Bachelor Thesis

Evolutionary Algorithms For Independent Sets

Sebastian Lamm

1633214

September 29, 2014

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

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

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen also solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 29.09.2014

Sebastian Lamm

Abstract

An independent set of a graph $G = (V, E)$ is a subset $S \subseteq V$, such that there are no adjacent nodes in S. The independent set problem is that of finding the maximum cardinality set among all possible independent sets.

We present a novel evolutionary algorithm for this problem. Our algorithm starts by generating a population of individuals by using greedy algorithms to create initial independent sets. We then use tournament selection for picking individuals, called parents, for mating. We introduce four new ways to combine the selected individuals. The first procedure creates new individuals by building the overlap of the parents independent sets. The other schemes make use of graph partitioning to produce new offsprings applying both 2-way and k-way partitions as well as node separators. We use them to quickly exchange whole parts of the given independent sets. For example, we make use of node separators obtained from the partitioner to generate new independent sets in the first of these combine operators. We also provide an indirect approach by using vertex covers and partitions in our third operator. Finally, we use k-way partitions and node separators to build a multiple-point combination operator. All newly created sets are afterwards improved by using the fast local search algorithm by Andrade et. al. [1].

We present an experimental evaluation of our algorithm using instances from the literature as well as ones proposed by Andrade et. al. [1]. Our algorithm performs particularly well on social network graphs and the meshes which were introduced by Andrade et. al. [1]. Notable improvements can also be found on instances from Walshaw's Partition Archive, which are common benchmarks for graph partitioners. In the initial phases of almost all test runs we performed, our algorithm is able to establish better solutions. Additionally, we are able to show that graph partitioners benefit our algorithm when compared to more simplistic methods.

Zusammenfassung

Eine unabhängige Menge eines Graphen $G = (V, E)$ ist eine Untermenge $S \subseteq V$, so dass sich in S keine adjazenten Knoten befinden. Das Problem maximaler unabhängiger Mengen beschreibt die Suche nach der unabhängigen Menge mit maximaler Mächtigkeit unter allen möglichen unabhängigen Mengen eines Graphen.

Wir präsentieren einen neuen evolutionären Algorithmus für dieses Problem. Unser Algorithms beginnt mit dem Erzeugen einer Population von Individuen, indem er mithilfe von Greedy-Algorithmen erste unabhängige Mengen erzeugt. Wir verwenden anschließend Turnierauswahl, um Individuen, Eltern genannt, zur Paarung auszuwählen. Wir führen vier neue Wege ein, um die ausgewählten Individuen zu kombinieren. Die erste Prozedur erzeugt neue Individuen, indem sie den Schnitt der unabhängigen Mengen der Eltern bildet. Die anderen Schemata nutzen Graphpartitionierung, um neue Nachkommen zu erzeugen. Hierfür machen sie sowohl von 2-wege, als auch von k-wege Partitionen und Knotenseparatoren Gebrauch. Diese dienen uns dazu, schnell ganze Teile der gegebenen unabhängigen Mengen auszutauschen. Beispielsweise nutzen wir im ersten dieser Operatoren Knotenseparatoren, die wir vom Partitionierer erhalten, um neue unabhängige Mengen zu erzeugen. In unserem dritten Operator stellen wir auch einen indirekten Ansatz zur Verfügung, der Knotenüberdeckungen und Partitionen verwendet. Zu guter Letzt setzen wir k-wege Partitionen und Knotenseparatoren ein, um einen Kombinationsoperator zu erzeugen, der in der Lage ist, mehrere Teile von Lösungen aufeinmal auszutauschen. Sämtliche unabhängige Mengen, die wir auf diese Weise neu erzeugen, werden anschließend durch die Anwendung der schnellen lokalen Suche von Andrade et. al. [1] verbessert.

Wir präsentieren eine experimentelle Auswertung unseres Algorithms unter Verwendung von Instanzen aus der Literatur. Ebenso testen wir einige der Instanzen, die bereits von Andrade et. al. [1] genutzt wurden. Unser Algorithmus liefert vorallem bei den von Andrade et. al. [1] verwendeten Meshes und sozialen Netzwerken merkbare Verbesserungen. Auch bei Instanzen aus Walshaw's Partition Archive sind deutliche Steigerungen erkennbar. In den Anfangsphasen der meisten Testdurchläufen hat unser Algorithmus bessere Lösungen erzeugt als unsere Konkurrenz. Zusätzlich werden wir experimentell nachweisen, dass Graph Partitionierer im Vergleich zu einfacheren Vorgehen, unserem Algorithmus einen Nutzen bringen.

Acknowledgments

I would like to give thanks to everyone who supported and encouraged me during my work on this thesis. Without all the people that cheered me up after the occasional setbacks, I wouldn't have gotten so far.

First of all, I owe special thanks to my supervisors Dr. Christian Schulz and Prof. Dr. Peter Sanders for the chance of doing research on such an interesting topic and a lot of good advice. Furthermore, I would like to thank Renato Werneck for providing me with the source code of their algorithm as well as graph data. Thanks again to my family and girlfriend for their love and comfort. Last but not least, I would like to thank my friends for their support and relief. My sincerest thanks to all of you!

Contents

1. Introduction

Over the last few years evolutionary algorithms have expanded their grasp at a breathtaking rate and our knowledge of the field today is more profound than some people might have ever imagined. Applying the principles of evolution and natural selection inspired by Darwin's theory of evolution has intrigued many researchers for decades to come. Along the rise of interest came a broader field of applications. Ranging from all kinds of problems, evolutionary algorithms are especially prominent in the fields of artificial intelligence and optimization. Even tough the simplicity of evolutionary algorithms can often be surprising at first, it is the combination of simple operations that gives them their strength and robustness.

Another important field of research nowadays are graph algorithms. Especially with the huge impact social networks had on our lives over the last couple of years, the interest for gaining and extracting information about them had a dramatic increase. Since a lot of these structures contain a huge amount of data and are fairly irregular, the room for new optimization methods, that exploit these characteristics, is being made.

One of these graph optimization problems is the search for the maximum independent set (MIS). This problem tries to find a maximum set of nodes in a graph, such that no two of these are connected to each other. Even tough plenty algorithms are dedicated to finding such a set in a given graph, little effort has been put into the application of the natural operators evolutionary algorithms have to offer. The MIS problem deserves our attention, not only because of its numerous direct applications in computer vision [2], pattern recognition [22], map labeling [30] and so forth, but also because it closely related to other optimization problems like finding a maximum clique or a minimum vertex cover. Since all of these problems are NP-hard [9], we are not able to solve all instances in polynomial time. Therefore, algorithms, that want to perform well on a broad range of graphs, have to use some sort of heuristic to get good results in a reasonable time. The problem with these algorithms is that they often provide insufficient approximations or are designed for a very specific set of graphs. Again the robustness of evolutionary algorithms sounds like a good match for this kind of problems. Most algorithms dedicated to finding a MIS on general graphs use and maintain a single solution and try to

improve it by applying a chosen heuristic. One of these algorithms, which forms the basis of the evolutionary algorithm presented in this thesis, is the local search algorithm by Andrade et. al. [1]. This algorithm tries to improve a single solution by using perturbations and simple 2-improvements. A k-improvement tries to remove $k-1$ nodes from the graph in such a way, that k new nodes can be added in turn. A method for using 3-improvements is also presented in their work [1], but will not be discussed much further.

1.1 Contribution

At first this thesis will define and cover the fast local search algorithm for finding independent sets by Andrade et. al. [1]. This will lay the foundation for the newly created evolutionary algorithm and help us understand what benefits we get from it. Additionally, we will briefly discuss the basics of evolutionary algorithms like selection and combination. In this context we will see how graph partitioners, in our case the KaHIP-framework (Karlsruhe High Quality Partitioning) by Peter Sanders and Christian Schulz [26], can help us build natural combine operators. We then present the evolutionary algorithm in detail ranging from the chosen selection-schemes to the combine operators and replacement techniques. Our algorithm uses partitions and node separators to generate sensitive combine operators that ensure a population of valid independent sets. Since we want to see how the algorithm works in practical applications, and especially how it compares to the iterated local search presented by Andrade et al. [1], a large emphasis is put on the experimental evaluation. To assure a fair competition they were performed on families of the original instances presented in [14, 32, 28] and extended by Andrade et al. [7, 23] as well as on some new instances that we added.

1.2 Overview

Section 2 will introduce the notation and necessary definitions this thesis will be using. Additionally, we present the reoccurring algorithms which are used for certain parts of our implementation. We will then cover the key aspects on which our work is based upon and give suggestions for further readings for each of these topics in Section 3. The main algorithm in all its details will be presented in Section 4. Section 5 concerns itself with the results of the experimental evaluation of our work in contrast to the iterated local search algorithm. Finally, we will conclude this thesis in Section 6 and present some possibilities for future work.

2. Preliminaries

This section introduces the basic algorithms, data structures and notation on which our evolutionary algorithm is build upon.

2.1 Graph Definitions

A *graph* G is a set of objects, called *nodes* (vertices), that are connected via *edges*. We define $V(G) = \{1, \ldots, n\}$ as the set of nodes belonging to a graph G and $E(G) \subseteq V(G) \times V(G)$ as its edges. Therefore, a graph can be written as a tuple $G = (V, E)$. We denote the number of nodes as $n = |V|$ and $m = |E|$ as the number of edges. Two nodes $u, v \in V$ are connected if there exists an edge $e \in E$ with $e = (u, v)$. Such an edge is called *directed*. Since our main focus is on undirected graphs, we denote edges as sets $e = \{u, v\}$, meaning that u is connected to v and vice versa. The set of neighbours for any node $v \in V(G)$ is defined as $N(v) = \{u \in V(G) \mid \exists e \in E(G) : e = \{u, v\}\}$. The *degree* of a node is denoted as $\Delta(v) = |N(v)|$ and the maximum degree of G as Δ or $\Delta(G)$. Furthermore, an undirected graph is called connected, if for any given nodes $u, v \in V$ there exists a path of edges from one node to the other. Throughout this thesis we limit ourselves to simple graphs, thereby leaving self-loops and parallel edges between nodes aside. The complement of a graph G is defined as $\overline{G} = (V, \overline{E})$ with $\overline{E}$ being the complement of E.

Given a simple undirected and connected graph $G = (V, E)$, an *independent set* is a subset of vertices $S \subseteq V$, such that there are no two adjacent nodes in S. An equivalent definition would be that no edge contains more than one start-/endpoint in S. An independent set is maximal, if it is not a subset of any larger independent set. Finally, the *maximum independent set* (MIS) problem is the search for the independent set that is the largest of these subsets in terms of cardinality. This problem is NP-hard [9] and therefore finding an optimal solution can take an exponential amount of time.

As mentioned before, the maximum independent set problem is also closely related to other common (NP-hard) graph problems like the maximum clique problem, which is also mentioned in [1]. Among these problems the minimum

vertex cover problem is the one which will be influencing our algorithm the most. A *vertex cover* is a subset of nodes $C \subseteq V(G)$, such that every edge $e \in E$ is at least incident to one node within the set. The *minimum vertex cover problem*, analogues to the MIS problem, searches for the vertex cover with the least amount of nodes. The interesting part about the vertex cover problem for our purposes is the fact that the complement of a vertex cover $V \setminus C$ always is an independent set by definition. Furthermore, complementing a minimum vertex cover results in a maximum independent set.

Another type of graphs that will be used in our algorithm are *bipartite graphs*. A bipartite graph consists of two disjoint sets, usually called U and V, which are independent sets on their own. Therefore, every edge in such a graph connects a node $u \in U$ to a node $v \in V$. Our main point of interest concerning bipartite graphs are bipartite matchings. A *matching* is a subset of edges $M \subseteq E$, such that no two edges share a common start-/endpoint. A node is called *matched*, if it is an start-/endpoint of an edge $e \in M$ and *unmatched* otherwise. A maximum matching is a matching of maximum cardinality, meaning that it contains the largest number of edges possible. In addition to that, a matching is called *perfect*, if it matches all nodes in a given bipartite graph.

2.2 Data Structures and Algorithms

This section describes the data structures and algorithms that are used throughout this thesis.

2.2.1 Graph Representation

The graph data structure we used in our work is an adjacency structure of the graph. This structure is a compressed sparse row format, which is a widely used scheme for storing graphs. Within this data structure nodes are numbered from 0 to $n - 1$, as opposed to our previous definition.

Given a graph G, that consists of n nodes and m edges, we store its adjacency structure in two arrays *xadj* and *adjncy*. The *adjncy* array is used to store the neighbours of every node. It therefore has to contain $2m$ elements, because we have to count edges from both directions. The *xadj* array is used to indicate where in the *adjncy* array the neighbours for a certain node are located. For example, the adjacency list of a node v starts at index $xadj[v]$ and ends at (but not includes) index $xadj[v + 1]$. This means that all neighbouring nodes for v can be found by starting at $adjncy[xadj[v]]$ and continuing until $adjncy[xadj[v + 1] - 1]$. Since *xadj* stores the start and end index for each node it has to be of size $n + 1$.

The file format we used for reading graphs is a simplification from the one used by Metis [16], Chaco [11], KaHIP [25] and during the 10th DIMACS Implementation Challenge. An example for a simple graph and its representation can be seen in Figure 2.1.

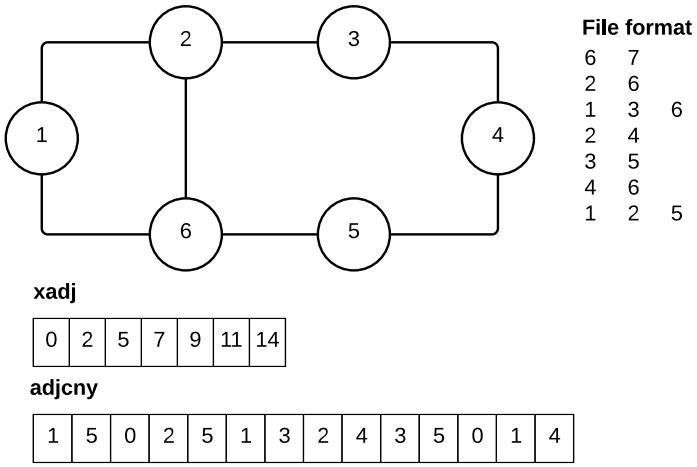


Figure 2.1: Example of a simple graph and its representation.

2.2.2 Priority Queues

A *Priority Queue* is an abstract data type that acts similar to a regular queue, but where each elements contains an additional priority. Priority queues are designed in such a way, that the element with the highest priority is always at the front. In general, the elements of the queue are sorted according to some strict weak ordering. The basic operations of a priority queue are:

- **push(element, priority)** Insert an element with an associated priority to the queue.
- **pop_highest_priority()** Returns the element with the highest priority and removes it from the queue.

In addition to this, most priority queues offer a *peek* operation, that allows to look at the element with the highest priority without removing it. The most

common implementation of a priority queue is using a heap data structure. However, we decided to use *bucket priority queue*, which maintains a number of buckets associated with each priority. This data structure is also used in the original work by Andrade et. al. [1]. It allows us to easily and efficiently pick a random element from a number of equally prioritized elements. For more information on priority queues the reader is referred to [4].

2.2.3 Breadth First Search

Breadth First Search or BFS is a graph search algorithm that starts at a root node and traverses the graph in a layered manner. BFS is usually implemented using a FIFO queue. The main algorithm removes the first node $s \in V(G)$ from the queue and then checks if any of the neighbouring nodes has already been looked at. If this is not the case it is added to the queue and marked as visited. This process is repeated until the target node has been found or no nodes are left to be added.

2.2.4 Hopcroft-Karp

The *Hopcroft-Karp* algorithm introduced in [13] is used to find a maximum matching in a bipartite graph. It uses augmenting paths to procedurally improve an initially empty matching. The main advantage of the Hopcroft-Karp algorithm to other augmenting path techniques is that, instead of searching for paths one by one, it simultaneously looks for several paths at once. This algorithm has the best known worst-case runtime $\mathcal{O}(|E|\sqrt{|V|})$ for the bipartite matching problem.

2.2.5 König's Theorem

König's Bipartite Matching Theorem states that for any bipartite graph G the cardinality of a maximum matching is equal to the cardinality of a minimum vertex cover. For a proof of this theorem the reader is referred to [17]. This theorem, combined with the Hopcroft-Karp algorithm, can be very helpful when building independent sets. We will discuss this aspect further in Section 4.4.3.

3. Related Work

3.1 Evolutionary Algorithms

An *evolutionary algorithm* can be classified as a generic optimization algorithm that tries to mimic the biological mechanisms of evolution and natural selection. They are praised for their robustness and universal applicability on different problems and search spaces alike [10]. Another thing that differentiates them from traditional optimization methods is that they work with a whole set of solutions as opposed to a single current solution. Each solution is encapsulated in a so called *individual*. The set of individuals on which every evolutionary algorithm is based upon is called the *population*.

