobox	12 328	330 500	12 328	330 500
bcsstk33	8 738	583 166	8 738	583 166
Road Networks				
eur	18 010 545	44 424 394	18 297 721	44 435 372

Table 6.1: Graph instances with total number of nodes $|V|$ and edges $|E|$. $|V_{cc}|$ and $|E_{cc}|$ denote the number of nodes and edges of the biggest connected component in each graph.

6.2.3 Instances

The graph instances we use for our experiments are obtained from Walshaw's Benchmark Archive [12] and from the 10th DIMACS Implementation Challenge [1, 2] where many graphs for benchmarking purposes are provided. These graphs are part of different categories such as sparse matrices, road networks and random geometric graphs which each have different properties. The graphs we used as well as their properties can be found in Table 6.1. Since our algorithm only works on connected graphs, we performed our experiments on the largest connected component, which is the largest subgraph that is connected, of each graph.

6.3 Node Separator Evaluation

We evaluate our algorithm by first showing the effect of different tuning parameters on solution quality and runtime. Through this evaluation we can define our *fast* and *strong* configurations, which we then compare to the KaHIP node separator (*KaHIP-NS*) algorithm by showing the improvement of our algorithm on different instances with different values of k. We also show the best found solutions of a variation of our algorithm where only our local search refinement is used and no evolutionary algorithm operations are applied. This variation will be referred to as *K-Plain* where as our normal algorithm will be referred to as *K-Evo* in the experimental evaluation.

6.3.1 Parameter tuning

In this section we look at the most important tuning parameters and show how they effect the solution quality and runtime of our algorithm. All results for parameter tuning are obtained by running our algorithm K-Evo with the specific parameter on multiple graph instances with three different random seeds. These three runs are averaged for each instance and then the geometric mean is computed over all averages. We repeat these tests with $k = \{2, 4, 16, 32, 64\}$ blocks and then average over all result values for each k to get the final results. In total the results therefore contain data from different instances, different values of k and different random seeds.

Maximum pair refinement steps

First we look at the tuning parameter *maximum pair refinement steps* as explained in Section 6.2.2. We refer to this parameter as M in this section. In Figure 6.1 we show how different values of M improve the separator size.

The improvement values were obtained by running our algorithm with $M = \{1, \ldots, 7\}$ as described above. The final results are shown in Figure 6.1. As we can see, a value of $M = 5$ gives the most improvement for our algorithm with a decrease in separator size of 1% which is why we choose this value for our strong configuration. Since bigger values for M actually increase the separator size, this tuning parameter seems counter productive but the final results in the following sections show that the positive effect of this parameter increases when it is used in conjunction with the other tuning parameters.

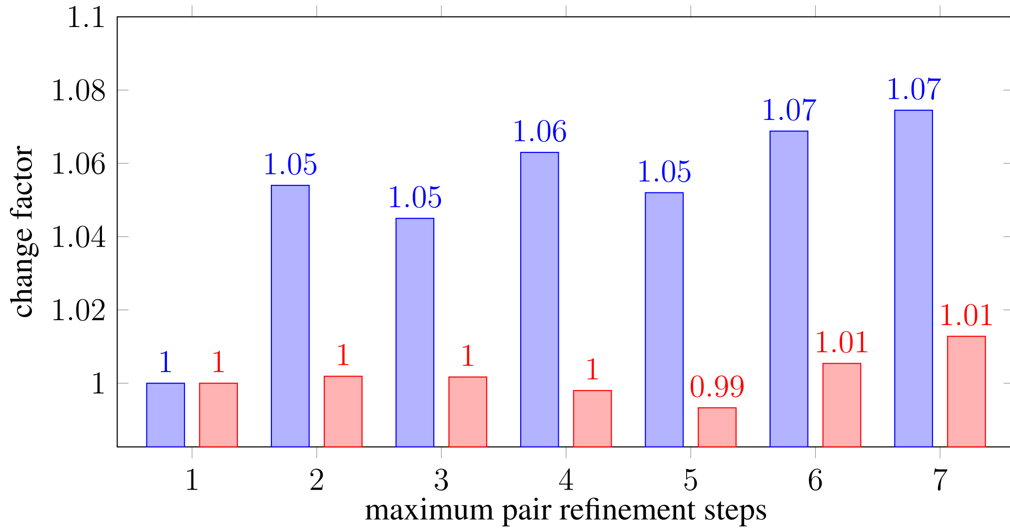


Figure 6.1: Node separator weight (red) and runtime (blue) change for a given value for the *maximum pair refinement steps* parameter compared to the minimum value of 1.

Maximum BFS expansion steps

Next we look at the tuning parameter *maximum BFS expansion steps* which is explained in Section 6.2.2. We refer to this parameter as B in this section. In Figure 6.2 we show how different values of B affect the separator weight as well as the runtime of our algorithm. The improvement values were obtained by running the parameter tuning procedure with parameter values $B = \{1, 3, 5, 7\}$. As we can see there is an increase in runtime duration of more than 50% while the separator weight goes down by at most 3%.

Maximum local search repetitions

We look at the tuning parameter *maximum local search repetitions* which is explained in Section 6.2.2. We refer to this parameter as L in this section. In Figure 6.3 we show how different values of L affect the separator weight as well as the runtime of our algorithm. The improvement values were obtained by running the above described procedure for parameter tuning with parameter values $L = \{1, \ldots, 5\}$. As the value for L increases the runtime of our algorithm increases linearly while the separator weight goes down by at most 2%.

