Universität Karlsruhe (TH)
Fakultät für Informatik
Lehrstuhl Prof. Dr. Wagner

Studienarbeit

Simultaneous Matchings Dynamic Graphs

Jonathan Dees

eingereicht am

Supervisors: Prof. Dr. Dorothea Wagner
Dipl. Inf. Ignaz Rutter
Dipl. Inf. Marcus Krug

Dynamic graphs, i.e. graphs that change over time, appear in diverse scenarios, for example in computer or social networks. When we solve a certain problem on a dynamic graph, the solution has to be adjusted as the graph changes. We require that the solution changes little over time, i.e. we require that the solutions are similar with respect to a given similarity measure. We consider the following problem SimMatch: Given two graphs, one being a modification of the other, find a maximum matchings in each graph, such that the maximum matchings are as similar as possible, i.e. the intersection of the matchings should be maximized. Since we are given two graphs, we know the modification in advance, so this is an offline problem.

In this work we show that SimMatch is NP-hard by a reduction from MaxCut. We also proof that it is NP-hard to approximate SimMatch better than 50/51. SimMatch remains NP-hard for rather restricted classes of graphs, e.g. cycles and trees. Instances in which the number of maximum matchings in one graph is bounded polynomially, can be solved in polynomial time: We present an algorithm running in $\mathrm{O}(\mathcal{MM}(G) \cdot \sqrt{n}m^2)$ time, where $\mathcal{MM}(G)$ denotes the number of maximum matchings of one graph.

We also present an ILP-formulation and a heuristic with a running time of $\mathrm{O}(n^2m + n^3 \log n)$ for SimMatch, which we both evaluate in the experimental part. In our test cases, the heuristic performs significantly faster than the ILP-approach while having a relative low error rate.

Acknowledgements

At this point i would like to thank

I hereby declare that this Studienarbeit is my own, unaided work and that i have not used sources or means without declaration in the text.

Karlsruhe, 13.11.08, Jonathan Dees

Contents

1 Introduction

A graph is an abstract representation of a set of objects (nodes) and the relation of the objects to each other (edges between the nodes). There are many applications in which graphs are used, e.g. networks. In a network we have a set of elements, in which any set of two elements may be connected. Often we have a dynamic setting, i.e. the network changes over time and with that the related graph. When solving a problem for a graph that changes dynamically, all future modifications may be known in advance, this is the *offline* problem, or the future modifications are not known at the time where we want to compute a solution for the current graph, this is the *online* problem. In this work we concentrate on the former case, the offline problem. One possible application of the offline problem is as follows: If we know the history of the modifications for a graph, we can analyze the development of the solution for a problem on the sequence of graphs produced by the modifications. Two successive graphs in the sequence, i.e. one graph is generated by modifications of the other, should have similar solutions for the problem. Beside that the solution should be as good as possible considering only the actual graph. Thus, we need to make a trade-off for our solutions: the solutions should be as good as possible according to the problem for each graph but should also not vary much over time. Summarized: We want to compute solutions for a problem on several instances, so that the solutions are as similar as possible (assuming that the instances are somehow related). In this theses we analyze this problem for *matchings*.

A matching in a graph $G = (V, E)$ is a subset of edges $M \subseteq E$, in which no two edges are adjacent, i.e. no two edges share the same endpoint. There are many applications for finding matchings, here we want to give one example: We are given a number of employees and we want to build teams of two people working together. Some employees are expected to work efficiently together and some are not. If we model the employees as nodes and insert edges between all nodes representing two employees which form a good team, a matching in that graph will return a possible pairing of employees. As we want to minimize employees without a team partner, the matching should contain as many edges as possible. If there exists no other matching with a larger number of edges, the matching is called maximum matching. In general, there exists more than one maximum matching.

There are quite a number of algorithms for computing maximum matchings available. It has been one of the first non-trivial combinatorial problems for which a polynomial time algorithm has been found [Edm65]. Today, one of the fastest known algorithm for finding maximum matchings runs in $O(\sqrt{n}m)$ time [MV80, Vaz94]. There exists an algorithm based on matrix multiplication with a running time of $O(n^{2.367})$ [MS04], but it is rather impractical.

As discussed before, we are analyzing the matching problem in a dynamic setting as an offline problem. We search for maximum matchings in two graphs, so that the matchings in both graphs are as similar as possible. As an input, we have two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ which may share some edges and nodes, i.e. $V_1 \cap V_2 \neq \emptyset$ and $E_1 \cap E_2 \neq \emptyset$. We want to find maximum matchings M_1 and M_2 for G_1 and G_2, respectively, so that the two maximum matchings are similar as possible, that means, $|M_1 \cap M_2|$ should be maximized. Note that although $|M_1 \cap M_2|$ should be maximized, M_1 and M_2 must remain *maximum* matchings. We call this problem SimMatch (Simultaneous Matching). In Section 1.1 we describe a more general version, but throughout this work we focus on the basic problem SimMatch.

A dynamic version of the previous example with the employees can be modeled as an instance of SimMatch. Again, we have a number of employees which should form teams of two and some work efficiently together and some not. A part of the employees is working the whole week (except Sunday), the other part is only working half-time, some of these half-time workers is working from Monday to Wednesday, some of them are working from Thursday to Saturday. We now want to find an admissible pairing of employees for the period from Monday to Wednesday and from Thursday to Saturday. In both periods, the number of paired employees should be maximum. Also, it is advantageous that pairings last the whole week and are preserved from Monday to Saturday. If we generate two graphs, one graph for each period, and model the employees as nodes and insert an edge between two nodes which represent employees that are good at working together, as we did in the previous example, we generated a SimMatch instance. The solution of the instance returns us pairings of employees, such that the number of pairings is maximum in each period and such that the number of pairings lasting the whole week is maximized.

Beside this example, we want to mention another possible application of SimMatch, related to graph drawing. Drawing a graph is the mapping of nodes and edges to graphical objects on the plain (usually points and lines, respectively) to visualize it. There exist a lot of different graph drawing algorithms, one interesting algorithm yielding good results can be found in [Wal03]. We propose to extend the algorithm in [Wal03] in order to create similar drawings drawings for two similar graphs. The algorithm in [Wal03] uses matchings to identify adjacent nodes and contracts the graph iteratively according to the matchings. The coarsest graph is given an initial layout and then the layout is refined for all graphs created during the contraction, ending with the original one. If we want to compute a layout for two graphs that are similar, we compute similar matchings in both graphs for each step, this will hopefully also result in a similar layout. Finding similar matchings for two graphs that share some edges is our problem SimMatch, so if we can solve SimMatch we can extend the graph drawing algorithm in [Wal03] to draw similar graphs with a similar layout.

Beside the motivation of possible practical applications, the pure theoretical problem is already very interesting. To the best of our knowledge, the problem SimMatch (or the more general problem listed in Section 1.1) has not yet been examined. This work closes this gap and analyses this problem. It is structured as follows. At first, we give a formal description of the problem in the next Section 1.1. Preliminaries can be found in Section 1.2.

In Chapter 2 we analyze the complexity of SimMatch. We will show that SimMatch is NP-hard and prove this by a reduction from MaxCut. We will also see that SimMatch remains NP-hard for certain restrictions on the graph structure, namely, both graphs consist solely of cycles, both graphs are planar, both graphs are connected, both graphs consist solely of simple paths and both graphs consist of a single tree. We will also see that SimMatch is hard to approximate which also applies to the restrictions.

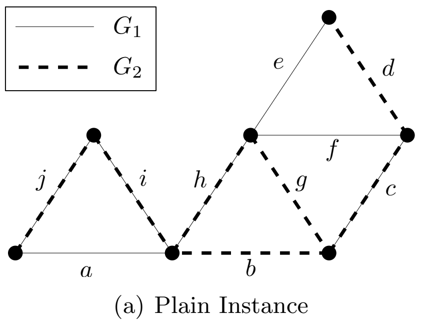
In Chapter 3 we propose some solutions for SimMatch despite its NP-hardness. We first give an ILP-formulation for SimMatch After that we present a heuristic for SimMatch which runs in $O(n^2m + n^3\log n)$ time, but, unfortunately, leaves us in general with no relative guarantee on the solution. Then we will show how to efficiently enumerate all maximum matchings in a graph and use this in order to solve certain instances of SimMatch optimally in polynomial time: Those instance have a polynomially bounded number of different maximum matchings.

In Chapter 4 we evaluate our heuristic as well as our ILP approach on real instances. We will see that the heuristic is very close to the optimal solution and beats the ILP approach concerning running time. The ILP approach is applicable on small instances while guaranteeing optimal results. The heuristic solves instances up to values of 10^6 for $n + m$ whereas the ILP is only able to solve instances up to values of 10^4 for $n + m$.

