Engineering Memetic Algorithms for Hypergraph b-Matching

Jakob Erben

March 27, 2025

3727893

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisor:
Henrik Reinstädtler

Co-Referee:
Prof. Dr. Michael Gertz

Acknowledgments

I would like to thank Professor Schulz and Henrik Reinstädtler for their expert supervision of this thesis and their invaluable support. I would also like to thank my friends and family for their support and encouragement.
I acknowledge support by the state of Baden-Württemberg through bwHPC.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, 27.03.2025

Jakob Erben

Abstract

A hypergraph extends the concept of a graph by allowing edges, called hyperedges, to connect more than two vertices, enabling the modeling of more complex relationships. The hypergraph b-matching problem is a generalization of the hypergraph matching problem. The objective is to select a subset of hyperedges such that each vertex v is incident to at most $b(v)$ of the chosen hyperedges.
In this work, a memetic approach to solve the hypergraph b-matching problem is introduced. By using partitions of the hypergraph, two hypergraph b-matchings of the same hypergraph can be combined into a potentially better one. This central idea is used as the crossover operator, around which the memetic algorithm is built. Extensive experimentation has been conducted to validate the efficacy of these algorithms. While the results obtained are not yet on par with the current state of the art, initial results are encouraging.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Graphs serve as a potent modeling instrument, enabling the formulation of numerous real-life problems as graph problems [68]. Subsequently, a suitable graph algorithm can be employed to efficiently solve these problems. A notable problem within this framework pertains to the identification of a matching in a graph. The applications of graph matching encompass domains such as protein structure prediction [8], computer network gossiping algorithms [9], and in the optimal solving of unit-demand auctions [24]. Nevertheless, the conversion of real-life problems into graph matching problems is not without its constraints.

One approach to address these limitations involves relaxing certain constraints from the original problem. A notable example is the generalization from graphs to hypergraphs, which has been applied to various problems, such as graph coloring [77] and minimum cuts [17]. The generalized problem considered in this thesis is the hypergraph b-matching problem, which extends the maximal matching problem by generalizing two aspects. First, it is formulated for hypergraphs, wherein hyperedges can connect any number of vertices instead of exactly two, as in graphs. Second, the concept of a matching is extended to a b-matching , which allows a vertex to be included in multiple matched edges. In this model, a vertex v can be included in at most $b(v)$ hyperedges of the matching, instead of being restricted to a single one. This generalization enables the modeling of many real-life problems that could not be captured by the traditional maximal graph matching problem.

The hypergraph b-matching problem has been shown to be NP-hard [30], therefore, efficient approximation solvers are required. The currently best-performing algorithm for solving this problem was presented by Großmann et al. [38], which combines novel data reduction techniques with a greedy solver and a local search algorithm.

The efficacy of memetic algorithms in numerous optimization problems has been well-documented. These algorithms are a type of genetic algorithm that integrates one or more

local search methods [62]. The incorporation of local search enables memetic algorithms to function as a hybrid of genetic approaches and methods that refine a single solution [61]. This hybridization aims to combine the best aspects of both techniques.

1.2 Our Contribution

This thesis builds upon the current best solution for the hypergraph b-matching problem presented by Großmann et al. [38]. To achieve this objective, components from their approach are incorporated and a genetic framework is developed around them, thereby forming a memetic algorithm. In particular, a novel crossover operator is introduced that combines two existing solutions into a better solution. This operator serves as the key component of the memetic approach.
The resulting algorithm, termed *HeiMemBMatcher*, is parallelized to utilize available computational resources and maximize solution quality. The algorithm is presented in multiple configurations. One is optimized for speed and the other for solution quality. Finally, the approach is evaluated on a set of benchmark instances to assess solution quality and computational performance. Although the results do not yet match the current state of the art, they still highlight some promising initial findings.

1.3 Structure

The remainder of this thesis is organized as follows. In Chapter 2, the fundamentals of the hypergraph b-matching problem, and parallel computation are introduced, and an overview of related work in these fields is provided in Chapter 3. The proposed algorithm and its various components, with a special focus on the memetic aspects, are presented in Chapter 4. The subsequent chapter, Chapter 5, delves into the parallelization of the algorithm. Chapter 6 then covers the most important data structures employed in the algorithms' implementation. Chapter 7 contains the experimental evaluation, including parameter tuning and final performance analysis. Finally, Chapter 8 summarizes our findings and proposes potential avenues for future research.

CHAPTER 2

Fundamentals

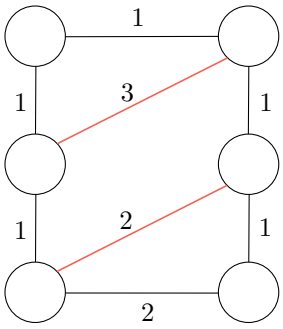
The following chapter introduces the basic concepts of graph theory and uses them to outline the graph matching and graph b-matching problems. These problem definitions are then extended to their corresponding formulations for hypergraphs. Additionally, the hypergraph partitioning problem is introduced. The chapter concludes with basic concepts of parallel computing.

2.1 General Definitions

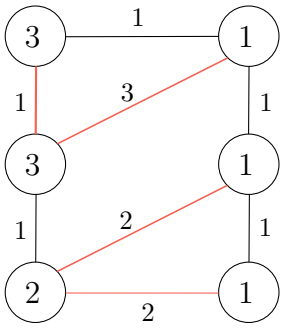
This section introduces graphs and hypergraphs, which serve as the foundational structures employed throughout this thesis.

Graphs. A *graph* G is defined as a pair of sets, V and E. The set of *vertices* is denoted by V and the set of *edges* is denoted by E. An edge $e \in E$ is a two-element subset of V. Therefore, the set of edges E can be expressed as follows: $E = [V]^2$. Two vertices $u, v \in V$ contained in an edge $e = \{u, v\}$ are called *adjacent*. For brevity, it is denoted as $e = uv$. A vertex v is called *incident* to an edge e if $v \in e$. The number of edges a vertex is incident to is called its degree. The neighborhood of a vertex v is defined as the set of all vertices adjacent to v. The closed neighborhood of v is denoted as $N[v] = \{v\} \cup \{u \in V \mid uv \in E\}$. A graph is called weighted if there exists a function $w : E \to \mathbb{R}_{>0}$ that assigns a weight to each edge.

Hypergraphs. A *hypergraph* H is defined as a pair (V, E), where V is a set of *vertices* and E is a set of *hyperedges*. A hyperedge e is defined as a non-empty subset of V of any size. A hypergraph in which each hyperedge contains exactly d vertices is called *d-uniform*. Therefore, graphs are 2-uniform hypergraphs.
A hyperedge $e \in E$ is called *incident* to a vertex v if $v \in e$. Two vertices that are shared



(a) Example of a maximal matching, with the matched edges highlighted in red.



(b) Example of a maximal b-matching, with the matched edges highlighted in red.

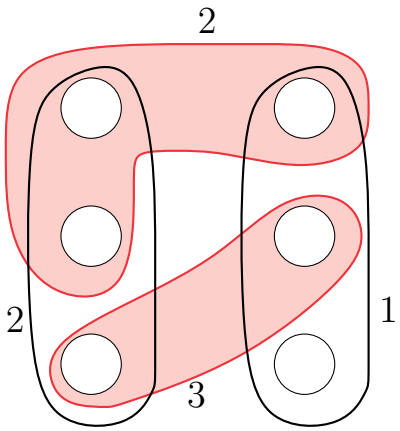
Figure 2.1: Examples of maximal matching and maximal b-matching.

by at least one hyperedge are called *adjacent*. If there exists a function $w : E \to \mathbb{R}_{>0}$ that assigns a weight to each hyperedge, the hypergraph is called *weighted*.
The *neighborhood* of a vertex v is the set of all vertices that are adjacent to v. The closed neighborhood is denoted as $N[v] = \{v\} \cup \{u \in V \mid \exists e \in E, v \in e, u \in e\}$.
A hypergraph is called d*-partite* if the vertices can be partitioned into d disjoint sets, such that each hyperedge contains precisely one vertex from each block.

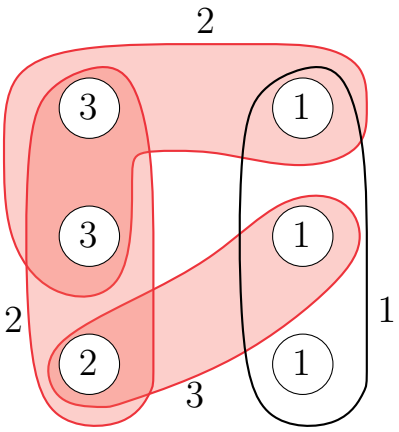
2.2 Problems Definitions

Following the establishment of graphs and hypergraphs, the problem definitions pertinent to this thesis can be introduced. The matching problem is initially presented in its fundamental form and subsequently expanded to define the hypergraph b-matching problem. Additionally, the hypergraph partitioning problem is introduced.

Graph Matching Problems. A matching $\mathcal{M} \subseteq E$ of a graph is a subset of its edges such that no vertex is incident to more than one edge in $\mathcal{M}$. If a graph is weighted, there exists a matching that has the highest possible weight among all possible matchings. The weight of a matching is defined as the sum of the edge weights of the contained edges, that is, $w(\mathcal{M}) = \sum_{e \in \mathcal{M}} w(e)$.
A matching is called *maximal* if no additional edge can be added without violating the matching property, and it is called *maximum* if it has the highest possible weight. A problem where $w(e) \equiv 1$ for every edge e is called a *cardinality* problem.
The matching problem can be extended by allowing each vertex to be incident to at most $b(v)$ edges, where $b : V \to \mathbb{N}$. This extended problem is called the b*-matching* problem. The value $b(v)$ is referred to as the *capacity* of a vertex v. The weight of a b-matching is defined in a similar manner. If $b(v) \equiv 1$, then the problem is identical to the maximum graph matching problem. An edge is called *matched* if it is contained in the matching. As



(a) Example of a maximal hypergraph matching, with the matched edges highlighted in red.



(b) Example of a maximal hypergraph b-matching, with the matched edges highlighted in red.

Figure 2.2: Examples of maximal matching and maximal b-matching.

illustrated in Figure 2.1a, a maximal matching for the specified weighted graph is demonstrated, with the matched edges highlighted in red. A similar representation is provided in Figure 2.1b, which depicts a maximal b-matching for the aforementioned graph. The capacities of the vertices are indicated within them, and once more, the matched edges are highlighted in red.

Hypergraph Matchings. A hypergraph matching is a subset of hyperedges such that no vertex is incident to more than one hyperedge in the matching. The weight of a hypergraph matching is defined as the sum of the weights of the contained hyperedges $w(\mathcal{M}) = \sum_{e \in \mathcal{M}} w(e)$.
A hypergraph matching is called *maximal* if no additional hyperedge can be added without violating the matching property, and *maximum* if it has the highest possible weight.
The hypergraph matching problem can be extended by allowing each vertex to be incident to at most $b(v)$ hyperedges, where $b : V \to \mathbb{N}$. This extended problem is called the *hypergraph b-matching* problem. The weight of a hypergraph b-matching is defined in the same way. If $b(v) \equiv 1$, then the problem is identical to the maximum hypergraph matching problem.
Figure 2.2a shows an example of a maximal hypergraph matching for the given weighted hypergraph, where the matched hyperedges are highlighted in red. Similarly, Similarly Figure 2.2b presents a maximal b-matching for the same hypergraph. The capacities are again indicated within the vertices, and the matched hyperedges highlighted in red.

Partitioning. The objective of the *partitioning* problem is to divide the vertices of a graph or hypergraph into k disjoint subsets. If $k = 2$, the partition is referred to as a

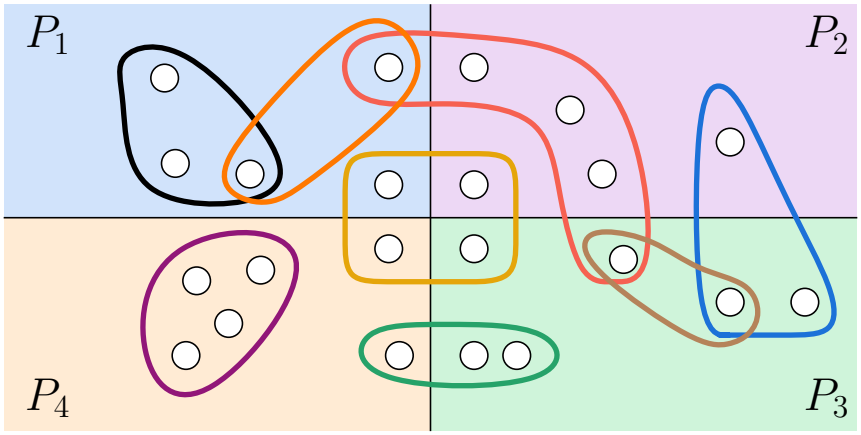


Figure 2.3: Example of a partitioning of a hypergraph into four blocks.

bipartition. The disjoint subsets are called *blocks* and are labeled $P_1, P_2, \ldots, P_k$.
The partitioning problem also classifies the edges: edges that contain only vertices from the same block are termed *inner partition edges*, while edges that contain vertices from different blocks are called *cut edges*. The quality of a partition is measured by an objective function. The two most significant objective functions aim to minimize the *weighted edge cut* and the *connectivity*, respectively. The weighted edge cut is the sum of the edge weights of all cut edges. The connectivity metric also takes into account the number of blocks that an edge connects. The connectivity of an edge e is denoted as λ_e. The connectivity of a partition is therefore given by $\sum_{e \in Cut}(\lambda_e - 1) \cdot w(e)$.
An additional constraint can be introduced, requiring the blocks to be of similar size. This constraint is called the *balance* constraint. The balance constraint, and it is satisfied if the size of any block is at most $(1 + \epsilon)$ times the average block size. The value ϵ is called the *imbalance*.
Figure 2.3 provides an illustration of a hypergraph partitioned into four blocks, with the yellow, red, green, and blue hyperedges representing the cut edges.

2.3 Parallel Computing

In the context of parallel computing, the solution to a given problem is achieved through the concurrent operation of multiple *processing elements* (PEs). Each PE is responsible for managing its own computation unit and storage. A problem that can be distributed across multiple PEs and consequently be solved without requiring communication between them is referred to as *embarrassingly parallel*. Theoretically, solving such problems on n PEs results in an n-fold acceleration in computation speed.
Different PEs can communicate with each other by sending *messages*. The time needed to exchange messages is called *communication time*. Messages can be exchanged in a *blocking* or *non-blocking* way. When using blocking communication, the PEs need to wait

until the message is either sent or received. In the non-blocking approach, the PEs can continue working on other tasks while waiting for both communication partners to become available.
Two metrics are frequently utilized when parallelizing a problem. The *speedup* $S = \frac{T(1)}{T(n)}$ is the ratio of the time needed to solve the problem on a single PE, $T(1)$, to the time needed to solve the problem with n PEs, $T(n)$. The *efficiency* E is the ratio of the speedup to the number of PEs. For n PEs, the efficiency is therefore $E = \frac{S}{n}$. The efficiency is a measure of how well the parallelization scales with the number of PEs. If the efficiency is close to 1, the parallelization scales well. Conversely, if the efficiency is close to 0, the parallelization does not scale well. Embarrassingly parallel problems have a theoretical efficiency of 1.

CHAPTER 3

Related Work

The subsequent chapter will present the related work for this thesis. The discussion will commence with a review of previous research on the maximum matching problem, followed by an examination of research on the hypergraph matching problem. Ultimately, the chapter will conclude with an exploration of current scientific progress for memetic algorithms for different related problems and research on the hypergraph partitioning problem.

3.1 Maximum Matching

The maximum matching problem is a well-known problem in graph theory. It was proven to be solvable in $O(|E||V|^2)$ by Edmonds [29]. There have been numerous improved solution approaches to the exact maximum graph matching problem [60, 26, 58]. Furthermore, several methods have been developed to provide quick approximation results, i.e., $\frac{1}{2}$-approximations [25, 69] and a $(1 - \epsilon)$ approximation by Duan et al. [27]. Brin et al. [13] presented an approximation with linear work in parallel. Efficient data structures for the problem were introduced by Gabow [33]. Additionally, multiple data reduction techniques have been proposed, for example, by Korenwein et al. [47]. For a comprehensive overview of data reduction results, readers are referred to [1]. Recent advances include a parallel algorithm presented by Schwing et al. for solving the maximum cardinality matching problem on GPUs [71].