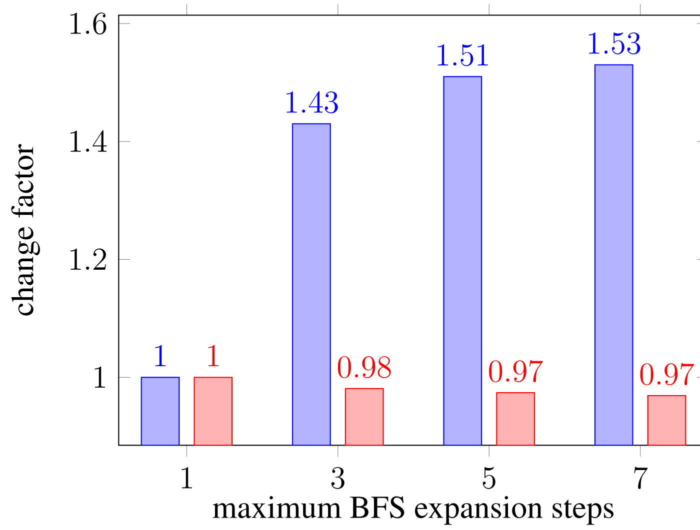


Figure 6.2: Node separator weight (red) and runtime (blue) change for a given value for the *maximum BFS expansion steps* parameter compared to the minimum value of 1.

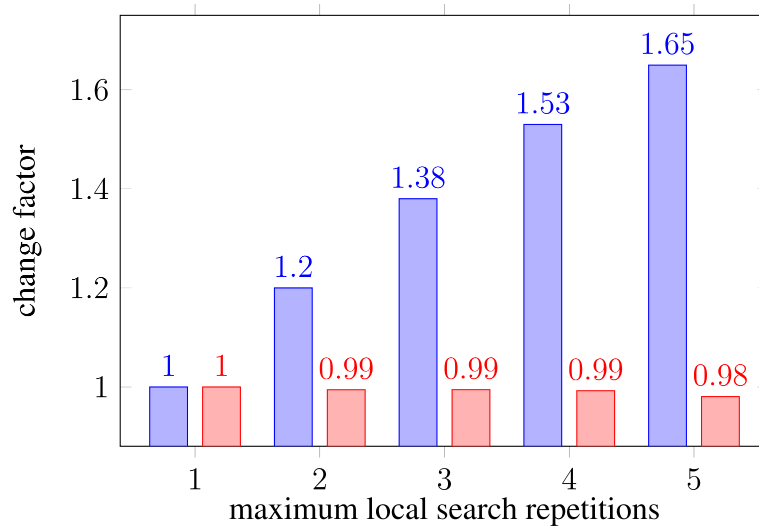


Figure 6.3: Node separator weight (red) and runtime (blue) change for a given value for the *maximum local search repetition* parameter compared to the minimum value of 1.

6.3.2 Effect of k on Runtime

Local Search

In Figure 6.4 we show the effect of different values of k on the runtime of the computation of a single individual for the initial population. Since no combine or mutation operation is performed, the runtime is mostly dominated by the computation of the initial separator and by our local search algorithm and the multilevel approach. We can see that as expected the runtime increases with increasing k but the increase is sub-linear. The reason for this is that although with increasing k the number of adjoint blocks and therefore the amount of local search refinements between two adjoint blocks increases, the blocks themselves are on average smaller. Since the number of nodes which are looked at by the refinement algorithm are bounded by the size of the blocks, the runtime per refinement operation decrease with increasing k.

Evolutionary Algorithm

In Figure 6.5, we show the effect of different values of k on the runtime of performing one combination operation. We can see that the runtime increases linearly for increasing values of k. A combination operation mainly prevents the matching algorithm in the coarsening phase of the multilevel approach from matching any edges between nodes that are part of the two node separators which are being combined. Larger values of k result in larger separators and therefore more edges that must not be matched by the matching algorithm. Since fewer edges are matched, the size reduction of the graph per coarsening step also decreases. This leads to more coarsening iterations and an increase in runtime.

When performing a combination operation we also use our local search refinement in order to improve the separator size. Comparing the effect of k on a combination operation and on our local search algorithm, we can see that although the combination operation is also performing local search refinement the overall runtime of one combination operation is only a fraction of the computation of a new individual with local search. One reason for this is that we do not have to compute a new initial separator during a combination operation. The second reason is that the runtime of our local search is smaller when performing a combination operation. This is due to the fact that only when the previous local search refinement iteration between two adjoint blocks improves the separator, will a potential next iteration be performed. This is less likely the case during the combination operation because the separator has already been refined when it was created during the computation of the individuals of the initial population.

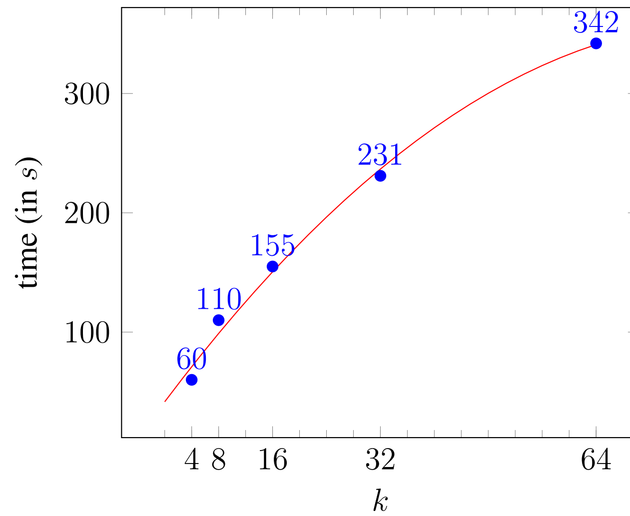


Figure 6.4: Time to generate a single node separator for a given value of k with a corresponding trendline (red). Time values are geometric means of multiple graph instances.