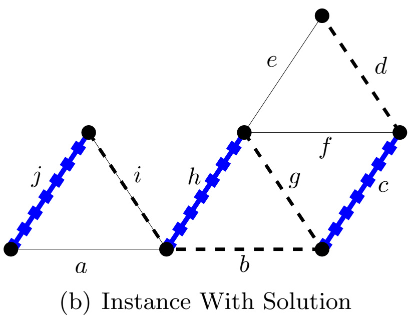
In the last Chapter 5 we conclude our work and give some ideas about future work, also pointing out some other possible simultaneous problems.

1.1 Problem

The formal definition of SimMatch is as follows:



(a) Plain Instance



(b) Instance With Solution

Figure 1.1: SimMatch example

SimMatch	
Input:	Two graphs G_1 and G_2 sharing some common edges and nodes
Output:	Maximum matching M_1 in G_1 and M_2 in G_2 such that $\|M_1 \cap M_2\|$ is maximized

In detail the problem can be described as follows: We are given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ which share some nodes $V' = V_1 \cap V_2$ and some edges $E' = E_1 \cap E_2 \subseteq \binom{V'}{2}$. We want to find two maximum matchings $M_1 \subseteq E_1$ and $M_2 \subseteq E_2$ in G_1 and G_2, respectively. M_1 and M_2 must be maximum, i.e. $|M_1|$ and $|M_2|$ are maximum under all matchings in G_1 and G_2, respectively. Additionally, the cardinality of the intersection of M_1 and M_2 must be maximum under all maximum matchings in G_1 and G_2, i.e. there are no maximum matchings M'_1 and M'_2 in G_1 and G_2, respectively, so that $|M'_1 \cap M'_2| > |M_1 \cap M_2|$.

In Figure 1.1 you find a small example instance of SimMatch. Figure 1.1(a) shows the plain instance: We see two graphs G_1 and G_2, both having the same nodes, only their edges differ. The size of a maximum matchings is 3 in both graphs since there are 7 nodes and a matching with 3 edges is possible in both graphs. Figure 1.1(b) shows the SimMatch instance with an optimal solution, the matched edges are marked bold (blue). We have the same matching in both graphs, $M_1 = M_2 = \{c, h, j\}$, i.e. the size of both matchings is 3, hence they are maximum matchings. Also, the size of the intersection of the matchings is 3, which is maximum, thus, the matchings M_1 and M_2 constitute an optimal SimMatch solution.

Considering our problem definition of SimMatch, there are some variations possible. For example, in SimMatch the matchings must be maximum, and under all pairs of maximum matchings, the cardinality of the intersection must be maximum. It could also be the other way around, i.e. $|M_1 \cap M_2|$ must be maximum and under all pairs of matchings that satisfy this requirement $|M_1| + |M_2|$ must be maximum. There are also other trade-offs imaginable, in addition to that; also the number of graphs could be extended. However, we focused on the basic problem version SimMatch during our work. As we will see this problem is already hard and some results for SimMatch can be transferred to the more general version. Before we describe the more general problem version, we want to state informally what kind of problems we want to be able to describe with our general problem version. All problems that can be described have the following in common: We search for matchings in several graphs, the matchings and their intersections with each other must satisfy certain requirements on the maximality of their cardinality in relation to each other. The following problem version allows to describe such problems, note that during this work only the specific version of SimMatch presented above is used unless otherwise stated. In what follows $\mathcal{P}(A)$ denotes the power set of A.

SimMatch (General Version)

Input: n graphs $G_1, \dots, G_n$ sharing some common edges and nodes, and a weight function $w : E \times \mathcal{P}(\{1, \dots, n\}) \to \mathbb{R}$ with $E = \bigcup_{k=1}^{n} E_k$

Output: Matchings $M_1, \dots, M_n$ in $G_1, \dots, G_n$, respectively, so that

$$\sum_{\substack{e \in E, \\ s \in \mathcal{P}(\{1,\dots,n\}), \\ e \in \bigcap_{k \in s} M_k}} w(e, s)$$

is maximum.

This formulation allows us, beside weighting each edge differently, to set different priorities for edges for different subsets of graphs. For example, if we have given a sequence of Graphs $G_1, \dots, G_n$, we can set $w(e, \{k, k+1\}) > 0$ for $1 \leq k < n$ to reward taking the same edge in two matchings of two successive graphs.

Of course, the simple version is included in the general version. If we have only $n = 2$ graphs G_1 and G_2 and set w for every $e \in E$ as

$$w(e, s) = \begin{cases} |E| + 1 & \text{if } s = \{1\} \text{ or } s = \{2\} \\ 1 & \text{if } s = \{1, 2\} \\ 0 & \text{if } s = \emptyset \end{cases}$$

this corresponds to the basic SimMatch with G_1 and G_2 as input. In an optimal solution M_1 and M_2 must be maximum matchings. Assume that, without loss of generality, M_1 would be not a maximum matching. Hence there exists a maximum matching M_1' with $|M_1'| > |M_1|$ and $\sum_{e \in M_1'} w(e, \{1\}) \geq \sum_{e \in M_1} w(e, \{1\}) + |E| + 1$. So replacing M_1 by M_1' will increase our sum due to $w(e, \{1\})$ by at least $|E| + 1$ and decrease our sum due to $w(e, \{1, 2\})$ by at most $|E|$ since there are at most $|E|$ edges. This means that the sum is not maximum, contradicting our assumption that M_1 and M_2 is an optimal solution. Hence M_1 and M_2 must be maximum matchings. $w(e, \{1, 2\}) = 1$ ensures that under all pairs of maximum matchings the one with the largest intersection $|M_1 \cap M_2|$ is taken.

1.2 Preliminaries

A *graph* G is represented by a pair (V, E) such that V is a set of nodes and $E \subseteq \binom{V}{2}$ is a set of edges which represents a relation between the nodes. For an edge $e = \{u, v\} \in E$ with $u, v \in V$, we say the nodes u and v are *adjacent* and e and u are *incident* (also e and v). In the following, we use only *undirected* graphs if not otherwise stated. The *degree* of a node u in an undirected graph is the number of its incident edges and is denoted by $\deg(u)$. If we refer to m or n within the text n usually represents the number of nodes and m the number of edges in the associated graph, i.e. for a graph $G = (V, E)$ $n = |V|$ and $m = |E|$. A *path* in a graph is a sequence of nodes $(v_1, v_2, \dots, v_n)$ with edges connecting the nodes $\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}$. By adding the edge $\{v_n, v_1\}$ we obtain a *cycle*. A *matching* $M \subseteq E$ in a graph $G = (V, E)$ is a subset of edges without two different edges sharing one endpoint. A matching $M \subseteq E$ in a graph $G = (V, E)$ is a *maximum matching* if there exists no matching M' in G with $|M'| > |M|$. A matching $M \subseteq E$ in a graph $G = (V, E)$ is a *maximal matching* if there exists no edge $e \in E$ which can be added to M so that $M \cup \{e\}$ is still a matching. We deal mostly with maximum matchings and not with maximal matchings during this work. $\mathcal{P}(A)$ denotes the power-set of a set A.

2 Complexity

There are often several ways to solve a problem. Sometimes it is useful to analyse the problem from a theoretical point of view before taking a path into a specific direction to solve the problem. The direction of the path may expose as a dead end and the used tools as inappropriate. Even the problem itself may turn out to be unsolvable.

SimMatch is not unsolvable. But there are some limitations if we want to compute an exact solution. It is helpful to know about this limitations if we want to understand algorithmic solutions for SimMatch. This chapter discusses those and the theoretical complexity of SimMatch. As we will see the problem is NP-hard, which means that there does not exist any algorithm which solves SimMatch in polynomial time, unless P=NP (see [GJ79] for some classic introduction to NP-completeness in general). This may be surprising as finding a maximum matching in a single graph is solvable in polynomial time [Edm65]. The fact that SimMatch is NP-hard requires us to use some alternative approaches to get a solution, which is part of Chapter 3 in which we propose a heuristic.

In the first section of this chapter, we will show that SimMatch is NP-hard by reducing an already known NP-hard problem, namely MaxCut, to it. We will also show that SimMatch remains NP-hard for some relatively strong restrictions on the graph structure. In the succeeding section we will see that the problem is even more difficult: Every polynomial time approximation can not approximate better than 50/51, unless P=NP.

2.1 NP-hardness

In the first part of this section we show that SimMatch is NP-hard by reducing MaxCut to it. Later we will show that the problem remains NP-hard for some restrictions on the graph structure, namely both graph consist of cycles, are planar, are connected, consist solely of simple paths or each consist of a single tree.

The common way to prove that a problem is NP-hard is to show that the related decision problem is NP-complete. So in the following we will analyze the related decision problem of SimMatch, which is stated as follows.

SimMatch (Decision Problem)	
Instance:	Two graphs G_1 and G_2 sharing some edges and nodes and a positive integer K.
Question:	Is there a maximum matching in G_1 and G_2 with both matchings sharing at least K matched edges?