3.2 Maximum Hypergraph Matching

The hypergraph matching on d-partite and d-uniform hypergraphs can be poorly approximated, as has been shown by Hazan et al. [40], who showed the poor approximation for the equivalent maximum d-set packing problem. They limit the quality of the approximation

to within a factor of $\mathcal{O}(d/\log d)$. For the more general problem of matchings in non-uniform hypergraphs, Håstad [42] showed that the problem is NP-hard and that there is no $n^{1-\epsilon}$ factor approximation unless $P = NP$. Therefore, the same holds for the maximum independent set problem. Cygan et al. [19] utilized local search to provide a $(k+1+\epsilon)/3$-approximation algorithm for the k-set packing problem. This result can be applied for the matching problem in d-uniform, d-partite hypergraphs. Fřer and Yu [32] refined their results by improving the runtime of the algorithm.
Anneg et al. [6] present an improved optimality bound for a relaxed linear programming (LP) approach for the non-uniform hypergraph matching problem. An approximation bound for the d-packing problem of 1.786 for $d = 3$ is presented by Thiery and Ward in [76]. This result can again be applied to d-partite and d-uniform hypergraphs. In recent advances, Neuwohner [63] proved a threshold below $\frac{k}{2}$ by $\Omega(k)$. Both these thresholds improve on the established local search approach by Berman [11] for the maximum weight independent set in d-claw-free graphs.
Effective heuristic approaches have been proposed by Dufosse et al. [28]. Their main idea is to reduce the complexity of the uniform problem. Therefore, they expand the two Karp-Sipser [46] rules in order to apply them to hypergraphs. Furthermore, they introduce the idea of utilizing the Sinkhorn-Knopp algorithm [74] to enable an additional rule. However, their experimental evaluation is limited to d-partite, d-uniform hypergraphs with uniform edge weights.

3.3 Hypergraph b-Matching

The non-existence of an approximation for the hypergraph b-matching cardinality problem was shown by El Ouali and Jäger [30]. They showed that there is no polynomial-time approximation within any ratio smaller than $\Omega(\frac{b}{\log b})$. The maximum weight b-matching problem has some approximation when restricting to k-uniform hypergraphs. Krysta [50] presented a greedy approach which results in a $k+1$ approximation. Parekh and Pritchard [64] used linear programming to construct a $(k-1+\frac{1}{k})$ approximation algorithm. Koufogiannakis and Young [48] developed a k-approximation by employing a distributed algorithm. However, these approximations are only shown theoretically, and no practical implementations of these algorithms are known.
Recently, Großmann et al. [38] engineered effective data reduction-based algorithms that are combined with local search. They present a first practical algorithm for the general hypergraph b-matching problem. In this work, their reductions and local search are utilized at various places and their solution is used as a baseline to compare against.

3.4 Memetic Approaches for Related Problems

In the case of $b \equiv 1$, the problem is reducible to the maximum independent set problem on its line graph. The maximum independent set problem is defined as identifying the heaviest subset of vertices in a graph such that no two selected vertices are connected by an edge. To construct the corresponding a line graph, each edge in the original graph is represented as a vertex in the line graph. Two vertices in the line graph are connected if their corresponding edges in the original graph are adjacent. Memetic algorithms exist for the maximum unweighted [51, 52] and weighted independent set problem [37]. However, we are unaware of any other algorithms for $b(v) > 1$.
Memetic algorithms, however, have been successfully applied to a variety of graph problems. For instance, Andres et al. [4] present a memetic algorithm for the graph partitioning problem. Lü et al. [57] propose a memetic algorithm for the graph coloring problem, and Biedermann et al. [12] introduce a memetic algorithm for graph clustering.

3.5 Hypergraph Partitioning

For an extensive overview of hypergraph partitioning algorithms, see [16]. The balanced partitioning problem itself is $\mathcal{NP}$-hard [14, 53]; however, effective heuristics such as the multilevel scheme [36] perform well in practice. In this work, the KaHyPar framework [70] is employed, a multilevel hypergraph partitioner that utilizes advanced coarsening, refinement, and preprocessing techniques to achieve high-quality partitioning results with a good tradeoff between runtime and solution quality.

CHAPTER 4

Memetic Hypergraph B-Matching Algorithm

The algorithm under consideration in this thesis is a memetic algorithm, a specialized form of a genetic algorithm. Memetic algorithms are population-based metaheuristics that extend the genetic framework with one or more local search algorithms [62]. Thus, they form a hybrid approach between population-based evolutionary algorithms and those aimed at improving a single individual [61]. The next section will outline the general structure of the algorithm. Sections 4.2–4.5 will then present each of the components in greater detail.

4.1 General Structure

Algorithm 1 delineates the general structure of the memetic approach. The algorithm's input parameters include a hypergraph H, a weighting function w that assigns a weight to each edge, and a b-function that assigns a capacity to each node. The algorithm first applies reductions to the hypergraph H to reduce the problem size.
The subsequent steps in the algorithmic process are solving the problem for the reduced formulation H_R. It is only at the termination of this process that the solution is transformed back to the original hypergraph H. It is imperative to note that, due to the reduction step being exact and deterministic, it needs to be executed only once. The reduction process is introduced in Section 4.2.
Subsequently, the *Initialize* procedure is called repeatedly to create the individuals of a population. The Initialize process receives the reduced problem H_R and creates a heuristic solution for this problem. It consists of a greedy matching algorithm that quickly generates an initial matching $\mathcal{M}_i$. A local search algorithm then improves the quality of the initial solution. The created individual solutions are then collected to form the initial population $\mathcal{P}_{init}$. Section 4.3 details this process and discusses the quality of the created population.
Once a population $\mathcal{P}$ has been established, it is then repeatedly altered through the genetic

Algorithm 1 A high level overview of the memetic hypergraph b-matching algorithm. After building a population, the individuals repeatedly cooperate, compete and get improved.

```
1:  procedure HeiMemBMatch(H = (V, E), w(e) : e ∈ E → ℕ, b(v) : v ∈ V → ℕ)
2:      H_R ← Reductions(H)
3:      P ← Initialize(H_R)
4:      best ← SelectBestIndivual(P )
5:      while not Stopping Criterion do
6:          P_new ← Cooperate(P )
7:          P_new ← Improve(P_new )
8:          P ← Compete(P_new ,P )
9:          if SelectBestIndivual(P )>best then
10:             best ← SelectBestIndivual(P )
11:     return best
12: procedure Initialize(H_R = (V_R, E_R, w()))
13:     for i ∈ 1..N do
14:         M_i ← GreedyMatching(H_R)
15:         M_i ← LocalSearch(M_i)
16:     return {M_i | i ∈ {1, . . . , N}}
```

process. The genetic approach was extended to form a memetic algorithm by adding a local search algorithm, and the required components are referred to as *memetic steps*. The first step, *cooperation*, receives a population $\mathcal{P}$ and creates a new population $\mathcal{P}_{new}$ by combining pairs of individuals. The details of the cooperation step are provided in Section 4.4.1. These newly created individuals collectively form $\mathcal{P}_{new}$. The new population is then further improved as part of the *improvement* step, which is described in Section 4.4.2. Finally, the old population $\mathcal{P}$ and the new population $\mathcal{P}_{new}$ compete to determine which individuals are kept for the next iteration. The competition step is detailed in Section 4.4.3.
Each iteration of the algorithm results in a new population, which is called a *generation*. The algorithm determines the best individual in each generation and compares it to the best individual found so far, storing the better of the two. This guarantees that the algorithm retains the best discovered solution upon termination. To determine whether the algorithm should terminate or if another generation will be computed, the stopping criterion is evaluated. Several stopping criteria are possible, and these are presented in Section 4.5. Once the stopping criterion is fulfilled, the algorithm terminates and returns the best solution discovered.

4.2 Reductions

The size of a hypergraph, for which a problem must be solved, can be reduced using so-called *data reduction rules*, or simply *reductions*. The output of such a process is called a

Reduction	Description
Abundant Vertices	Remove nodes with a capacity higher than their degree.
Neighborhood Removal	Add edges, which are heavier than the $b(v)$-th heaviest edges adjacent to v.
Weighted Isolated Edge Removal	From edges, which form a clique that contains a node of capacity 1, the heaviest edge is selected for the solution.
Weighted Edge Folding	If an edge has exactly 2 non-adjacent neighbors (the shared vertices have capacity 1) with lower weight, than the three edges can be combined to form a shared edge.
Weighted Twin Folding	If two edges share exactly 2 non-adjacent neighbors (the shared vertices have capacity 1) with lower weight, than the four edges can be combined to form a shared edge.
Weighted Domination	If an edge is the superset of another edge with higher weight, and they share a vertex with capacity 1 it can safely be removed.

Table 4.1: Overview over the data reduction rules presented by Großmann et al. [38]

reduced hypergraph, and if the reductions are exact, they guarantee that an optimal solution for the original hypergraph can be computed on the reduced instance [1].
The data reductions used as part of this thesis were presented by Großmann et al. [38] and are exact and deterministic reductions. They present several data reduction rules for the hypergraph b-matching problem, which are presented in Table 4.1. The table additionally provides a concise description for each rule, with more detailed explanations in [38].
These rules adopt various approaches. The first approach is to remove unnecessary vertices or edges. The *Abundant Vertices* rule removes vertices with excess capacity, ensuring they never impact the solution. Conversely, *Weighted Domination* eliminates edges that directly compete with another edge that is always preferred.
The subsequent approaches involve identifying edges that are invariably part of an optimal solution. *Neighborhood Removal* and *Weighted Isolated Edge Removal* actualize this approach by storing the corresponding edges separately, removing them from the hypergraph, and reducing the capacities of contained vertices accordingly. The reduced instance no longer necessitates consideration of these edges.
Lastly, *Weighted Edge Folding* and *Weighted Twin Folding* identify sets of edges that can be decided as a single entity. These edges are then combined to form a single edge in the reduced hypergraph and can be decided at once.
The reductions are applied repeatedly, as the successful application of a single rule may modify the hypergraph in a way that allows another reduction to be applied. Großmann et al. [38] determined that the effectiveness of the reductions varies significantly from instance to instance, with some being reduced significantly while many are not reduced at all. Additionally, they determined that when solving the b-matching problem with an ILP solver,

the reductions lead to a speedup for 80% of instances. Notably, 40% of the executions were accelerated by a factor of at least 2.

4.3 Population Initialization

This step receives a reduced hypergraph H_R, which is generated using the reductions presented in Chapter 4.2. It generates the initial population $\mathcal{P}_{init}$ through repeatedly creating individual heuristic solutions, which are then collected to form the population. The creation of an individual can be divided into two separate steps. First, a matching $\mathcal{M}$ is constructed using a greedy approach, where edges are iteratively selected and added to the matching. The specifics of this greedy selection process are detailed in Section 4.3.1. Section 4.3.2 explains the secondary step, in which a local search algorithm is applied to refine the solution, enhancing its quality. This process is repeated N times to generate the initial population. Finally, an analysis of the quality of the initial population is presented in Section 4.3.3.

4.3.1 Greedy Matching

A greedy algorithm always makes the most promising choice at a given time, assuming that a combination of these locally optimal solutions leads to a globally good solution [18]. For a hypergraph b-matching problem, the greedy matching is given the edges E as well as an evaluation function $r() : e \in E \rightarrow \mathbb{R}$ (also sometimes called a *reward function*). It starts with an empty matching $\mathcal{M} \leftarrow \emptyset$. In step t, it then selects an edge $e_t \in E^t$, where E^t denotes the set of matchable edges at that time. In addition, e_t maximizes the evaluation function:

$$e_t = \arg\max_{e_j \in E^t} r(e_j)$$

This thesis utilizes different evaluation functions proposed and compared by Großmann et al. [38]. They analyzed different evaluation functions and found that the best solution on average was the *pin*-evaluation function, which divides the weight of an edge by its size. This function has the added advantage of being a stable value regardless of which edges have already been selected.
The greedy process therefore works as follows: First, all edges are sorted based on the evaluation function. Then, they are processed according to this order, adding each edge when possible. This approach is deterministic, producing identical solutions upon repeated execution. Since the memetic algorithm benefits from an initial population containing diverse individuals, as will be discussed in Section 4.3.3, additional randomness is introduced to the solutions.
To this end, three methods were developed to randomize the greedy matching process. The first method involves the introduction of random tie-breaking, which is achieved by shuffling the edges before sorting them according to the evaluation function. The second

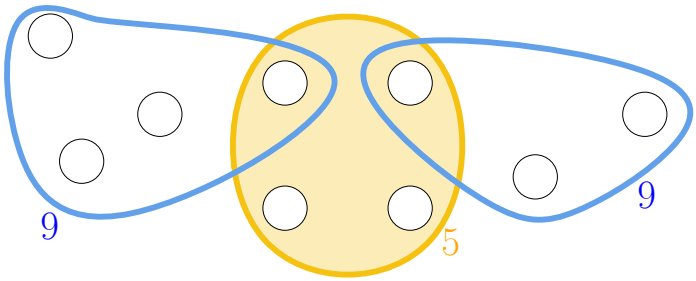


Figure 4.1: Example of a (1,2)-swap. The edge yellow edge, currently in the matching, is removed. Subsequently, the two blue edges, which previously were only blocked by the yellow edge, are added to the matching.

method involves the addition of random noise to the values computed by the evaluation function. Finally, a method is employed that randomly pushes edges to a later position in the order with a given probability, and the distance by which an edge is pushed is also determined probabilistically. A comparison of these approaches is presented in Section 7.3.

4.3.2 Local Search

Local search has been identified as a highly effective solving solution whenever an exhaustive search is impractical. The underlying principle of local search is to incrementally enhance a solution by systematically incorporating and subsequently excluding components of the solution [45]. Rather than addressing the problem on a global scale, local decisions are made and then integrated to form a global solution.
The local search employed in this study was introduced by Großmann et al. [38] and is an iterative approach, which repeatedly executes two distinct algorithms to modify the solutions. Firstly, a *swapping algorithm*, which swaps edges currently in the solution for blocked edges, if it improves the solution quality. Secondly, the local search incorporates a *perturbation* component, which modifies the solution to facilitate the avoidance of local optima.
Prior to initiating the local search, the current solution is stored, and after each iteration, the solution quality is evaluated. If an improvement is detected, the current solution becomes the starting point for the subsequent iteration. Conversely, if no improvement is detected, the solution is typically discarded. However, in some cases, slightly worse solutions are accepted as the starting point for the next iteration, thereby enhancing the algorithm's capability to escape local optima. The local search is terminated by either the completion of a predetermined number of iterations or by surpassing a predefined timeout.

(1,2)-Swaps. The fundamental concept underlying (1,2)-Swaps is illustrated in Figure 4.1. Assume that the yellow edge is currently in the matching. Furthermore, assume that the two blue edges are not in the matching and are only blocked by the yellow edge.

Given that the combined weight of the two non-adjacent blue edges exceeds that of the yellow edge, they are deemed a superior option for the matching. Hence, the yellow edge is removed and the blue edges are incorporated into the matching.
The swapping algorithm is designed to identify triplets of edges that can undergo a (1,2)-swap.
Once a successful swap is executed, the matching is maximized, ensuring that all edges, which have been freed up in the process are incorporated.

Perturbation. The solution is additionally perturbed. This is achieved by iterating through the unselected edges and forcing some of them into the matching. This approach was draws inspiration from the work of Andrade et al. [3], who presented a similar method for the independent set problem. Typically, when the perturbation is applied, a single edge is forced into the matching. However, with a certain probability, more edges are forced into the matching. The process of forcing edges into a matching is further explained in Section 4.4.2, wherein the insertion is executed during the mutation.