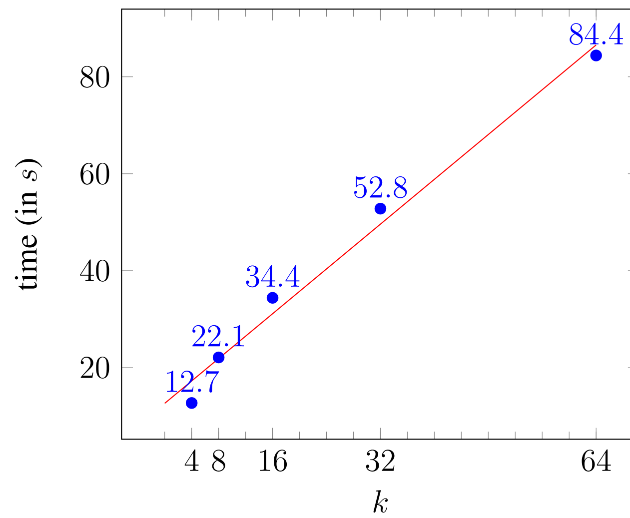


Figure 6.5: Time to perform to a single combination operation for a given value of k with a corresponding trendline (red). Time values are geometric means of multiple graph instances.

6.3.3 KaHIP Comparison

We now compare our algorithms K-Evo and K-Plain to KaHIP-NS, the already existing k-way node separator algorithm in KaHIP. KaHIP-NS first computes a k-way partition and then transforms the partition into a node separator by computing each partial separator between all adjacent blocks and combining these into the final separator. This method is also used to compute our initial separator which is described in more detail in Section 3.1.

The results were obtained by running each instance and parameter combination three times with different random seeds and then averaging the three results. All algorithms are running for the same amount of time in each particular instance we compare. Our results are divided into subtables for each value of k that was used. Each table shows the graph instance together with the node separator weight of the solution provided by all three algorithms. The results shown are averaged over three runs with different random seeds. Each table also shows the geometric mean over all results for each algorithm as well as the separator weight improvement of our algorithms over KaHIP-NS. We used an imbalance value of 4 for all experiments. Since our algorithms as well as KaHIP-NS have different standard configurations which determine the solution quality and runtime, we show results for two of these configurations, the *strong* and the *fast* configuration.

The results show that K-Evo produces on average about 18% and 23% smaller node separators respectively, for our two different configurations than KaHIP-NS. The biggest improvement over KaHIP-NS is 64% in *fast* configuration on the instance *vibrobox* with $k = 8$. Detailed improvement results are shown in Tables 6.4 and 6.5.

Strong Configuration

The *strong* configuration is used to produce the best solution quality at the cost of longer runtime of the algorithms. The parameters in this configuration for our algorithms can be seen in Table 6.2.

In Table 6.4 we compare the results of K-Evo and K-Plain to the results of KaHIP-NS for each instance and each value of k. Next to the node separator sizes we also show the improvement that K-Evo achieves. We can see that for the graph *vibrobox* and $k = 16$ we achieve the biggest improvement in this configuration of 57% which means our algorithm K-Evo found a separator 57% smaller than the separator computed by KaHIP-NS. We also show the geometric means of the results for each value of k which are also depicted in Figure 6.6a for K-Evo. The improvement values of the geometric means indicate an average improvement of 18% across all tested instances. In Figure 6.7 we show the improvement ratios for each instance and value of k sorted by improvement. There we can see that one graph (*eur*) has a worse separator for each value of k in comparison to KaHIP-NS. Since this graph is the only graph where our algorithm consistently performs worse and also the only road network graph we tested, it is likely that the lack of improvement is due to the inherent structure of road networks.

Fast Configuration

The *fast* configuration is designed to run as fast as possible while still producing acceptable results. The main difference between this configuration and the *strong* configuration is that we reduced the number of iterations for multiple operations as can be seen in the parameter setting in Table 6.3.

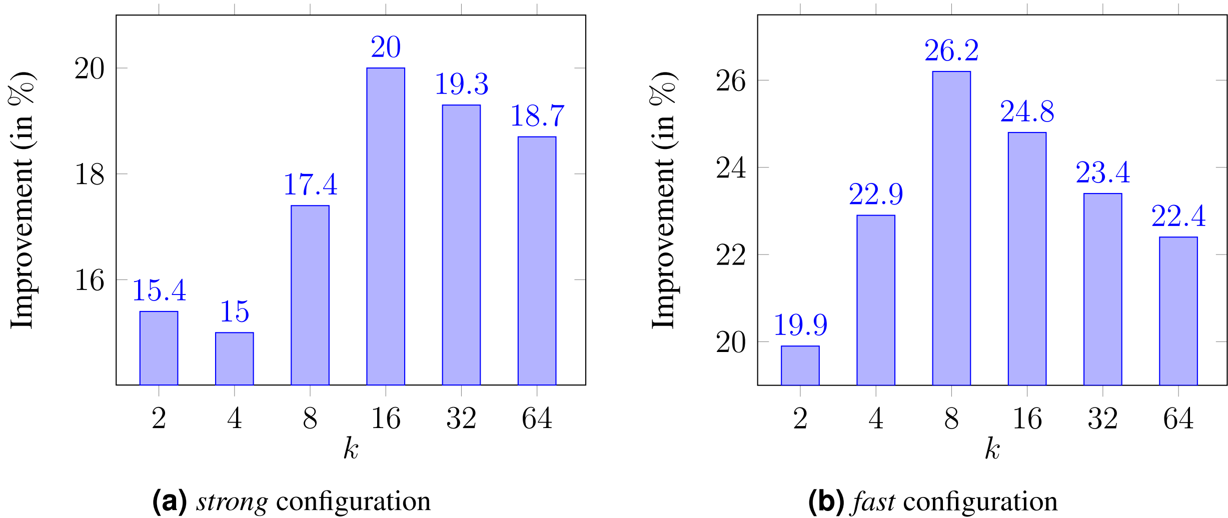
In Table 6.5 we compare the results of K-Evo and K-Plain to the results of KaHIP-NS in the same way as for the *strong* configuration. We can see that, for the graph *vibrobox* and $k = 8$, K-Evo achieves an improvement of 64% which is the largest improvement value for all tested instances and parameters in both configurations. The improvement values of the geometric means shown in Figure 6.6b indicate an average improvement of 23% across all tested instances. In Figure 6.8 we show the improvement ratios for each instance and value of k sorted by improvement. There we can see that no tested instance has an improvement ratio less than 1 which means K-Evo achieves better results than KaHIP-NS on all instances in the *fast* configuration.