Note that in the further text we do not explicitly differentiate between the decision problem and the original SimMatch problem introduced in Section 1.1 as it should be quite clear which problem is referred to in the text.
The first part to proof the NP-completness of SimMatch is to show that SimMatch is in NP. That means we need to check if SimMatch can be solved by a non-deterministic algorithm in polynomial time.

Lemma 2.1.1. SimMatch $\in$ *NP*

Proof. Given an instance of SimMatch, we non-deterministically "guess" a solution which consists of two sets with edges including the matched edges. To verify that this solution is a "Yes"-instance we need to check the following:

1. Both sets are matchings, i.e. no two edges in each set share the same endpoints.
2. Both matchings are maximum, i.e. the number of edges in each set comply with the maximum number of edges possible, this value can be computed in polynomial time, see [Edm65].
3. The cardinality of the intersection is at least K.

These steps can be done in polynomial time. Hence it follows that SimMatch $\in$ NP. □

Next we show the NP-hardness of SimMatch by reducing an already known NP-hard problem to it. We present a reduction from MaxCut to SimMatch. MaxCut is known to be NP-complete [GJS74] and can be stated as follows:

MaxCut (Decision Problem)	
Instance:	Graph $G = (V, E)$, positive integer K
Question:	Is there a partition of V into disjoint sets A and B such that the number of edges from E that have one endpoint in A and one endpoint in B is at least K?

Lemma 2.1.2. MaxCut *is reducible to* SimMatch

Proof. First, we give a *polynomial* transformation from a MaxCut instance to a SimMatch instance. After that we will show that the transformation yields a "Yes"-instance if and only if the MaxCut instance is a "Yes"-instance.

Without loss of generality we consider only MaxCut instances in which every node has at least one incident edge. Nodes with degree zero can be included into an arbitrary partition in the solution as they do not have further influence. Before we go into the detail of the transformation we give a short overview of the general idea: The nodes of the MaxCut instance are transformed into cycles, those cycles build up G_1. The edges are also transformed into cycles forming the graph G_2. The important part is that node cycles of G_1 and edge cycles of G_2 share some edges. Edges are shared between the cycle of an edge and the two cycles of the nodes which are incident to the edge. Now we explain how this transformation corresponds to the original problem. All constructed cycles have an even number of edges. For such cycles there are exactly two maximum matchings possible. In a node cycle the matching represents whether we assigned the node to set A or to set B. If two nodes that are connected by an edge are assigned to different sets, the matching in the cycle that corresponds to the connecting edge can have a higher correspondence with the matchings in the two cycles of the nodes and therefore leads to a higher objective value. In this case the matchings of the edge cycle and the two node cycles can have an intersection of two. If the two nodes are assigned to the same set, the intersection can only be one. This is quite similar to the MaxCut instance, the only difference here is that the solution value is increased by only one for an edge if its two incident nodes lie in different sets. If the incident nodes lie in the same sets the solution value is not increased at all. Thus the objective value needs to be modified slightly. Now we show that such a transformation is really possible by describing its construction in detail.
We transform the MaxCut instance $(G = (V, E), K)$ into a SimMatch instance $(G_1 = \bigcup_{v \in V} \{\text{node cycle } c_v\}, G_2 = \bigcup_{\{u,v\} \in E} \{\text{edge cycle } c_{\{u,v\}}\}, K')$.

Node Cycles Every $v \in V$ is transformed into a node cycle c_v of length $4 \cdot \deg(v)$ located in G_1. The cycles have an even number of edges and thus, for each cycle there are exactly two possible maximum matchings (half of the edges can be matched). Each of those matchings in a cycle c_v corresponds to whether we put v to the set A or B (in MaxCut). Let the edges in this cycle be numbered from 1 to n, so that two consecutive numbered edges are adjacent (including n and 1). We call every odd numbered edge an A-edge, every even numbered edge a B-edge. In a maximum matching either all A-edges are matched, indicating that v is included in set A, or all B-edges are matched, indicating that v belongs to set B. If the A-edges are matched we say v is assigned to set A and vice versa. As the length of the node cycle is four times the degree of the node, c_v has four edges for each incident edge. We group them such that for each incident edge we have four consecutive edges on our cycle.

Edge Cycles To get some relation with the size of a certain partition in the MaxCut instance, we transform every edge $\{u, v\} \in E$ of the MaxCut instance

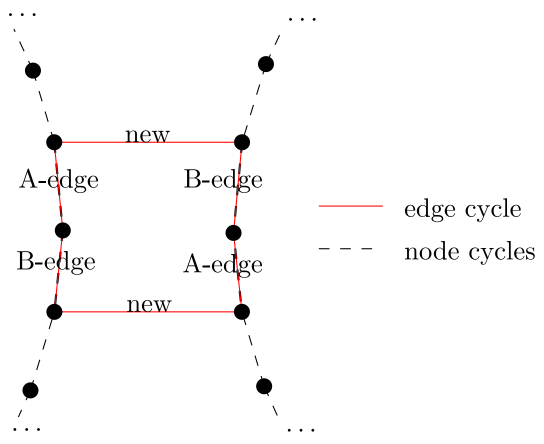


Figure 2.1: Edge Cycle

to a cycles $c_{\{u,v\}}$ of length 6. These cycles are named edge cycles and build up G_2. They do not solely consist of new edges, they share edges *(and their incident nodes)* with node cycles of G_1. Both node cycles c_v and c_u in G_1 hold four consecutive edges for our edge $\{u, v\}$. We only use the inner two edges of such four consecutive edges. This ensures that they are not influenced by neighbouring edge cycles and thus the edge cycle remain independent within G_2. The two inner edges consist of one A-edge and one B-edge. So the edge cycle is made up of two A-edges and two B-edges, one each of c_v and c_u, plus two new edges. One new edge is adjacent to the A-edge of c_v and the B-edge of c_u, the other one connects the B-edge of c_v and the A-edge of c_u.
In summary, a cycle $c_{\{u,v\}}$ looks like this: The A-edge of c_v followed by the B-edge of c_v, a new edge, the A-edge of c_u followed by the B-edge of c_u, the other new edge closing the cycle. The assembly of the edge cycle can also be seen in Figure 2.1 (In Figure 2.2 you see an example of a complete transformation). The idea about this cycle is the following: As for the node cycles, we also have two possible maximum matchings in the edge cycles in G_2. The edge cycle $c_{\{u,v\}}$ in G_2 connects the two node cycles c_u and c_v of G_1. The size of the intersection of the matching for G_1 and G_2, considering one of these edge cycles $c_{\{u,v\}}$ in G_2, can be either one (1), two (2) or zero (0) depending on how we match the corresponding node cycles c_u and c_v in G_1. There are two possibilities for each node cycle: match the A- or the B-edge. This represents whether we put the node into set A or into set B. The intersection has size one (1), if both corresponding node cycles c_u and c_v are matched in the same way (both A-edge or both B-edge). In this case it is irrelevant how we match the edge cycle. The size of the intersection is two (2), if the corresponding node cycles are matched different and the edge cycle is matched in the right way, zero (0) if the matching in the edge cycle is misplaced. This is easy to check by analysing all possible cases for an edge cycle seen in Figure 2.1. The edge cycle is independent of other cycles and therefore we will never take the matching with an intersection of zero elements.

Looking back to the MaxCut instance, if we have an edge with one endpoint in A and one in B in a solution, the objective value (number of edges having one endpoint in A and one in B) increases by one. In the SimMatch instance, the objective value (cardinality of the intersection of the matchings in G_1 and G_2) may increase by two. But in the SimMatch instance the objective value increases also by one for every edge not satisfying the condition that the endpoints are in different sets. That means we have one extra element in the intersection of M_1 and M_2 for every edge, independent of how we partitioned the nodes. Therefore we set $K' = K + |E|$.

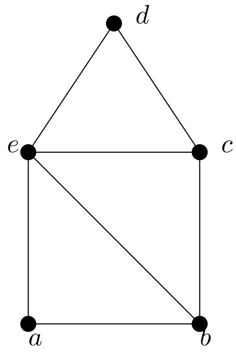
Clearly the transformation can be performed in polynomial time. In Figure 2.2 you find a small sample transformation, an optimal solution for the example MaxCut and SimMatch instance is shown in Figure 2.3 where the (*blue*) bold edges belong to the solution set. After the proof, we give some information about the example.