4.3.3 Population Quality

Previous work shows that a better initial population increases the probability that a good final solution is reached [15, 78]. A pivotal factor in evaluating the quality of a population is its size. Achieving an optimal population size necessitates finding a balance. Insufficient population size can result in suboptimal solutions due to premature convergence to a local optimum [66, 56, 49]. Conversely, an overly large population necessitates a greater computational expense and a longer time duration to reach a solution [66, 56].
Lobo and Goldberg [56] present an approach to solving this issue. They propose determining the population size in relation to the problem size. This thesis uses a slightly altered version of this approach. The b-matching problem may have varying levels of difficulty for the same hypergraph, depending on the given capacities. For example, one could pick a b-function that assigns each node a capacity equal to its degree. In this scenario, the solution is simply to add all edges to the matching. Regardless of the hypergraph's size, this can be solved trivially. Hence, the population size is proportional to the time required to create a single instance, including the reduction. The underlying principle is that if the process takes a significant amount of time, then the problem offers numerous decision and is considered to be hard.
Another important metric for evaluating the quality of a population, both initially and throughout the entire execution, is its diversity. A population with high diversity is more likely to produce a good solution, as it is more likely to explore a more extensive portion of the search space [23]. To promote diversity in the population, the greedy matching function is modified to introduce additional variation. Furthermore, the non-deterministic local search contributes to the diversity. The diversity of a population can be measured in different ways. In this thesis, the *Jaccard distance* is used to measure similarity between

different sets [72]. The Jaccard distance for a pair of matchings $\mathcal{M}_i$ and $\mathcal{M}_j$ is defined as:

$$J(\mathcal{M}_i, \mathcal{M}_j) = 1 - \frac{|\mathcal{M}_i \cap \mathcal{M}_j|}{|\mathcal{M}_i \cup \mathcal{M}_j|}$$

The diversity of the population is then described as the average Jaccard distance between all pairs of individuals. Section 7.3 presents experimental results regarding the diversity of the population.

4.4 Memetic steps

After an initial population $\mathcal{P}$ is created, the memetic algorithm attempts to improve the quality of the population over time. To this end, the following process is executed iteratively. Individuals are selected and subsequently crossed over to form improved solutions in a process called *Cooperation*, explained in Section 4.4.1. The solution quality of these newly created individuals is then further refined in the *Improvement* step, which is delineated in Section 4.4.2. Finally, the new and old populations are combined in the *Competition* step, detailed in Section 4.4.3. The resulting population then serves as the starting point for the next iteration. This process is repeated until a stopping criterion is met, at which point the algorithm terminates. Potential stopping criteria are presented in Section 4.5.

4.4.1 Cooperation

The objective of the cooperation step is to combine different individuals. A parent pair, consisting of two individuals, is selected for the purpose of being combined into a new individual, which is referred to as an offspring. The purpose of cooperation is for the offspring to inherit the advantageous characteristics of their parents. Furthermore, this should enable the algorithm to explore previously unexplored areas of the search space [62].
To perform the *cooperation*, it is first necessary to select parent pairs. This process is referred to as *parent selection* and will be explained subsequently. The parents are then combined using a process called *crossover*. This thesis uses a crossover operator that combines individuals with the aid of a bipartition of the underlying hypergraph. This crossover operator constitutes the central idea around which this thesis is structured. A well-designed crossover operator has been shown to substantially improve the performance of memetic algorithms [65, 43].

Parent Selection

The *parent selection* process receives a population $\mathcal{P}$ which is then used to calculate the qualities of the individuals, also known as fitness values. In the context of the hypergraph b-matching problem, the weight of the different matchings is employed as a measure of

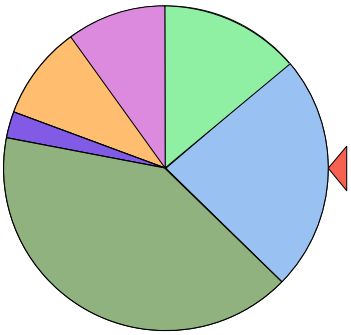


Figure 4.2: Example of fitness proportional sampling. The individual corresponding to the blue section is selected.

their fitness. This approach is considered conventional as the weight of the matching corresponds to the problem's objective function, which is frequently selected to be the fitness function [54]. The objective is to select multiple parent pairs in a manner that ensures the creation of high-quality offspring. One intuitive approach to achieving this objective is to cross over individuals who are particularly fit. However, this objective is in direct competition with the need to maintain diversity within the population. This challenge arises because repeatedly selecting the fittest individuals leads to offspring that are also fit. However, this reduces genetic diversity. This trade-off can be managed by adjusting the *selective pressure*. Selective pressure determines how strongly higher-quality solutions are favored in selection, while lower-quality solutions are rarely chosen [73]. The literature presents multiple selection strategies, and this thesis explores different parent selection strategies. Section 7.3 presents experimental results comparing these different strategies.

Naive Selection. The naive selection method involves the random selection of individuals from a population to establish parent pairs. This approach is characterized by its simplicity and speed. However, it does not generate any selection pressure. It functions as a baseline that should be outperformed by subsequent solutions.

Fitness Proportional Selection. *Fitness proportional selection* (FPS) is a process that links the likelihood of an individual being selected to its degree of fitness. This process can be conceptualized as repeatedly spinning a roulette wheel, as illustrated in Figure 4.2. The wheel is spun multiple times, with each revolution selecting one parent at a time. The roulette wheel is divided into sections, with larger sections representing fitter instances and smaller sections representing less fit individuals [39, 73].
In the figure, the individuals corresponding to the green and blue sections are very fit, while the orange and purple ones are relatively unfit. Consequently, the individuals in the green and blue sections are more likely to be selected, while those in the orange and purple sections are less likely to be selected.
This method has a weak point when all sections have very similar weights. In such cases,

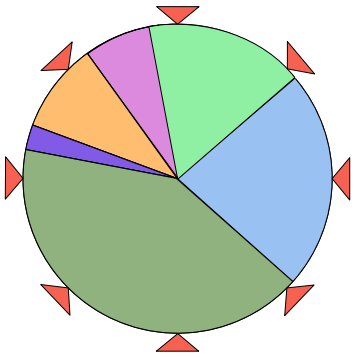


Figure 4.3: Example of stochastic universal sampling. The eight pointers sample at once, some are sampled multiple times.

the method tends to behave similarly to the naive approach, which provides no selection pressure toward fitter instances [39]. This issue can be mitigated by converting fitness values into ranks, where each individual's fitness value is replaced by its rank after sorting by fitness. This modified approach is referred to as *rank-based fitness proportional selection* (FPS-Rank).

Stochastic Universal Sampling. *Stochastic Universal Sampling* (SUS) can be understood as an extension of the FPS method. Rather than utilizing a single pointer as in the FPS method, there are N evenly spread pointers, where N is the number of individuals to be selected. The primary advantage of this approach is its lack of bias [67].
Figure 4.3 shows a roulette wheel with eight evenly spaced pointers, which select eight individuals at once. These individuals are then randomly combined to form parent pairs.

Tournament Selection. In tournament selection, a hyperparameter k, called the *tournament size*, is required. For the selection of each parent pair, a random subset of k individuals is picked from the population. These individuals enter a tournament, from which the two fittest individuals are selected [34]. In instances where multiple individuals possess identical fitness values, the selection among them is randomized.
In this parent selection strategy, the selection pressure can be adjusted by setting k. For larger values of k, the fittest individuals are more likely to participate in many tournaments and, therefore, are strongly represented in the chosen parent pairs. For smaller values of k, fitter individuals statistically appear in fewer tournaments, reducing the selection pressure. However, choosing a tournament size that is too large can lead to a loss of diversity [20].

Crossover

The *crossover* operator receives a pair of individuals, designated as the parents, along with a bipartition of the hypergraph. Subsequently, the operator generates an offspring. The validity of the offspring is shown in the following theorem.

Theorem 1
Given a pair of valid hypergraph b-matchings , $\mathcal{M}_1$ and $\mathcal{M}_2$, and a bipartition (P_0, P_1) of the corresponding hypergraph, any combination of two edge sets corresponding to the two blocks form a valid matching $\mathcal{M}_{new}$.
An edge set $\mathcal{M}_{i,j}$ corresponding to a block P_j can be constructed by removing all edges from the matching $\mathcal{M}_i$ that do not only contain vertices from the corresponding block.

Proof: Assume, for contradiction, that this is not the case. Then there must be at least one vertex v that is incident to at least $b(v) + 1$ edges from $\mathcal{M}_{new}$. Further, assume that v is part of the block P_j. Then these edges must originate from some edge set $\mathcal{M}i, j$, which can only contain edges from the valid matching $\mathcal{M}_i$. However, this would imply that $\mathcal{M}_i$ is invalid, contradicting our assumption.

Algorithm 2 The crossover operator, with emphasis on producing fit offsprings.

```
1:  procedure CROSSOVER(M_1, M_2, (P_0, P_1))
2:      M_{1,0}, M_{1,1}, M_{1,cut} ← partitionEdgeSets(M_1)
3:      M_{2,0}, M_{2,1}, M_{2,cut} ← partitionEdgeSets(M_2)
4:      M_new ← ∅
5:      if w(M_{1,0}) > w(M_{2,0}) then
6:          M_new ← M_new ∪ M_{1,0}
7:      else
8:          M_new ← M_new ∪ M_{2,0}
9:      if w(M_{1,1}) > w(M_{2,1}) then
10:         M_new ← M_new ∪ M_{1,1}
11:     else
12:         M_new ← M_new ∪ M_{2,1}
13:     M_cut ← M_{1,cut} ∪ M_{2,cut}
14:     for e ∈ M_cut do
15:         if isMatchable(M_new, e) then
16:             M_new ← M_new ∪ e
17:     return M_new
```

Algorithm 2 demonstrates how to apply the crossover operator. The algorithm receives two parent matchings, $\mathcal{M}_1$ and $\mathcal{M}_2$, as well as a bipartition (P_0, P_1) of the hypergraph. The partition is generated using the *Karlsruhe Hypergraph Partitioner* (KaHyPar) [70].
Using this partition, the edges of a matching can be divided into three distinct parts. The categorization of edges for matching $\mathcal{M}_1$ is as follows:

- $\mathcal{M}_{1,0}$: Contains all edges whose vertices are fully contained in P_0.
- $\mathcal{M}_{1,1}$: Contains all edges whose vertices are fully contained in P_1.
- $\mathcal{M}_{1,cut}$: Contains all edges that are part of the cut set according to the given partition.

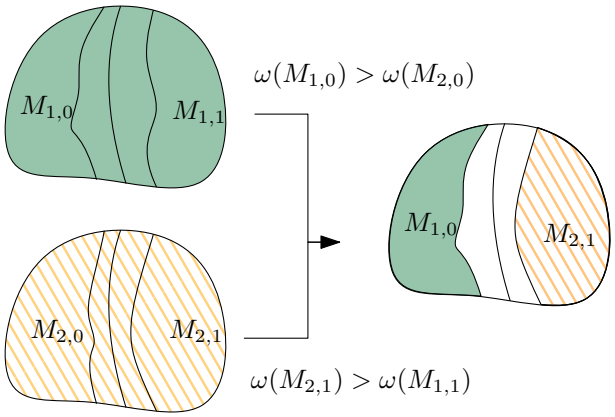


Figure 4.4: Example crossover operation. $M_{1,0}$ and $M_{2,1}$ have higher weight than their counterpart and are therefore selected.

This subdivision of edge sets is then repeated for the second matching, $\mathcal{M}_2$. The new matching is constructed by determining, for each partition, which edge set has the higher weight and adding the subsets with higher weight to the new matching.
This combination can be performed safely, as established by Theorem 1. However, it is highly improbable that this new matching is maximal. To address this, a set of edges to explore is formed by combining the cut-edge sets. This set is then traversed, and edges are added to the new matching in a greedy fashion.

Figure 4.4 visualizes this process. On the left, the subdivisions of the matchings are shown. For the green matching, the area between $\mathcal{M}_{1,0}$ and $\mathcal{M}_{1,1}$ represents the cut. The middle line inside the cut indicates the partitioning boundary. It is important to note that while the partition remains stable, the cut may differ between the two matchings, as seen when inspecting the orange matching.
The figure operates under the assumption that the weight of $\mathcal{M}_{1,0}$ is greater than that of $\mathcal{M}_{2,0}$, while $\mathcal{M}_{2,1}$ is heavier than $\mathcal{M}_{1,1}$. Consequently, the new matching is created by combining $\mathcal{M}_{1,0}$ and $\mathcal{M}_{2,1}$ to produce the offspring, which is depicted on the right side of the figure. This offspring is then further improved by maximizing it.
The crossover operator contrasts with local improvements that occur during initialization and the improvement step through repeated execution of the local search. While these improvements are inherently local, the crossover operator integrates high-quality local decisions on a global scale.
Additionally, it introduces a slight perturbation in the cut area, which may help escape certain local optima. However, it is possible for the selected subsets to originate from the same matching, in which case no global improvement is achieved.

Diversity Crossover Step. There is a certain risk that the crossover step reduces the population's overall diversity. To illustrate this point, consider the following example. Assuming that a population contains some matchings that perform very well with respect to

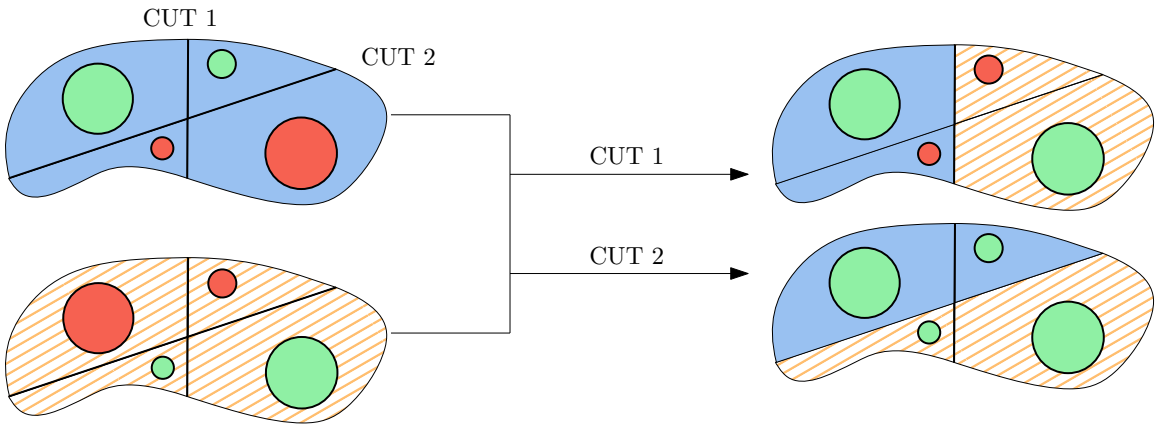


Figure 4.5: Example of crossover process for different partitions

a specific partition P. Specifically, the edge sets $\mathcal{M}_{i,0}$ and $\mathcal{M}_{i,1}$ are heavy, while the cut-edge set $\mathcal{M}_{i,cut}$ is relatively light. These heavy subsets are very likely to be selected during crossover steps. As a result, the next generation is likely to inherit these edge sets. Over multiple generations, all individuals in the population may become increasingly similar to these well-performing individuals, even if they represent only a local optimum. However, this effect can be avoided by using different partitions across generations.

Another advantage of using different partitions is illustrated in Figure 4.5. The red regions within the matchings signify locally suboptimal zones, i.e., areas where no local optimum has been identified. Conversely, the green circles denote locally optimal regions.

It is evident that cut 2 combines more optimal regions than cut 1. If cut 1 were used repeatedly, the offspring of the depicted matchings could inherit the same small local suboptimal areas, making them less likely to be improved. However, using different partitions could help identify these weak spots and ultimately remove them from the matching.

Moreover, the figure underscores the fact that different partitions can produce different resulting matchings. This demonstrates that using varied partitions can enhance overall diversity. A more diverse population is more likely to produce a high-quality final solution [23].

4.4.2 Improvement

The *improvement* step is aimed at improving the quality of newly created offspring. This is achieved through two mechanisms. First, *mutation* randomly alters individuals in the hope of producing a beneficial mutation. Only a certain percentage of individuals are selected from mutation, as determined by the *mutation probability*. After mutation, all individuals in the population are further refined by local search. The improved matchings then become the new population.

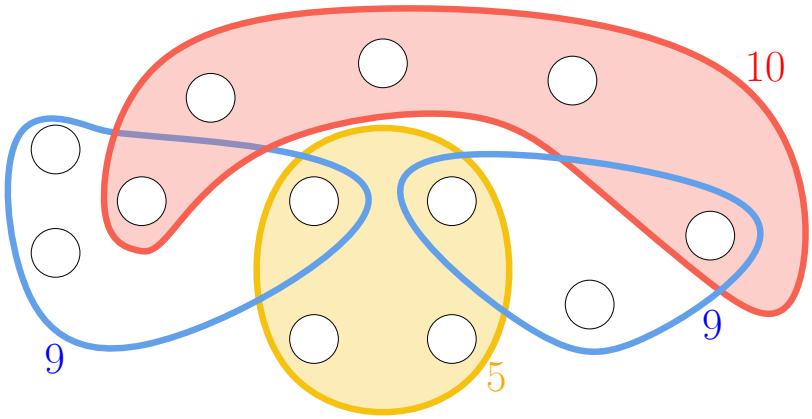


Figure 4.6: Example for an improvement facilitated through the *Edge Removal Mutation*. The red edge is removed allowing for a (1,2)-swap to be executed, hence improving the solution by selecting the two blue edges.