To reproduce the effects of evolution, these algorithms use a combination of simple operations, which basically consist of selection, mating or reproduction and mutation. Newly generated individuals, called *offsprings*, are then inserted into the population based on certain criteria. If a certain number of iterations has passed, the resulting population is usually called a *generation*. A simple overview over the main steps of an evolutionary algorithm can be seen in Figure 3.1.

The selection of individuals for reproduction is based on a so-called *fitness function* $f : P \mapsto \mathbb{N}_0$, which returns a numeric value for each individual. After the best-fit individuals are chosen as *parents*, they create one or more new offsprings based on crossover operations. Additionally, the new offsprings are then able to undergo mutation with a certain probability. Mutation adds some additional diversity or perturbation to the mating process. The probability for mutation in evolutionary algorithms is generally very low to reduce the amount of randomness introduced.

Using these operations evolutionary algorithms enforce strong (well-fit) individuals to contribute their genetic material and weak individuals to slowly fade out of existence. The algorithm ends when a certain termination criterion is met. This criterion can be anything from a fixed number of iterations to a time limit. For more information about evolutionary algorithms and how they work the reader is referred to [10].

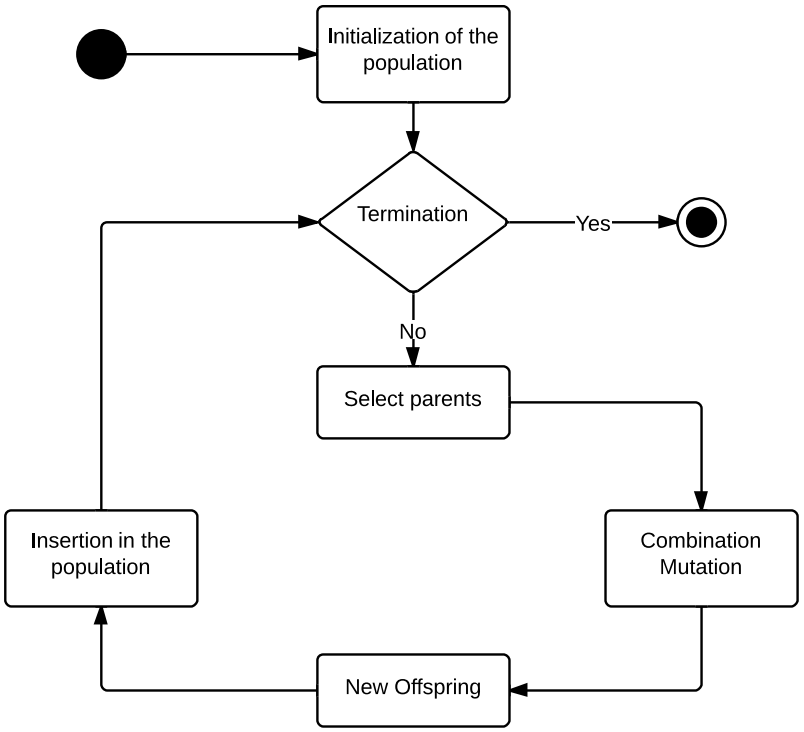


Figure 3.1: Overview over a simple evolutionary algorithm.

3.2 Iterated Local Search

One of the foundations of our evolutionary algorithm and our main competitor is the fast local search for maximum independent sets presented by Andrade et. al. [1]. This natural local search algorithm uses simple 2-improvements or $(1, 2)$-swaps to gradually improve a single current solution. The intention of a (j, k)-swap is to remove j nodes from the solution and then insert k new nodes into it. A $(1, 2)$-swap in particular tries to remove a single node from the solution in hopes of adding two other free nodes instead. A node is called *free*, if none of its neighbouring nodes can be found the current solution. We can also define freeness in terms of tightness. The tightness of a node $t(v)$ is the number of neighbouring solution nodes. Hence, free nodes can be described as nodes with a tightness of zero. Local search then basically iterates over all nodes $v \in V(G)$ and looks for such a swap. A simplified

pseudocode representation of the local search can be seen in Algorithm 1.

The data structure that is used to maintain the current solution consists of an array that is split into three different blocks. The array is a permutation of all nodes in our graph. The first block contains the vertices of the solution, whereas the second block stores any remaining free nodes. Finally, the third block contains all non-free nodes, that are not in the solution. We explicitly store the position and tightness of each node. In addition to that, we also store the sizes of the first two blocks. This data structure allows a node to be moved between blocks in constant time as well as the insertion and removal of new nodes in time proportional to their degree.

It is shown, that the local search procedure can find a valid $(1, 2)$-swap in linear time $\mathcal{O}(m)$, if it exists. We can explain this runtime by looking at the number of nodes that are processed by the algorithm. A solution node $x \in V$ is only examined if it is a candidate for a 2-improvement. This includes removing and possibly reinserting it into the solution, which can be done in $deg(x)$ time. A non-solution node on the other hand is only processed if it becomes free during the algorithm, which only happens when its unique solution neighbour is examined. This way, each node is looked at $\mathcal{O}(1)$ times. In turn, every edge is also visited $\mathcal{O}(1)$ times, which results in the aforementioned runtime. Different implementations of this procedure are presented by Andrade et. al. [1]. One of these is an incremental version, which maintains a list of candidates that is extended and reduced depending on the swaps made. This version of the local search is also used in this thesis and is known to run in sublinear time. Furthermore, other methods for improving the solution using $(2, 3)$-swaps are also proposed but we limit ourself to the simpler version with $(1, 2)$-swaps. This is because the use of 3-improvements is too slow, since there is no incremental algorithm for this version.

3.2.1 Metaheuristic

The fast local search is tested in a heuristic which is based on a metaheuristic called the *Iterated Local Search* (ILS) [8]. This heuristic starts from a random solution S and repeatedly applies a number of steps until a termination criterion is met. An overview over the single steps that are used can be seen in Algorithm 2.

The perturbation step is represented by the $force(k)$ routine, which inserts k nodes to the solution and removes any conflicting neighbouring nodes. Additional mechanisms are implemented to maintain a certain diversity. The first of these is *soft tabu*, which keeps track of the iteration in which each node was inserted into the solution. If there is more than one possible node to insert, the $force$ routine basically chooses the one that has not been in the solution for the longest time. The second mechanism is directly attributed with the local search algorithm. If the $force$ procedure selected a single node for insertion ($k = 1$), that node is not

allowed to be removed by the following local search iteration. The node can only be removed if all other possible nodes have already been checked without success. The ILS then proceeds by applying the local search algorithm. The final step consists of deciding whether or not to keep the new solution S'. To avoid straying from any optimal solutions found so far, the number of times the algorithm is allowed to go to a worse solution is limited by the current solution size. Interested readers are referred to the original paper [1] for more detailed explanations.

3.3 Graph Partitioning

The graph partitioning problem takes a graph G and a number $k > 1$ and tries to divide the nodes of the graph $V(G)$ into k subsets $V_1, V_2, \ldots, V_k$, called blocks. This is done in such a way, that an objective function is minimized. The most common form of graph partitioning is balanced partitioning, where the blocks should be of roughly equal size and the number of edges that runs between the individual blocks is the minimization criterion. A more thorough explanation on graph partitioning can be found in [27]. Graph partitioning proofed to have many applications ranging from route planning [6] to VLSI design [15]. In later sections, we will also see how it can help us find good cuts for the reproduction step of our evolutionary algorithm. Not only do graph partitioners benefit evolutionary algorithms, they are also subjects of evolutionary algorithms themselves [24]. Another important aspect for our work is that graph partitions allow the efficient generation of node separators. For more information on graph partitioning and its application, we refer the reader to [3].

3.3.1 KaHIP

The *Karlsruhe High Quality Partitioning Framework (KaHIP)* is a family of graph partitioning programs described in [27]. Its main components are:

- *KaFFPa*, a multilevel graph partitioning framework
- *KaFFPaE*, a distributed evolutionary algorithm for solving the graph partitioning problem
- *KaBaPE*, which provides balancing variants for the aforementioned techniques.

It also includes programs that output node separators for a given partition. For a more detailed explanation the reader is referred to [25]. An overview over the techniques of the framework can be found in Figure 3.2.

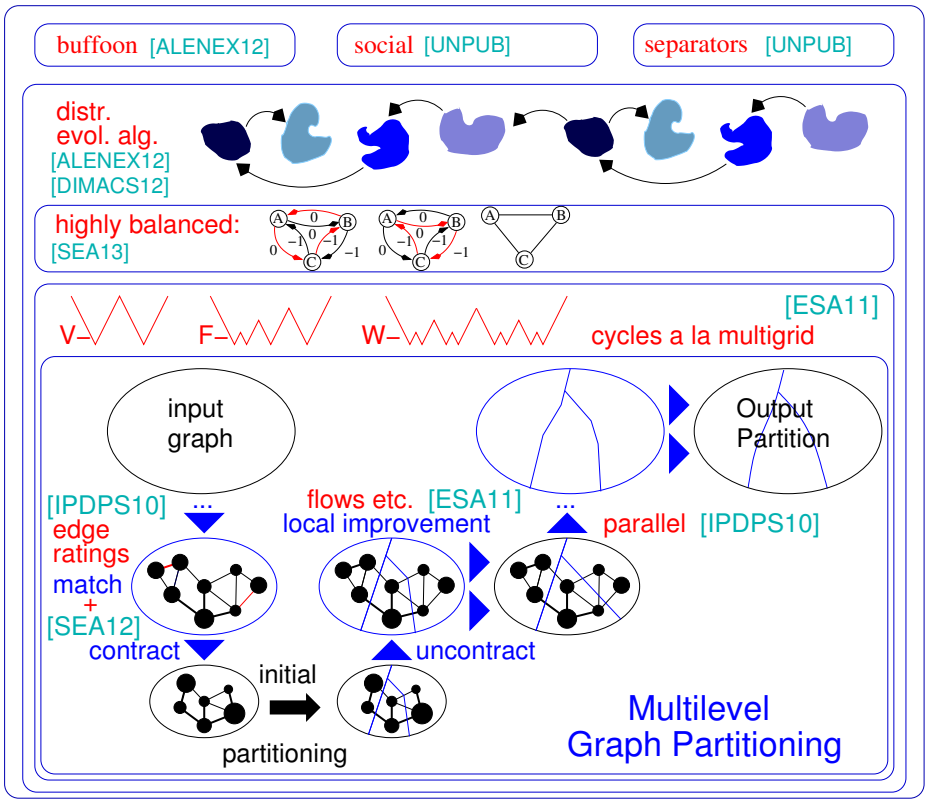


Figure 3.2: Overview over the techniques used in the KaHIP framework [26].

4. EvoMIS

This section will present our novel evolutionary algorithm (EA) for independent sets in all its details. We will see, how the previously discussed topics such as graph partitioning and the fast local search can help us build good combination operators for our particular problem domain. Following the general scheme depicted in Figure 3.1 step by step, we explain our choices of algorithms for each stage. Our population is denoted as P and consists of individuals $\mathcal{I}_1, \ldots, \mathcal{I}_p$, with $p = |P|$ being the total population size. The notation for an individual and its solution representation, which is presented in the next section, are used interchangeably throughout this thesis. When using partitions or node separators for our combination operators we define $B = \{V_1, \ldots, V_k\}$ as the set of blocks created by a k-way partitioning and S as the resulting separator of size $s = |S|$. As a stopping criterion for our algorithm we choose a simple timer, that can be set to an arbitrary time limit.

4.1 Solution Representation

Since we count nodes from 1 to n ($V(G) = \{1 \ldots n\}$), the most natural representation of our problem is a string $s = \{0, 1\}^n$. In this format a 1 at position j ($1 \leq j \leq n$) means that node j is part of the independent set and a 0 means it is not. An example for this representation is shown in Figure 4.1. Another possible way of storing a solution would be a (sorted) list of all the nodes that represent the independent set. We decided to go with the more natural representation because of its underlying simplicity, which really benefits us when building advanced combine operators. Additionally, when using actual bit-vectors to store an independent set, efficient bitwise operations can be performed on these solutions. An example of this would be applying the *AND* operator on two solution representations to get their intersection.

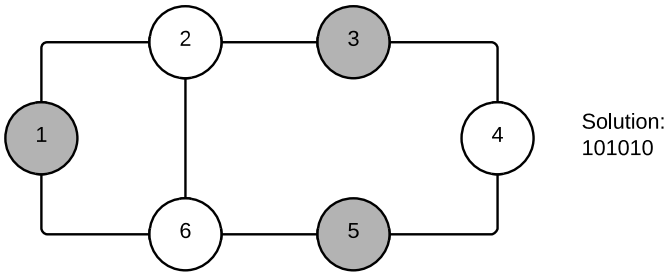


Figure 4.1: Example for the solution representation based on our simple graph from Section 2.2.1.

4.2 Initial Solutions

The process of creating initial solutions for the independent set problem can be done in a number of different ways. The three methods we used in our work are presented in the following.

The first and most simplistic way is to start from an empty (independent) set and add nodes at random until no further nodes can be added. Before a node can be added to the solution, we need to make sure it results in a valid independent set. We do this by simply checking if any of the surrounding nodes have already been added to the solution. This is essentially the same as checking if the node is not free. If this is not the case and the node is free, we are able to add the node to the solution. Despite the straightforwardness of this method, it adds a decent amount of diversity into the initial stages of the EA, which over an extended period of time can lead to good solutions. The pseudocode for this procedure can be seen in Algorithm 3.

The other two options we provide in our EA are taking a greedy approach to the independent set problem. The first of these two is principally the same as the one used by Andrade et al. [1]. Starting from an empty solution, we always add the node with the least residual degree. Given a subgraph $G' = (V', E') \subseteq G$ the residual degree of a node $v \in V'$ can be defined as $deg_{G'}(v) = |\{u \mid u \in V' \wedge \{u, v\} \in E'\}|$. The node that gets added to the independent set therefore is $x = \min_{v \in V'} deg_{G'}(v)$. We do this by using a bucket priority queue which groups the nodes into buckets based on their residual degree in the graph. The bucket queue allows us to pick a

random candidate each time multiple nodes share the same residual degree. After a node has been chosen and added to the solution, we remove all its neighbouring nodes from our graph G, since they can no longer be part of the independent set. As a final step we need to decrement the residual degrees of all next neighbours to leave us with a reduced graph. We then repeat the procedure using this graph, until our bucket queue is ultimately empty. The pseudocode for this method can be seen in Algorithm 4.

The second greedy algorithm tries to generate a maximum independent set by using the detour across vertex covers. We first create a solution C to the minimum vertex cover problem and then calculate the complement $V \setminus C$ to reach our goal of an independent set. By definition of the minimum vertex cover problem this complement is a valid maximum independent set. The rest of the procedure is very similar to our first greedy algorithm. The only real difference is that we now group the nodes in order of the maximum residual degree instead of the minimum residual degree. The pseudocode for this procedure can be seen in Algorithm 5.

Once we now know how to generate the initial solutions for our individuals, we can simply repeat the creation process a certain number of times until we have a population that fits our needs. Each of these methods can also be further extended by applying the iterated local search algorithm to the resulting solutions. A comparison between the individual methods can be found in Section 5.

4.3 Selection

After we have established the initial generation of individuals, we focus our attention on the selection schemes. This step is essential in any EA and can be tackled in many different ways. We now present the selection scheme we decided to use for our work.

Deterministic tournament selection selects parents for mating based on a number of so called tournaments, that are held between the individuals of the current generation. Each tournament selects a certain number k of random individuals, who then "fight" for a mating spot using their fitness function as a weapon. Even tough the selection of participants is strictly random, the tournament format itself allows for some variation. Either the amount of desired parents q determines the number of tournaments or we only schedule one tournament and return q individuals from it. In the first case, each winner from one of the tournaments is selected as a mating candidate. In the second case, the q best-placed individuals are selected. Even tough these two versions follow the same principle, the selection pressure introduced in the first one is greater than in the second one. *Selection pressure* describes any causes that might have an effect on the success of reproduction for a certain population or individual. An example of the second method is demonstrated

in Figure 4.2. In our work we decided to go for the first version. In general, the tournament selection scheme has several benefits for EAs. For example, the selection pressure can easily be adjusted by choosing different tournament sizes k. More precisely, bigger tournaments tend to increase the selection pressure, since unfit individuals are almost certain to lose in the early rounds of the tournament. On the other hand, smaller tournament sizes reduce the selection pressure. It is notable that a tournament size of 1 results in a simple random selection. Tournament selection can also be easily adjusted for the use on parallel architectures and ensures a certain degree of convergence when used in conjunction with diversification methods. Additional information on this selection scheme can be found in [21]. Other selection schemes, like linebreeding and random selection, were also tested during our work, but did not generate a distinct advantage over tournament selection. For more information about selection schemes in general the reader is referred to [10].