Now we show that the transformation is sound, i.e., the transformed instance of SimMatch is a "Yes"-instance if and only if the original instance of MaxCut is a "Yes"-instance.
If the MaxCut instance is a "Yes"-instance, there exists a partition of the nodes in A and B, so that we have at least K edges with one endpoint in A and the other in B. We are now able to set the matching in the transformed SimMatch instance as follows: In every node cycle we set the matching according to the partition in the MaxCut instance. As seen above, the intersection with the matching of the edge cycles is two for every edge with one endpoint in A and one in B and one for every other edge. Therefore the size of the intersection is at least $K + |E|$. Hence, we have a "Yes"-instance.
If the MaxCut instance is a "No"-instance, the transformed SimMatch instance is also a "No"-instance, for the following reason: If the SimMatch instance was a "Yes" instance, there would exist matchings so that the size of the intersection is at least $K + |E|$. That means there must be at least K edge cycles which have a matching intersection of two because every edge cycle has at most an intersection of two and there are $|E|$ ones of them. To have an intersection of two one of its related node cycles is assigned to set A and the other one to set B. If we use a partition in MaxCut which is determined by the matchings of the cycles in SimMatch, we have K edges having one endpoint in A and one endpoint in B. That means the MaxCut instance was a "Yes"-instance, which contradicts the assumption. □

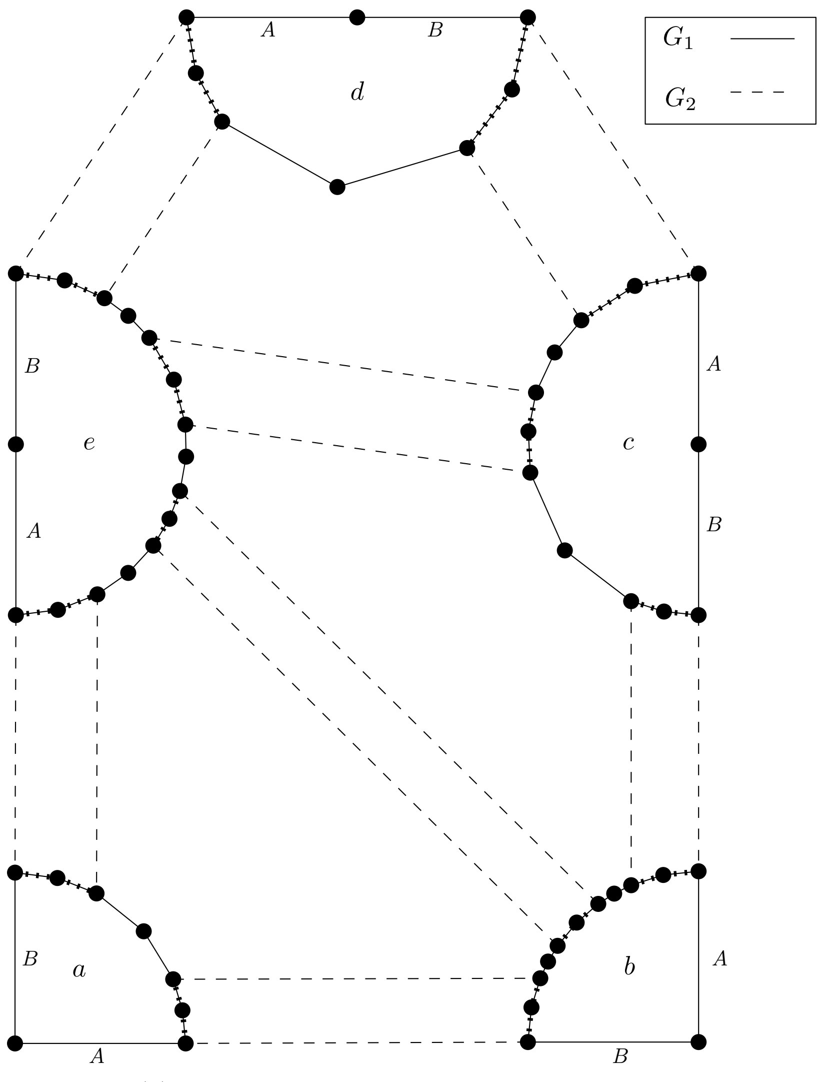
The following theorem follows directly from Lemma 2.1.1 and 2.1.2:

Theorem 2.1.3. SimMatch *is NP-complete.*

In addition to the formal description of the transformation we give a concrete example of a transformation shown in Figure 2.2. Figure 2.2(a) shows the orig-

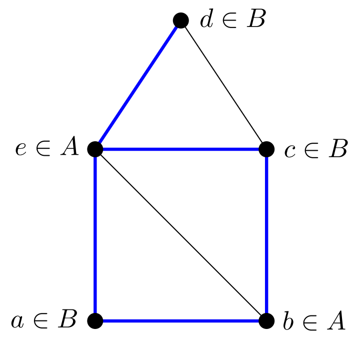


(a) MaxCut Instance

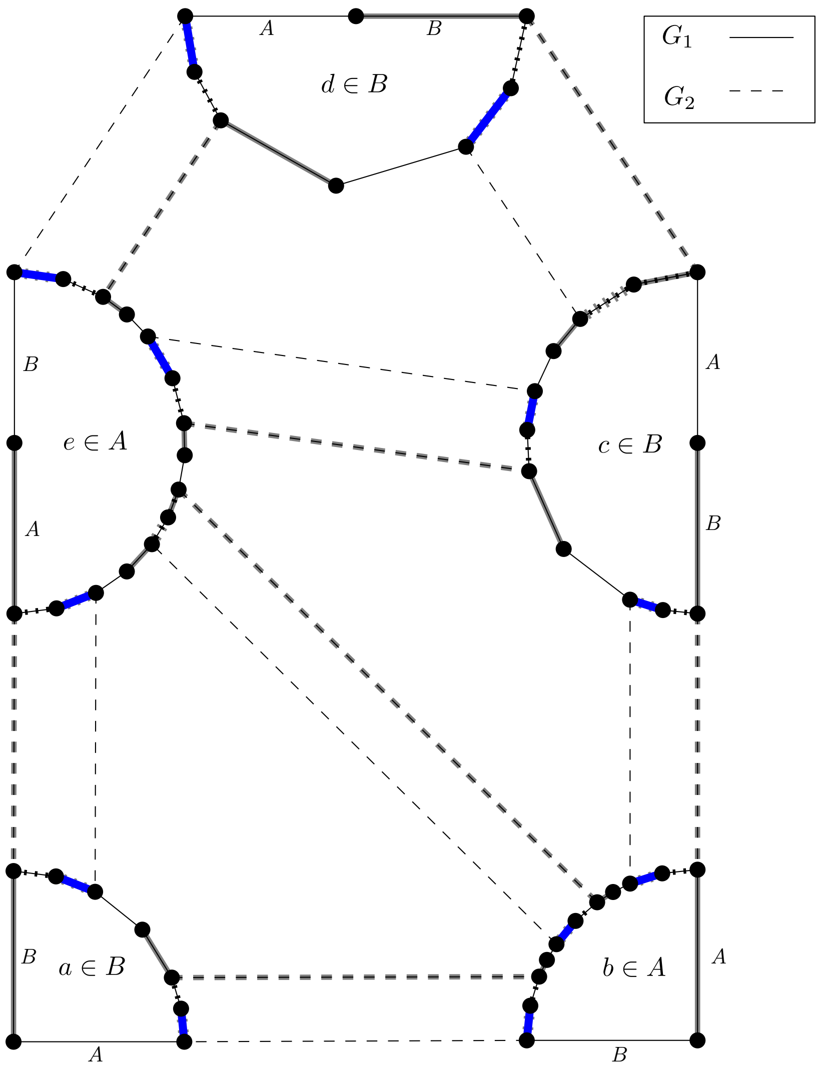


(b) Transformed To SimMatch Instance

Figure 2.2: Example transformation to SimMatch instance.



(a) Solved MaxCut Instance



(b) Solved SimMatch Transformation

Figure 2.3: Example transformation with (an) optimal solution.

inal MaxCut instance. This is transformed to the SimMatch instance in Figure 2.2(b), there you can see the five node cycles corresponding to a, b, c, d and e which make up the graph G_1. Between those, there are seven edge cycles which are part of G_2. The node cycles consist of A-edges and B-edges and in every node cycle there is one marked with A and one with B. Whether an edge is an A-edge or an B-edge can be deduced from those (A-edges and B-edges are alternating). In Figure 2.3 an optimal solution is shown for both instances. In 2.3(a) the solution for the MaxCut is illustrated, every node is set to partition A or to partition B. The bold (*blue*) edges have one endpoint with a node in set A and one in set B, these edges are part of the solution. The size of the solution is 5, two edges could not be taken. Below in 2.3(b) we see the corresponding solution in the SimMatch instance, in every cycle every second edges is matched, those matched edges are marked bold. The thickest (*blue*) edges lie within the intersection of the matchings of G_1 and G_2, these edges are part of both matchings. Every edge cycle, which corresponds to an edge of MaxCut where one endpoint is a node part of set A and one is a node part of set B, has two edges which lie in the intersection. All other edge cycles have only one edge lying in the intersection. This sums up to the solution size $5 \cdot 2 + 2 = 12$.

NP-hard Restrictions After we have seen that SimMatch is NP-hard we may ask if there are cases which are easy to solve. Or the other way around, we might ask which restrictions leave the problem NP-hard. As we will see, the restrictions on the graph structure can be relatively strong without violating NP-hardness for the respective class of graphs. However, if the number of different maximum matchings is fixed or bounded polynomially, it is possible to solve the problem in polynomial time, see Section 3.3. Here we will concentrate on restrictions which leave the problem NP-hard. We first list all observed restrictions which leave the problem NP-hard, after that we explain why this holds.