Mutation

Mutation, similar to crossover, is an operator that allows the algorithm to explore new solutions in the search space [22]. This thesis proposes two different approaches to mutation. The first approach removes random edges from the matching. The second approach, similar to the perturbation outlined in Section 4.3.2, forces edges that are currently unmatched into the matching. The number of edges affected by mutation is determined by the *mutation strength*. Given that the mutation step has the potential to diminish population quality, which could prove disadvantageous to the algorithm, an elitist approach is adopted. In this approach, a mutated individual is eliminated if its performance is inferior to that of the original solution. However, akin to the perturbation, there is a slight probability that a less optimal solution is retained. The two mutation approaches are presented below, and their performance is compared, along with the overall impact of the mutation step, in Section 7.3.

Edge Removal Mutation. For a given b-matching , a random subset of the solution's edges is removed, thereby inducing an initial reduction in the solution quality. Subsequent to the edge removal, the local search algorithm is executed. The underlying principle of this approach is that removing some edges relaxes constraints, potentially leading to a better solution by enabling previously blocked (1,2)-swaps.
Figure 4.6 offers a visual representation of the manner in which this process can improve solution quality. Suppose the red and yellow edges are part of the matching, providing a total solution weight of 15. If a (1,2)-swap were executed on either the yellow or red edge, it would identify the two blue edges as a better replacement, since their combined weight is 18. However, the blue edges are blocked by both the yellow and red edges, preventing the swap from occurring. By removing the yellow edge through mutation, the swap is enabled. Subsequently, the yellow edge is then evaluated for (1,2)-swaps, resulting in its

Algorithm 3 Edge Insertion Mutation

```
1:  procedure EdgeInsertionMutation(M, isFiltering)
2:      E_unmatched ← M.blockedEdges()
3:      if isFiltering then
4:          E_unmatched ← removeEdgesWithManyPins(E_unmatched)
5:      E_insert ← randomSubset(E_unmatched)
6:      for e ∈ E_insert do
7:          for n ∈ containedNodes(e) do
8:              if residualCapacity(n) < 1 then
9:                  M ← M.removeEdgesToUnblockNode(n)
10:         M ← M.addToMatching(e)
11:     M ← applyLocalSearch(M)
12:     M ← M.maximize()
13:     return M
```

replacement by the two blue edges. This process culminates in a new total solution weight of 18, an improvement over the initial solution of weight 15.

Edge Insertion Mutation. The second approach is based on the perturbation step of the local search, as presented in Section 4.3.2. The process is outlined in Algorithm 3. The algorithm operates by drawing random edges from the set of currently unmatched edges. Additionally, it can be configured to discard unlikely choices by filtering out edges, that contain a high number of vertices, or to pick heavier edges with a higher probability.
After selecting a random subset of these edges, the algorithm iterates over them, attempting to force each into the matching. As not all edges can be inserted immediately, the algorithm must examine each node in the edge. For each node, it checks whether the residual capacity is at least one. If not, the lightest edge containing that vertex is removed from the matching. Once these removals have been performed, the edge can be inserted. This process is repeated for all selected edges.
After inserting the edges, local search is applied. It is important to note that this step is deliberately executed prior to maximizing the matching, as local search may benefit from the removed constraints, similar to the other mutation approach. Finally, the matching is maximized to further improve the mutation results.

Local Search

As previously mentioned in Section 4.3 the local search algorithm is crucial to the memetic algorithm, as it allows the algorithm to exploit all available knowledge to improve the quality of the population [61]. Local search serves as the mechanism to leverage knowledge of locally optimal solutions, enabling targeted improvements to individuals.
However, it is imperative to exercise caution when applying local search to the population.

An excessive number of iterations can lead to high computational costs. Additionally, it increases the risk of converging to local optima. Therefore, local search should be applied wisely. One should use it enough to achieve significant improvements with each iteration but not excessively, as to avoid the previously mentioned drawbacks.

4.4.3 Competition

Another important factor in the performance of a genetic algorithm is selecting a suitable population for the next generation. This selection process again requires balancing diversity and solution quality [62].
The next generation can be selected from both the previous population and the newly created and improved individuals. Various selection strategies exist, and a salient feature of these strategies is whether they are *elitist* or not. An elitist strategy ensures that the fittest individuals are always selected for the next generation, allowing the algorithm to further refine them and exploit their advantageous traits. However, this can reduce population diversity and increase the risk of getting trapped in a local optimum [44].
This thesis presents two selection strategies: an elitist approach called the *plus selection strategy* and a non-elitist approach called the *comma selection strategy*. Both strategies operate on an existing population of size μ and a newly generated population of size λ. The details of these approaches are discussed in the following, and their performance is evaluated in Section 7.3

Plus Selection Strategy $(\mu + \lambda)$**.** The *plus selection strategy* is an elitist approach to selection. In this method, individuals from both the new and the old population compete directly. The top individuals are selected from the combined pool of size $\mu + \lambda$. This ensures that the fittest individuals from the previous generation can be preserved in the next generation without being altered by crossover or improvement steps. However, as previously discussed, this approach increases the risk of getting trapped in a local optimum due to being more likely to reduce the population's diversity.

Comma Selection Strategy (μ, λ)**.** The *comma selection strategy* takes a more explorative approach to selection. Here, the old population is entirely discarded, and solely the top individuals from the newly generated population of size λ are selected. This strategy enables the algorithm to circumvent local optima by permitting the selection of some lower-quality individuals, thereby fostering diversity and facilitating additional exploration.
Although the best individuals from the previous generation are not explicitly retained, their advantageous characteristics are likely to persist in the new population, albeit slightly altered through crossover and improvement steps.

4.5 Stopping Criterion

This thesis puts forth two distinct stopping criteria. The first is a rudimentary *time-based* criterion, which terminates the algorithm after a set duration. The second approach aims to stop executions that are unlikely to yield further improvements, thereby avoiding the computation of the *long tail* of the solution quality. The long tail refers to a phenomenon, where improvements become increasingly smaller over time, leading to diminishing returns despite continued computational effort. To determine when the long tail begins, the algorithm observes how the solution quality changes over the course of multiple generations. Once one of the stopping criteria is met, the best solution found during the execution is returned.

CHAPTER 5

Parallelization

The subsequent chapter describes the parallelization of the algorithm. Section 5.1 explains how different components of the algorithm can be parallelized. Section 5.2 explores strategies for enhancing utilization through non-blocking communication. Finally, Section 5.3 combines these approaches, introducing different configurations referred to as parallelization levels.

5.1 Parallel components

The ensuing sections will delineate the ways in which the various components of the algorithm can be made to execute in a parallel manner.

Parallel Population Initialization

The different individuals of the initial population are independent of one another. Consequently, the computation of the initial population is *embarrassingly parallel*. Each of the n PEs computes $\frac{N}{n}$ individuals and stores them in a local population.

Parallel Cooperation

The cooperation step is the first component that can be parallelized in different ways. The simplest approach is to have each PE perform crossover operations exclusively within its local population. This configuration is referred to as *distributed cooperation*.
Thus, cooperation is executed independently for each local population rather than treating the population as a globally shared resource. This method requires no communication between different PEs and is therefore *embarrassingly parallel*. However, it also prevents the exchange of solutions between PEs. This hinders the optimization process, as it limits the exchange of solutions among PEs, leading to a scenario where the PEs essentially execute

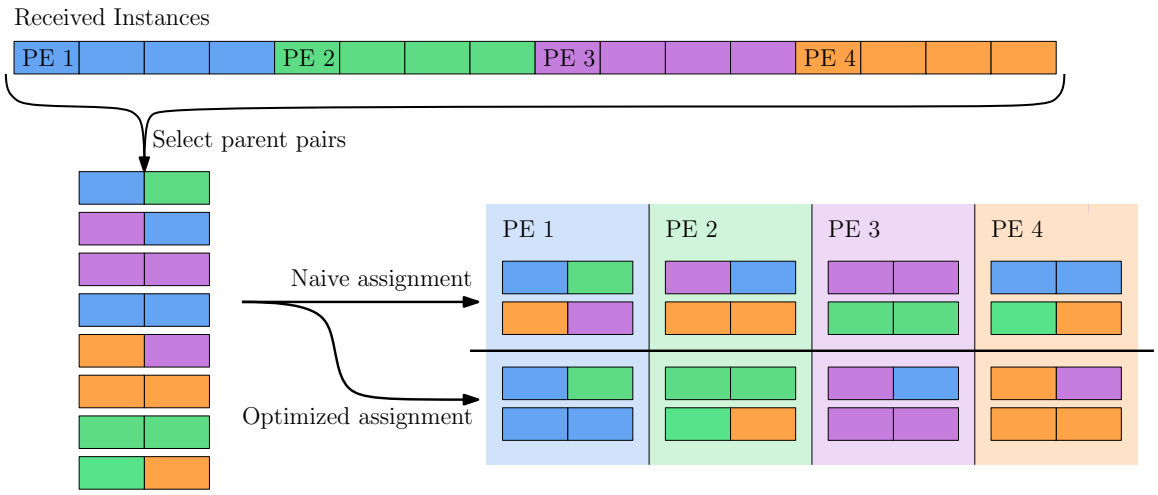


Figure 5.1: Example assignment for the naive and optimized parallel cooperation. The top indicates the current location of the individuals. On the left, these individuals were combined to create the parent pairs. On the right, potential assignments for the naive and the optimized approach can be compared.

the algorithm independently for specific subsets of the population.
The second approach, termed *shared cooperation*, conceptualizes the population as a globally shared resource, allowing the cooperation step to be computed collectively. Initially, each PE calculates the fitness values for its local population. Subsequently, these values are collected by a designated PE, which selects parent pairs and distributes them among the PEs for computation. The assignment process is designed to balance the load, ensuring that each PE computes approximately the same number of pairs.
A notable challenge in this approach pertains to the communication overhead. Each PE is responsible for receiving all unknown individuals from other PEs that are part of the assigned parent pairs. If parent pairs are assigned randomly, this communication can quickly become a bottleneck. To address this issue, an optimized assignment strategy is introduced. This strategy introduces a second constraint: a PE should only receive a parent pair only if its local population already contains at least one of the individuals in the pair. This constraint may be broken if enforcing it would lead to significant workload imbalance.
This optimization is expected to significantly reduce communication costs, as illustrated in Figure 5.1. At the top of the figure, a set of individuals representing the best solutions from different PEs is shown. A parent selection function then produces pairs, as depicted on the left. The different assignment strategies determine how many individuals need to be exchanged. The naive approach necessitates the exchange of 12 individuals, whereas the optimized approach reduces this number to only 4.

Parallel Improvement

The improvement step is once again *embarrassingly parallel*. Each PE can independently compute the improvement step for its local population. However, it is crucial to ensure that each PE processes a similar number of individuals, especially if the algorithm synchronizes at any point. Otherwise, PEs with fewer computations would be forced to wait for those with heavier workloads, leading to more idle time and reduced efficiency.
This issue is exemplified by the assignment of parent pairs in the shared cooperation approach. In the event that the workload distribution is not balanced, some PEs may be inclined to commence cooperation at an earlier time than their counterparts. Consequently, these PEs would be required to wait for the other PEs to reach the same execution point.

Parallel Competition

The competition step can be handled similarly to the cooperation step. The naive approach, referred to as *distributed competition*, treats the population as a local resource. This results in an *embarrassingly parallel* approach, as each PE independently performs the competition step. However, this method again prevents any exchange of information between the PEs, with the aforementioned drawbacks.
In contrast, the *shared competition* approach views the population as a globally shared resource. Analogous to the cooperation step, one PE is designated to collect the fitness values and decide which individuals will be part of the new population. This decision triggers the other PEs to exchange individuals to construct the updated population.
A naive implementation of this approach would randomly assign the new individuals to the PEs, but this is likely to cause significant communication overhead. To mitigate this, a second constraint is again introduced aiming to keep individuals on their original PE whenever possible. This optimization reduces the amount of data exchanged and helps maintain computational balance, similar to the cooperation step.

Parallel stopping criterion

Each PE can independently check whether the stopping criterion is met. If the criterion is satisfied, the memetic process is terminated for that PE.

5.2 Non-blocking approaches

The idea behind non-blocking communication is to enable PEs to continue computations while awaiting the conclusion of the data exchange [7]. When implemented correctly, this approach allows PEs to fully utilize their computational resources. This becomes particularly important for large problem sizes, where communication overhead can substantially impact performance.

In the following, two different locations for integrating non-blocking communication are explored.

Non-blocking parallel competition. The *shared competition* approach is examined first, as it is the simpler of the two options. Each PE computes the fitness values for its local population, which are then collected. After receiving information about which individuals to retain, the PE sends and receives individuals as is needed.
Consider the following example: there are three PEs. Assume further that PE 1 needs to send two individuals to PE 2 and PE 3, respectively. In this scenario, either PE 2 or PE 3 would need to wait for the completion of the other communication process of the other PE. As they are still waiting for individuals required for the new population, they are unable to already begin computing the next generation. Instead of being idle, a PE can instead perform additional iterations of the local search algorithm on some randomly chosen individuals from the next generation, which it already possesses.
While this approach reduces idle time, it does not fully utilize computational resources, as the PE now executes local search computations that may not be necessary. However, this is still preferable to remaining idle, as long as it is not excessively applied, which could lead to excessive exploitation through additional local search operations.

Non-blocking parallel cooperation. Once more, the population is regarded as a globally accessible resource. Subsequent to the collection of all fitness values by one PE, the determination of pairs is made, and their assignment to the processors is carried out as described above. The state of all PEs is now as follows: they have some individuals to send and receive, and they have some work to compute.
Algorithm 4 shows how a PE handles this process. It sets up two queues: one for the pairs that are ready to be crossed over and one for the individuals that have already undergone the crossover. Additionally, it creates a new population in which the new individuals are stored. The algorithm then enters a loop that is exited once all communications have been completed. In each iteration, it checks if there are any pairs ready to be crossed over. If so, it performs the crossover and adds the new individual to the queue containing the crossed-over individuals. Conversely, if no pair is available, it next checks if there are any new individuals available. If so, that individual is improved and added to the new population. If neither of these steps can be performed, it randomly selects an individual and applies additional local search to it. Subsequent to the completion of all communications, the algorithm clears out the queues through executing the remaining crossover and improvement steps.
The advantages of this approach are twofold. Firstly there is significantly more work to be performed while waiting for the communication to finish. Secondly, the crossover and the improvement steps are required to be performed regardless of the communication status.
When using the *distributed competition* approach, the algorithm can be further optimized by checking in the while loop whether the competition step can be computed and, if so,

Algorithm 4 Non-blocking parallel crossover

```
1:  procedure CROSSOVERNONBLOCKING(parentPairsRelevant [ ], P)
2:      P_new ← [ ]
3:      readyPairsQueue ← [ ]
4:      crossedOverQueue ← [ ]
5:      while not allCommunicationsFinished( ) do
6:          readyPairsQueue ← readyPairsQueue ∪ determinePairsReady( )
7:          if readyPairsQueue not empty then
8:              newIndividual ← crossover(readyPairsQueue.pop())
9:              crossedOverQueue.insert(newIndividual)
10:             continue
11:         if crossedOverQueue not empty then
12:             improvedInidvidual ← improve(crossedOverQueue.pop())
13:             P_new.insert(improvedInidvidual)
14:             continue
15:         if P_new not empty then
16:             randomIndividual ← removeRandomIndividual(P_new)
17:             improvedIndividual ← localSearch(randomIndividual)
18:             P_new.insert(improvedIndividual)
19:
20:     for pair ∈ readyPairsQueue do
21:         crossedOverQueue.insert(crossover(pair))
22:     for individual ∈ crossedOverQueue do
23:         P_new.insert(improve(individual))
24:     return P_new
```

computing it directly. This would allow the next generation to start immediately upon completion of the communication.