4.4 Combination

Combination tries to implement the notion of natural reproduction in EAs and takes the selected parents from the previous section to generate one or more new offsprings. These offsprings in turn try to spread the genetic material of their parents throughout the next generations. The operators we use in this stage are the following:

- **Intersection combine**, based on the overlap of two parents
- **Node separator combine**, which uses 2-way node separators
- **Vertex cover combine**, which uses 2-way partitions and vertex covers
- **Multiway combine**, which uses k-way partitions or k-way node separators.

We define the set of q parents, that are returned from our selection scheme, as a subset $\{\mathcal{P}_1, \ldots, \mathcal{P}_q\} \subseteq P$. Besides the multiway combine operator, which can take an arbitrary number of parents, all other operators require exactly two parents for combination. The resulting offsprings from this step are a set $\{\mathcal{O}_1, \ldots, \mathcal{O}_r\}$ based on the number of created offsprings r.

Most traditional EAs use a simple crossover operator to combine two parents. Simple crossover takes the solution representations of the parents, which are in most cases bit-strings, and selects an arbitrary cutting point. At the selected point the bit-strings of the parents are swapped and the resulting strings are the encoded representations of the offsprings [10]. An example of this process can be seen in Figure 4.3. Since our representation of independent sets underlies certain

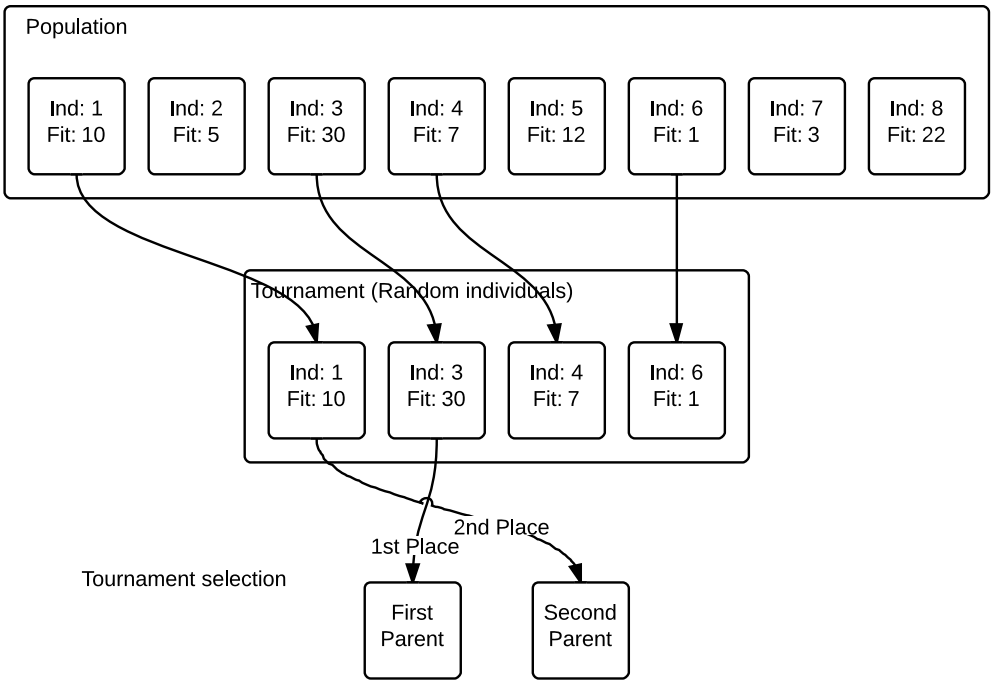


Figure 4.2: Example for tournament selection with one tournament and the selection of the top candidates.

restrictions, we are not allowed to use such an arbitrary operator on our individuals. In more detail, neighbouring nodes are not allowed to be in a valid solution and therefore these nodes cannot be set in the solution representation at the same time. This could possibly occur when using simple crossover. We therefore need to make sure that such representations cannot be generated. This can be done in one of two ways: either through arbitrary crossover and the addition of penalties, which are placed on invalid individuals, or through more sensitive combination operators. We decided to go with the second way, because we want to ensure that only valid solutions are present in our population at any given time.

The general scheme, which all of our combination operators follow, tries to generate new, not particularly maximal, independent sets, which then undergo a number of post-processing steps. The first of these is a maximization step, that tries to add any free nodes that are left in the graph. Afterwards, we apply a single

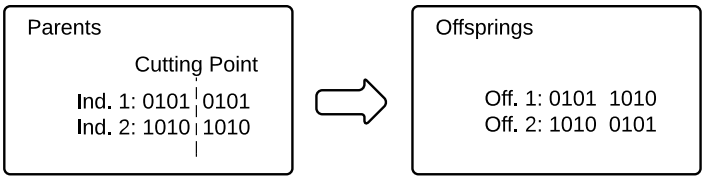


Figure 4.3: Example of the simple crossover.

iteration of the fast local search to our solutions to ensure that they reach local optima [1]. As a final step we encode this newly created solution in an offspring, which can subsequently be inserted in the population. An overview of this scheme can be seen in Figure 4.4. The next sections will be concerned with describing all of the combination operators in detail. Each of these operators is equiprobable to be chosen for combination.

4.4.1 Intersection

Our first combination operator takes the solution nodes of the two parents $\mathcal{P}_1, \mathcal{P}_2$ and builds their intersection $\mathcal{P}_1 \cap \mathcal{P}_2$. When using actual bit-vectors for the solution representation, this operation can be performed in linear time. Building the intersection of the nodes from two independent sets is guaranteed to return another valid independent set by definition of the independent set problem. The major drawback of this method is that the resulting independent set is in most cases smaller than the ones of the parents, except when $\mathcal{P}_1 = \mathcal{P}_2$. To compensate for this fact we use our aforementioned post-processing steps. We first maximize our solution and then apply the fast local search algorithm to make sure we reach a local optimum. Finally, a new offspring carrying this solution is generated, as depicted in our general scheme. The pseudocode for this algorithm can be seen in Algorithm 6. Additionally, an example for the procedure is shown in Figure 4.5.

4.4.2 Node Separator

The next combine operator will show us why graph partitioning is especially useful when working with context-sensitive (restricted) combine operators on graphs. Like its name implies, this operator uses node separators, that are obtained from a graph partitioner or via a simpler BFS, to generate new offsprings. In Section 5 we

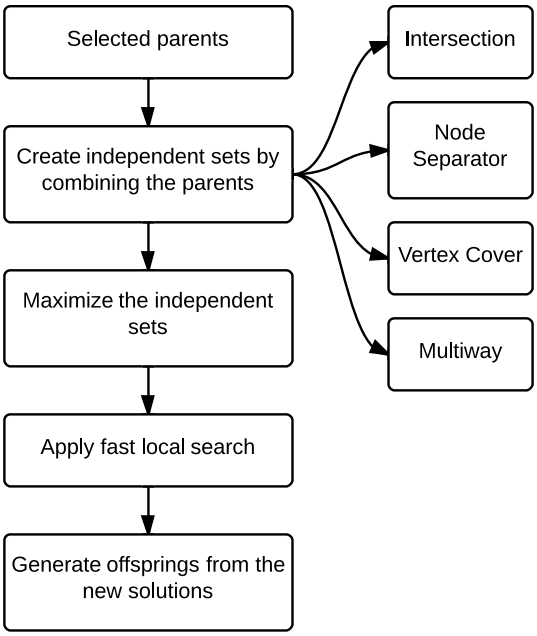


Figure 4.4: Overview of the general combination scheme.

will study what differences result from using each of these methods. For now, we are primarily concerned on how to generate new and possibly improved solutions using node separators.

We do this by first creating a node separator S that divides our graph G into two subsets A and B ($V = A \cup B \cup S$). This is done in such a way, that there are no edges running from A to B or vice versa. S can therefore serve as a crossover point for our operation. An example of a node separator can be seen in Figure 4.6. We then generate several new independent sets using S and our two parents $\mathcal{P}_1, \mathcal{P}_2$. The first of these sets results from using the independent set nodes of our first parent that are also present in A. We call this set S_1. Analogues, we get the second set S_2 by using the independent set nodes of $\mathcal{P}_2$ that are available in B. We then combine S_1 and S_2 by building their union $S_1 \cup S_2$ to get a combined independent set, which forms the basis of our first offspring $\mathcal{O}_1$. Again, this unification can be done in linear time when using bit-vectors for the solution representation. The validity of this independent set is directly implied by the definition of a node separator. Since S_1 and S_2 are separated by S, there is no possibility of solution nodes interfering

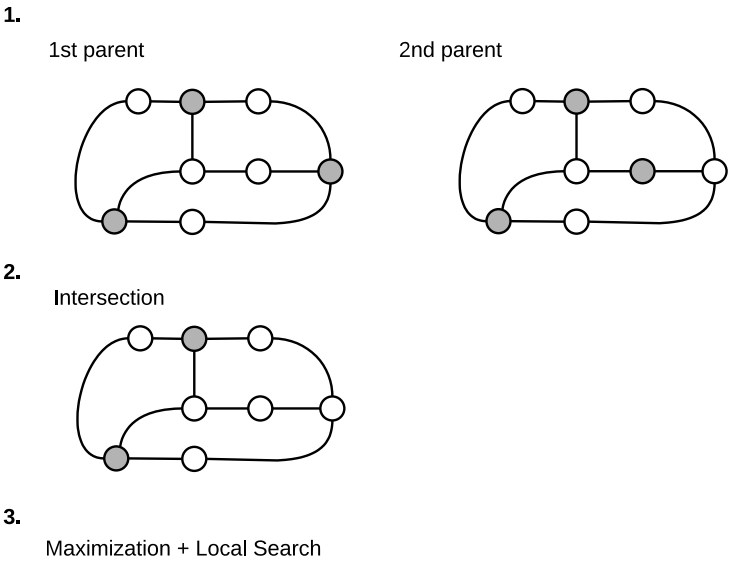


Figure 4.5: Combination of parents using the intersection combine operator.

with each other. The implied validity is one of the major advantages when working with this operator to create constrained solutions. The only nodes which are now left to be added are the nodes of S, because we did not include them in our creation process. As a result, our offspring is definitely an independent set, but might not be maximal. To make it so, we use a greedy algorithm similar to one of those discussed in Section 4.2. We simply put the nodes of S in a priority bucket queue, that is sorted in order of decreasing residual degree. By always adding the node with the least residual degree within S, we are able to create our new solution. Following our general scheme, we then add any free nodes that are left and afterwards perform a single iteration of the fast local search algorithm. In tandem to the first offsprings, we create a second offspring $\mathcal{O}_2$. This one spawns from the overlap of the first parents independent set nodes and B as well as the overlap of the second parents independent set nodes and A. This leaves us with two new offsprings $\mathcal{O}_1$ and $\mathcal{O}_2$ that can be added to the population. The pseudocode for the whole procedure can be seen in Algorithm 7. Figure 4.7 shows an example of this combine operator.

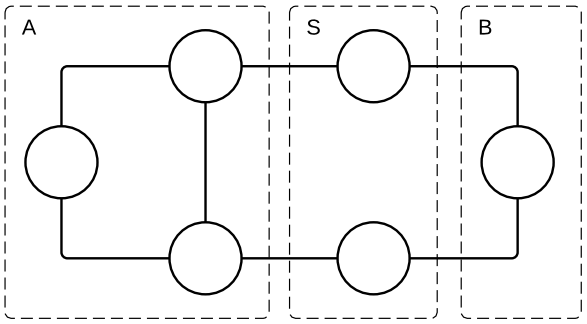


Figure 4.6: Example of a simple node separator.

4.4.3 Vertex Cover

The third combine operators tries to tackle the MIS problem using an indirect approach. It does so by taking advantage of the fact that the complement of a maximal independent set is a minimal vertex cover and vice versa [17]. Thus, by complementing our solutions, we are able to obtain minimal vertex covers, which in turn can be combined in different ways to produce new vertex covers. One of these ways will be discussed in the following. The newly created vertex covers can then be transformed into new independent sets by building their complements once again. The combination operator also uses partitions ($V = A \cup B$) to acquire its crossover points. The necessary partitions can once more be generated either by a graph partitioner or via a simple BFS.

Starting from the usual pair of parents $\mathcal{P}_1, \mathcal{P}_2$ and a graph partitioning $V = A \cup B$, we first generate the parents complements $\overline{\mathcal{P}}_1, \overline{\mathcal{P}}_2$. The complement of an individual is a new individual that contains the complemented solution representation of the original one. The bitwise complement of the solution representation allows us to easily build vertex covers from existing independent sets. We then use these vertex covers $\overline{\mathcal{P}}_1$ and $\overline{\mathcal{P}}_2$ in a similar fashion as presented in the previous section to obtain new solutions. These can then be extended to result in valid minimal vertex covers.

The first offspring is generated by taking the vertex cover nodes of $\overline{\mathcal{P}}_1$, that are present in A, as well as the vertex cover nodes of $\overline{\mathcal{P}}_2$, that are present in B. The second offspring is created the other way round using the overlap of $\overline{\mathcal{P}}_1$ and B in combination with the overlap of $\overline{\mathcal{P}}_2$ and A. Unlike the previous section

where we used a node separator, that separates the blocks A and B, we now use a partitioning, meaning A and B are connected. This fact can lead to issues at the partitioning cut. More precisely, the resulting set might consist of too little nodes to be a valid vertex cover, because some edges between the partitions are not covered after crossover. To fix this problem, we want to add as little as possible nodes from the cuts boundary to our newly created sets. We do this by extracting the bipartite graph that represents the boundary of the cut and then using the Hopcroft-Karp algorithm to get a minimum vertex cover of it. After fixing the boundary of the partition cut, we can once again complement our now valid vertex covers to get a fresh set of independent sets. To continue as before, we then follow our general scheme by going through the maximization phase and applying the fast local search to reach a local optimum. Finally, we put our maximum independent sets in two new offsprings $\mathcal{O}_1$ and $\mathcal{O}_2$. The pseudocode for this combine operator can be seen in Algorithm 8. Additionally, an example of this method can be found in Figure 4.8.

4.4.4 Multiway

Our last and probably most sophisticated operator extends the repertoire of available combination mechanism by k-way partitions and k-way node separators. In contrast to the operators discussed so far, which used 2-way partitions and node separators, it divides the given graph in an arbitrary number of blocks k. This kind of operator closely resembles the multiple-point crossover or generalized crossover model [10, 5]. More precisely, instead of selecting only a single crossing point, like in simple crossover, it can select a number of different crossing points at once. In distinction from to the traditional multiple-point crossover, which often only combines two parents, we are also able to select an arbitrary number of parents $2 \leq p \leq |P|$. We will see in Section 4.8, how this positively affects the diversity of our population. The k-way partitions and separators used for this procedure are obtained from a graph partitioner as usual. Generating k-way partitions and separators ($k > 2$) via BFS is possible, but requires a more sophisticated procedure than the simple BFS used so far.

Before the actual combination step, we first have to select a certain number of parents q. We then take a (random) k-way partition or separator from our pool of partitions and calculate the score for every possible pair $(\mathcal{P}_i, V_j)$ of a parent $\mathcal{P}_i$ and a block V_j. The score of a pair $(\mathcal{P}_i, V_j)$ is calculated by counting the number of the parents solution nodes inside the given block. We then select the parent with the highest score for each of the blocks. An example of this can be seen in Figure 4.9.

The subsequent procedure is dependent on whether a partition or separator was selected. Each of these variants is very similar to their respective 2-way combine operators. If a k-way node separator was chosen, we start by simply taking the

highest scoring parents for each block, like mentioned before, but exclude the separator itself. We then take their respective solutions nodes and create a new solution by combining the nodes for each block. Since we left out the separators nodes, this solution will not be maximal in most cases and we are able to improve it even further. This can be done by using a bucket queue and the same greedy algorithm we already used in the node separator combine operator.

Secondly, if a partition was chosen, we start by repeating the same selection process for the parents that we used for the previous variant. Afterwards, instead of fixing our solution right away, we complement our newly created solution to get a set of nodes, that can be transformed into a valid vertex cover. Once again, we utilize the more or less same greedy algorithm, that helped us to create the initial solutions based on vertex covers, for fixing the solution. We cannot use the Hopcroft-Karp algorithm that we mentioned in the vertex cover combine operator. This is due to the fact, that we cannot be sure that the boundary of the partitioning cut is a bipartite graph. This reduces the quality of the vertex cover we obtain by a meager amount, which is comparably small to the overall solution size. We then once again complement our vertex cover to get a new independent set.

After either of the two methods has finished, we reiterate the usual post-processing steps. This includes the maximization of our solution and the application of the local search algorithm. This way we receive our final solution, which can be forged into an offspring. The pseudocode for these procedures can be found in Algorithm 9 and Algorithm 10. The differences of the two methods in terms of performance and solution quality, as well as good choices for the number of parents and blocks, will be discussed in Section 5.

4.5 Partition Pool

It can be quite costly to generate a new partition/node separator with the graph partitioner every single time a combination operation is performed. To alleviate this effect, we can build a number of partitions/node separators in advance and maintain or refresh them while the evolutionary algorithm takes place. The data structures that realizes this idea is called the *separator pool*.