- G_1 and G_2 consist of cycles
- G_1 and G_2 are planar
- G_1 and G_2 are connected
- G_1 and G_2 consist solely of simple paths
- G_1 and G_2 each consist of a single tree

G_1 and G_2 consist of cycles Take a look back at the reduction in Section 2.1 and to Figure 2.2. The construction returns instances in which the graphs consist solely of cycles.

G_1 and G_2 are planar Similar to the class of graphs above, the reduction in Section 2.1 already returns instances in which the graphs are planar and therfore belong to this class.

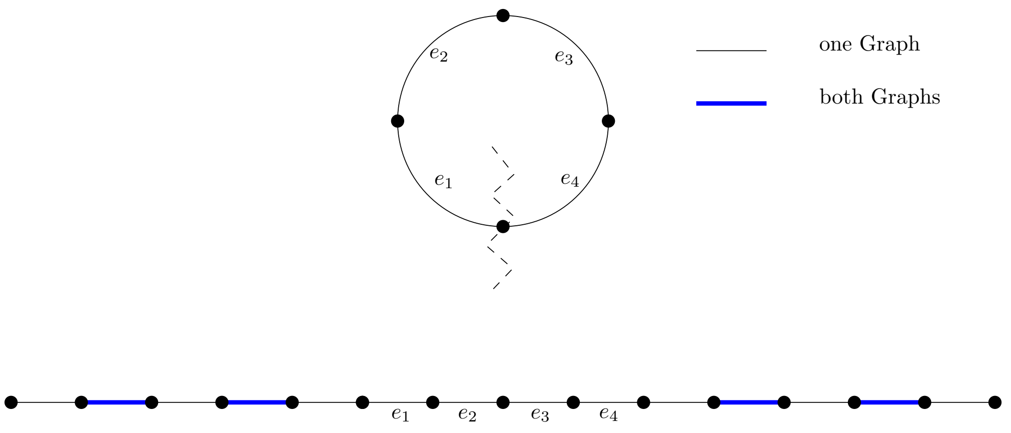


Figure 2.4: Converting A Cycle (*Top*) To A Path (*Bottom*)

G_1 and G_2 are connected To show that the problem remains NP-hard if each graph is connected we just modify the resulting graphs from the reduction in Section 2.1. Those graphs consist of simple cycles which have a perfect matching. Now we modify each graph. In both graphs, G_1 and G_2, we add two new nodes that are connected by a new edge. We now connect one of these nodes to a node of each cycle by inserting additional edges. It is clear that the resulting graph still has a perfect matching. By taking the newly inserted edge, all nodes can be matched. For a maximum matching it is necessary to match this edge, otherwise we would not be able to get a perfect matching, as we would miss one of the new nodes. This modification does not change anything except that there is one additional edge which is always matched, there is no further influence. That means that the reduction still works. The modification connects the graph, so this restriction leaves the problem NP-hard.

G_1 and G_2 consist solely of simple paths This is a quite strong restriction for our graphs. Nevertheless this restriction leaves the problem NP-hard. We will again modify our reduction from Section 2.1 in a way that it meets our restriction and still works. The transformed graphs from the reduction consist solely of cycles. We transform each cycle from graphs G_1 and G_2 to a simple path. An important property of the cycles was that there are exactly two different maximum matchings possible. In both possible matchings every second edge is included (alternating). In our transformation to a path, we want to retain this property for these edges, we name them cycle edges. Let m_c be the number of cycle edges, m_c is even. We transform each cycle to a path in which the cycle edges form the middle of the path ordered by their sequence within the cycle starting at one arbitrary edge. We enlarge this path by adding m_c plus one edges to each end of the path. The result for a cycle with four edges can be seen in Figure 2.4. Note that we also insert edges (=paths with length one) to the graph in which the transformed cycle is not located. These edges are part of the constructed path and therefore present in both graphs. Every second edge from the edges which where added at both ends from the path to the cycle edges is part of both graphs. Those edges are marked bold (and blue) in Figure 2.4.

Now we explain the idea of the constructed path. The number of nodes in this path is odd, therefore exactly one node is unmatched in a maximum matching (*none of the matched edges is incident to this node*). The unmatched node should *not* lie within the cycle edges, because if it does, the cycle edges are not matched alternatingly, and this is what we want to preserve. We try to force that the unmatched node is at one end of the path as then the cycle edges are matched alternatingly. At which end the unmatched node is located will decide which half of the cycle edges is matched. To assist letting an end note unmatched, we added the single edges to the other graph. These edges must always be included in a maximum matching in the other graph. With those, the matching intersection regarding only the transformed cycle is increased by $m_c/2$, since when we let the appropriate end node unmatched, the matched cycle edges stay the same and the intersection is enriched by the added single edges at one side of the path.

So note that with this selection, the size of the intersection of the matchings within one transformed cycle can be as high as $m_c/2 +$ size of intersection of the cycle. In contrast, when we let a node in the path unmatched which is incident to a cycle edge, the intersection can be at most $m_c/2$ since none of the single edges can be matched in both graphs and there are m_c cycle edges where at most half of them can be matched.

We will now use parts of the reduction from Section 2.1 and modify it in a way that the SimMatch instance has two graphs consiting solely of simple paths and we are still able to solve MaxCut with it. So let us consider an instance of MaxCut, $(G = (V, E), K)$. We apply the transformation used in the proof of Lemma 2.1.2 on this instance. This give us the SimMatch instance $T = (G_1 = \bigcup_{v \in V} \{\text{node cycle } c_v\}, G_2 = \bigcup_{\{u,v\} \in E} \{\text{edge cycle } c_{\{u,v\}}\}, K + |E|)$, where G_1 and G_2 consist solely of cycles. We now transform every single cycle to a path as described above, this results in $G_1' = \bigcup_{v \in V} \{\text{node cycle } c_v \text{ transformed to path}\}$ and $G_2' = \bigcup_{\{u,v\} \in E} \{\text{edge cycle } c_{\{u,v\}} \text{ transformed to path}\}$. Additionally we need to change $K + |E|$. As described above, we are able to get the same size for the matching intersection for the paths as for the cycles plus $m_c/2$ for every transformed cycle, where m_c is the number of edges in the cycle. Looking back to the MaxCut instance, we are able to observe that every edge in E generates 4 edges in 2 node cycle each and 6 edges in one edge cycle. This makes a total of 14 edges. That means, every edge in E increases our objective value by 7, so we change $K + |E|$ to $K + 8|E|$. So this is our SimMatch instance which consists only of paths and can be used to solve MaxCut: $T' = (G_1', G_2', K + 8|E|)$.

We now show that a "Yes"-instance of MaxCut is transformed to a "Yes"-instance of SimMatch and a "No"-instance is transformed to a "No"-instance.

First we show that a "Yes"-instance of MaxCut is transformed to a "Yes"-instance of SimMatch. So let the MaxCut instance be a "Yes"-instance. Then, as shown in the proof of Lemma 2.1.2, the transformation to a SimMatch instance T consisting of cycles produces "Yes"-instance, too. That means, the intersection of our matchings is at least $K + |E|$. As shown above, in our instance T' where the graphs consists solely of paths, we are then able to find maximum matchings in both graphs with an intersection of $K + 8|E|$: Just match the cycle edges in T' as in T, leaving one node unmatched at the end of a path where it is possible like described previously. This gives us an extra intersection of $m_c/2$ edges per path and thus $7|E|$ edges, over all. This leads us to an intersection of $K + 8|E|$, so T' is a "Yes"-instance.

Now consider the MaxCut instance to be a "No"-instance. Then, as stated in the proof of Lemma 2.1.2, the transformation to the SimMatch instance T produces a "No"-instance, that means, there are no matchings M_1 and M_2 for G_1 and G_2, respectively, so that $|M_1 \cap M_2| \geq K + |E|$. We now assume that T' is a "Yes"-instance and lead it to a contradiction by proving that T is a "Yes"-instance. If T' is a "Yes"-instance there are matchings for G_1 and G_2 with an intersection of at least $K + 8|E|$. With those matchings, we can construct a solution for T although T is a "No"-instance. Consider a path in the graph where the cycle edges are not matched alternatingly. This is only possible if a node incident to a cycle edge is unmatched. We change the unmatched node on this path so that the cycle edges are matched alternatingly. Considering the construction of the path, we see that the maximal matching intersection is at most $m_c/2$ if an edge incident to the cycle edges is unmatched. We now change the matched edges on that path so that the unmatched node is located at an arbitrary end of the path. The new matching intersection for that path is now at least $m_c/2$, that means the size of the intersection of the matchings did not decrease.