5.3 Parallelization levels

In general, the parallelization levels are situated in between the following extremes. On the one side, the different PEs act independently of each other. Each of them works on their own local population. On the other side, all PEs work on the same population and for each of the steps the population is understood as a globally shared resource.
The Table 5.1 provides an overview of the different parallelization levels. They are ordered by the amount of communication required. The shared approaches to the cooperation and competition step are both assumed to use optimized assignments and additionally to use non-blocking approaches as outlined previously. These levels impact the communication cost and the amount of cooperation and Section 7.3 will explore the impact of these levels

on the performance of the algorithm.
Additionally, it should be noted, that it is possible to reduce the amount of communication by using the following idea.

T-Sync approach. The T-Sync approach arises from the idea of finding a middle ground between the distributed and shared approaches. For the improvement method, i.e. , one could use the distributed approach for $T - 1$ iterations in a row and then switch to the shared approach for the T-th iteration. This approach significantly reduces communication costs.
There are, however, two key considerations. As T approaches the total number of generations that the algorithm computes within a given timeout, this approach increasingly resembles the distributed approach. Additionally, for large values of T, the probability increases that the different local populations will diverge in solution quality, which may necessitate greater communication either during the improvement or the competition step.
Nonetheless, this approach can provide a well-balanced compromise, as will be further demonstrated in Section 7.3.

Level	Cooperation	Competition	Advantages	Disadvantages
Distributed Level	Distributed cooperation	Distributed competition	No synchronization, no communication	No exchange between PEs
T-Sync Level	Distributed cooperation T-Sync cooperation	T-Sync competition Distributed competition	Very little communication but still allows for cooperation between PEs	Requires synchronization every T generations
Partially shared Level	Distributed cooperation Shared cooperation	Shard competition Distributed competition	Medium amount of communication	Requires synchronization every iteration
Shared Level	Shared cooperation	Shared competition	Allows for steady cooperation population can be understood globally	Requires a lot of communication

Table 5.1: Overview over different levels of parallelization applicable to the algorithm. Sorted by amount of communication required from top (no communication) to bottom (most communication)

CHAPTER 6

Datastructures

This chapter presents efficient data structures for the algorithm introduced in Chapter 4. Two different data structures for storing a hypergraph b-matching are introduced. The first is a modifiable version, originally presented by Großmann et al. [38], designed to facilitate the efficient computation and storage of reductions. Since these reductions can alter the underlying hypergraph, as shown in Section 4.2, this data structure allows modifications to be applied to the stored hypergraph accordingly.
The second approach is a reduced version of the first data structure, which does not permit modifications to the hypergraph. This version is conceptually similar to the data structure proposed by Schlag et al. [70] as part of the *Kahypar* project, which also served as inspiration for the modifiable b-matching .
The following sections introduce these data structures in detail and describe their respective capabilities. Finally, a brief overview is provided of how the data evolves throughout the execution of the algorithm.

6.1 Reduced b-Matching

The process of computing a b-matching requires storing both the hypergraph and the matching. The hypergraph can be represented as a list of vertices, along with a list of lists, where each inner list contains the vertices associated with a specific hyperedge.
Schlag et al. [70] proposed that, for a fixed hypergraph, these lists can be combined into a single structure by storing only the start and endpoint for each edge. This approach reduces the number of cache misses, improving performance.
The b-matching itself is stored as proposed by Großmann et al. [38], where all edges are initially stored in a single list. The edges are bundled by their current state, which is either matched, free, or blocked. The different sections are differentiated by storing the start and endpoint for each of the states. At the beginning, all edges are considered free. When an edge is moved to the solution, it is swapped with the first free edge, increasing

the matched size while decreasing the free size. The start and endpoints are updated accordingly. Similarly, when an edge becomes blocked, it is swapped with the last free edge, reducing the free size. Additionally, Großmann et al. [38] proposed storing the number of edges blocking a given edge, as well as the matched and blocked edges for each vertex. These enhancements facilitate efficient computations in the local search described in Section 4.3.2 and the mutation process from Section 4.4.2.
A population contains multiple such reduced b-matchings. Since the corresponding hypergraph remains fixed, it can be shared among different individuals, eliminating the need for redundant storage.

6.2 Modifiable b-Matching

The reductions introduced in Section 4.2 can modify the hypergraph. Vertices may need to be removed, while edges may be merged, removed, or added. To support these modifications, a dynamic data structure is required. Großmann et al. [38] proposed a hypergraph data structure that enables efficient modifications.
An edge is removed by setting its size to a negative value. When two edges are combined, one edge is deactivated, and their vertex lists are merged.
These changes remain reversible, as a history of modifications is maintained.

6.3 Data Progression

As outlined in Chapter 4, the algorithm begins by computing the reductions once using the modifiable data structure. The resulting reduced graph is then converted into the reduced data structure, which serves as the working representation for the genetic algorithm. After the genetic algorithm completes, the solution is translated back to the modifiable data structure, and the reductions are reverted. The final solution is then returned.
The progression is illustrated in Figure 6.1. After loading the given problem into a modifiable b-matching structure, the reductions are applied. These reductions remove or transform the yellow areas, store the green areas as part of the solution, and leave the red areas undecided. The reduced data structure is then constructed based only on the red areas. Once the memetic algorithm computes an approximate solution for the reduced problem, this solution is integrated back into the modifiable data structure. Finally, the yellow area removed by the reductions is reconstructed, and the final solution is obtained.

Translation to reduced data structures. After computing the reductions, the modifiable b-matching is converted into a reduced b-matching. This process involves copying all vertices along with their remaining free capacities. Additionally, all still-free edges are included in the reduced b-matching. Some of these edges may represent the combination of multiple edges from the original problem formulation.

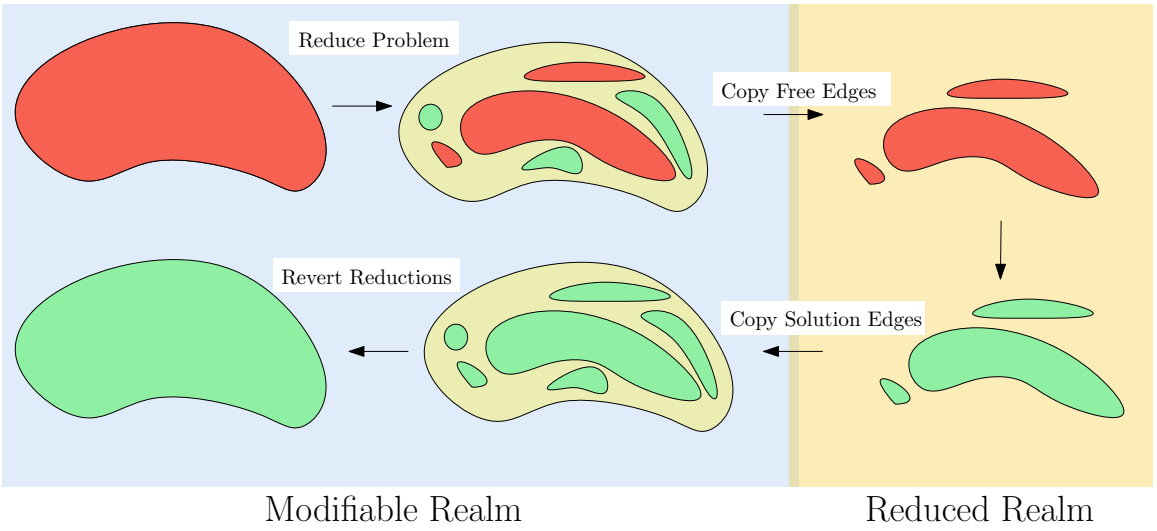


Figure 6.1: Progression of a b-matching through the data-structures during the algorithm's execution. The blue area represents the modifiable b-matchings, while the yellow one represents the reduced data-structure.

Solution edges from the reductions are always part of the final solution and therefore do not need to be considered, meaning they are not copied. Similarly, edges deactivated by the reductions are also excluded.

Translation from reduced data structures. Once the genetic algorithm completes, the solution is translated back to the original problem formulation. The matched edges from the reduced b-matching are copied into the modifiable b-matching, and the combined solution is returned.

CHAPTER 7

Experimental Evaluation

In the initial section of this chapter, the methodology is outlined. This is followed by an exposition of the instances used in the experiments. Next, the performed experiments and their results are presented. The chapter concludes with a discussion of the results.

7.1 Methodology

This section first outlines the implementation of the algorithm presented in Chapters 4-6 and introduces its key configurations. It then describes the experimental setup and execution. Next, the competitor algorithm is presented, along with details of its execution. Finally, the evaluation metrics used in the analysis of the experiments are introduced.

Implementation Details

The code is written in C++ and compiled using `mpic++` version 12.3.0 with the compiler flags `-O3 -march=native`.

External Libraries. The modifiable graph structure from Großmann et al. [38] is incorporated, along with their reductions and local search algorithms. Additionally, the *KaHyPar* partitioning library, introduced by Schlag et al. [70], is employed to partition the hypergraphs as required.

Configurations. Several algorithmic components introduced in Chapter 4 have been implemented and can be configured for use in the final algorithm. Similarly, the parallelization approaches presented in Chapter 5 are also implemented and accessible through configuration. The selection of components, a parallelization approach, and hyperparameters defines a final configuration, of which two are highlighted below.

The configuration *fHeiMemBMatch* is optimized for speed, while *qHeiMemBMatch* is optimized for solution quality. For the final evaluation, both configurations were slightly adjusted to accommodate a 3-hour timeout, showcasing the full potential of the algorithm. The results of these evaluations are presented in Section 7.3.

Experimental Execution

The experiments were conducted on a single node of the BwUniCluster2.0, equipped with an *Intel Xeon Gold 6230* CPU featuring 20 physical cores, a 27.5 MB cache, and a clock speed of 2.1 GHz. The system has 96 GB of RAM and runs RHEL 8.4.
In general, experiments were executed with a timeout of 1 hour using the corresponding configurations. Additionally, some runs with a 3-hour timeout and the adapted configurations were performed to highlight the parallelization effects more clearly.
Each execution processed five instances simultaneously, with each instance allocating four processing elements (PEs), fully utilizing the 20 available cores. The experiments were repeated three times, and the results were averaged to mitigate random noise.

Competitors

The algorithm presented by Großmann et al. [38] is used as a baseline for comparison. Since this algorithm is also utilized within *HeiMemBMatch* to compute the initial solution, it is used with a much shorter local search. The comparison uses a local search only limited by the timeout it will be used for the final comparison. We use the reductions, then the `S,pin` greedy matching function and as many local search iterations as possible within the timeout. Following the naming convention by Großmann we call this config `Reductions+S,pin+ILS3600sec`, where the local search runs for up to 3600 seconds.
For the convergence experiments we compare against repeated execution of the `Reductions+S,pin+ILS20` algorithm, which uses 20 iterations of the local search algorithm. As this configuration is used to compute the individual solutions in the population. This comparison showcases how calculating more individual solutions compares to the memetic process.

Evaluation Metrics

The key evaluation metrics used to compare different configurations and assess the resulting performance are presented below.

Convergence. Analyzing the convergence of the algorithm is essential for understanding its behavior over time. To evaluate convergence across repeated runs and different instances, the analysis follows the approach described by Andre et al. [5].

Consider a single instance I. Whenever the algorithm discovers an improved solution, it records a pair (ω, t), where ω represents the new solution weight and t denotes the time at which the solution was found. Given a specific random seed s, this sequence is referred to as $T_{I,s}$.
Sequences from different seeds are combined by iteratively computing an average. Initially, the first pair from each seed is added to a pool, where the average quality and time are computed, forming the first value of the combined sequence. Subsequently, result pairs from all seeds are processed in chronological order. For each new pair, the existing pair with the same seed in the pool is replaced. The average is then recomputed, and the updated pair is added to the combined sequence. This process generates a single sequence that describes the average behavior of the algorithm for instance I.
To normalize solution times within this sequence, each time value is divided by t_{init}, which represents the time required to compute the initial solution for instance I using the comparison algorithm. The normalized timestamp is denoted as t_n, and the resulting sequence is referred to as $\overline{T}_{I,t_n}$.

To generalize across different instances, a similar process is applied to combine sequences, with the key difference that geometric averaging is used to mitigate the disproportionate influence of large instances. Finally, the resulting sequence is normalized by dividing each value by the first recorded average weight.

Speedup and Scalability. To assess the benefits of algorithm parallelization, speedup and efficiency are analyzed. Assume that the algorithm is executed on a set of instances with different numbers of PEs. However, due to the use of a time-based stopping criterion, conventional speedup and efficiency metrics must be adapted.
Speedup is typically defined as $S(p) = \frac{T(1)}{T(p)}$, where $T(p)$ is the runtime with p PEs, and efficiency as $E(p) = \frac{S(p)}{p}$. Since termination may only occur after a fixed duration, an alternative approach compares running times for specific solution qualities.
For a single instance I, the initial and the best results from each execution with k PEs are extracted, referred to as $\min_k$ and $\max_k$. A solution window $[\text{low}, \text{high}]_I$ is defined as $\text{low} = \max(\min_k)$ and $\text{high} = \min(\max_k)$. Thresholds within this window are then computed. For example, $th_5 = \text{low} + 0.05 \cdot (\text{high} - \text{low})$ represents 5% of the window. The speedup for this threshold is given by

$$S_{th_5,I} = \frac{T_{th_5,I}(1)}{T_{th_5,I}(p)}$$

where $T_{th_5,I}(p)$ is the time required for p PEs to reach the threshold on instance I. Efficiency is then computed as

$$E_{th_5} = \frac{S_{th_5}}{p}.$$

Finally, speedups and efficiencies are computed for all instances, and their geometric average is taken.

Diversity. To evaluate the diversity of different b-matchings and partitions, the Jaccard distance introduced in Section 4.3.3 is used. This measure allows for assessing how diverse the different matchings or partitions are.

7.2 Problem instances

This thesis utilizes the well-established test set presented by Heuer and Schlag [41], which combines hypergraphs from three different sources: the ISPD98 VLSI Circuit Benchmark Suite [2], the International SAT Competition 2014 [10], and the University of Florida Sparse Matrix Collection [21]. The edge weights are provided by Großmann et al. [38].
A random sample of 10 instances is selected as the tuning set and used to generate different configurations. The final performance evaluation is conducted on the entire test set. Additionally, a random subset of 70 instances is selected from the smaller half of the benchmark set, where an optimized configuration tailored for smaller graphs is applied.
The instances provide no explicit values for the b-function. Hence, the tests were performed for so-called capacities c where $b(v) \equiv c$ for all vertices v. Additionally, some tests were executed, where the provided weights of a node were used as the values for the b-function.

7.3 Experiments

The conducted experiments can be divided into three groups. The first group focuses on tuning the algorithm's configurations. The second group evaluates the parallelization of the algorithm. The final group assesses the performance of the final configurations.

Configuration Tuning Experiments

The experiments aim to either identify the best-performing components or determine the most suitable hyperparameters for the algorithm. The following experiments do not provide an exhaustive list of all configurations tested but instead highlight the most significant decisions made during the tuning process.