The separator pool generates a number of arrays, that contain all the necessary partitions and separators for use by the combination operators, including:

- (2-way) partitions for the vertex cover combine,
- (2-way) separators for the node separator combine and
- k-way partitions and separators for the multiway combine.

Each of these arrays can store an individual amount of partitions/separators. A single partition is represented by an id, the number of blocks k and the actual partition map of size $n = |G|$. Analogues, a single separator consists of an id, the number of blocks k (excluding the separator block itself) and a separator map of size n. In addition to that, the size of the separator block is also stored explicitly. The partitions/separators are generated using the KaHIP framework [26] and can be obtained from the separator pool at random. The pool also offers the possibility to apply any given partition or separator to the underlying graph data structure. These methods can be used to quickly prepare the graph for the individual needs of each combination operator.

A feature that is especially useful for the multiway combine operator is that the pool is also able to keep track of the scores for every possible combination of an individual and a block $(\mathcal{I}_i, V_j)$ $(1 \leq i \leq |P|, 1 \leq j \leq |B|)$. This allows us to quickly retrieve the best candidates within a given set of parents. The scores are calculated as soon as the pool is initialized in the beginning of the EA and recalculated every time the pool gets renewed. Additionally, if a new individual is inserted in the population after combination, it suffices to only calculate the respective scores. This way we do not have to recompute all scores for every chosen parent if the multiway combine operator was selected during the evolutionary algorithm, which benefits its runtime.

The renewal of the pool is dependent on two different aspects. The first is a threshold, which triggers the replacement of the current (k-way) partitions and separators. This threshold limits the number of iterations that can pass without resulting in a new best solution. The second one is a timer, that measures the time taken for building the pool time_{build} as well as the time taken for combination operations after the pools creation $\text{time}_{combination}$. Based on a arbitrary factor r, we refresh the pool if the combination timer exceeds the building timer ($\text{time}_{combination} > r \cdot \text{time}_{build}$).

4.6 Mutation

The next stage we have to examine is the mutation operator. As described in Section 3.1, this operator is able to make changes to an offspring after the combine operators. It therefore is able to add additional perturbation and diversity to the population. Since often times bit-vectors are used for the solution representation in EAs, the usual mutation operator works as follows: The solution is processed from front to back and every bit has a certain probability of being flipped. The probability for flipping a bit is usually very small to limit the effects of randomness introduced by mutation. For more information on mutation, we refer the reader to [10]. Since our representation underlies certain restrictions and we want to

have a valid maximum independent set at all stages during our algorithm, we cannot perform mutation using this simple procedure. Instead we have to design an operator, that allows the mutation of offsprings and still generates valid maximum independent sets. The mutation operator we used in our algorithm to facilitate these characteristics is pretty straightforward. We take our unmutated offspring and apply another round (or several rounds) of the iterated local search algorithm to its solution representation. Since a big part of the ILS is the forceful insertion of new nodes into the solution, this operator generates enough diversion from the original solution to serve as a good mutation operator. We will later see how different mutation probabilities effect our algorithm in Section 5.

4.7 Replacement

We now explain the criteria under which an offspring is allowed to contribute its genetic material to the population. In addition to that, we will be discussing our strategy for removing individuals from the population, in case an offspring is successfully selected for insertion.

Traditional EAs often use two overlapping populations as a strategy for establishing new generations [10]. Overlapping populations work in the following way: all new offsprings are inserted in one of the two populations. If this first population is full, we use its individuals to generate new offsprings, which are then put in the second one. If the second population is full, the first one is emptied, the populations are swapped and the process repeats. The main advantage of this strategy is that it removes the necessity for complex replacement methods which can be seen in single population variants. On the downside, maintaining diversity using overlapping populations usually needs complex selection schemes. We implemented an option that allows us to use overlapping populations, but finally decided to go with a single population and variable replacement schemes. Before we discuss these in detail, we will explain when an offspring is added to the population anyway.

We start by looking at the currently worst individual, in terms of solution size, within our population. If our new offsprings solution is greater than this minimum it is allowed to enter the population for the time being. Depending on the results of the replacement scheme, the offspring might still be excluded from sharing its genetic material. The one exception to this, is a threshold on the number of insertions, that do not result in any new offspring entering the population. If this threshold is exceeded, the individual is inserted under all circumstances, even if it is smaller than the current minimum. In this case the minimum will be replaced by it.

After an offspring is allowed to enter the population, we apply another round of the ILS to its solution to enforce a stronger convergence of our algorithm. We then

concern ourselves with finding an appropriate replacement for our new offspring. Our first and generally used replacement scheme tries to find the most similar individual to our offspring in terms of overlapping solution nodes. We achieve this, by calculating the number of differences between our offspring $\mathcal{O}$ and each individual in our population. This can be done using the Hamming distance. We then select the individual with the least differences for removal, which is also the most similar one. The one exception to this procedure is the currently best solution. The best solution is not allowed to be selected for replacement, because we want to ensure that our overall solution quality does not degrade over time. After we are done selecting an individual, we remove it from the population and simply put the offspring in its place. An pseudocode algorithm for this process is presented in Algorithm 11. The other two schemes try to maintain a certain level of diversity in addition to selecting a suitable replacement and will be discussed in the next section.

4.8 Diversification

Like mentioned before, evolutionary algorithms work by enforcing strong individuals (in terms of fitness) to share their genetic material and weak individuals to vanish over time. Strictly following this approach can lead our population to strife in a certain direction that might lacks distinct traits of weak individuals. However, these traits could come in handy in a later stage of the algorithm. To attribute this aspect, we should try to keep a certain portion of weak individuals, if their solution representation differs distinctly from the stronger ones. The size of this subpopulation is crucial, since it should be big enough to carry enough diverse traits, but not to big to disrupt the convergence of the EA. In literature this process is often denoted as niching and speciation [10]. There are a wide number of methods for incorporating this concept in practical algorithms, some of which can also be found in our work.

Crowding is a replacement scheme proposed by De Jong [5] and a generalization of the similarity based replacement method we described in Section 4.7. Using this scheme, instead of outright taking the most similar individual, we first selected a random subpopulation consisting of a certain of number of individuals, called the *crowding factor*. We then choose the most similar individual from that subpopulation. Our strict version resembles crowding with a crowding factor of the population size p. Smaller crowding factors might help a certain species of weak individuals to occupy a larger portion of the population than they would have otherwise.

Another well established method of maintaining diversity is to use *sharing*. Sharing allows us to form and stabilize different subpopulations over time. To

implement this approach in our algorithm, we introduce a sharing function that calculates the degree of similarity for each individual in our population. The similarity is measured by calculating the Hamming distance of an individual with all the others and then summing up these values. Individuals with a high degree of similarity should then be given a higher chance of being replaced than individuals with high variousness and lower similarity. To attribute this fact, we simply divide the fitness value of each individual by its normalized degree of sharing $f_{sharing}(\mathcal{I}_i) = \frac{f(\mathcal{I}_i)}{\frac{1}{p}\sum_{j=1}^{p} d(\mathcal{I}_i,\mathcal{I}_j)}$. Our modified fitness function is then used in a new replacement scheme that always chooses the individual with the smallest shared fitness value for removal. The exception to this is again the currently best individual, which should not be removed in any case. Besides the increased diversity and speciation, the major drawback of using sharing mechanisms is that the calculation of the shared fitness function is rather costly and has to be performed whenever a new individual is inserted in the population

In addition to the aforementioned two approaches, we added a third replacement scheme for maintaining diversity. Our scheme creates a subpopulation by selecting individuals that are in certain range of similarity to our offspring based on an arbitrary percentage. We again measure similarity using the Hamming distance of the individuals. From this subpopulation, we then choose an individual for replacement at random. One issue using this method is that in the beginning the similarity of individuals might still be too small for a percentage based subpopulation to form, in which case we simply select the most similar individual.

A more implicit way of generating variousness is the multiway combine operator. Since the operator can take an arbitrary number of parents and picks the best parent for each block, overall weak individuals, that still have distinct strong traits in some regions, are able to be selected to contribute their genetic material for certain blocks. Instead of having to maintain a niche for weak individuals, which could be helpful in a later phase of the algorithm, this allows us to select important traits early on. This way, these individuals can still contribute their genetic material and be removed as usual, without focusing on keeping them a longer time. This especially holds true, when selecting a great number of parents for combination. For further reading on diversification techniques the reader is referred to [10]. The effects of the different diversification techniques will be discussed in the following section.

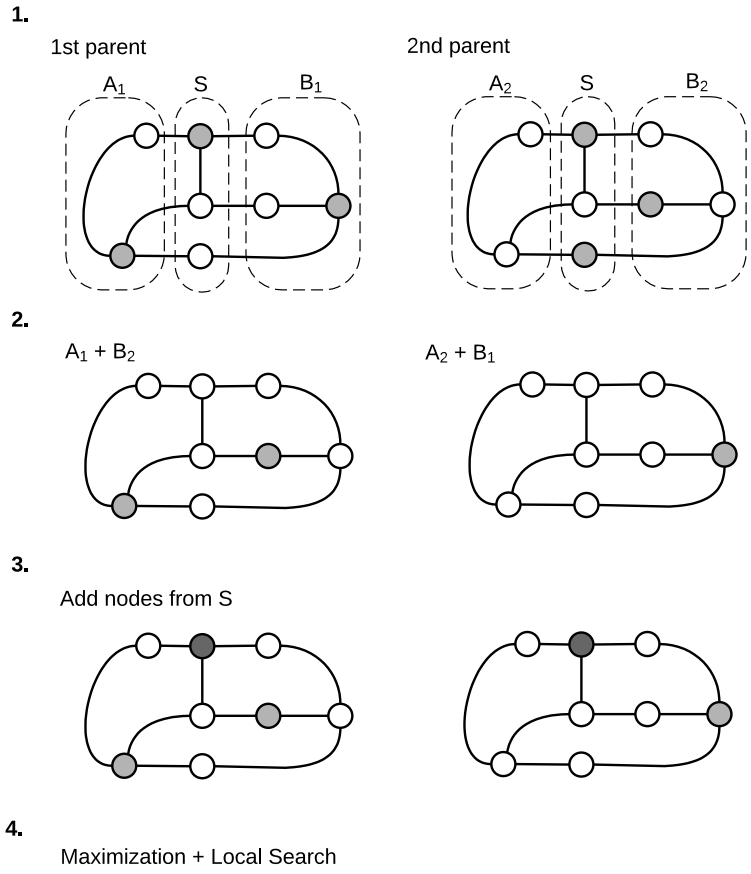


Figure 4.7: Combination of parents using the node separator combine operator.

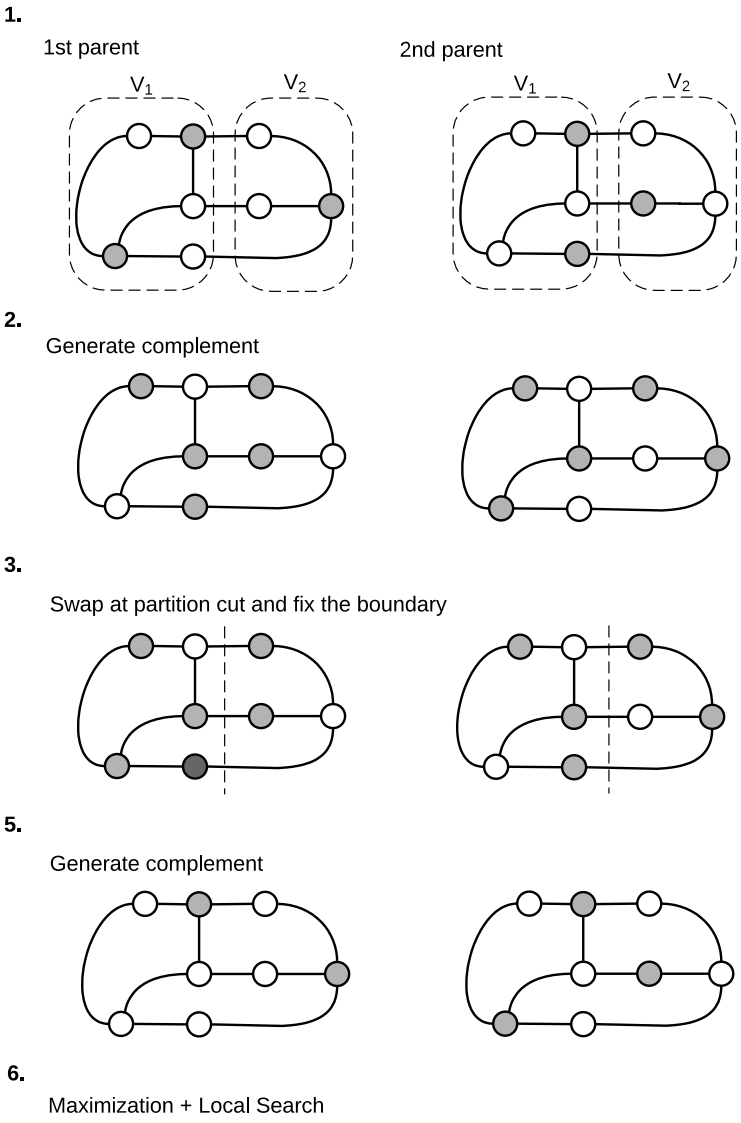
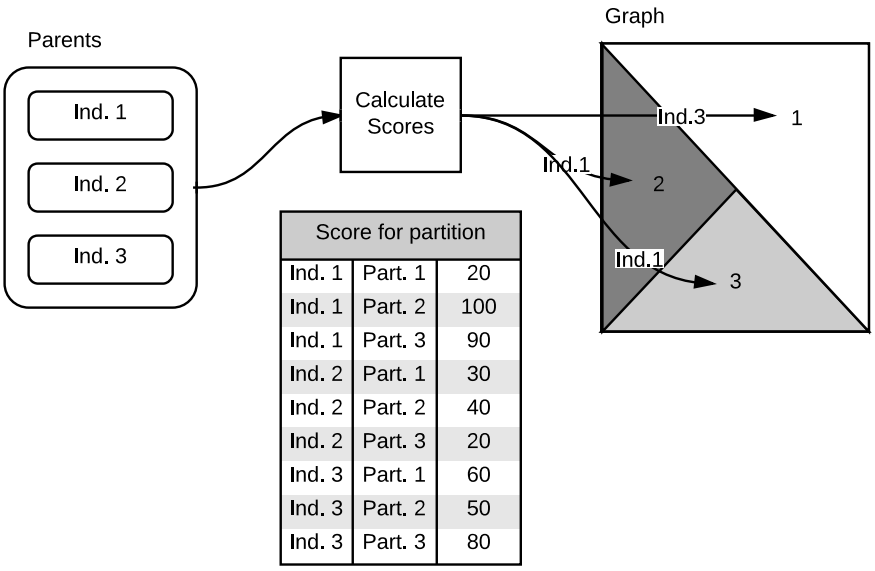


Figure 4.8: Combination of parents using the vertex cover combine operator.



Score for partition		
Ind. 1	Part. 1	20
Ind. 1	Part. 2	100
Ind. 1	Part. 3	90
Ind. 2	Part. 1	30
Ind. 2	Part. 2	40
Ind. 2	Part. 3	20
Ind. 3	Part. 1	60
Ind. 3	Part. 2	50
Ind. 3	Part. 3	80

Figure 4.9: Example of a parent distribution for the multiway combine operator.

5. Experimental Results

We now examine the performance of our algorithm on various instances for the maximum independent set problem. We start by presenting the instances we chose for our experiments and then describe our efforts to further improve our algorithm by fine tuning several of its parameters. In the end we will compare the final results of the tuned evolutionary algorithm with the ones obtained from the ILS algorithm of Andrade et. al. [1]. The measurements for both algorithm, the ILS and our EA, do not include reading the graph and building the necessary data structure associated with it.

5.1 Test Graphs

The families of instances, we chose for our experiments, consist of graphs from the work of Andrade et. al. [1] (MESH) as well as new instances that we added (STREET, SOCIAL, WALSHAW, DELAUNAY and GEOMETRIC). The first family, MESH, was kindly provided by Renato Werneck and consists of the dual graphs of triangular meshes. The dual graph of a graph G contains nodes for each face of G and edges for each pair of nodes that corresponds to neighbouring faces. Their importance for the maximum independent set problem comes from an application in Computer Graphics described by Sander et. al in [23].

A close relative to the ROAD family, that is used by Andrade et. al. [1], is the STREET family. It contains road networks of various countries including Germany and Great Britain. They are unweighted and undirected versions of the largest strongly connected component of the corresponding Open Street Map road networks and have been used in the 10th DIMACS Implementation Challenge, on graph partitioning and graph clustering. Both the ROAD and STREET family gain their importance for the maximum independent set problem from the application in map labeling algorithms.