Parameter	Value
inital population size	18
selection operation	tournament selection
combine to mutation ratio	9 : 1
time factor	20
maximum pair refinement steps	5
maximum BFS expansion steps	10
maximum local search repetitions	5

Table 6.2: Parameter settings for our *strong* configuration. A detailed explanation for each parameter can be found in Section 6.2.2.

Parameter	Value
inital population size	8
selection operation	tournament selection
combine to mutation ratio	1 : 0
time factor	10
maximum pair refinement steps	1
maximum BFS expansion steps	1
maximum local search repetitions	1

Table 6.3: Parameter settings for our *fast* configuration. A detailed explanation for each parameter can be found in Section 6.2.2.



(a) *strong* configuration

(b) *fast* configuration

Figure 6.6: Improvement of K-Evo over KaHIP-NS for the geometric means for a given value of k in *strong* and *fast* configuration.

Graph	$k = 2$				$k = 4$				$k = 8$			
	KaHIP-NS	K-Plain	K-Evo	Impr.	KaHIP-NS	K-Plain	K-Evo	Impr.	KaHIP-NS	K-Plain	K-Evo	Impr.
2cubes_sphere.mtx	1 859	1 380	1 368	26.4 %	5 000	3 456	3 430	31.4 %	8 815	6 298	6 311	28.4 %
auto	2 433	2 253	2 253	7.4 %	6 622	6 021	6 085	8.1 %	11 422	10 436	10 396	9.0 %
bcsstk33	436	418	418	4.1 %	966	815	804	16.8 %	1 791	1 328	1 330	25.7 %
eur	128	135	130	−1.6 %	302	399	383	−26.8 %	642	776	797	−24.1 %
fe_body	103	91	91	11.7 %	329	281	280	14.9 %	588	476	475	19.2 %
fe_tooth	1 179	940	952	19.3 %	2 139	1 729	1 722	19.5 %	3 515	2 886	2 861	18.6 %
G2_circuit.mtx	312	312	312	0 %	701	701	699	0.3 %	1 482	1 486	1 480	0.1 %
vibrobox	1 188	610	610	48.7 %	1 855	1 018	1 018	45.1 %	2 651	1 332	1 309	50.6 %
wave	2 353	2 160	2 160	8.2 %	4 537	4 233	4 224	6.9 %	7 680	7 125	7 078	7.8 %
Geometric Mean	668	568	566	15.4 %	1 507	1 290	1 281	15.0 %	2 683	2 220	2 217	17.4 %

Graph	$k = 16$				$k = 32$				$k = 64$			
	KaHIP-NS	K-Plain	K-Evo	Impr.	KaHIP-NS	K-Plain	K-Evo	Impr.	KaHIP-NS	K-Plain	K-Evo	Impr.
2cubes_sphere.mtx	12 395	8 866	8 613	30.5 %	17 000	12 163	11 714	31.1 %	22 588	16 729	15 825	29.9 %
auto	19 316	18 127	17 715	8.3 %	30 630	29 815	29 112	5.0 %	43 654	43 427	41 051	6.0 %
bcsstk33	2 957	2 147	2 116	28.4 %	4 134	2 966	2 840	31.3 %	5 511	4 015	3 910	29.1 %
eur	1 338	1 606	1 563	−16.8 %	2 349	2 822	2 751	−17.1 %	4 044	5 096	4 949	−22.4 %
fe_body	928	755	752	19.0 %	1 522	1 221	1 185	22.1 %	2 385	1 863	1 805	24.3 %
fe_tooth	5 366	4 272	4 112	23.4 %	7 699	6 206	5 957	22.6 %	10 646	8 598	8 125	23.7 %
G2_circuit.mtx	2 655	2 683	2 717	−2.3 %	4 425	4 375	4 287	3.1 %	7 043	6 985	6 727	4.5 %
vibrobox	5 016	2 308	2 180	56.5 %	6 189	3 328	3 227	47.9 %	6 761	3 867	3 715	45.1 %
wave	11 433	10 568	10 481	8.3 %	16 498	15 557	14 952	9.4 %	22 815	21 304	20 482	10.2 %
Geometric Mean	4 473	3 656	3 581	20.0 %	6 687	5 575	5 397	19.3 %	9 459	8 018	7 692	18.7 %

Table 6.4: Comparison of KaHIP-NS, K-Plain and K-Evo for $k = \{2, 4, 8, 16, 32, 64\}$, all with *strong* configuration enabled. The Improvement values refer to the comparison between KaHIP-NS and K-Evo.

Graph	$k = 2$				$k = 4$				$k = 8$			
	KaHIP-NS	K-Plain	K-Evo	Impr.	KaHIP-NS	K-Plain	K-Evo	Impr.	KaHIP-NS	K-Plain	K-Evo	Impr.
2cubes_sphere.mtx	2 175	1 453	1 405	35.4 %	5 269	3 886	3 559	32.5 %	9 064	6 819	6 789	25.1 %
auto	2 653	2 352	2 333	12.1 %	7 317	6 360	6 216	15.0 %	12 280	10 764	10 481	14.6 %
bcsstk33	466	432	422	9.4 %	1 163	845	855	26.5 %	1 884	1 355	1 334	29.2 %
eur	198	165	190	4.0 %	587	494	466	20.6 %	1 149	949	932	18.9 %
fe_body	119	91	91	23.5 %	346	298	283	18.2 %	633	509	498	21.3 %
fe_tooth	1 294	1 057	1 045	19.2 %	2 317	1 836	1 769	23.7 %	3 789	3 050	2 943	22.3 %
G2_circuit.mtx	360	313	312	13.3 %	745	704	700	6.0 %	1 580	1 527	1 498	5.2 %
vibrobox	1 083	610	628	42.0 %	1 987	1 086	1 066	46.4 %	3 726	1 620	1 337	64.1 %
wave	2 443	2 161	2 160	11.6 %	4 727	4 448	4 313	8.8 %	8 396	7 285	7 186	14.4 %
Geometric Mean	753	597	603	19.9 %	1 741	1 388	1 342	22.9 %	3 126	2 399	2 309	26.2 %