We apply this change to every path where the cycle edges are not matched alternatingly and we still have an intersection of at least $K + 8|E|$. As now all cycle edges are matched alternatingly, we can use the matching solution from T' for T by just dropping all non-cycle edges. Due to the construction, at most $7|E|$ edges in the matching solution from T' are non-cycle-edges (at most $m_c/2$ for each transformed cycle resulting in a total of $7|E|$ edges), that means that we have deduced maximum matchings for T with an intersection of at least $K + |E|$. It follows that T is a "Yes"-instance. This contradicts our assumption that T is a "No"-instance.

Hence, SimMatch is still NP-hard when both input graphs are restricted to solely consist of paths.

G_1 and G_2 each consist of a single tree Maybe surprising, the SimMatch even remains NP-hard if we restrict both graphs to be a single tree, i.e. connected graphs without cycles. One could have expected that there is some dynamic programming

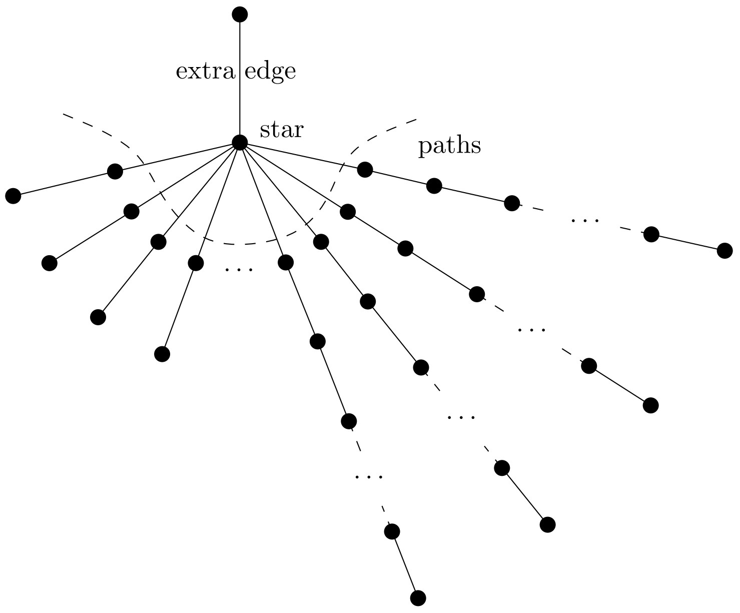


Figure 2.5: Connecting Paths To A Tree

approach as in general finding matchings in a tree is relatively easy. However, the problem remains NP-hard. To prove this, we modify the reduction above in which we restricted Graph G_1 and G_2 to consist solely of simple paths. We will transform each graph to a single tree without changing the essential structure. In each graph, we insert a new node. We also insert an edge between one endpoint of each path and the new inserted node. Additionally, we insert another new node and an edge, called extra edge, connecting the node with the other new nodes. So the added edges and nodes on its own form a star graph, where all edges but one share one node with one path. The transformation can be seen in Figure 2.5, the part below the dashed line consists of the paths, the part above are the newly inserted edges and nodes. The resulting graph is a tree.

Now consider a matching in such a graph. If the extra edge is included in such a matching, the possibilities for the rest of the matching do not differ at all from the matching in the graph consisting solely of paths. If the extra edge is not matched, one of its adjacent edges need to be matched. This edge is adjacent to a path with an even number of edges. It can not be adjacent to a path with an odd number of edges since then there would be an additional node unmatched. Instead of this edge, we could have also matched the extra edge without making any difference for the intersection of the maximum matchings. That means that the modification of the graph did not change the possibilities of the matchings within the paths nore did it modify the resulting intersection of the matchings. Hence, the previously presented reduction still works. Thus SimMatch is NP-hard even if both input graphs are trees.

2.2 Inapproximability

As seen in the last section SimMatch is NP-hard and unless P=NP there is no efficient way to find solutions that are close to the optimum. But we may still find a method to solve the general problem near-optimally. Unfortunately, this is restricted to some level as we will see in this section.
As the problem is NP-hard, we are interested in finding a PTAS (*polynomial-time approximation scheme*) which would give us a an ϵ-approximation for any fixed ϵ in polynomial time. Unless P=NP, there does not exist such an algorithm and even an approximation within a specific range is not possible as we will see in the following. To prove this, we will draw on the fact that MaxCut has an approximation gap [TSSW00].

Theorem 2.2.1 ([TSSW00])**.** *For any $\epsilon > 0$ it is NP-hard to approximate* MaxCut *to within $\frac{16}{17} + \epsilon$*

Based on this theorem, we can deduce a lower bound for SimMatch. This lower bound value will be different from $\frac{16}{17}$ as we will argue with the transformation of Section 2.1 and during the transformation the optimal objective value changes. The techniques used to prove the following inapproximability bound can be found in detail in Vazirani's book about approximation algorithms [Vaz03].

Theorem 2.2.2. *For any $\epsilon > 0$ it is NP-hard to approximate* SimMatch *to within $\frac{50}{51} + \epsilon$.*

Proof. Let I be a MaxCut instance and I' the corresponding SimMatch instance generated by the reduction described in Section 2.1. Here $OPT(I)$ and $OPT(I')$ denote the size of an optimal MaxCut and an optimal SimMatch solution, respectively. It is clear that for a class of MaxCut instances I and some fixed value k, it is NP-hard to decide whether $\text{OPT}(I) \geq k$ or $\text{OPT}(I) < \frac{16}{17}k$. If it was possible, we could find a better approximation than $\frac{16}{17}$ which would contradict Theorem 2.2.1. We will see that the reduction from an instance I of MaxCut to an instance I' of SimMatch preserves this gap, in the sense that it is also NP-hard to approximate SimMatch better than a certain factor as a constant factor approximation of SimMatch leads to a constant factor approximation of MaxCut due to the reduction. When we transform an instance I of MaxCut to an SimMatch instance I', $\text{OPT}(I)$ changes to $\text{OPT}(I') = \text{OPT}(I) + m$ where $m :=$ number of edges in I. Now we can not decide in polynomial time for all k whether $\text{OPT}(I') \geq k + m$ or $\text{OPT}(I') < \frac{16}{17}k + m$. If we were able to decide it for every k, we could use this to decide $\text{OPT}(I) \geq k$ or $\text{OPT}(I) < \frac{16}{17}k$ for every k in polynomial time, which contradicts Theorem 2.2.1.
We know that m and k are related, of course $m \geq k$ since a maximum cut can not be greater than the number of edges in a graph. Maybe less obvious it holds that $k \geq \frac{m}{2}$. The size of a maximum cut is always at least as large as half of the number of its edges: If we scan the nodes in arbitrary order and place each node to the set where fewer of the adjacent nodes are part of already yields a cut with a size of

at least $\frac{m}{2}$. Hence $k = \beta \cdot m$ with $\beta \in [\frac{1}{2}..1]$. That means we can not distinguish between $\text{OPT}(I') \geq k + m = (\beta + 1)m$ and $\text{OPT}(I') < \frac{16}{17}k + m = (\frac{16}{17}\beta + 1)m$. The factor within which we can not approximate is therefore

$$\frac{(\frac{16}{17}\beta + 1)}{(\beta + 1)}$$

This term is maximized for $\beta = \frac{1}{2}$ and therefore always lower than $\frac{50}{51}$. This implies our approximation gap. □

Note that this inapproximability gap remains for some of the restrictions on SimMatch presented at the end of Section 2.1. The 50/51-inapproximability remains for instances where the graphs consisting solely of cycles, instances with planar graphs and instances with connected graphs. The reduction for those instances is similar to the reduction used in Theorem 2.2.2, that means, during the reduction $OPT(I') = OPT(I) + m$ still holds (with I being an MaxCut instance, I' being the reduced SimMatch instance and m the number of edges in the MaxCut instance).
For the other two restrictions, the graphs consist solely of simple paths and the graphs consist of a single tree, the reduction gives us $OPT(I') = OPT(I) + 8m$. We are still able to give an inapproximability factor with the approach in Theorem 2.2.2, but the factor is weaker. For those instances it is NP-hard to approximate for every $\epsilon > 0$ within $288/289 + \epsilon$.

Due to Theorem 2.2.2, it is unlikely to find a PTAS for SimMatch and there even exists a gap which is preventing one from getting too close to the optimal solution. At the utmost we are able to get as close as $\frac{50}{51} = 0.9803$ for a set of instances but there is still a chance of getting a polynomial algorithm which provides us with, e.g., a 0.5-approximation. Even that may be impossible because there could exists a tighter lower bound. An interesting question would be if $\frac{50}{51}$ is tight. It is possible that this lower bound may be improved, for example by using another gadget or by sparing the detour on MaxCut. The approximation gap could also be improved by using a specific class of MaxCut instances for which Theorem 2.2.1 still holds but the optimum size of MaxCut is always a factor bigger than half of the edges. To show that this bound can not be considerably further improved we have to find an algorithm with an approximation factor as high as possible. This has not been done yet and at the moment it is unclear wether such an approximation actually exists. Hence the gap between 0 and $\frac{50}{51}$ is still open.