As mentioned in our introduction, the big influence of social networks on our everyday life cannot be denied. Since they are also an interesting subject for various problems, including graph partitioning, we added the SOCIAL family to our experiments. It consists of various social network graphs, including citation

networks from the 10th DIMACS Implementation Challenge, autonomous system graphs [20] and network graphs from Wikipedia or Google [19, 18].

An often used benchmark for graph partition algorithms is Chris Walshaw's graph partitioning archive. Our WALSHAW family consists of instances that are found in this archive. Many of the graphs in this family come from typical partitioning applications and because graph partitioners play a big role in our algorithm we wanted to incorporate it into our work. The graphs were also used in the 10th DIMACS Implementation Challenge. For more information about the archive the reader is referred to [31, 29].

The DELAUNAY family features a number of graphs that have been generated using the Delaunay triangulation of random points in the unit square. A Delaunay triangulation of points P is a triangulation $DT(P)$, in such a way, that no point of P is within the circumcircle of any triangle in $DT(P)$. This family has been introduced by Holtgrewe et. al. [12] and is also part of the 10th DIMACS Implementation Challenge.

Our last family, GEOMETRIC, consists of random geometric graphs and has also been introduced by Holtgrewe et. al. [12] and used in the 10th DIMACS Implementation Challenge. Each graph of this family is made up of $n = 2^X$ (in our case $15 \leq X \leq 20$) nodes, which are random points in the unit square. In this type of graph two nodes are connected, if their Euclidean distance is below $0.55\frac{ln(n)}{n}$.

5.2 Methodology

Our algorithm was implemented using C++ and was compiled using gcc v.4.6.3 with full optimizations turned on (-O3 flag). Each run was made on one core of a 2.4 GHz Intel Xeon E5-4640 CPU with 528 GB of RAM running Ubuntu 12.04.5 LTS 64-bit Edition. The test results for the ILS were obtained by using the original algorithm from Andrade et. al. [1], which was kindly provided by Renato Werneck. For our evaluation we present average and maximal values as well as convergence plots. Convergence plots depict the evolution of the solution quality over time.

To gather our data we used three repetitions (seeds) per instance. Each repetitions was set to a time limit of ten hours. Average values were obtained by computing the geometric mean over all repetitions for each instance. For the generation of the convergence plots we follow the methodology of Peter Sanders and Christian Schulz [24]: whenever a new best solution was found by our algorithm a triple $(t, \text{solution}, \text{seed})$ is generated. The tuple consists of the current timestamp t, the newly found optimum solution and the seed that was used. Once the algorithm is done, we collect all of these triples and put them in a sequence T_{seed} that is sorted by timestamp in ascending order. If more than one repetition was used, we merge the individual sequences together in a new sequence S, which is once again sorted

by timestamps. Afterwards, we create a final sequence S_g, that consists of event based geometric means. This can be done by iterating through S and updating the geometric mean G alongside. G is initialized by using the first solution value for each repetitions. When processing a tuple $(t, \text{solution}, \text{seed})$ we update G by replacing the value of T_{seed} that took place in its initial computation. We then add the pair (t, G) to S_g.

To generate convergence plots for a whole family we repeat this procedure by using the sequence S_g of each instance. We start by adding an instance label to each pair, that indicates which instance it belongs to. After merging these, we sort the resulting sequence by timestamp in ascending order. Finally, we can calculate a sequence S_{family} by computing the geometric means of this sequence using the same approach as presented above.

5.3 Parameter Tuning

Before we can start the experimental evaluation of our algorithm, we have to make sure the parameters are chosen in a reasonable way. For this purpose, we performed a number of tuning experiments. The parameters we decided to tune were the following:

- Population size
- Separator pool sizes
- KaHIP mode
- Initial solutions
- ILS Iterations
- Mutation rate

In addition to that, we also present an evaluation of the performance of each combine operator on its own and examine the results of our diversification methods. As a final step, we compare the results of our algorithm to a version that uses a simple BFS instead of a graph partitioner. Each convergence plot used in this section also includes a zoomed subplot that shows the last 30% (time-wise) of the plot in greater detail.

As a starting point for our tests, we used the configuration seen in Table 5.1. The values for this configuration were chosen based on the observations we made during the implementation of our algorithm. Finally, the parameter tuning was conducted on the graphs shown in Table 5.2.

Parameter	Value
Population size	30
Pool size	15
KaHIP mode	fastsocial
Initial solutions	Greedy (minimal residual degree)
Mutation rate	10%
ILS iterations	10000
Tournament size	2
Multiway blocks	64
Multiway parents	$\|P\|$
Insertion threshold	150
Pool threshold	180
Pool renewal factor	10.0

Table 5.1: Starting configuration.

Family	Graph
DELAUNAY	`del_n15`
GEOMETRIC	`rgg_n15`
MESH	`bunny`
SOCIAL	`skitter`
SOCIAL	`cnr-2000`
SOCIAL	`in-2004`

Table 5.2: Graphs used for the parameter tuning.

The first parameter we chose for our tuning experiments was the population size $|P|$. Starting from 30 individuals, we tried smaller and bigger populations. Smaller populations often result in a faster convergence, but most of the time they lack variety in the later stages of the algorithm. Bigger populations on the other hand, start off slow, but might have a distinct advantage after a longer period of time, due to the higher amount of different subpopulations and individuals. Since we test instances over a long period of time, this effect can also be observed in our tuning results shown in Figure 5.1. The same principally goes for the separator pool size. A bigger number of partitions and separators increases the amount of possible cuts, that each combine operator can choose from. This in term leads to a bigger variety of resulting individuals and over a longer time to a better solution quality. On the downside, building a larger number of partitions and separators also reduces the amount of time spent for the actual combination of individuals. The effects of different pool sizes can be seen in Figure 5.2.

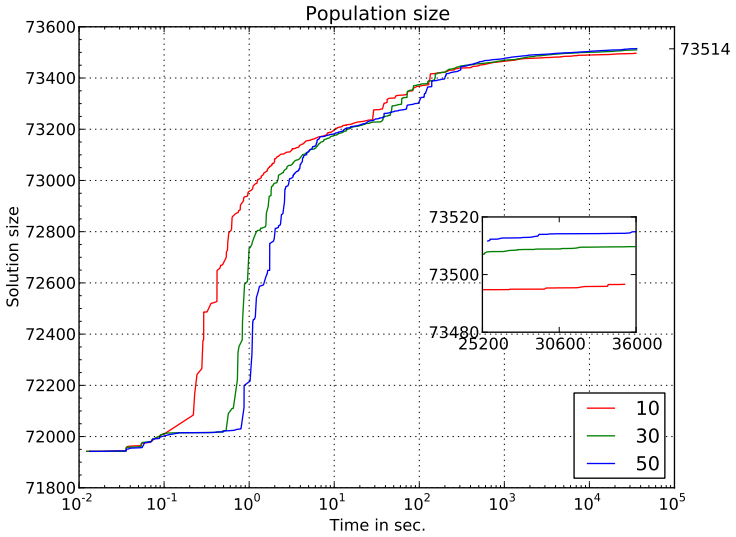


Figure 5.1: Effects of different population sizes. The global maximum is shown at the right.

Another important aspect for the quality and variety of the partitions/separators we use comes from the configuration of the graph partitioner itself. This also influences the time that is used for the separator pool generation. As described in the KaHIP user guide [25], there are 6 different modes for the KaHIP framework to choose from. The effects of each of these methods can be seen in Figure 5.3. We excluded the social network graphs from this experiment, since the KaHIP framework has problems processing them with the non-social configurations. Our evaluation shows, that for our purposes the *fast* and *fastsocial* configurations are the best. In addition to testing different modes, we also used a randomized imbalance (between 3% and 85%) to add further variety. The iterated local search by Andrade et. al [1] can also be parameterized in a number of ways. We decided to examine the effects of different numbers of iterations the ILS is able to perform each time it is invoked. The results of which are shown in Figure 5.4. A higher number of iterations enables offsprings to climb to better solutions before they enter the population. On the downside, this again reduces the amount of time we spent combining individuals. For our initial solutions, we decided to stay with the greedy algorithm that uses the least residual degree of nodes. The start of the different methods might differ greatly, but the convergence is fairly similar. This even holds true when starting from solutions that have an iteration of the ILS applied to them. The results of using different initial solutions can be seen in Figure 5.5. The last tuning parameter we mentioned before is the mutation rate. Since we use the ILS

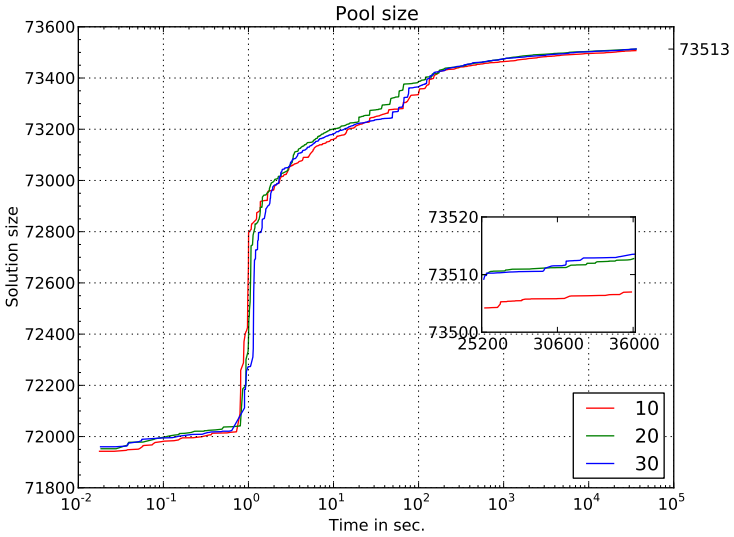


Figure 5.2: Effect of different pool sizes. The global maximum is shown at the right.

as a mutation operator, we wanted to make sure that the mutation rate is relatively low. Otherwise, our algorithm spends too much time performing the ILS and too little time on its core mechanics, like the combination of individuals. As mentioned in Section 4.6, the mutation rate of evolutionary algorithms should nonetheless be kept at a low level. We therefore decided to stay with our initial mutation rate. The plot for this parameter can be found in Figure 5.6. After all necessary tuning parameters were tested, we settled on the configuration shown in Table 5.3.

We proceed with evaluating the impact each of our diversification methods has on our algorithm. The results of these experiments can be seen in Figure 5.7. First of all, it is notable that the version of our algorithm that uses sharing seems to be inferior to the standard version. This probably results from the fact, that calculating the shares for each individuals after every insertion takes too much time. Other than that, crowding and our own replacement procedure, that generates subpopulations based on a percentage of similarity, seem to be equal to the standard replacement method. We therefore decided to omit any additional diversification methods in our evolutionary algorithm. The next experiments were concerned with the effects of each combine operator when taken as the only possible combination operator. Figure 5.8 shows, that both versions of the multiway combine operator, based on partitions as well as separators, seem to have the biggest impact on the solution size over time. After that, the node separator and vertex cover combine operators seem to be fairly equal in terms of quality and convergence. The intersection combine

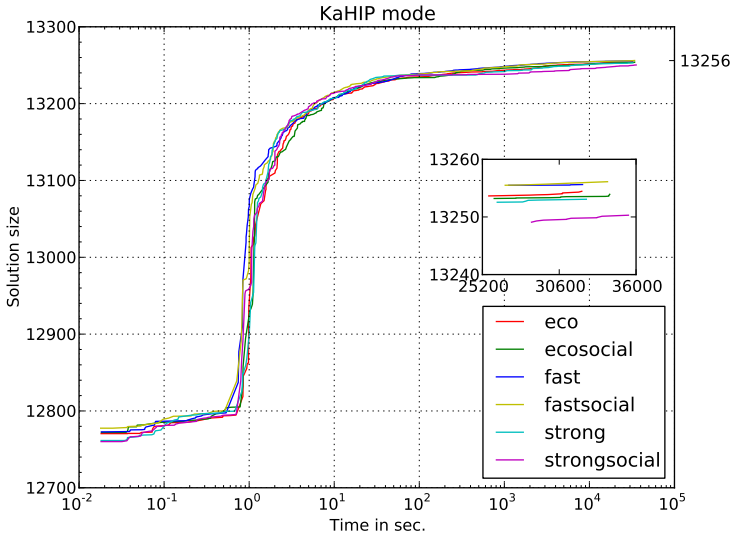


Figure 5.3: Evaluation of the different KaHIP modes. The social network graphs were excluded from this experiment, because the KaHIP framework is not able to process them with the non-social configurations. The global maximum is shown at the right.

operator comes in last in this comparison. This most probably results from the fact, that the intersection of two different independent sets is in most cases relatively small. Finally, we examine if graph partitioners really benefits our algorithm or if a simple BFS would be enough for our needs. As seen in Figure 5.9 the positive impact of using a graph partitioner cannot be denied, especially in the long run. It has to be noted, that both version were restricted to intersection, node separator and vertex cover combine operators, since we did not implement a version of the multiway combine operator that uses a BFS.

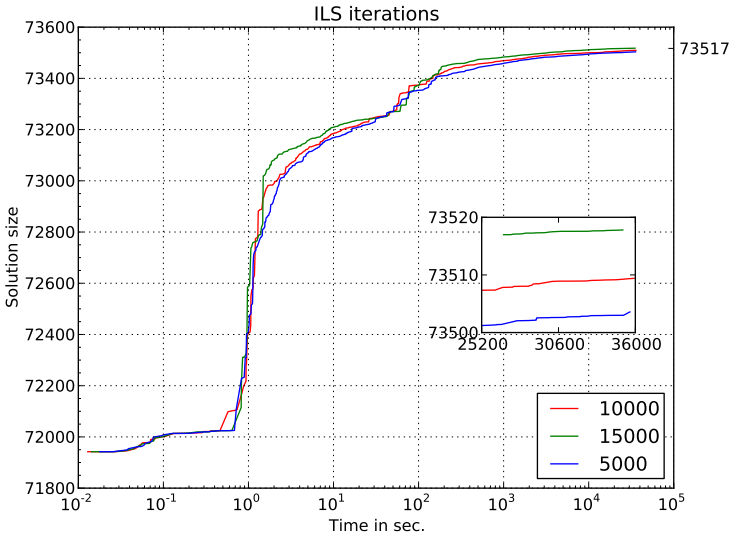


Figure 5.4: Effects of different numbers of ILS iterations. The global maximum is shown at the right.

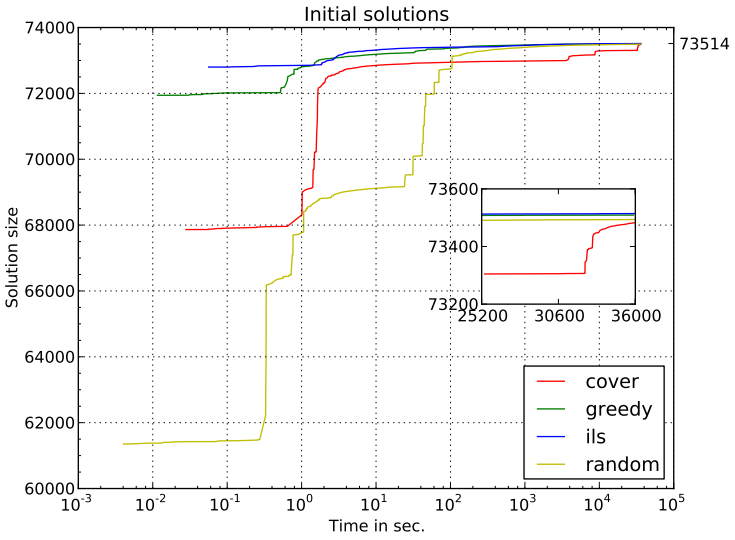


Figure 5.5: Evaluation of the different methods for generating initial solutions. The global maximum is shown at the right.

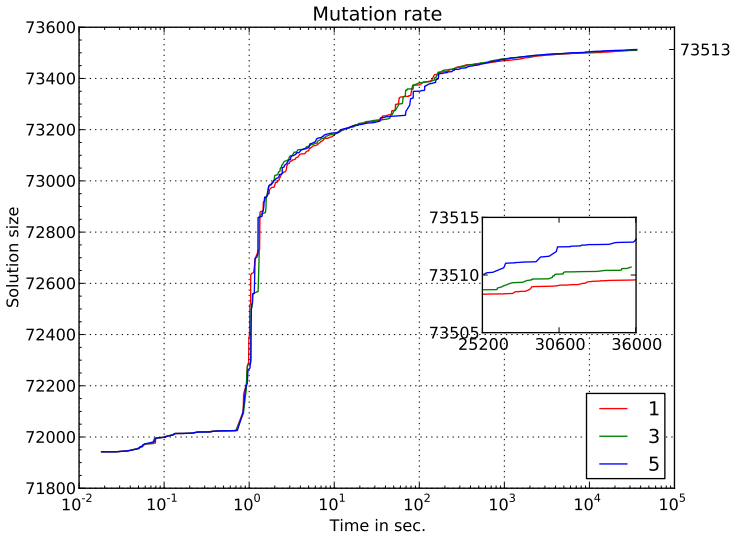


Figure 5.6: Effects of different mutation rates. The global maximum is shown at the right.