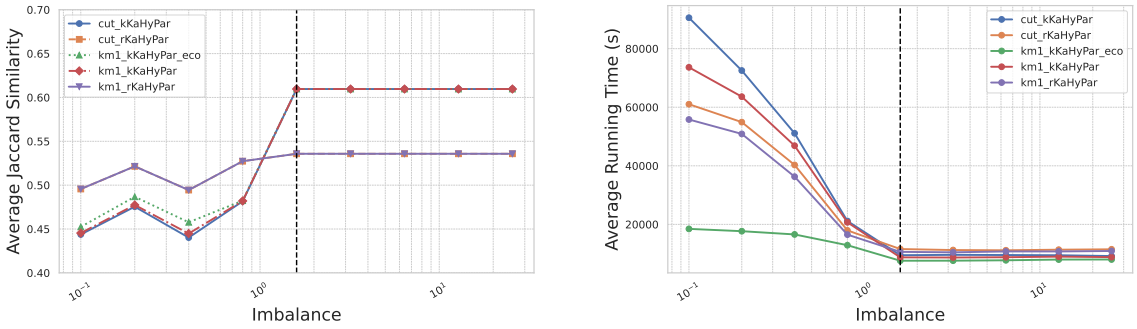
Diversification Methods. As discussed in Section 4.3.3, population diversification is crucial to the success of the algorithm. Therefore, various approaches to enhancing the diversity of the initial population were introduced as part of the greedy matching step, outlined in Section 4.3.1.
To compare these methods, a population of size 25 was generated for each instance, and the average Jaccard distance between all pairs from the population was computed. The average Jaccard distances for multiple runs was again averaged. A Jaccard distance of 1 indicates that two matchings share no edges, while a distance of 0 means they are identical. The loss

Method	Parameter	Loss	average Jaccard distance
TieBreak	-	**0.028630**	0.165868
Gaussian	stddev=1.0	**1.213405**	0.460213
Gaussian	stddev=2.0	2.230221	0.525477
Gaussian	stddev=3.0	3.933404	0.591026
Pushback	1 positions	**14.782910**	0.600157
Pushback	2 positions	13.906353	0.628862
Pushback	4 positions	13.396370	**0.653669**

Table 7.1: Average solution quality loss and Jaccard distance for the different diversification methods.

in solution quality was also recorded as the difference between a not-diversified individual and the population.
Table 7.1 presents the results of the diversification experiment. It can be observed that the tiebreak method results in only a 0.03% loss in solution quality, with some instances even showing improvements. However, its Jaccard distances are the lowest compared to the other methods.
In the *Pushback* method, an edge was pushed back with a 10% probability by an average of 1, 2, or 4 positions, respectively. Even at this low rate, solution quality degraded by up to 14.78%, making this method unsuitable, although it provided high distances between individual solutions.
The Gaussian noise method, with a standard deviation of 1.0, resulted in only a 1.21% loss in solution quality. Its Jaccard distances are significantly higher than those of the tiebreak method and only slightly lower than those of the pushback method. Therefore, the Gaussian noise method was selected to diversify the population.

Partitioning Configurations. As shown in Section 4.4.1, the crossover method benefits from utilizing different partitions. To explore this effect, various partitioning configurations and settings for Kahypar were tested against each other. To facilitate comparison 25 partitions were generated for each configuration. Again the averages over multiple runs were computed. Exponentially increasing imbalances, ranging from 0.1% to 25.6%, were used in the experiment, as well as all configurations provided by Gottesbüren et al. [35]
The primary objectives for partitioning were twofold: ensuring that partitions were as distinct as possible while minimizing computational time.
Figure 7.1a presents the Jaccard distances for different partitioning configurations. Across varying imbalance values, all configurations reach a plateau at approximately 1.6% imbalance, beyond which the distances no longer notably increase. The configurations split into 2 groups, with the better ones reaching about a 0.61 Jaccard distance. Similarly, the plot in Figure 7.1b illustrates the average running time in seconds for each configuration, showing



(a) Average Jaccard distance between partitions for different configurations with increasing imbalance.

(b) Average running time required to partition the graphs for different partitioning configurations and imbalances.

Figure 7.1: Comparison of partitioning diversity and running times across different configurations and imbalances.

Mutation Rate	Edge Insertion Mutation	Edge Insertion Mutation (Weighted)	Edge Removal Mutation
1.00%	**0.00%**	0.02%	**7.81%**
0.10%	0.03%	0.78%	9.05%
0.01%	0.67%	5.25%	12.93%
0.001%	6.54%	9.18%	**20.89%**

Table 7.2: Success rates of different mutation methods and rates compared using 2 local search iterations.

that running time stabilizes beyond 1.6% imbalance. Here all configurations show similar convergence and tend to similar running times.
In terms of running time, the configuration `km1-kKaHyPar_eco` provides the fastest partitioning. Furthermore, it ties for the highest average Jaccard distance. This configuration was introduced by Gottesbüren et al. [35] and will be used with an imbalance of 1.6% in the final algorithm.

Mutation Methods. The mutation methods, introduced in Section 4.4.2, were evaluated by comparing their success rates, to determine the most effective approach. A mutation is considered successful, if the solution after mutation is better than the solution before mutation. Since the effectiveness of mutation depends on the local search process, this aspect was analyzed in conjunction with the local search operator.
Table 7.2 presents the success rates of different mutation methods and rates. Only a short local search process is applied to avoid biasing the results towards the local search operators effectiveness.
The results indicate, that the *Edge Insertion Mutation* performs the worst, with a success

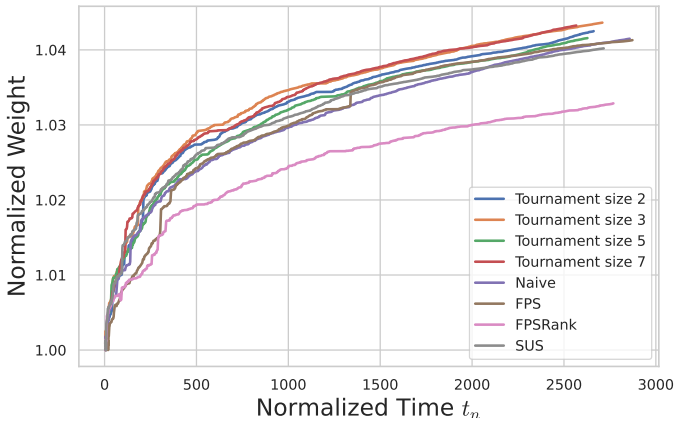


Figure 7.2: Convergence plots for different parent selection methods.

rate of 0% at a mutation rate of 1.00%. Only at a mutation rate of 0.001% does the success rate increase significantly. Likely because, at such small mutation rates, the outcome primarily reflects the quality of the local search algorithm rather than the mutation itself. A slightly modified version, *Edge Insertion Mutation (Weighted)*, performs better by prioritizing edges with higher weights.
The *Edge Removal Mutation* achieves the highest success rate, reaching 7.81% at a mutation rate of 1.00% and increasing further to 20.89% at a rate of 0.001%. The success rate consistently improves as the mutation rate decreases, which is expected, since removing fewer edges reduces the improvements required from the local search operator.
Consequently, the *Edge Removal Mutation* was chosen for the final configurations. However, the fast final configuration (`fHeiMemBMatch`) avoids mutation entirely, as it requires significant computation for a relatively low success rate.
Section 4.4.2 also discusses a variant of the removal mutation that filters out edges containing many vertices. This approach was tested but did not yield significant improvements.

Parent Selection. As discussed in Section 4.4.1, *HeiMemBMatch* supports different parent selection functions. These functions were evaluated by comparing their performance, with the results presented in Figure 7.2.
The results indicate, that most parent selection methods converge to similar solution qualities. Only the *ranked-based fitness proportionate* approach performs significantly worse than the naive baseline. In contrast, tournament selection yields good results for all tournament sizes. Additionally, it offers greater configurability through the tournament size parameter, allowing the selective pressure to be adjusted. This places it as the most suitable choice for the final configuration.

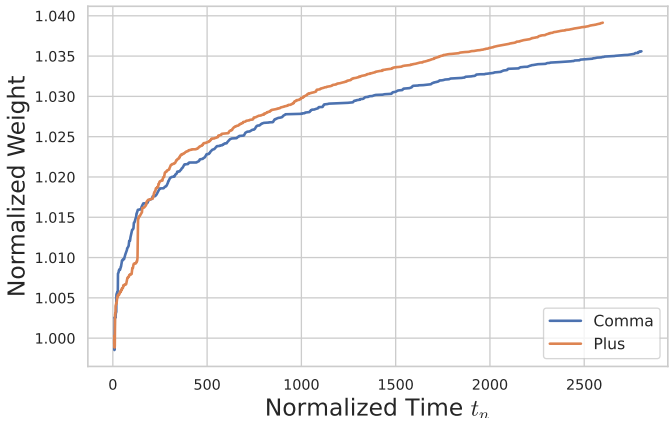


Figure 7.3: Convergence plots for different competition strategies.

Competition Strategies. *HeiMemBMatch* offers two different competition strategies, presented in Section 4.4.3, which were compared against each other. The results, shown in Figure 7.3, indicate that the *plus selection* strategy not only outperforms the *comma selection* strategy but also leads to faster convergence. Therefore, the plus selection strategy is preferred for this dataset.

Parallelization Evaluation

To evaluate the performance of parallelization, two experiments were conducted. The first experiment examines the impact of different parallelization levels on the algorithm's performance. The second experiment tests the scalability of the quality focused configuration.

Parallelization Levels. Different parallelization configurations were created and evaluated based on their performance. The experiments were conducted on the tuning set with a one-hour timeout. The results for 2 PE are presented in Figure 7.4 and for 8 PE in Figure 7.5.
The `distributed` approach used no communication. The `T-sync` approaches invoked the shared competition, introduced in Section 5.1, every T iterations. The `shared` approach used the non-blocking version of shared cooperation, which is detailed in Section 5.2, in every iteration, while the `fully` configuration applied both shared cooperation and competition at each iteration. These configurations represent different parallel levels presented in Section 5.3.
For both PE counts, configurations with higher communication between PEs outperformed those with less interaction. This effect is particularly noticeable for short execution times, where approaches that share individuals more frequently find better solutions more quickly. As execution time increases, these differences become less pronounced. As populations

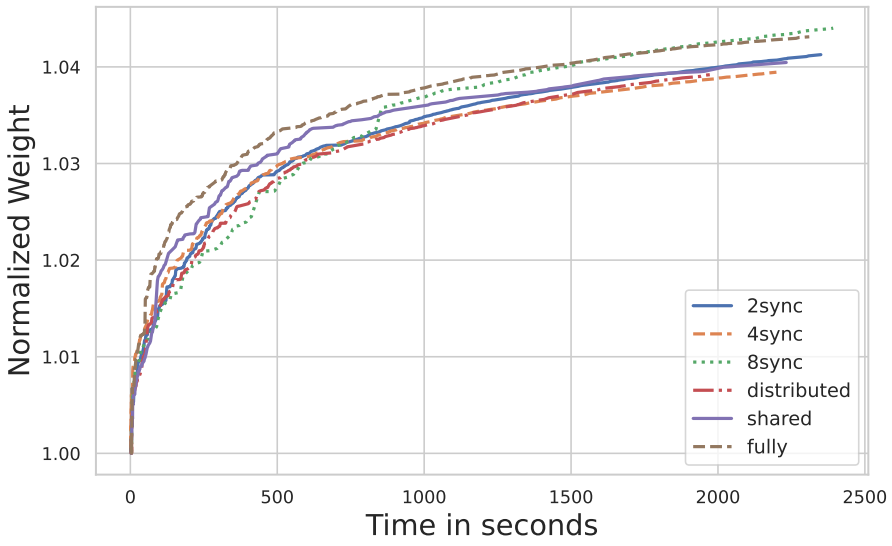


Figure 7.4: Performance comparison between different parallelization levels for 2 PEs.

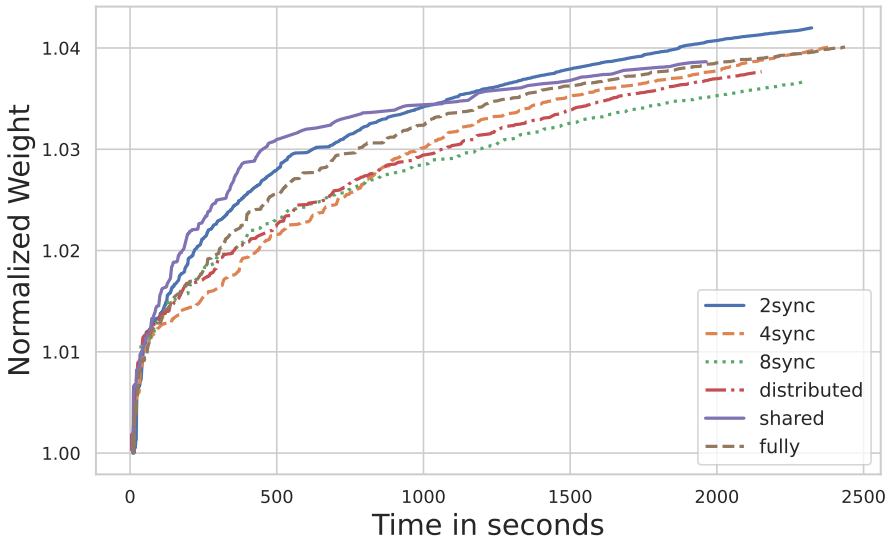


Figure 7.5: Performance comparison between different parallelization levels for 8 PEs.

grow more similar, improvements increasingly rely on local search. In this phase, less collaborative approaches catch up, as they can perform more generations and, consequently, more local search iterations. This effect is increased by the diminishing return behavior, which exhibited by the algorithm as it approaches the optimal solution.

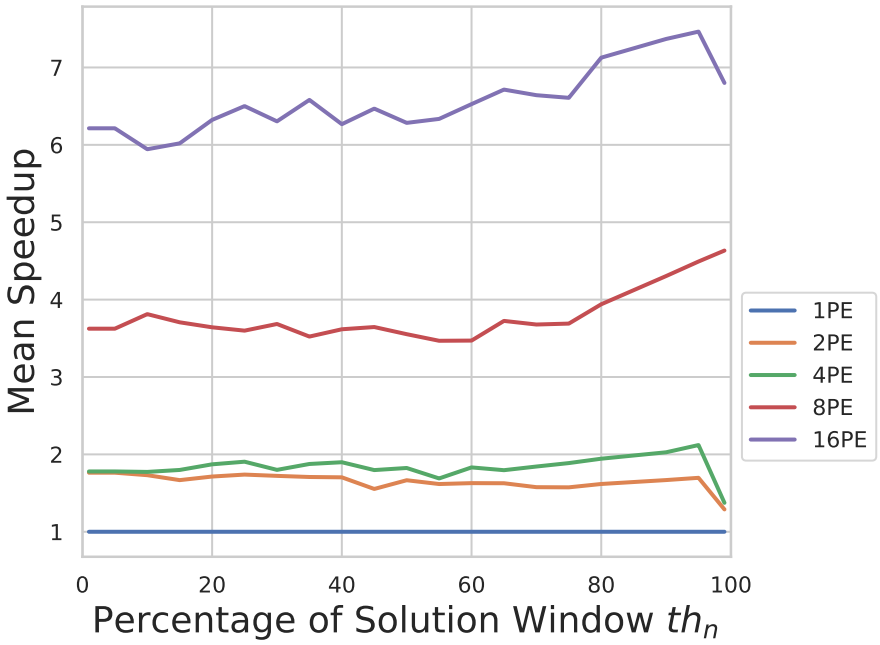


Figure 7.6: Speedup for the quality configuration with 3hout timeout.

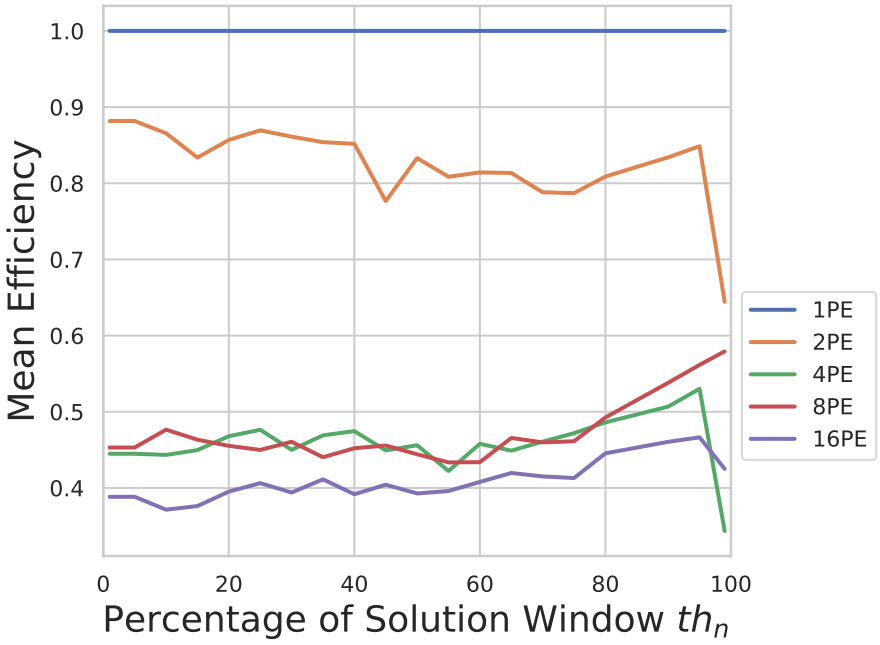


Figure 7.7: Efficiency for the quality configuration with 3hout timeout.