Graph	$k = 16$				$k = 32$				$k = 64$			
	KaHIP-NS	K-Plain	K-Evo	Impr.	KaHIP-NS	K-Plain	K-Evo	Impr.	KaHIP-NS	K-Plain	K-Evo	Impr.
2cubes_sphere.mtx	13 245	9 943	8 970	32.3 %	17 988	14 485	12 626	29.8 %	24 102	20 006	16 852	30.1 %
auto	21 194	18 680	18 165	14.3 %	33 273	31 841	31 131	6.4 %	47 316	48 271	43 967	7.1 %
bcsstk33	2 917	2 297	2 246	23.0 %	4 125	3 211	3 080	25.3 %	5 453	4 266	4 162	23.7 %
eur	2 090	1 841	1 773	15.2 %	3 521	3 178	3 019	14.3 %	6 002	5 638	5 374	10.5 %
fe_body	1 025	816	783	23.6 %	1 659	1 304	1 241	25.2 %	2 607	2 001	1 891	27.5 %
fe_tooth	5 862	4 542	4 313	26.4 %	8 317	6 547	6 106	26.6 %	11 481	9 068	8 450	26.4 %
G2_circuit.mtx	3 297	2 931	2 799	15.1 %	5 319	4 623	4 402	17.2 %	7 938	7 303	6 904	13.0 %
vibrobox	5 000	2 666	2 453	50.9 %	6 051	3 657	3 361	44.5 %	6 770	4 136	3 842	43.2 %
wave	12 345	10 881	10 741	13.0 %	18 010	16 490	15 416	14.4 %	24 651	23 781	21 612	12.3 %
Geometric Mean	5 038	3 974	3 790	24.8 %	7 436	6 070	5 696	23.4 %	10 445	8 756	8 110	22.4 %

Table 6.5: Comparison of KaHIP-NS, K-Plain and K-Evo for $k = \{2, 4, 8, 16, 32, 64\}$, all with *fast* configuration enabled.
The Improvement value refers to the comparison between KaHIP-NS and K-Evo.

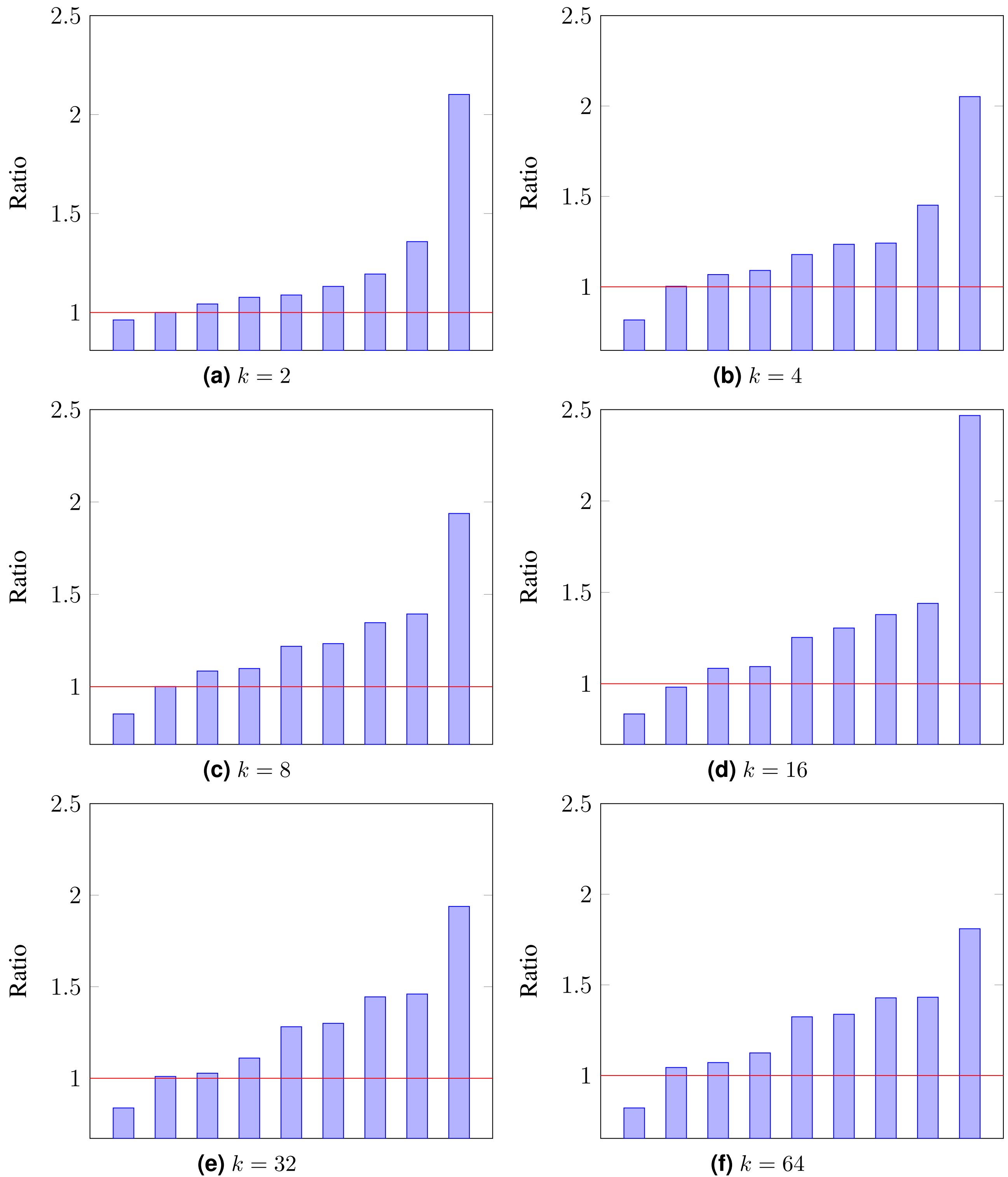


Figure 6.7: Improvement ratios of K-Evo per instance over KaHIP-NS for $k = \{2, 4, 8, 16, 32, 64\}$ in *strong* configuration.