Parameter	Value
Population size	50
Pool size	30
KaHIP mode	fastsocial
Initial solutions	Greedy (minimal residual degree)
Mutation rate	10%
ILS iterations	15000
Tournament size	2
Multiway blocks	64
Multiway parents	$\|P\|$
Insertion threshold	150
Pool threshold	180
Pool renewal factor	10.0

Table 5.3: Configuration after the parameter tuning.

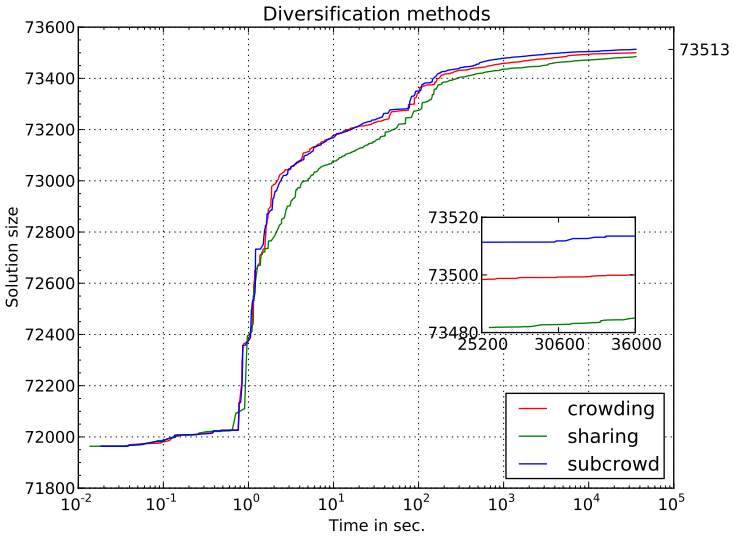


Figure 5.7: Impact of the diversification methods. The global maximum is shown at the right.

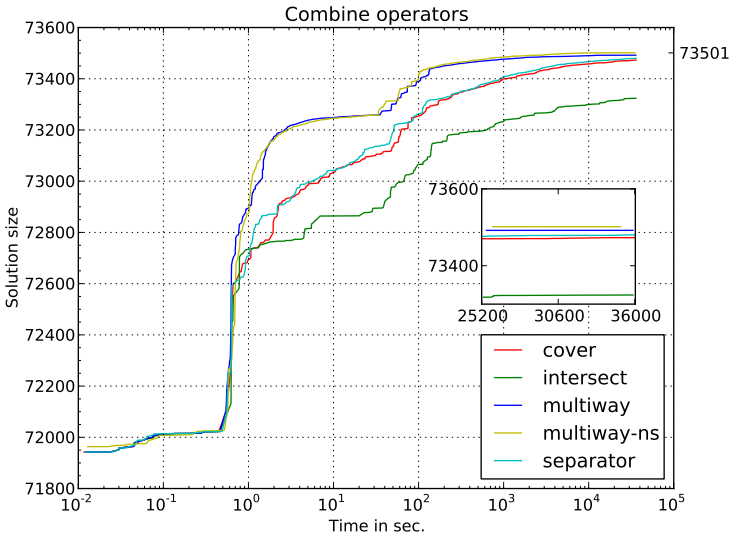


Figure 5.8: Results of our algorithm using only one combine operator at a time. The global maximum is shown at the right.

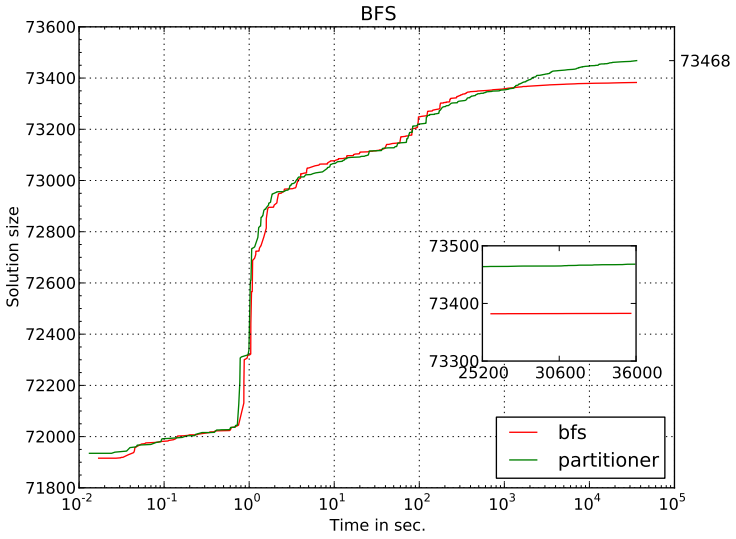


Figure 5.9: Comparison of our algorithm using the KaHIP framework and a BFS. The global maximum is shown at the right.

5.4 Comparison with ILS

In this section, we compare our evolutionary algorithm with the iterated local search algorithm by Andrade et. al. [1]. We do this by examining the convergence of both algorithms on the aforementioned graph families.

First off all, we want to take a closer look at the MESH family. On several instances, most notably `bunny` and `gameguy`, EvoMIS surpassed the ILS. As seen in Figure 5.10 and Figure 5.11 our algorithm starts of slower than the ILS, but is able to fully unfold its potential in the later stages. The slow start is due to the fact, that our algorithm first needs to establish a whole population of individuals that share a similar level of quality. After the population has stabilized, we can start improving the overall solution quality, which then leads to better individuals. This aspect holds true for most of our experiments and evolutionary algorithms in general. As noted by Andrade et. al. both of these instances are highly regular [1]. On irregular instances like `buddha` or `dragon` our algorithm falls short. Convergence plots for these instances can be found in Figure 5.12 and Figure 5.13. A detailed overview for all instances of this family can be found in Table 5.4.

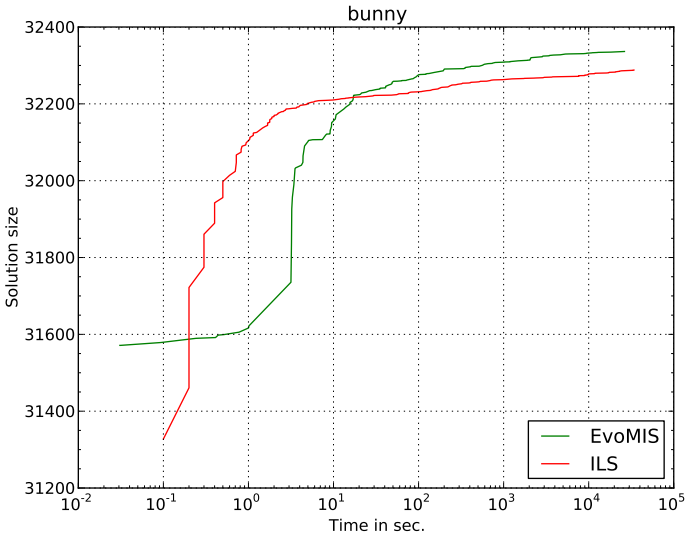


Figure 5.10: Comparison of EvoMIS and ILS for the `bunny` graph.

A family of graphs, our algorithm performs particularly well upon, is the SOCIAL family. Especially on instances like `cnr-2000` and `in-2004` it delivers a distinct advantage over the ILS. On `cnr-2000` we are able to improve the maximum found by the ILS by nearly 100 additional nodes. The convergence plot for this instance can be found in Figure 5.14. This plot shows, that due to the variety in

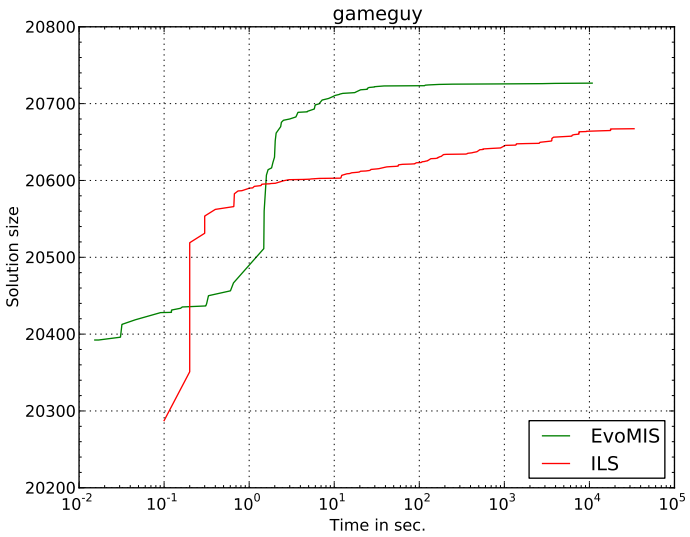


Figure 5.11: Comparison of EvoMIS and ILS for the `gameguy` graph.

starting solutions, we are able to head off from an already better level than our competitor. In particular, this holds true for most of the large instances, like the ones from SOCIAL or STREET. Throughout the rest of the test runs we were able to successfully maintain this advantage. The same goes for the convergence plot of `in-2004`, which is shown in Figure 5.15. For the remaining instances of this family the ILS and our algorithm are mostly equal. The detailed overview for all tested instances can be found in Table 5.5.

Another family of graphs, were we can report considerable improvements, is the WALSHAW family. As depicted in Table 5.9 we were able to exceed the ILS or be at least equal to it, on more than half of the instances from this test set. Some examples of this are `fe_ocean`, `fe_rotor` and `wave`. The according convergence plots can be seen in Figure 5.16, Figure 5.17 and Figure 5.18 respectively. The only graphs which were losses compared to the ILS in this family are `auto` (Figure 5.19) and `598a` (Figure 5.20). The significant increase of solution size our algorithm is able to show in this family, as well as the SOCIAL family, could result from the fact, that the partitioner we use, processes this kind of instances in a very profitable way.

On the instances from the GEOMETRIC (Table 5.7), DELAUNAY (Table 5.8) and STREET family (Table 5.6) the ILS proved to be superior to our evolutionary algorithm. The STREET family in particular is a clear win for the ILS. Unlike the SOCIAL family, where we also start from an advantageous point, we are not able to maintain it over the course of the test runs. For example, this can be seen in the

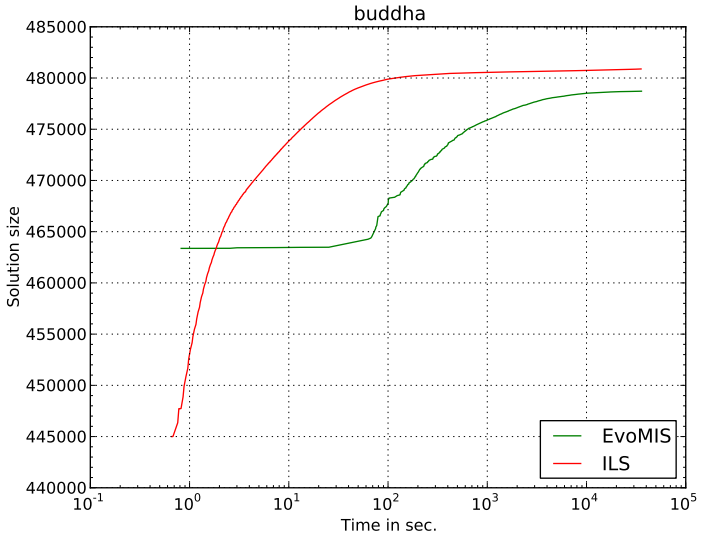


Figure 5.12: Comparison of EvoMIS and ILS for the `buddha` graph.

convergence plot of `nl` shown in Figure 5.21. In comparison to the SOCIAL family, street networks usually have a greater diameter and most nodes have a significantly lower degree. Our algorithm might not be able to handle such instances in a very good way. This aspect could be attributed in future enhancements of our algorithm. Our experiments for the MESH family indicated that our algorithm does really well on regular meshes. Since the GEOMETRIC and DELAUNAY family consist of fairly regular instances, the advantage the ILS has for these graphs can be attributed to the fact, that our algorithm was probably not able to fully converge in the given time limit. This also comes in play for many of the larger instances. As seen in Table 5.10 and Table 5.11, our algorithm performs really well on the smaller instances and loses its edge for most of the larger ones. Therefore, longer test runs for these instances might be necessary.

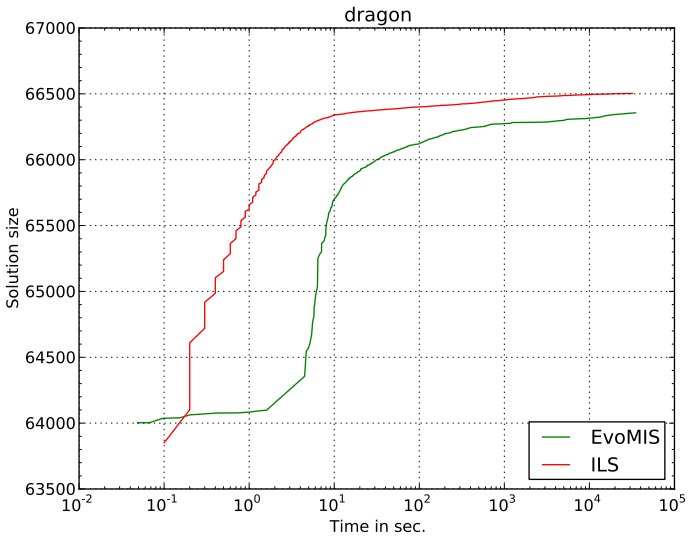


Figure 5.13: Comparison of EvoMIS and ILS for the `dragon` graph.

Graph		EvoMIS			ILS		
Name	n	Avg.	Max.	Min.	Avg.	Max.	Min.
`beethoven`	4 419	**2 004**	*2 004*	2 004	**2 004**	*2 004*	2 004
`blob`	16 068	7 248	7 249	7 247	**7 249**	*7 250*	7 249
`buddha`	1 087 716	478 713	478 760	478 678	**480 883**	*480 918*	480 816
`bunny`	68 790	**32 336**	*32 341*	32 330	32 288	32 292	32 285
`cow`	5 036	**2 346**	*2 346*	2 346	**2 346**	*2 346*	2 346
`dragon`	150 000	66 356	66 365	66 343	**66 503**	*66 505*	66 501
`dragonsub`	600 000	281 190	281 260	281 076	**282 029**	*282 066*	282 000
`ecat`	684 496	322 117	322 182	322 079	**322 263**	*322 290*	322 242
`face`	22 871	10 215	10 216	10 215	**10 217**	*10 217*	10 217
`fandisk`	8 634	**4 075**	*4 075*	4 075	4 073	4 074	4 072
`feline`	41 262	**18 847**	*18 851*	18 843	18 846	18 847	18 844
`gameguy`	42 623	**20 726**	*20 727*	20 726	20 669	20 690	20 659
`gargoyle`	20 000	8 850	8 851	8 849	**8 852**	*8 853*	8 852
`turtle`	267 534	122 275	122 305	122 247	**122 504**	*122 574*	122 435
`venus`	5 672	**2 684**	*2 684*	2 684	**2 684**	*2 684*	2 684

Table 5.4: Test results for the MESH family.

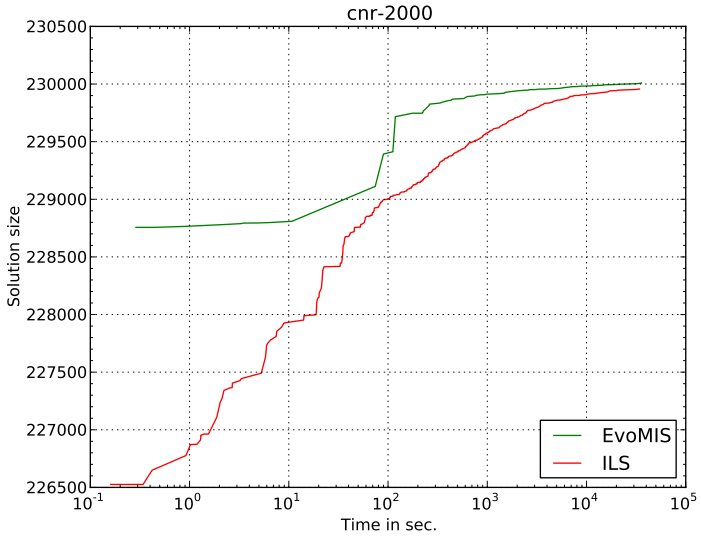


Figure 5.14: Comparison of EvoMIS and ILS for the `cnr-2000` graph.