3 Algorithmic Approaches

In Chapter 2 we have seen that it is hard to compute an optimal solution for SimMatch. In this chapter we propose ways to find solutions for SimMatch instances despite their NP-hardness. For some instances it may be an option to search for the optimum solution even if there is the possibility of getting an exponential running time. When instances are small or the instances are not really "hard", this approach may be applicable. We will formulate SimMatch as an integer linear program (ILP) which can be solved with classic methods. An optimal solution of the ILP yields an optimal solution of SimMatch. We will also show how the ILP can be improved if we use the fact that we know the size of the maximum matchings. After this we will present a heuristic for solving the SimMatch problem. The heuristic may be more applicable in real world instances. We first introduce the heuristic and how it works. Then we show that, unfortunately, it is not an approximation and leaves us in general with no relative guarantee to the optimum solution. In the last section we present a class of SimMatch instances that can be solved in polynomial time. We also give an algorithm that solves instances of this class polynomially. This class consists of instances where (at least) one graph has a polynomial bounded number of different maximum matchings. In general the number of different maximum matchings is exponential and therefore this algorithm is only applicable for special cases.

3.1 ILP formulation of SimMatch

A straightforward solution for SimMatch is modeling it as an integer linear programming (ILP) problem. This is a fast and intuitive way to obtain a solution which is optimal. There are a lot of techniques and programs available for solving ILPs, often only requiring a reasonable amount of time. However, solving ILPs is known to be NP-hard and is therefore in general not efficient, unless P=NP. We now give the constraints and variables of the ILP-formulation based on the SimMatch instance (G_1, G_2, K), with $G_2 = (V_2, E_2), G_1 = (V_1, E_1)$ and E_1, E_2 sharing some edges.

For every edge $\{i, j\}$ in E_k for $k \in \{1, 2\}$ let $X_{\{i,j\},k}$ be the corresponding binary variable. If $X_{\{i,j\},k}$ is 1, this means that the edge is part of the matching in graph G_k.

$$X_{\{i,j\},k} \in \{0, 1\} \qquad k \in \{1, 2\}, \{i, j\} \in E_k, i < j$$

To ensure that the matching is valid in G_1 and G_2, we add the following constraints for every node $i \in V_k$:

$$\sum_{\{i,j\} \in E_k} X_{\{i,j\},k} \leq 1 \qquad\qquad k \in \{1,2\}, i \in V_k$$

We also need to measure the size of the solution, that is, the number of edges which are matched in both graphs. So for all edges $\{i,j\} \in E_1 \cap E_2$ that exist in both graphs, we introduce a new binary variable $B_{\{i,j\}}$ (B='Both') that should only be 1 if the corresponding edge is matched in both graphs.

$$B_{\{i,j\}} \in \{0,1\} \qquad\qquad \{i,j\} \in E_1 \cap E_2, i < j$$

With the following constraints, $B_{\{i,j\}}$ can only be 1 if the corresponding edge is matched in G_1 and G_2, that means if $X_{\{i,j\},1}$ and $X_{\{i,j\},2}$ are 1.

$$X_{\{i,j\},1} + X_{\{i,j\},2} - 2 \cdot B_{\{i,j\}} \geq 0 \qquad\qquad \{i,j\} \in E_1 \cap E_1, i < j$$

Maximizing the sum over all $X_{\{i,j\},k}$ would already induce a valid maximum matching for G_1 and G_2. However, among all these matchings in G_1 and G_2, we want to find the matchings with the biggest intersection, which is measured by the variables $B_{\{i,j\}}$. To get a valid solution, we need to take care that the sum over all edge variables $X_{\{i,j\},k}$ is maximum independent of any reward given for taking the same edge in both graphs with variable $B_{\{i,j\}}$. After it is ensured that the sum over all $X_{\{i,j\},k}$ variables is maximum, the rewards from variable $B_{\{i,j\}}$ should be maximized. We can achieve this by taking care that all rewards from $B_{\{i,j\}}$ together are not as good as increasing a single $X_{\{i,j\},k}$ variable by one. The sum of all $B_{\{i,j\}}$ is at most $\lfloor |V_1 \cap V_2|/2 \rfloor$. This leads us to the following sum which has to be maximized by the ILP solver:

$$\left(\left\lfloor \frac{|V_1 \cap V_2|}{2} \right\rfloor + 1 \right) \cdot \sum_{\substack{k \in \{1,2\} \\ \{i,j\} \in E_k, i<j}} X_{\{i,j\},k} + \sum_{\substack{\{i,j\} \in E_1 \cap E_2 \\ i<j}} B_{\{i,j\}} \qquad \text{(basic)}$$

As already said, the objective function ensures that the matchings in both graphs are maximum. This can also be achieved with a constraint if we already know the size of the maximum. We now give an alternative ILP-formulation which has one more constraint and a different objective function. If we insert a constraint which forces the maximum size of the matchings, the maximum size does not need to be ensured by the objective function. Instead, the constraint directly ensures that the right number of $X_{\{i,j\}}$ variables is set to 1. In the experimental studies only this improved version is used, because it can be solved much faster by the ILP-solver than the basic version. However, we must determine the maximum size of the matchings. This can be done polynomially in $\mathrm{O}(\sqrt{n}m)$ time [MV80, Vaz94]. Let $\mathrm{M}(G_1)$ and $\mathrm{M}(G_2)$ be the size of a maximum matching in G_1 and G_2, respectively. For the improved version we replace the basic objective function from above with the following inequality

$$\sum_{\{i,j\}\in E_k, i<j} X_{\{i,j\},k} = \mathrm{M}(G_k) \qquad k \in \{1,2\} \quad \text{(improved)}$$

and a new objective function which has to be maximized:

$$\sum_{\{i,j\}\in E_1\cap E_2} B_{\{i,j\}} \qquad \text{(improved)}$$

In both versions we can directly obtain a solution for the SimMatch instance after solving the ILP:

$$M_1 := \{\{i,j\} \in E_1 \mid X_{\{i,j\},1} = 1\}$$

$$M_2 := \{\{i,j\} \in E_2 \mid X_{\{i,j\},2} = 1\}$$

The ILP enables us to get an optimal solution for SimMatch. However, solving the ILP may require exponential running time. In the next section we present a heuristic for SimMatch with running time $\mathrm{O}(n^2 m + n^3 \log n)$.

3.2 Heuristic

We now present and analyze a heuristic for the SimMatch problem. A heuristic is a method that tries to find a solution to a problem but does not necessarily give any formal guarantees on the running time of the algorithm or on the quality of the solution. As we will see, our heuristic runs in polynomial time but is not able to provide a relative guarantee compared to the optimal solution.
The algorithm works in turns where it fixes the matching in one graph and optimizes the matching in the second graph to fit better with the currently fixed matching of the other graph. The best fitting matching is a matching where the following properties are maximized, the first named properties with higher priority: number of edges in the matching, number of edges in the matching shared with the fixed matching from the other graph, number of edges in the matching

shared with the other graph. After one turn, the two graphs switch their roles. This process is iterated until no further improvement can be done. Now we give a more formal description of the heuristic.

The heuristic will do the following steps with $G_1 = (V_1, E_1)$ and $G_2 = (V_1, E_1)$ as input: At first, set M_1 and M_2 to be the empty set, both representing matchings in G_1 and G_2, respectively. Now recalculate M_1 in G_1, but take a maximum matching where the number of matched edges also appearing in the matching of G_2 (which is fixed at the moment and may contain edges in later steps) is maximum. With second priority, among all those matchings take only those where the number of the matched edges which are also in G_2 (not caring if the edge is matched in G_2) is maximum. Before we will explain how to achieve this, we state the constraints more formally:
Set M_1 to a maximum matching such that for all other maximum matchings M_1' of G_1 both of the following hold:

$$|M_1 \cap M_2| \geq |M_1' \cap M_2| \tag{3.1}$$
$$|M_1 \cap M_2| = |M_1' \cap M_2| \Rightarrow |M_1 \cap E_2| \geq |M_1' \cap E_2| \tag{3.2}$$

Now we show how to find such a matching. We use a matching where some edges have higher priorities, this is possible by using weighted matchings. We give a short introduction to weighted matchings: In a weighted matching, a number is assigned to every edge (its weight) by a function $w : E \to \mathbb{N}_0$. A maximum matching M in the weighted case is a matching which maximizes $w(M) = \sum_{e \in M} w(e))$. Such a matching can be computed in $\mathrm{O}(nm + n^2 \log\, n)$ time [Gab90].
To obtain the desired matching we create a weighted matching instance whose solution satisfies the two properties above. In order to accomplish this, we set the edge weights $w : E_1 \to \mathbb{N}_0$ as follows[1]. Let $k := \lfloor |V_1 \cap V_2|/2 \rfloor$.

$$w(e) = \begin{cases} k(k+2) + k + 3 & \text{if } e \in M_2 \\ k(k+2) + 2 & \text{if } e \in E_2 \setminus M_2 \\ k(k+2) + 1 & \text{if } e \notin E_2 \end{cases} \tag{3.3}$$

Lemma 3.2.1. *A maximum weight matching M_1 in G_1 with weight function w is a maximum matching and satisfies requirements (3.1) and (3.2).*

Proof. First we want to state that the number of edges which can be matched in $E_1 \cap E_2$ is at most k, since in every matching the number of matched edges is at most half of the number of nodes. For a valid weighted matching result M_1, the weight function w ensures the following three properties with decreasing priority:

(1) The number of edges in M_1 is maximum: If we took one edge less, our objective sum would be decreased by at least $k(k+2)+1$ for this edge. This loss can not be

[1]lowering the weights (a little) is possible, but it seems not to be a significant improvement

compensated with higher weighted edges, as there can be taken at most k edges with higher weights and each of those can at most return an extra of $k+2$, resulting in $(k+2)k$ which is not enough. This ensures that M_1 is a maximum matching.