Speedup and Scalability. To assess the impact of parallelization, the speedup and efficiency of the algorithm was evaluated. The standard formulas for speedup and efficiency were extended, as outlined in Section 7.1, to account for the use of a timeout criterion.
The experiments were conducted on the tuning instances with a three-hour timeout to allow solutions to reach convergence. The quality configuration was used, as it produces better results, providing a larger window for measurement. The quality configuration uses the non-blocking shared cooperation every iteration and hence falls into the *shared parallelization level*. Runs were performed using 1, 2, 4, 8, and 16 processing elements (PEs).
Figure 7.6 presents the speedup for the quality configuration, which employs the non-blocking competition approach. A single PE remains stable at a speedup of 1, serving as the baseline. With 2 PEs, the speedup reaches approximately 2. For 4 PEs, it increases slightly to around 2.5. With 8 PEs, it reaches at about 4, and for 16 PEs, it becomes approximately 6.5.
The increase in speedup is not strictly linear with the number of PEs. However, this is expected, as the configured communication between PEs introduces overhead. Additionally, speedup slightly decreases when approaching the upper bound of the solution window, due to the algorithm's diminishing returns behavior. Despite these factors, the observed speedup remains promising, indicating that the algorithm scales well. Increasing the number of PEs consistently leads to clear performance improvements, demonstrating the effectiveness of the parallelization.
Figure 7.7 illustrates the corresponding efficiencies. The highest efficiency is observed for 2 PEs, ranging from approximately 0.8 to 0.9. For higher PE counts, efficiency stabilizes between 0.4 and 0.5. This behavior is expected as the efficiency is best when the PE's do not have to wait for another. An effect, that does not occur for only 2 PEs, as they always want to communicate with another. The stabilization in efficiency for higher PE counts is a good sign, as it indicates that the algorithm scales well and maintains efficiency as the number of PEs increases.

Overall Evaluation

For the final performance evaluation, the *HeiMemBMatch* algorithm is tested using two different configurations. These configurations are derived by tuning various hyperparameters on the tuning set, with some of these experiments described in previous sections.
The first configuration, optimized for speed, is referred to as *fHeiMemBMatch*. The second configuration prioritizes solution quality, potentially yielding better results but requiring more time to converge. This configuration is called *qHeiMemBMatch*. [1]
A key distinction between the two configurations is that *fHeiMemBMatch* follows a more distributed approach. It falls under the T*-sync Level* category, exchanging solutions via shared cooperation only every five iterations. Additionally, the fast configuration does not utilize any mutation methods, sacrificing some diversity in favor of increased efficiency.

[1] A complete list of the chosen hyperparameters for both configurations can be found in the appendix.

Capacity	Graph	Improvement (%)
1	dac2012_superblue14	-10.00 (Worst)
1	Trefethen_20000	1.90 (Best)
3	ESOC.mtx	-7.52 (Worst)
3	Trefethen_20000	0.45 (Best)

Table 7.3: Best and worst results for capacities 1 and 3 when qHeiMemBMatch against Reductions+S,pin+3600sec.

The quality configuration enhances diversity through three distinct mechanisms. First, it generates a more diverse initial population, though at the cost of lower initial solution quality. Second, it incorporates the *Edge Removal Mutation* method to introduce further variation. Third, it employs the *shared cooperation* method, which promotes diversity by facilitating cooperation among a broader range of individuals.
A comparison of the total running times for the convergence experiments outlined in the subsequent section reveals that, on average, the fast configuration reduces runtime by 10% (for capacity 1) and 15% (for capacity 3) compared to the quality configuration with a one-hour timeout for the tuning set. This discrepancy in running times becomes even more pronounced when the timeout is increased to three hours, where `fHeiMemBMatch` achieves a speed improvement of 36% (for capacity 1) and 28% (for capacity 3) over `qHeiMemBMatch`.

General Performance On Complete Test Set. First, the performance on the entire test set is examined. Since computing results for the full dataset is highly resource-intensive, only the quality configuration was executed, with runs repeated twice. Table 7.3 highlights the best and worst instances within the dataset. The average geometric improvement was -0.89% (capacity 1) and -0.31% (capacity 3), showing that the state-of-the-art algorithm performs better. With a worst-case improvement of -10.00% (capacity 1) and -7.52% (capacity 3) and a best-case improvement of 1.90% (capacity 1) and 0.45% (capacity 3). For both capacities, approximately 20% of instances showed improvements. Additionally, about 3% of instances did not produce any results within the timeout, meaning that they failed to compute the initial population and first generation within the allocated time.
The instances that showed improvements had an average size 63% smaller than the overall dataset average. Furthermore, these improving instances exhibited about 25% less improvement when comparing the best individual from the initial population to the final result of the comparison algorithm.
Finally, additional computational resources could further enhance performance. However, due to resource limitations, the experiments were constrained to ensure they could be conducted within a reasonable timeframe.

Configuration	Graph	Improvement (%)
qHeiMemBMatch_Small	sat14_MD5-28-4.cnf.primal	-0.37 (Min)
qHeiMemBMatch_Small	gupta3	5.68 (Max)
fHeiMemBMatch_Small	airfoil_2d.mtx	-0.64 (Min)
fHeiMemBMatch_Small	bbmat.mtx	2.54 (Max)

Table 7.4: Best and worst results when comparing qHeiMemBMatch and fHeiMemBMatch against Reductions+S,pin+3600sec with 1-hour timeouts for capacities 1,2 and 3.

General Performance On Smaller Graphs Test Set. Based on these results, the algorithm was optimized for a test set with a higher likelihood of success. To achieve this, 70 random instances were selected from the smaller half of the benchmark set, and the configurations were further adapted. These optimized versions of both configurations for smaller instances are named `fHeiMemBMatch_Small` and `qHeiMemBMatch_Small`. The primary difference is that these versions allocate significantly more time to initial population generation. This adjustment is feasible because each generation runs considerably faster, requiring less time to compute the initial generations, which contribute the most significant improvements.
For the selected test set, the most notable improvements were observed at capacity 1. A average geometric improvement of 0.37% (fast) and 0.56% (quality) was achieved, improving approximately 64% (fast) and 73% (quality) of the instances.
Table 7.4 presents the best- and worst-performing instances for both configurations. The minimum recorded improvement was -0.37% for the quality configuration and -0.64% for the fast configuration, while the maximum improvement reached 2.75% for `qHeiMemBMatch_Small` and 2.54% for `fHeiMemBMatch_Small`.
However, performance decreases significantly for higher capacities. With capacity 3, the average improvements for three-hour runs drop to 0.27% for the quality configuration and 0.16% for the fast configuration.
Furthermore, when using the provided weights as the b-function values, no notable differences in results were observed, regardless of the configuration. This is likely because the comparison algorithm efficiently reaches near-optimal solutions, resulting in an initial population that is also already close to this solution.

Convergence. The convergence plots for different timeouts and capacities are shown in Figures 7.8 to 7.10. In general, the quality configuration outperforms the fast configuration but requires more time to converge. The repeated executions of the `Reductions+S,pin+ILS20` algorithm stops finding improved solutions significantly earlier than both memetic approaches. This algorithm is utilized for comparison, as it computes the initial population.
The fast configuration exhibits an initial surge in solution quality, which is less pronounced

in the quality configuration. This difference arises because the quality configuration starts with a more diverse initial population, resulting in a lower initial solution quality. However, this diversity enables the algorithm to discover better solutions over time.
The convergence plots further illustrate, that the chosen stopping criteria prevent the memetic algorithms from fully exploring the long tail of the diminishing returns curve, especially for the fast configuration

7.4 Discussion

The memetic algorithm demonstrates encouraging initial results. While it does not yet match the state-of-the-art algorithm on the entire test set, it still exhibits first promising results. In particular, the algorithm shows significant potential for smaller instances. A key factor in achieving good results is balancing the time allocated to population initialization with the time dedicated to memetic steps. In the optimized configuration, a relatively large portion of the available time was devoted to initialization, leading to strong initial solutions. However, this approach cannot be directly applied to larger instances, as they require more time to compute each generation.
Additionally, the memetic framework demands substantial computational resources. As shown in the convergence plots, the memetic algorithm can take over 1,000 times longer than a single execution of `Reductions+Spin+ILS20` with capacity 1.
Beyond time complexity, the algorithm also requires significantly more memory. This is an inherent necessity, as it must maintain multiple individuals in the population, with memory usage peaking just before the competition step. Reducing these requirements would enable the algorithm to process larger populations sizes, which has shown potential for greater improvements.
Despite these drawbacks, the memetic framework remains highly adaptable and tunable for different instance types. This adaptability is evident when comparing the general performance to the optimized results. Key factors contributing to this flexibility include the adjustable interplay between initialization and the memetic process, as well as the ability to control diversity and selective pressure, such as through tournament size or population size adjustments.
Parallelization proves to be an effective approach, particularly for larger population sizes, as demonstrated in the speedup and efficiency analysis. Speedup remains nearly linear, and efficiency remains relatively stable as the number of processing elements (PEs) increases. However, for longer execution times, the benefits of increased inter-PE communication diminish. Initially, higher communication between PEs enhances performance, but as generations progress, solution diversity decreases, reducing the advantages of further communication.

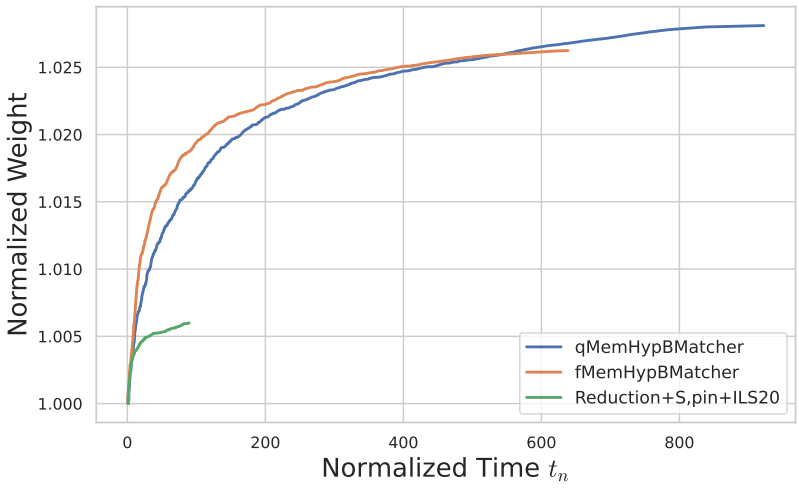


Figure 7.8: Capacity 1 (1 hour)

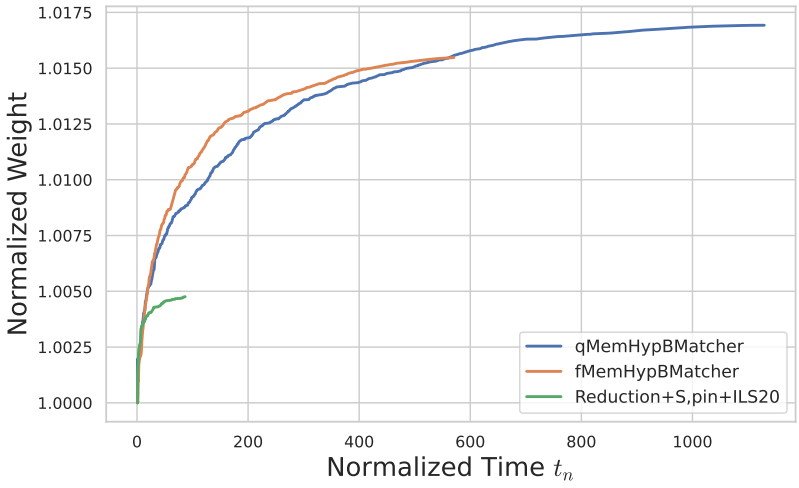


Figure 7.9: Capacity 2 (1 hour)

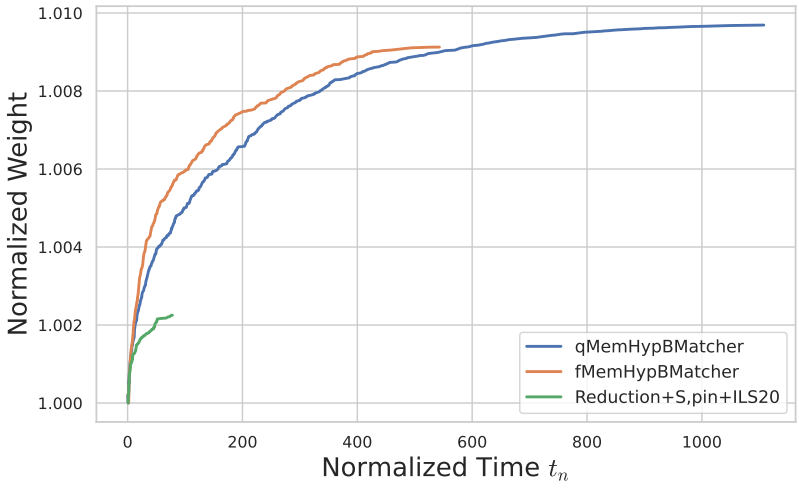


Figure 7.10: Capacity 3 (1 hour)

CHAPTER 8

Conclusion and Future Work

This thesis introduces a memetic algorithm for solving the hypergraph b-matching problem, building upon fundamental concepts and prior work. A novel crossover method is introduced, allowing offspring solutions to inherit favorable traits from both parents. Using this central idea an algorithm, extending the state-of-the-art solution by Großmann et al. [38], was formulated. It uses the data reductions, local search, and greedy solver from their work. As it incorporates an approach to local search it falls into the category of memetic algorithms.

The memetic algorithm first applies problem reductions to minimize the input size. It then generates an initial population by repeatedly applying a greedy solver and local search to the reduced problem. The population is iteratively refined through crossover, where individuals are selected using a parent selection strategy and recombined through the novel crossover operator. The resulting offspring undergo mutation and further local search optimization before competing with the previous generation. All the while ensuring that the best solution is retained. This cycle continues until a termination criterion is met, upon which the best solution is returned.

The algorithm is highly adaptive, providing fine-grained control over key components such as parent selection, mutation, and local search. Additionally, it has been parallelized, as part of this work, to leverage modern multi-core systems efficiently. Different parallelization strategies allow for adjustable levels of communication between processing elements (PEs), further enhancing its flexibility. The thesis also details the data structures and transformation methods used to represent hypergraph b-matchings.

Experiments were conducted to provide key insights into optimal algorithmic component choices and parallelization performance. Findings from different tuning experiments were combined to create good configurations for the algorithm. The results indicate that parallelization scales well, maintaining efficiency as the number of PEs increases. Additionally, the more communicative parallelization strategies show improved performance especially for short durations.

The effectiveness of the proposed approach is evaluated through extensive experiments.

While the algorithm is not yet competitive with the state-of-the-art, it demonstrates promising initial results. In particular, for smaller instances, an optimized average improvement of 0.37% (for `fHeiMemBMatch_SMALL`) and 0.56% (for `qHeiMemBMatch_SMALL`) was achieved for capacity 1 ($b(v) \equiv 1$). These results highlight the algorithms potential, especially when sufficient computational resources are available. Thus, it presents a promising approach for solving the hypergraph b-matching problem and, with further refinements, could potentially compete with state-of-the-art algorithms.

Future Work. Several potential improvements could enhance the memetic algorithm. Given its memetic nature, the algorithm allows for the integration of different local search strategies. If new local search techniques for hypergraph b-matching are developed, they could be incorporated to extend the algorithm. Other algorithms have had success using multiple local search algorithms [31, 55].
Furthermore, the relationship between the time allocated for population initialization and the time devoted to memetic steps, warrants further investigation. Optimally allocating only the necessary time for computing the initial generations, as those contribute the most significant improvements. All the while reserving the remaining time for refining high-quality initial solutions could provide a robust framework applicable not only to the selected smaller subset but to a broader range of instances. Additionally, improving the efficiency of the memetic steps could mitigate the impact of this balance, further enhancing overall performance.
Another promising enhancement is the inclusion of a restart mechanism to mitigate convergence stagnation. This strategy would restart the algorithm when convergence is detected, retaining the best solutions while introducing new randomly initialized individuals. This approach helps escape local optima and restores diversity in populations that have become too homogeneous. Other memetic algorithms have demonstrated the effectiveness of this method [75, 59].
Lastly, an alternative partitioning strategy could be explored by increasing the number of blocks and adapting the crossover operator accordingly. This would divide a matching into smaller, independent segments, effectively treating them as separate genes within a solution. This structure could i.e. enable gene-level tournament selection, allowing for a more fine-grained exploration of the solution space. Such an approach may improve diversity and facilitate a more adaptive search process.