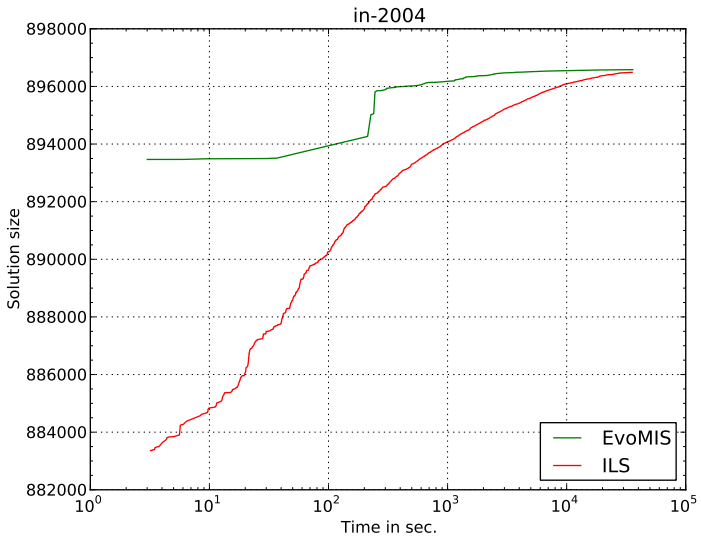


Figure 5.15: Comparison of EvoMIS and ILS for the `in-2004` graph.

Graph		EvoMIS			ILS		
Name	n	Avg.	Max.	Min.	Avg.	Max.	Min.
amazon	735 323	309 721	309 725	309 719	**309 792**	*309 793*	309 792
skitter	554 930	328 519	328 520	328 519	**328 596**	*328 614*	328 582
citation	268 495	**150 380**	*150 380*	150 380	**150 380**	*150 380*	150 380
cnr-2000	325 557	**230 008**	*230 016*	230 001	229 956	229 965	229 943
coPapers	434 102	**47 996**	*47 996*	47 996	**47 996**	*47 996*	47 996
enron	69 244	**62 811**	*62 811*	62 811	**62 811**	*62 811*	62 811
in-2004	1 382 908	**896 581**	*896 585*	896 580	896 488	896 495	896 475
gowalla	196 591	**112 369**	*112 369*	112 369	**112 369**	*112 369*	112 369
google	356 648	**174 072**	*174 072*	174 072	**174 072**	*174 072*	174 072

Table 5.5: Test results for the SOCIAL_NETWORK family.

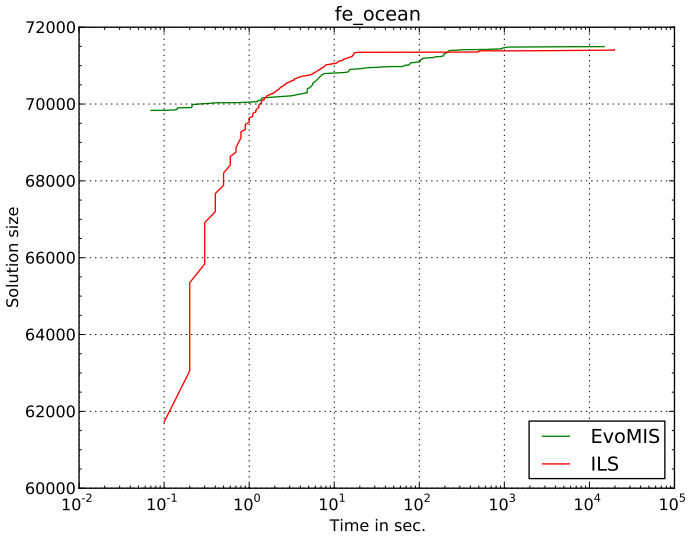


Figure 5.16: Comparison of EvoMIS and ILS for the fe_ocean graph.

Graph		EvoMIS			ILS		
Name	n	Avg.	Max.	Min.	Avg.	Max.	Min.
ger	11 548 845	5 836 807	5 836 966	5 836 583	**5 841 385**	*5 841 400*	5 841 375
gb	7 733 822	3 956 916	3 956 977	3 956 841	**3 958 507**	*3 958 512*	3 958 503
italy	6 686 493	3 351 464	3 351 531	3 351 398	**3 353 382**	*3 353 390*	3 353 378
lux	114 599	**57 663**	*57 663*	57 663	**57 663**	*57 663*	57 663
nl	2 216 688	1 116 215	1 116 255	1 116 179	**1 116 713**	*1 116 716*	1 116 709

Table 5.6: Test results for the STREET_NETWORK family.

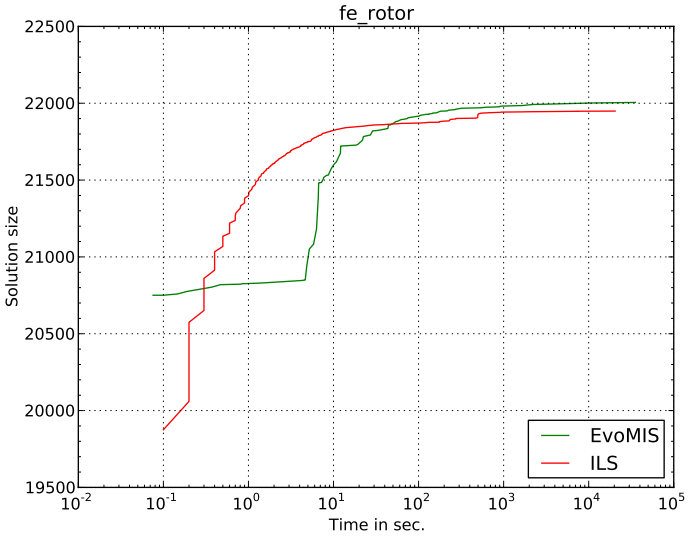


Figure 5.17: Comparison of EvoMIS and ILS for the `fe_rotor` graph.

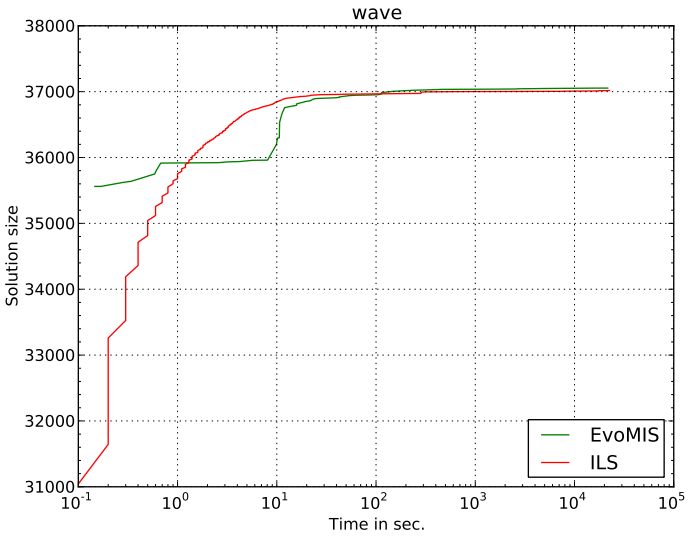


Figure 5.18: Comparison of EvoMIS and ILS for the `wave` graph.

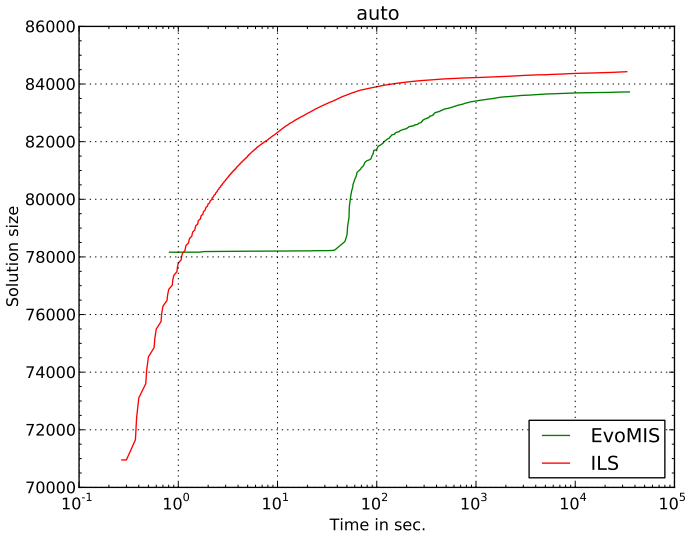


Figure 5.19: Comparison of EvoMIS and ILS for the `auto` graph.

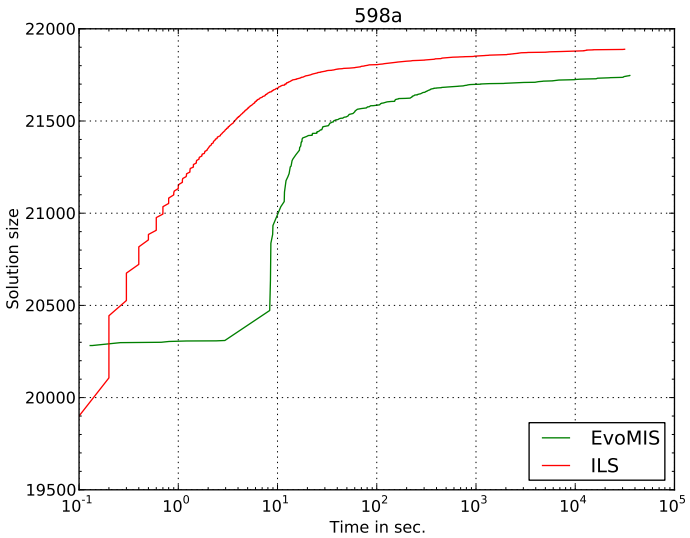


Figure 5.20: Comparison of EvoMIS and ILS for the `598a` graph.

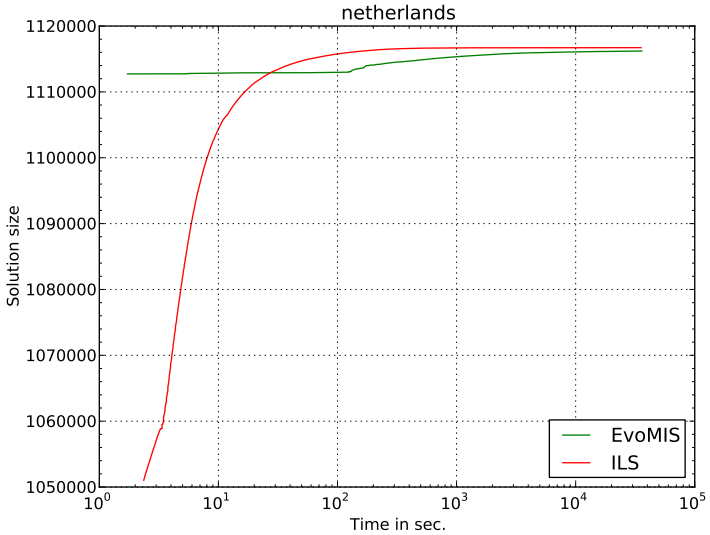


Figure 5.21: Comparison of EvoMIS and ILS for the `nl` graph.

Graph		EvoMIS			ILS		
Name	n	Avg.	Max.	Min.	Avg.	Max.	Min.
`rgg_n15`	32 768	**6 977**	*6 979*	6 977	6 974	6 974	6 974
`rgg_n16`	65 536	**13 277**	*13 278*	13 276	13 275	13 277	13 274
`rgg_n17`	131 072	25 389	25 395	25 384	**25 400**	*25 401*	25 399
`rgg_n18`	262 144	48 570	48 573	48 567	**48 638**	*48 639*	48 638
`rgg_n19`	524 288	93 042	93 072	93 020	**93 331**	*93 335*	93 328
`rgg_n20`	1 048 576	178 561	178 602	178 514	**179 322**	*179 328*	179 312

Table 5.7: Test results for the GEOMETRIC family.

Graph		EvoMIS			ILS		
Name	n	Avg.	Max.	Min.	Avg.	Max.	Min.
`del_n15`	32 768	**10 330**	*10 333*	10 328	**10 330**	10 330	10 330
`del_n16`	65 536	20 683	20 684	20 682	**20 688**	*20 689*	20 688
`del_n17`	131 072	41 292	41 294	41 289	**41 327**	*41 328*	41 326
`del_n18`	262 144	82 478	82 484	82 475	**82 601**	*82 603*	82 600
`del_n19`	524 288	164 959	164 972	164 949	**165 277**	*165 279*	165 276
`del_n20`	1 048 576	329 768	329 790	329 742	**330 599**	*330 603*	330 593

Table 5.8: Test results for the DELAUNAY family.

Graph		EvoMIS			ILS		
Name	n	Avg.	Max.	Min.	Avg.	Max.	Min.
4elt	15 606	**4 944**	*4 944*	4 944	4 941	4 942	4 940
598a	110 971	21 746	21 761	21 739	**21 889**	*21 891*	21 886
auto	448 695	83 726	83 764	83 702	**84 428**	*84 442*	84 402
bcsstk30	28 924	**1 783**	*1 783*	1 783	**1 783**	*1 783*	1 783
bcsstk31	35 588	**3 487**	*3 488*	3 487	3 486	3 487	3 486
brack2	62 631	21 413	21 414	21 413	**21 415**	*21 416*	21 415
crack	10 240	**4 603**	*4 603*	4 603	**4 603**	*4 603*	4 603
cs4	22 499	9 155	9 160	9 151	**9 172**	*9 173*	9 172
fe_ocean	143 437	**71 494**	71 587	71 447	71 411	*71 614*	71 291
fe_pwt	36 519	9 308	*9 310*	9 308	**9 309**	*9 310*	9 308
fe_rotor	99 617	**22 006**	*22 013*	21 999	21 949	21 979	21 902
fe_tooth	78 136	**27 793**	*27 793*	27 793	27 791	27 792	27 791
vibrobox	12 328	**1 851**	*1 851*	1 851	1 850	*1 851*	1 850
wave	156 317	**37 054**	*37 058*	37 050	37 016	37 031	36 999

Table 5.9: Test results for the WALSHAW family.

Graph		EvoMIS			ILS		
Name	n	Avg.	Max.	Min.	Avg.	Max.	Min.
beethoven	4 419	**2 004**	*2 004*	2 004	**2 004**	*2 004*	2 004
cow	5 036	**2 346**	*2 346*	2 346	**2 346**	*2 346*	2 346
venus	5 672	**2 684**	*2 684*	2 684	**2 684**	*2 684*	2 684
fandisk	8 634	**4 075**	*4 075*	4 075	4 073	4 074	4 072
crack	10 240	**4 603**	*4 603*	4 603	**4 603**	*4 603*	4 603
vibrobox	12 328	**1 851**	*1 851*	1 851	1 850	*1 851*	1 850
4elt	15 606	**4 944**	*4 944*	4 944	4 941	4 942	4 940
blob	16 068	7 248	7 249	7 247	**7 249**	*7 250*	7 249
gargoyle	20 000	8 850	8 851	8 849	**8 852**	*8 853*	8 852
cs4	22 499	9 155	9 160	9 151	**9 172**	*9 173*	9 172
face	22 871	10 215	10 216	10 215	**10 217**	*10 217*	10 217
bcsstk30	28 924	**1 783**	*1 783*	1 783	**1 783**	*1 783*	1 783
del_n15	32 768	**10 330**	*10 333*	10 328	**10 330**	10 330	10 330
rgg_n15	32 768	**6 977**	*6 979*	6 977	6 974	6 974	6 974
bcsstk31	35 588	**3 487**	*3 488*	3 487	3 486	3 487	3 486
fe_pwt	36 519	9 308	*9 310*	9 308	**9 309**	*9 310*	9 308
feline	41 262	**18 847**	*18 851*	18 843	18 846	18 847	18 844
gameguy	42 623	**20 726**	*20 727*	20 726	20 669	20 690	20 659
brack2	62 631	21 413	21 414	21 413	**21 415**	*21 416*	21 415
del_n16	65 536	20 683	20 684	20 682	**20 688**	*20 689*	20 688
rgg_n16	65 536	**13 277**	*13 278*	13 276	13 275	13 277	13 274
bunny	68 790	**32 336**	*32 341*	32 330	32 288	32 292	32 285
enron	69 244	**62 811**	*62 811*	62 811	**62 811**	*62 811*	62 811
fe_tooth	78 136	**27 793**	*27 793*	27 793	27 791	27 792	27 791
fe_rotor	99 617	**22 006**	*22 013*	21 999	21 949	21 979	21 902
598a	110 971	21 746	21 761	21 739	**21 889**	*21 891*	21 886
lux	114 599	**57 663**	*57 663*	57 663	**57 663**	*57 663*	57 663
del_n17	131 072	41 292	41 294	41 289	**41 327**	*41 328*	41 326
rgg_n17	131 072	25 389	25 395	25 384	**25 400**	*25 401*	25 399
fe_ocean	143 437	**71 494**	71 587	71 447	71 411	*71 614*	71 291
dragon	150 000	66 356	66 365	66 343	**66 503**	*66 505*	66 501
wave	156 317	**37 054**	*37 058*	37 050	37 016	37 031	36 999
gowalla	196 591	**112 369**	*112 369*	112 369	**112 369**	*112 369*	112 369
del_n18	262 144	82 478	82 484	82 475	**82 601**	*82 603*	82 600
rgg_n18	262 144	48 570	48 573	48 567	**48 638**	*48 639*	48 638
turtle	267 534	122 275	122 305	122 247	**122 504**	*122 574*	122 435

Table 5.10: Test results for all graphs sorted by the number of nodes n.