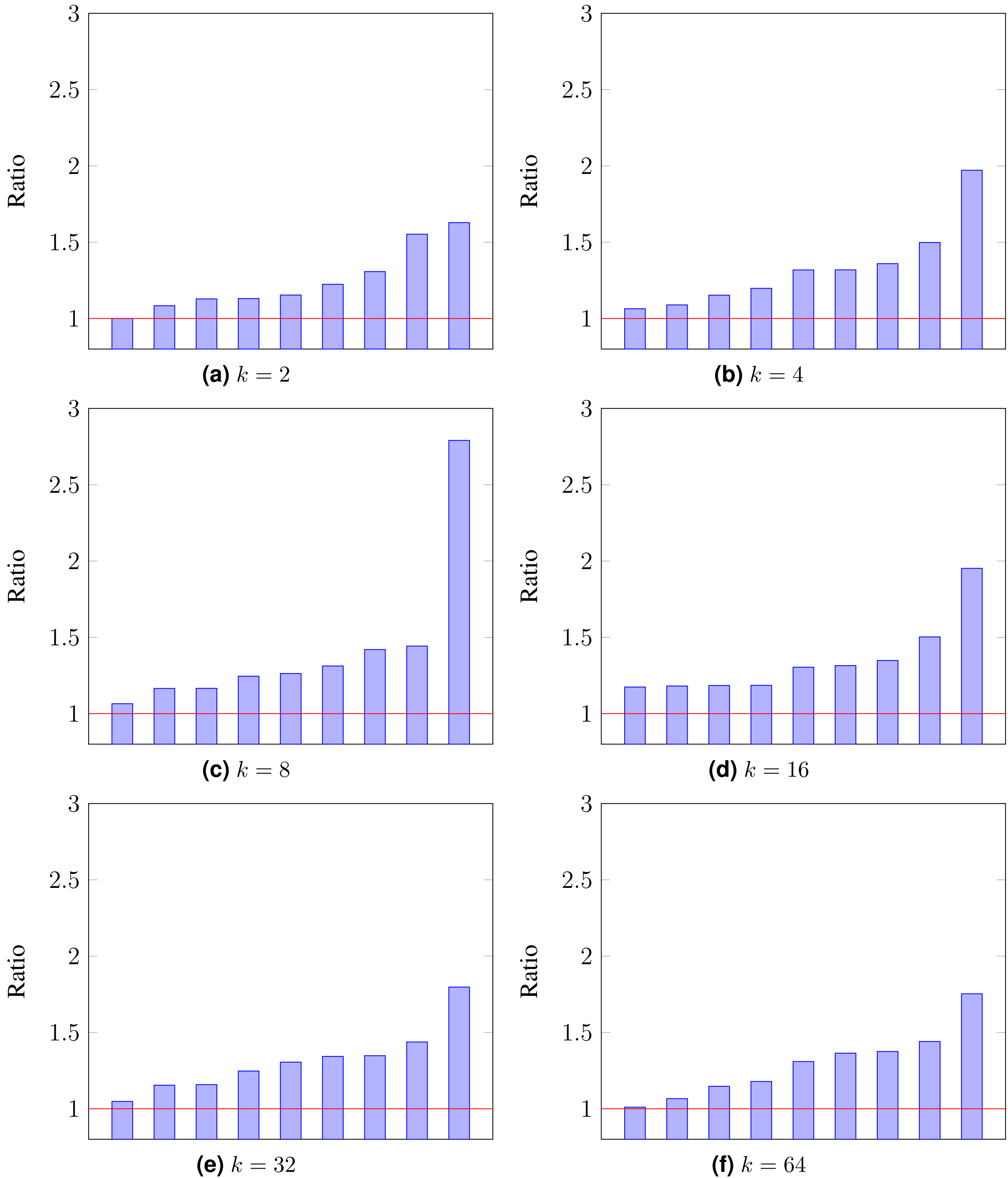


Figure 6.8: Improvement ratios of K-Evo per instance over KaHIP-NS for $k = \{2, 4, 8, 16, 32, 64\}$ in *fast* configuration.

Analysis

The Figures 6.6a and 6.6b, which show the improvement means of K-Evo over KaHIP-NS for different values of k, have a distinct peak at $k = 16$ for the *strong* configuration and at $k = 8$ for the *fast* configuration where the maximum improvement values are reached. This peak arises due to the two main techniques we use to improve the separator solution quality: local search and the evolutionary algorithm.

Our local search refinement, as the name suggests, only locally improves the separator because it only looks at two adjoint blocks at a time. This means that if there are only few blocks in total, the local search looks at a larger percentage of the graph at a time and is therefore more likely to find solutions close to the global optimum. With increasing k the final separator size is predominantly determined by the initial separator used in the multilevel approach and the local search refinements on the coarser levels of the graph hierarchy where big parts of the graph are represented by just a few nodes. Local search on the finer levels only looks at a small percentage of the graph at a time and can not modify the overall structure of the separator. Therefore the overall improvement for bigger values of k decreases.

On the other hand there is the combination operation of the evolutionary algorithm which acts more globally than the local search. In conjunction with the multilevel approach it can exchange large parts of the separator with a second separator. For bigger values of k the separator is larger and more complex and is thus represented by many nodes even in the coarsest level of the graph hierarchy where the combination operation takes place. This means that we can more easily exchange parts of the separator when it is large in comparison to the graph itself which happens when k is also large. When k is small then the separator is only represented by very few nodes on the coarsest level and it is therefore more unlikely that the combination operation can exchange any relevant parts of the separators and bring a significant improvement in separator size.