Command-Line Arguments

The following table shows the command line arguments `fHeiMemBMatch` and `qHeiMemBMatch`. The configurations were used in Chapter 7 with a timeout of 3600 sec (1 hour) and a maximum population size of 100 for shorter tests. For the longer tests a 10800 sec timeout (3 hour) timeout was used and the maximum population size of was increased to 200.

The subsequent table presents the command line arguments `fHeiMemBMatch_SMALL` and `qHeiMemBMatch_SMALL`. The configurations were used in Chapter 7 with a timeout of 3600 sec (1 hour). The biggest differences are highlighted. Additionally, it should be noted that the population size was reduced to only 50.

Category	fHeiMemBMatch	qHeiMemBMatch
Parallel Features		
–use_shared_cooperation	false	true
–use_shared_competition	true	false
–use_non_blocking_coop	false	true
Diversification Greedy Matchings		
–diversification_methode	GAUSSIAN	GAUSSIAN
–gaussianStdDevPopInit	2	4
–diversity_factor_initial_population	0.4	0.7
Timeouts		
–initial_creation_time_percentage	0.1	0.1
Genetic Algorithm Parameters		
–parent_selection_function	Tournament	Tournament
–tournament_size	5	3
–exchange_every_nth_generation	5	5
Local Search Parameters		
–runs_local_search_short	6	4
–runs_local_search_long	20	15
–runs_local_search_after_mutation	0	7
Mutation Parameters		
–mutation_function	EdgeRemovalMutation	EdgeRemovalMutation
–mutation_probability	0	0.25
–mutation_rate	-	0.001
–smart_mutation_enabled	true	true
Competition Parameters		
–comma_replacement_rate	1	0.75
–competition_strategy	PLUS	PLUS

Command line arguments for `fHeiMemBMatch` and `qHeiMemBMatch`

Category	fHeiMemBMatch_SMALL	qHeiMemBMatch_SMALL
Parallel Features		
–use_shared_cooperation	false	true
–use_shared_competition	true	false
–use_non_blocking_coop	false	true
Diversification Greedy Matchings		
–diversification_methode	GAUSSIAN	GAUSSIAN
–gaussianStdDevPopInit	2	4
–diversity_factor_initial_population	**0.7**	0.7
Timeouts		
–initial_creation_time_percentage	**0.4**	**0.4**
Genetic Algorithm Parameters		
–parent_selection_function	Tournament	Tournament
–tournament_size	5	3
–exchange_every_nth_generation	5	5
Local Search Parameters		
–runs_local_search_short	8	8
–runs_local_search_long	50	50
–runs_local_search_after_mutation	0	0
Mutation Parameters		
–mutation_function	EdgeRemovalMutation	EdgeRemovalMutation
–mutation_probability	0	0
–mutation_rate	-	-
–smart_mutation_enabled	true	true
Competition Parameters		
–comma_replacement_rate	1	1
–competition_strategy	PLUS	PLUS

Command line arguments for `fHeiMemBMatch_SMALL` and `qHeiMemBMatch_SMALL`

Zusammenfassung

Ein Hypergraph erweitert das Konzept eines Graphen, indem er den Kanten, sogenannte Hyperkanten, erlaubt, mehr als zwei Knoten zu verbinden und somit komplexere Beziehungen zu modellieren. Das Hypergraph b-Matching-Problem verallgemeinert das Hypergraph-Matching-Problem, indem es darauf abzielt, eine Teilmenge von Hyperkanten auszuwählen, sodass jeder Knoten v in höchstens $b(v)$ der ausgewählten Hyperkanten enthalten ist.
In dieser Arbeit stellen wir einen memetischen Ansatz zur Lösung des Hypergraph b-Matching-Problems vor. Durch die Verwendung von Partitionen des Hypergraphen sind wir in der Lage, zwei Hypergraph b-Matchings desselben Hypergraphen zu einem potenziell besseren zu kombinieren. Diese zentrale Idee wird als Crossover-Operator verwendet, um den wir unseren memetischen Algorithmus aufgebaut haben. In umfangreichen Experimenten zeigen wir die Effektivität und Verbesserungen gegenüber den aktuellen Algorithmen. Umfangreiche Experimente wurden durchgeführt, um die Wirksamkeit dieser Algorithmen zu validieren. Während die erzielten Ergebnisse noch nicht mit dem aktuellen Stand der Technik mithalten können, sind die ersten Ergebnisse vielversprechend.

Bibliography

[1] Faisal N. Abu-Khzam, Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash. Recent advances in practical data reduction. In Hannah Bast, Claudius Korzen, Ulrich Meyer, and Manuel Penschuck, editors, *Algorithms for Big Data: DFG Priority Program 1736*, pages 97–133. Springer Nature Switzerland, Cham, 2022.

[2] Charles J. Alpert. The ispd98 circuit benchmark suite. In *Proceedings of the 1998 International Symposium on Physical Design*, ISPD '98, page 80–85, New York, NY, USA, 1998. Association for Computing Machinery.

[3] Diogo V Andrade, Mauricio GC Resende, and Renato F Werneck. Fast local search for the maximum independent set problem. *Journal of Heuristics*, 18:525–547, 2012.

[4] Robin Andre, Sebastian Schlag, and Christian Schulz. Memetic multilevel hypergraph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 347–354, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Robin Andre, Sebastian Schlag, and Christian Schulz. Memetic multilevel hypergraph partitioning. In Hernán E. Aguirre and Keiki Takadama, editors, *GECCO*, pages 347–354. ACM, 2018.

[6] Georg Anegg, Haris Angelidakis, and Rico Zenklusen. *Simpler and Stronger Approaches for Non-Uniform Hypergraph Matching and the Füredi, Kahn, and Seymour Conjecture*, pages 196–203.

[7] Aaron Becker, Ramprasad Venkataraman, and L Kale. Patterns for overlapping communication and computation. *Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, IL*, 3(4), 2009.

[8] Julien Becker, Francis Maes, and Louis Wehenkel. On the relevance of sophisticated structural annotations for disulfide connectivity pattern prediction. *PLoS One*, 8(2):e56621, 2013.

[9] Rene Beier and Jop F Sibeyn. *A powerful heuristic for telephone gossiping*. Citeseer, 2000.

[10] A. Belov, D. Diepold, M. Heule, and M. Järvisalo. The sat competition 2014, 2014.

[11] Piotr Berman. A d/2 approximation for maximum weight independent set in d-claw free graphs. In *Algorithm Theory - SWAT 2000*, pages 214–219, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[12] Sonja Biedermann, Monika Henzinger, Christian Schulz, and Bernhard Schuster. Memetic graph clustering, 2018.

[13] M. Birn, V. Osipov, P. Sanders, C. Schulz, and N. Sitchinava. Efficient parallel and external matching. In *Proc. of Euro-Par 2013*, volume 8097 of *LNCS*, pages 659–670. Springer, 2013.

[14] T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *IPL*, 42(3):153–159, 1992.

[15] E.K. Burke, S. Gustafson, and G. Kendall. Diversity in genetic programming: an analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[16] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *CoRR*, abs/2205.13202, 2022.

[17] Chandra Chekuri and Chao Xu. *Computing minimum cuts in hypergraphs*, pages 1085–1100.

[18] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[19] Marek Cygan. Improved approximation for 3-dimensional matching via bounded pathwidth local search. In *FOCS*, pages 509–518. IEEE, 2013.

[20] Dilip Datta, Kalyanmoy Deb, and Carlos M Fonseca. Solving class timetabling problem of iit kanpur using multi-objective evolutionary algorithm. *KanGAL, Report*, 2006006:1–10, 2006.

[21] Timothy A. Davis and Yifan Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[22] I De Falco, A Della Cioppa, and E Tarantino. Mutation-based genetic algorithm: performance evaluation. *Applied Soft Computing*, 1(4):285–299, 2002.

[23] Pedro Diaz-Gomez and Dean Hougen. Initial population for genetic algorithms: A metric approach. pages 43–49, 01 2007.

[24] Nevzat Onur Domanic, Chi-Kit Lam, and C. Gregory Plaxton. Bipartite Matching with Linear Edge Weights. In Seok-Hee Hong, editor, *27th International Symposium on Algorithms and Computation (ISAAC 2016)*, volume 64 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[25] Doratha E Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[26] Andre Droschinsky, Petra Mutzel, and Erik Thordsen. Shrinking trees not blossoms: A recursive maximum matching approach. In Guy E. Blelloch and Irene Finocchi, editors, *Proc. of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020*, pages 146–160. SIAM, 2020.

[27] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1), jan 2014.

[28] Fanny Dufossé, Kamer Kaya, Ioannis Panagiotas, and Bora Uçar. Effective heuristics for matchings in hypergraphs. In *International Symposium on Experimental Algorithms*, pages 248–264. Springer, 2019.

[29] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

[30] Mourad El Ouali and Gerold Jäger. The b-matching problem in hypergraphs: Hardness and approximability. In Guohui Lin, editor, *Combinatorial Optimization and Applications*, pages 200–211. Springer, 2012.

[31] Héctor Joaquín Fraire Huacuja, Guadalupe Castilla Valdez, Claudia G. Gómez Santillan, Juan Javier González Barbosa, Rodolfo A. Pazos R., Shulamith S. Bastiani Medina, and David Terán Villanueva. Multiple local searches to balance intensification and diversification in a memetic algorithm for the linear ordering problem. In Emilio Corchado, Marek Kurzyński, and Michał Woźniak, editors, *Hybrid Artificial Intelligent Systems*, pages 18–25, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[32] Martin Fürer and Huiwen Yu. Approximating the k-set packing problem by local improvements. In *ISCO*, LNCS, pages 408–420, Germany, 2014. Springer.

[33] Harold N Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 434–443, 1990.

[34] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms*, volume 1, pages 69–93. Elsevier, 1991.

[35] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. In *18th International Symposium on Experimental Algorithms (SEA)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 11:1–11:15, 2020.

[36] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced flow-based multilevel hypergraph partitioning. In *18th International Symposium on Experimental Algorithms, SEA*, pages 11:1–11:15, 2020.

[37] Ernestine Großmann, Sebastian Lamm, Christian Schulz, and Darren Strash. Finding near-optimal weight independent sets at scale. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 293–302, 2023.

[38] Ernestine Großmann, Felix Joos, Henrik Reinstädtler, and Christian Schulz. Engineering hypergraph b-matching algorithms, 2024.

[39] Lingaraj Haldurai, T Madhubala, and R Rajalakshmi. A study on genetic algorithm and its applications. *Int. J. Comput. Sci. Eng*, 4(10):139–143, 2016.

[40] Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-set packing. *computational complexity*, 15(1):20–39, 2006.

[41] Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[42] Johan Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182(1):105 – 142, 1999.

[43] Prasanna Jog, Jung Y Suh, and Dirk Van Gucht. The effects of population sizeheuristic crossover and local improvement on a genetic algorithm for the traveling salesman problem. In *Proceedings of the 3rd international conference on genetic algorithms*, pages 110–115, 1989.

[44] Joost Jorritsma, Johannes Lengler, and Dirk Sudholt. Comma selection outperforms plus selection on onemax with randomly planted optima. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1602–1610, 2023.

[45] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. A local search approximation algorithm for k-means clustering. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry*, SCG '02, page 10–18, New York, NY, USA, 2002. Association for Computing Machinery.

[46] Richard M. Karp and Michael Sipser. Maximum matching in sparse random graphs. In *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symp. on*, pages 364–375. IEEE, 1981.

[47] Viatcheslav Korenwein, André Nichterlein, Rolf Niedermeier, and Philipp Zschoche. Data reduction for maximum matching on real-world graphs: Theory and experiments. In *ESA*, volume 112 of *LIPIcs*, pages 53:1–53:13, 2018.

[48] Christos Koufogiannakis and Neal E Young. Distributed fractional packing and maximum weighted b-matching via tail-recursive duality. In *DISC*, pages 221–238. Springer, 2009.

[49] Vlasis Koumousis and Christos Katsaras. A saw-tooth genetic algorithm combining the effects of variable population size and reinitialization to enhance performance. *Evolutionary Computation, IEEE Transactions on*, 10:19 – 28, 03 2006.

[50] Piotr Krysta. Greedy approximation via duality for packing, combinatorial auctions and routing. In *MFCS*, pages 615–627. Springer, 2005.

[51] Sebastian Lamm, Peter Sanders, and Christian Schulz. Graph partitioning for independent sets. In *Experimental Algorithms: 14th International Symposium, SEA 2015, Paris, France, June 29–July 1, 2015, Proceedings*, pages 68–81. Springer, 2015.

[52] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F Werneck. Finding near-optimal independent sets at scale. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 138–150. SIAM, 2016.

[53] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990.

[54] J.A. Lima, N. Gracias, H. Pereira, and A. Rosa. Fitness function design for genetic algorithms in cost evaluation based problems. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 207–212, 1996.

[55] Jiaqi Liu, X. Z. Gao, X. Wang, and K. Zenger. A multiple local search strategy in memetic evolutionary computation for multi-objective robust control design. In *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 645–650, 2012.

[56] Fernando G. Lobo and David E. Goldberg. The parameter-less genetic algorithm in practice. *Information Sciences*, 167(1):217–232, 2004.

[57] Zhipeng Lü and Jin-Kao Hao. A memetic algorithm for graph coloring. *European Journal of Operational Research*, 203(1):241–250, 2010.

[58] Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 519–528. IEEE, 2014.

[59] Rafael Martí, Ángel Corberán, and Juanjo Peiró. Scatter search. In *Handbook of heuristics*, pages 717–740. Springer, 2018.

[60] Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *21st Symposium on Foundations of Computer Science*, pages 17–27. IEEE Computer Society, 1980.

[61] Pablo Moscato, Carlos Cotta, Alexandre Mendes, et al. Memetic algorithms. *New optimization techniques in engineering*, 141:53–85, 2004.

[62] Ferrante Neri and Carlos Cotta. Memetic algorithms and memetic computing optimization: A literature review. *Swarm and Evolutionary Computation*, 2:1–14, 2012.

[63] Meike Neuwohner. *Passing the Limits of Pure Local Search for Weighted k-Set Packing*, pages 1090–1137.

[64] Ojas Parekh and David Pritchard. Generalized hypergraph matching via iterated packing and local ratio. In *WAOA*, pages 207–223. Springer, 2015.

[65] G. Pavai and T. V. Geetha. A survey on crossover operators. *ACM Comput. Surv.*, 49(4), December 2016.

[66] Martin Pelikan, David Goldberg, and Erick Cantu-Paz. Bayesian optimization algorithm, population sizing, and time to convergence. pages 275–282, 09 2000.

[67] Tania Pencheva, Krassimir Atanassov, and A. Shannon. Modelling of a stochastic universal sampling selection operator in genetic algorithms using generalized nets. pages 1–7, 01 2010.

[68] Anandhan Prathik, K Uma, and J Anuradha. An overview of application of graph theory. *International Journal of ChemTech Research*, 9(2):242–248, 2016.

[69] R. Preis. Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In *STACS*, volume 1563 of *LNCS*, pages 259–269. Springer, 1999.

[70] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM J. Exp. Algorithmics*, mar 2022.

[71] Gregory Schwing, Daniel Grosu, and Loren Schwiebert. Parallel maximum cardinality matching for general graphs on gpus. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 880–889, 2024.

[72] Mushfeq-Us-Saleheen Shameem and Raihana Ferdous. An efficient k-means algorithm integrated with jaccard distance measure for document clustering. In *2009 First Asian Himalayas International Conference on Internet*, pages 1–6, 2009.

[73] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. Comparative review of selection techniques in genetic algorithm. In *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, pages 515–519, 2015.

[74] Richard Sinkhorn and Paul Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348, 1967.

[75] Jianyong Sun, Jonathan Garibaldi, Natalio Krasnogor, and Qingfu Zhang. An intelligent multi-restart memetic algorithm for box constrained global optimisation. *Evolutionary computation*, 21, 02 2012.

[76] Theophile Thiery and Justin Ward. *An Improved Approximation for Maximum Weighted k-Set Packing*, pages 1138–1162.

[77] Ian M Wanless and David R Wood. A general framework for hypergraph coloring. *SIAM Journal on Discrete Mathematics*, 36(3):1663–1677, 2022.

[78] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195, 2000.