Graph		EvoMIS			ILS		
Name	n	Avg.	Max.	Min.	Avg.	Max.	Min.
citation	268 495	**150 380**	*150 380*	150 380	**150 380**	*150 380*	150 380
cnr-2000	325 557	**230 008**	*230 016*	230 001	229 956	229 965	229 943
google	356 648	**174 072**	*174 072*	174 072	**174 072**	*174 072*	174 072
coPapers	434 102	**47 996**	*47 996*	47 996	**47 996**	*47 996*	47 996
auto	448 695	83 726	83 764	83 702	**84 428**	*84 442*	84 402
del_n19	524 288	164 959	164 972	164 949	**165 277**	*165 279*	165 276
rgg_n19	524 288	93 042	93 072	93 020	**93 331**	*93 335*	93 328
skitter	554 930	328 519	328 520	328 519	**328 596**	*328 614*	328 582
dragonsub	600 000	281 190	281 260	281 076	**282 029**	*282 066*	282 000
ecat	684 496	322 117	322 182	322 079	**322 263**	*322 290*	322 242
amazon	735 323	309 721	309 725	309 719	**309 792**	*309 793*	309 792
del_n20	1 048 576	329 768	329 790	329 742	**330 599**	*330 603*	330 593
rgg_n20	1 048 576	178 561	178 602	178 514	**179 322**	*179 328*	179 312
buddha	1 087 716	478 713	478 760	478 678	**480 883**	*480 918*	480 816
in-2004	1 382 908	**896 581**	*896 585*	896 580	896 488	896 495	896 475
nl	2 216 688	1 116 215	1 116 255	1 116 179	**1 116 713**	*1 116 716*	1 116 709
italy	6 686 493	3 351 464	3 351 531	3 351 398	**3 353 382**	*3 353 390*	3 353 378
gb	7 733 822	3 956 916	3 956 977	3 956 841	**3 958 507**	*3 958 512*	3 958 503
ger	11 548 845	5 836 807	5 836 966	5 836 583	**5 841 385**	*5 841 400*	5 841 375

Table 5.11: Test results for all graphs sorted by the number of nodes n. Continuation of Table 5.10.

6. Conclusion and Future Work

6.1 Conclusion

In this thesis we presented a novel evolutionary algorithm for the maximum independent set problem. In this context, we introduced four new combine operators to generate independent sets out of existing ones. These operators include methods that make use of intersections, vertex covers and node separators. For finding good crossover cuts, we utilized the KaHIP framework by Peter Sanders and Christian Schulz [26]. To make sure the offsprings of our algorithm are of sufficient quality, we used the fast local search by Andrade et. al. [1]. Their iterated local search serves as both, a tool to further improve our offsprings, as well as our main competitor in the experimental evaluation.

First off, we were able to show that using a partitioner instead of a simple BFS to acquire (multiple-point) cuts is beneficial for this kind of algorithm. Our experimental comparison further showed that our algorithm is able to exceed the ILS on a number of instances, which are interesting for the MIS problem. It performed especially well on parts of the MESH family, which was introduced by Andrade et. al. [1] and consists of triangular meshes. Other families that our algorithm handled very well were the SOCIAL family, which consists of different types of social networks, as well as the WALSHAW families, which is a well-known benchmark for graph parititoners. As mentioned before, this could be due to the fact, that the KaHIP framework really benefits us, when processing this kind of instances.

6.2 Future Work

A possible extension for our algorithm would be to incorporate the Hopcroft-Karp algorithm in our multiway combine operator. This would help us to fix the solutions, that are generated by this operator, in a possibly better way. Another interesting approach would be to parallelize our algorithm with the goal of further increasing its performance. During the implementation of our algorithm we kept this possibility

in mind and therefore this task should be attended easily. One could also try to use other graph partitioners besides the KaHIP framework and see if they produce comparable results. The next step we want to take with our algorithm is to design combine operators that work recursively on the blocks of a partition or separator. Another promising idea that we want to try is to break down the graph into smaller subgraphs, e.g. using a node separator, and then use our algorithm on these subgraphs to get good partial solutions. Afterwards, these solutions can be put together and further improved by the ILS. This procedure could also be performed recursively to get an even smaller set of graphs, for which the MIS problem can be solved efficiently. The additional granularity an approach like this adds to our algorithm might allow us to produce even better independent sets. We also want to try to fine-tune our algorithm for different kinds of graph families and perform longer test runs.

A. Algorithms

Algorithm 1 Pseudocode for the local search

```
for all nodes n in the solution do
    REMOVE n
    for all neighbour-pairs (u, v) of n do
        if (u, v) can be inserted into the solution then
            INSERT (u, v) into the solution
        end if
    end for
    if no new nodes could be added then
        REINSERT n
    end if
end for
```

Algorithm 2 Pseudocode for the ILS

```
while TERMINATION criterion not satisfied do
    solution' ← PERTURB solution
    apply LOCAL SEARCH on solution'
    if change conditions are satisfied then
        solution ← solution'
    end if
end while
```

Algorithm 3 Pseudocode for random initial solutions

```
while nodes left in G do
    PICK a random node n
    if no neighbour of n is in the solution then
        ADD n to the solution
    end if
    REMOVE n from G
end while
```

Algorithm 4 Pseudocode for greedy initial solution (least residual degree)

```
INITIALIZE the priority queue with all nodes
while nodes left in the priority queue do
    PICK a random node n with least residual degree
    REMOVE n from G and the queue
    for all neighbours k of n do
        REMOVE k from G and the queue
        for all neighbours l of k do
            UPDATE the degree of l
        end for
    end for
end while
```

Algorithm 5 Pseudocode for greedy initial solution (maximum residual degree)

```
INITIALIZE the priority queue with all nodes
while nodes left in the priority queue do
    PICK a random node n with maximum residual degree
    REMOVE n from G and the queue
    for all neighbours k of n do
        if k is completely covered then
            REMOVE k from G and the queue
        end if
    end for
end while
GENERATE the complement of the solution
```

Algorithm 6 Pseudocode for the intersection combine operator.

for all nodes n in G **do**
  if $n \in \mathcal{P}_1$ and $n \in \mathcal{P}_2$ **then**
    ADD n to the new solution $\mathcal{O}$
  end if
end for
$\mathcal{O}' \leftarrow$ *MAXIMIZE* $\mathcal{O}$
$\mathcal{O}'' \leftarrow$ *LOCAL SEARCH* applied to $\mathcal{O}'$
CREATE offspring with $\mathcal{O}''$

Algorithm 7 Pseudocode for the node separator combine operator.

GENERATE a node separator S
for all nodes n in G **do**
  if $n \in \mathcal{P}_1$ and $n \in A$ **then**
    ADD n to the new solution $\mathcal{O}_1$
  end if
  if $n \in \mathcal{P}_2$ and $n \in B$ **then**
    ADD n to the new solution $\mathcal{O}_1$
  end if
  if $n \in \mathcal{P}_1$ and $n \in B$ **then**
    ADD n to the new solution $\mathcal{O}_2$
  end if
  if $n \in \mathcal{P}_2$ and $n \in A$ **then**
    ADD n to the new solution $\mathcal{O}_2$
  end if
end for
$\mathcal{O}'_j \leftarrow$ apply *GREEDY* heuristic on $\mathcal{O}_j$ ▷ $j \in \{1, 2\}$
$\mathcal{O}''_j \leftarrow$ *MAXIMIZE* $\mathcal{O}'_j$
$\mathcal{O}'''_j \leftarrow$ *LOCAL SEARCH* applied to $\mathcal{O}''_j$
CREATE offspring with $\mathcal{O}'''_j$

Algorithm 8 Pseudocode for the vertex cover combine operator.

GENERATE a partition $V = A \cup B$
$\overline{\mathcal{P}}_j \leftarrow$ *COMPLEMENT* $\mathcal{P}_j$ ▷ $j \in \{1, 2\}$
for all nodes n in G **do**
  if $n \in \overline{\mathcal{P}}_1$ and $n \in A$ **then**
    ADD n to the new solution $\overline{\mathcal{O}}_1$
  end if
  if $n \in \overline{\mathcal{P}}_2$ and $n \in B$ **then**
    ADD n to the new solution $\overline{\mathcal{O}}_1$
  end if
  if $n \in \overline{\mathcal{P}}_1$ and $n \in B$ **then**
    ADD n to the new solution $\overline{\mathcal{O}}_2$
  end if
  if $n \in \overline{\mathcal{P}}_2$ and $n \in A$ **then**
    ADD n to the new solution $\overline{\mathcal{O}}_2$
  end if
end for
bound $\leftarrow$ *EXTRACT* the partitioning cut boundary
$H \leftarrow$ apply *HOPCROFT-KARP* on *bound*
$\overline{\mathcal{O}}'_j \leftarrow H \cup \overline{\mathcal{O}}_j$ ▷ $j \in \{1, 2\}$
$\mathcal{O}_j \leftarrow$ *COMPLEMENT* $\overline{\mathcal{O}}'_j$
$\mathcal{O}'_j \leftarrow$ *MAXIMIZE* $\mathcal{O}_j$
$\mathcal{O}''_j \leftarrow$ *LOCAL SEARCH* applied to $\mathcal{O}'_j$
CREATE offspring with $\mathcal{O}''_j$

Algorithm 9 Pseudocode for the multiway combine operator using partitions.

SELECT a number of parents q
SELECT a number of blocks k
$B \leftarrow$ *GENERATE* a k-way partition ▷ $i \in \{1, \ldots, q\}$
for all parents $\mathcal{P}_i$ **do** ▷ $j \in \{1, \ldots, k\}$
  for all blocks $V_j \in B$ **do**
    CALCULATE the score of $(\mathcal{P}_i, V_j)$
  end for
end for
best $\leftarrow$ *SELECT* the best parent for each block ▷ $j \in \{1, \ldots, k\}$
for all blocks $V_j \in B$ **do**
  SELECT highest scoring parent from *best*
  ADD nodes of parent inside V_j to new solution $\mathcal{O}$
end for
$\overline{\mathcal{O}} \leftarrow$ *COMPLEMENT* $\mathcal{O}$
$\overline{\mathcal{O}}' \leftarrow$ apply *GREEDY* algorithm to $\overline{\mathcal{O}}$
$\mathcal{O}' \leftarrow$ *COMPLEMENT* $\overline{\mathcal{O}}'$
$\mathcal{O}'' \leftarrow$ *MAXIMIZE* $\mathcal{O}'$
$\mathcal{O}''' \leftarrow$ *LOCAL SEARCH* applied to $\mathcal{O}''$
CREATE offspring with $\mathcal{O}'''$

Algorithm 10 Pseudocode for the multiway combine operator using node separators.

SELECT a number of parents q
SELECT a number of blocks k
$(S, B) \leftarrow$ *GENERATE* a k-way node separator ▷ $i \in \{1, \ldots, q\}$
for all parents $\mathcal{P}_i$ **do** ▷ $j \in \{1, \ldots, k\}$
  for all blocks $V_j \in B$ **do**
    CALCULATE the score of $(\mathcal{P}_i, V_j)$
  end for
end for
best $\leftarrow$ *SELECT* the best parent for each block ▷ $j \in \{1, \ldots, k\}$
for all blocks $V_j \in B$ **do**
  SELECT highest scoring parent from *best*
  ADD nodes of parent inside V_j to new solution $\mathcal{O}$
end for
$\mathcal{O}' \leftarrow$ apply *GREEDY* algorithm to $\mathcal{O}$
$\mathcal{O}'' \leftarrow$ *MAXIMIZE* $\mathcal{O}'$
$\mathcal{O}''' \leftarrow$ *LOCAL SEARCH* applied to $\mathcal{O}''$
CREATE offspring with $\mathcal{O}'''$

Algorithm 11 Pseudocode for assuring the similarity of individuals.

Require: Offspring $\mathcal{O}$
  $best \leftarrow ACQUIRE$ the best individuum so far
  $differences \leftarrow 0$
  $max\_differences \leftarrow MAX\_INT$
  $replacement\_found \leftarrow false$
  for all individuum $\mathcal{I}_i$ in the population **do** ▷ $i \in \{1, \ldots, p\}$
    if $\mathcal{I}_i == best$ **then**
      $SKIP$
    end if
    for all nodes j in G **do**
      if $\mathcal{I}_i[j] \neq \mathcal{O}[j]$ **then**
        $differences++$
      end if
    end for
    if $differences < max\_differences$ **then**
      $max\_differences = differences$
      $replacement \leftarrow ind_i$
      $replacement\_found \leftarrow true$
    end if
  end for

Bibliography

[1] Diogo Vieira Andrade, Mauricio G. C. Resende, and Renato Fonseca F. Werneck. Fast Local Search for the Maximum Independent Set Problem. In *Proceedings of the 7th International Workshop on Experimental Algorithms (WEA 2008)*, pages 220–234, 2008.

[2] William Brendel, Mohamed R. Amer, and Sinisa Todorovic. Multiobject Tracking as Maximum Weight Independent Set. In *Proceedings of the 24th IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2011)*, pages 1273–1280, 2011.

[3] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. *CoRR*, abs/1311.3144, 2013.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[5] Kenneth Alan De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, 1975. AAI7609381.

[6] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks - Design, Analysis, and Simulation [DFG priority program 1126]*, pages 117–139, 2009.

[7] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. *The Shortest Path Problem: 9th DIMACS Implementation Challenge*, volume 74. American Mathematical Soc., 2009.

[8] Matthijs den Besten, Thomas Stützle, and Marco Dorigo. Design of Iterated Local Search Algorithms. In *Proceedings of the EvoWorkshops 2001: EvoCOP, EvoFlight, EvoIASP, EvoLearn, and EvoSTIM on Applications of Evolutionary Computing*, pages 441–451, 2001.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[10] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[11] Bruce Hendrickson. Chaco. In *Encyclopedia of Parallel Computing*, pages 248–249. 2011.

[12] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a Scalable High Quality Graph Partitioner. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010)*, pages 1–12, 2010.

[13] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Scientific Computing*, 2(4):225–231, 1973.

[14] David S. Johnson and Michael A. Trick. *Cliques, Coloring, and Satisfiability: 2nd DIMACS Implementation Challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

[15] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Trans. VLSI Syst.*, 7(1):69–79, 1999.

[16] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[17] Heinz König. Über das Von Neumannsche Minimax-Theorem. *Archiv der Mathematik*, 19(5):482–487, 1968.

[18] Jure Leskovec, Daniel P. Huttenlocher, and Jon M. Kleinberg. Predicting Positive and Negative Links in Online Social Networks. In *Proceedings of the 19th International Conference on World Wide Web (WWW 2010)*, pages 641–650, 2010.

[19] Jure Leskovec, Daniel P. Huttenlocher, and Jon M. Kleinberg. Signed Networks in Social Media. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems (CHI 2010)*, pages 1361–1370, 2010.

[20] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: Densification Laws, Shrinking Diameters and possible Explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 177–187, 2005.

[21] Brad L. Miller and David E. Goldberg. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems*, 9:193–212, 1995.

[22] José Ruiz-Shulcloper and Gabriella Sanniti di Baja, editors. *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, volume 8258 of *Lecture Notes in Computer Science*. Springer, 2013.

[23] Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. Efficient Traversal of Mesh Edges using Adjacency Primitives. *ACM Trans. Graph.*, 27(5):144, 2008.

[24] Peter Sanders and Christian Schulz. Distributed Evolutionary Graph Partitioning. In *Proceedings of the 14th Meeting on Algorithm Engineering & Experimentation (ALENEX 2012)*, pages 16–29, 2012.

[25] Peter Sanders and Christian Schulz. Kahip v0.53 - Karlsruhe High Quality Partitioning - User Guide. *CoRR*, abs/1311.1714, 2013.

[26] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA 2013)*, volume 7933 of *LNCS*, pages 164–175, 2013.

[27] Christian Schulz. *High Quality Graph Partitioning*. epubli GmbH, Karlsruhe Institut of Technology, 2013.

[28] N. J. A. Sloane. Challenge Problems: Independent Sets in Graphs, 2000. `http://www.research.att.com/~njas/doc/graphs.html`.

[29] Alan J. Soper, Chris Walshaw, and Mark Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *J. Global Optimization*, 29(2):225–241, 2004.

[30] Tycho Strijk, Bram Verweij, and Karen Aardal. Algorithms for Maximum Independent Set Applied to Map Labelling. Technical report, 2000.

[31] Chris Walshaw. The Graph Partitioning Archive, 2000. `http://staffweb.cms.gre.ac.uk/~wc06/partition/`.

[32] K. Xu. BHOSLIB: Benchmarks with Hidden Optimum Solutions for Graph Problems, 2004. `http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm`.