To summarize, local which is more effective for smaller values of k because is acts locally and we have the evolutionary algorithm which is more effective for bigger values of k because it acts more globally. Since we combine these two techniques in our algorithm, the effects of both techniques on the separator size accumulate. This results in a value of k where both techniques noticeably contribute to the improvement, which is represented as the peaks in the aforementioned improvement figures of both configurations.

Convergence

In Figures 6.9 and 6.10 we show the convergence behavior of K-Evo and K-Plain compared against KaHIP-NS for different values of k in both configurations. Each curve in each convergence plot corresponds to one of the three algorithms. A curve depicts the behavior of the geometric mean of all tested instances over time. For each tested instance we record the best separator size for each point in time throughout the runtime of the algorithm. The time values of each instance are normalized by linearly mapping them to the interval $[0, 100]$ where 0 represents the start and 100 the end of the algorithm. The geometric mean is computed by using the normalized time values. This means that each point $P = (t, g)$ on the curve represents the geometric mean g of the separator sizes of all instances at $t\%$ of their runtime. This allows us to compare all algorithms at different percentages of their runtime duration.

We can see that KaHIP-NS converges very fast in both configurations although the final solution is always significantly worse than the solutions of K-Evo and of K-Plain. When we compare K-Evo against K-Plain we notice that they converge similarly fast for the first 40% of the duration. In this time period K-Evo computes the initial population which is basically the same procedure as K-Plain uses for the computation of the node separators. In the remaining 60% of the duration K-Evo applies the evolutionary algorithm which can be seen in the figures as a sudden improvement of the separator size. The improvement arising from the evolutionary algorithm is more noticeable for bigger values of k which is consistent with the conclusions of the analysis Section 6.3.3.

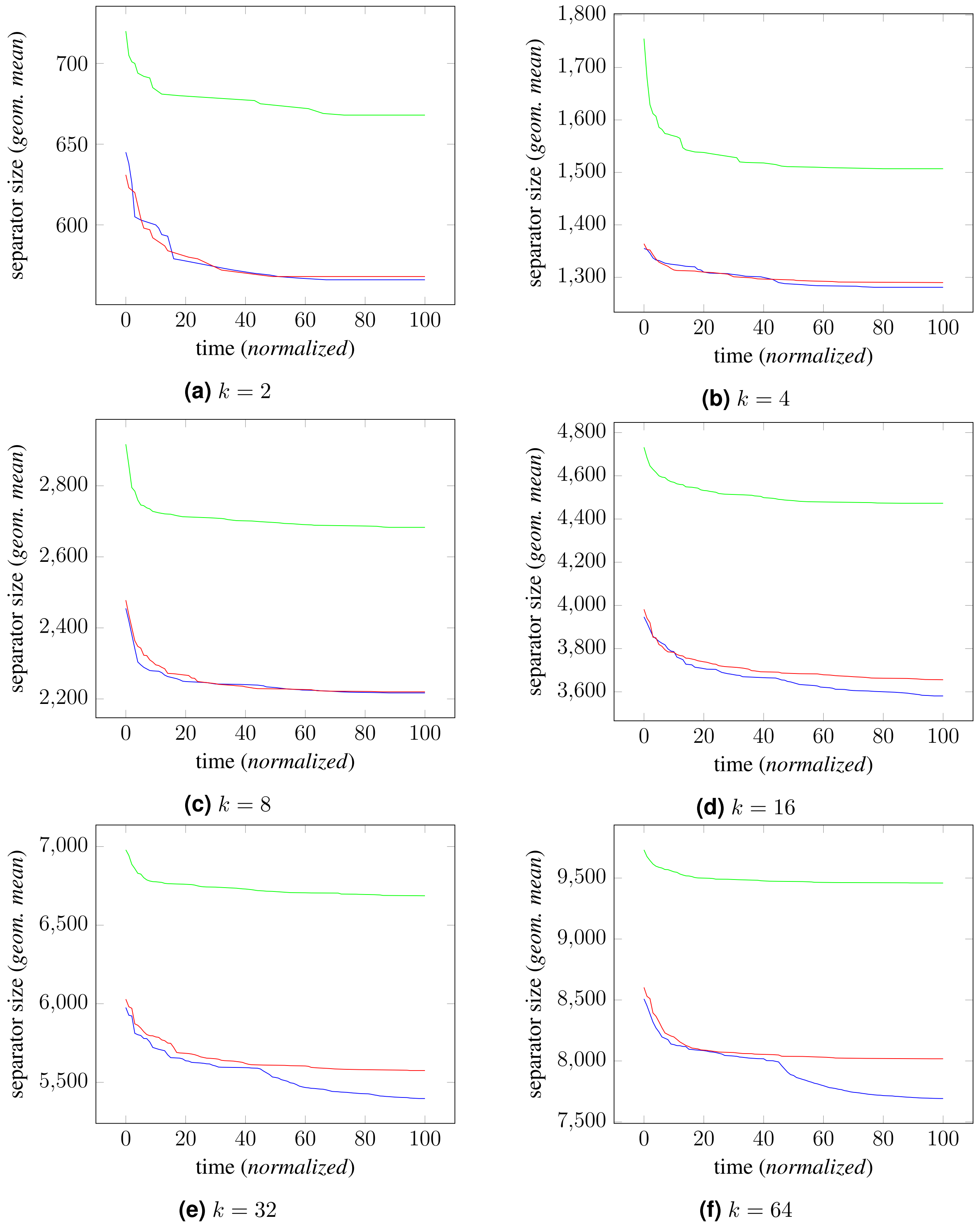


Figure 6.9: Convergence comparison of KaHIP-NS (green), K-Plain (red) and K-Evo (blue) for multiple values of k in *strong* configuration.

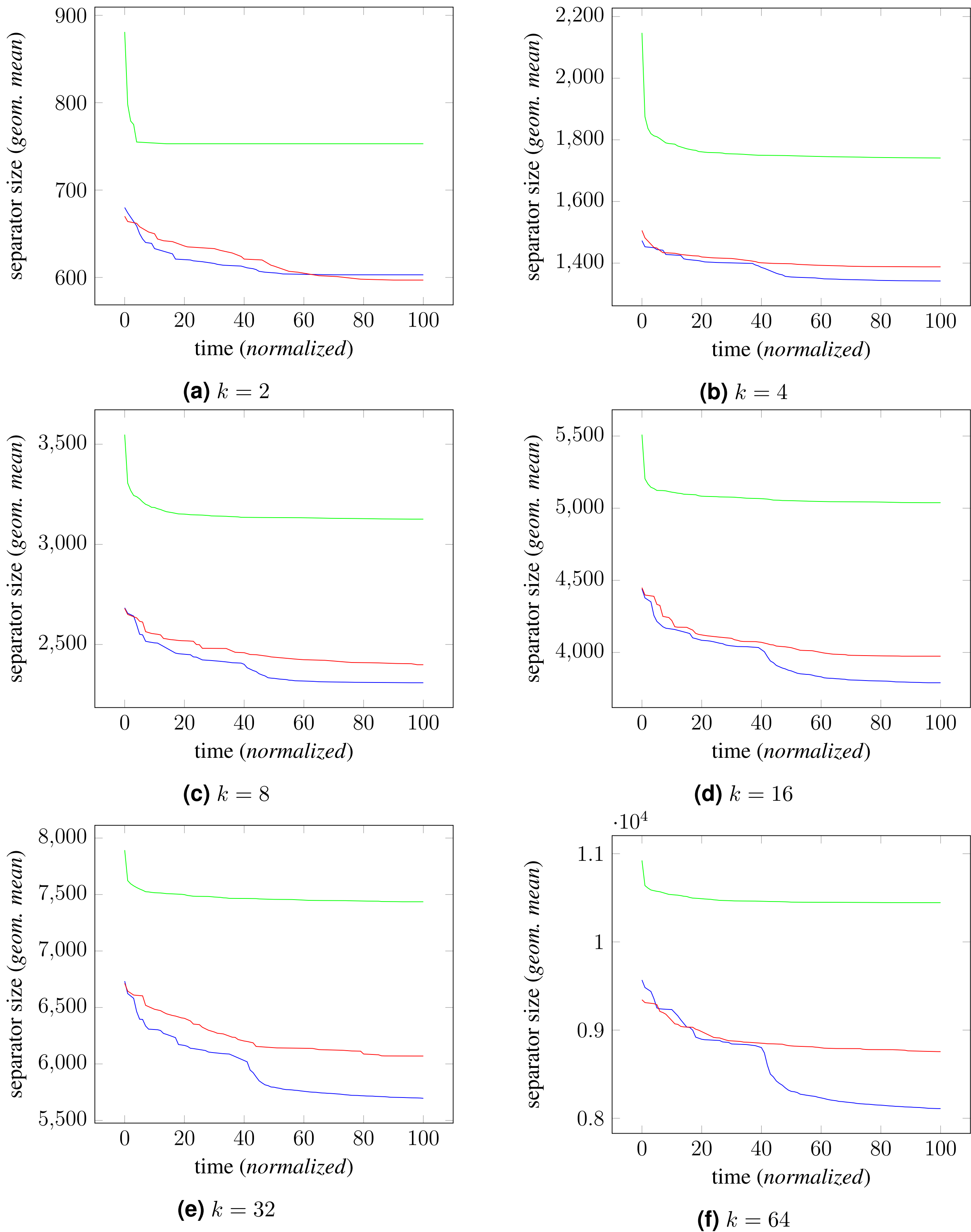


Figure 6.10: Convergence comparison of KaHIP-NS (green), K-Plain (red) and K-Evo (blue) for multiple values of k in *fast* configuration.

7 Conclusion

The goal of this thesis was to design and implement a new algorithm for computing small k-way node separators which fulfill a given balance constraint. We developed a novel k-way local search refinement method which also incorporates a new way of balancing k-way node separators. Additionally, we presented an evolutionary algorithm which operates on node separators. The idea of our algorithm is to combine the two ways of improving node separators in order to improve our solutions locally with the local search and globally with the evolutionary algorithm.

The experimental evaluation of our algorithm in comparison to the KaHIP k-way node separator algorithm shows that we can improve almost all node separator solutions of KaHIP-NS for our tested instances with an average improvement of 18% and 23% respectively for our *strong* and *fast* configuration. The evaluation also shows that the local search and the evolutionary algorithm work well in conjunction with each other. The k-way local search works best with smaller values of k where local optimization dominates the final separator size whereas the evolutionary algorithm works best for large values of k where global optimization dominates the separator size. This allows our algorithm to produce small k-way node separators for any value of k.

7.1 Future Work

A straightforward way to make our algorithm faster is to parallelize the computation of the individuals of the initial population as well as perform multiple combination and mutation operations in parallel. The local search refinement could also be parallelized using a shared memory model. In order to increase the solution quality, other methods for the different operations of the evolutionary algorithm and different gain functions for k-way balancing could be implemented.

The balancing operation can be expanded to work on arbitrary graphs so that our whole algorithm can be applied to arbitrary disconnected graphs. This enables our algorithm to be used for other applications like shortest path computation for route planning or graph compression.

Bibliography

[1] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. *Benchmarking for Graph Clustering and Partitioning*, pages 73–82. Springer New York, New York, NY, 2014.

[2] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. *Graph Partitioning and Graph Clustering*. Contemporary Mathematics. American Mathematical Society, 2013.

[3] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996.

[4] G. Brassard and P. Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[5] T. Nguyen Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[6] K. A. De Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, 2006.

[7] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177 – 189, 1979.

[8] B. L. Miller and D. E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[9] P. Sanders and C. Schulz. Distributed evolutionary graph partitioning. In *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments (ALENEX '12)*, pages 16–29, 2012.

[10] P. Sanders and C. Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[11] P. Sanders and C. Schulz. Advanced multilevel node separator algorithms. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*, pages 294–309, 2016.

[12] A. J. Soper, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.

[13] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

[14] C. Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1):325–372, 2004.