(2) The number of edges in M_1 which are also in M_2 is maximum: If we took one edge less which exists in M_2, that would be a loss of at least $k+1$. This can not be compensated by changing some edges in $E_1 \setminus E_2$ to higher weighted edges in $E_1 \cap E_2 \setminus M_2$, as there can be taken at most k edges changed to higher weights and those weigh only 1 more. This ensures requirement (3.3).

(3) The number of edges in M_1 which are also in E_2 but not in M_2 is maximum: Taking one edge less which exists in E_2 but not in M_2 would yield at least 1 less compared to other edges.

This corresponds exactly to our previously required properties and therefore we can find a compatible matching M_1 by solving the stated weighted matching. □

Above, we computed a new matching M_1 while fixing the matching M_2. Now we do the same the other way round. M_1 is fixed and we compute M_2 as described above (M_1 and M_2 switch roles). After this, we check the size of the solution $|M_1 \cap M_2|$ and compare it to the value before the recomputation of M_1 and M_2. If the size did increase, we repeat the recomputation of M_1 and M_2 until the size does not increase any more. During one step, $|M_1 \cap M_2|$ can not decrease as the newly computed matching M_1 will not have fewer edges which are also in M_2. This is due to the fact that the old M_1 is also a valid solution and therefore the new matching has at least as many common edges with M_2 as the previous one. The same argument holds for M_2 and therefore each step either increases the size of the solution or levels with it. If the size does not change the algorithm stops and returns M_1 and M_2. So the maximum number of iterations is bounded by the size of the optimum solution which is bounded by $\min(n_1, n_2)$, where $n_1 = |V_1|$ and $n_2 = |V_2|$. The result may depend on which matching is computed first. Therefore we start the process once with G_1 and start it afterwards with G_2. Then we take the best of both results. The complete algorithm Sim-Match-Heuristic is listed in pseudo code below.

The running time for the whole heuristic is determined by the computation of the weighted matching during the iterations. The weighted matchings can be computed for each Graph G_1 and G_2 in $O(nm + n^2 \log n)$, see [Gab90]. As mentioned above, the number of iterations is at most $\min(n_1, n_2)$, so the worst case running time of the heuristic is $O(\min(n_1, n_2)(n_1 m_1 + n_1^2 \log n_1 + n_2 m_2 + n_2^2 \log n_2)) = O(\min(n_1, n_2)(nm + n^2 \log n)) \subseteq O(n^2 m + n^3 \log n)$, with $n = |V_1| + |V_2|$ and $m = |E_1| + |E_2|$.

```
SIM-MATCH-HEURISTIC($Graph : G_1, G_2$)
 1  $f \leftarrow 1$                ▷ $first$
 2  $s \leftarrow 2$                ▷ $second$
 3  $M'_1, M'_2 \leftarrow \emptyset$
 4  for $i \leftarrow 1$ to 2
 5      do $M_f, M_s \leftarrow \emptyset$
 6         repeat
 7                 $s \leftarrow |M_f \cap M_s|$
 8                 for $j \leftarrow 1$ to 2
 9                     do $k \leftarrow \lfloor |V_1 \cap V_2|/2 \rfloor$
10                        for each $e \in E_f$
11                            do if $e \notin E_s$ then $w[e] \leftarrow k(k+2)+1$
12                               if $e \in E_s \setminus M_s$ then $w[e] \leftarrow k(k+2)+2$
13                               if $e \in E_s \cap M_s$ then $w[e] \leftarrow k(k+2)+k+3$
14                        $M_f \leftarrow$ maximum weighted matching in
                                   $G_f$ with weightings $w$
15                        exchange $f \leftrightarrow s$
16           until $s = |M_f \cap M_s|$
17         if $|M_f \cap M_s| \geq |M'_1 \cap M'_2|$
18           then $M'_1 \leftarrow M_f$ and $M'_2 \leftarrow M_s$
19         exchange $f \leftrightarrow s$
20  return $(M'_1, M'_2)$
```

The algorithm above is not an approximation but a heuristic. This implies that in general we can not give any general guarantee on the quality of the heuristic solution in comparison to the optimum solution. We will give an example where the optimum solution has size 1 and the heuristic may yield a solution of size 0. This example can easily be extended by simply duplicating the graphs, leading us to an optimal solution of size n and a heuristic returning 0. Thus, the quality of our heuristic can be arbitrarily bad compared with the optimum. The idea of the example has been attained by a random test case during the experimental tests where the heuristic returned 0 instead of an optimal 1. We extracted the essential structure and simplified the test case in a way that it becomes human readable preserving the difference between heuristic and optimum. The example graph with its optimum solution and heuristic solution is shown in Figure 3.1.

First take a look at the plain graph on the left side. Both graphs, G_1 and G_2, share the same ten nodes, graph G_1 consists of the solid edges, G_2 of the dashed edges. They share three edges in the middle part. In (b) a matching for both graphs is illustrated, the bold marked edges represent the matched edges. This is an optimal solution for SIMMATCH as we have three matched edges in both graphs which satisfies an maximum matching and we have got one common matched edge, which cannot be further improved. Any attempt to match two edges in common will not allow us to take a maximum matching for both graphs.

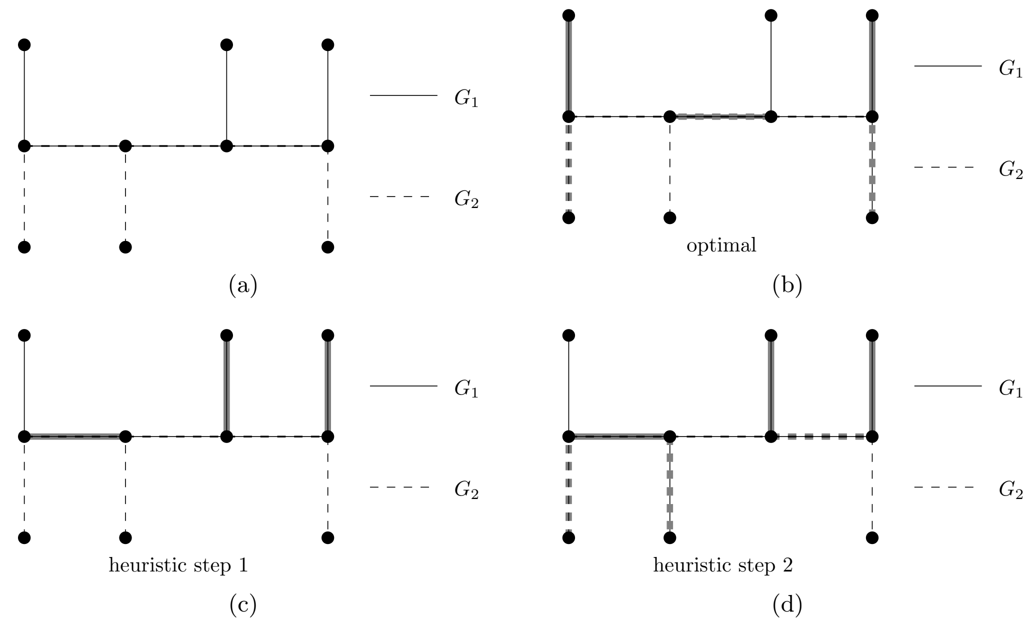


Figure 3.1: Heuristic Worst Case (a) base graph (b) optimal solution (c) heuristic step 1 (d) heuristic step 2

In (c) and (d) a potential run of the heuristic is illustrated by an intermediate result in (c), where G_1 is matched, and by the final in (d), where additionally G_2 is matched. In step 1 the maximum matching of G_1 includes one edge which is also present in G_2, more is not possible. In step 2 a maximum matching in G_2 can not include any edge that is currently matched in G_1. The given maximum matching maximizes the number of edges which also exist in G_1 to one. The heuristic algorithm would stop and return those two matchings, as the size $|M_1 \cap M_2|$ is equal to zero. It does not matter in which graph we start to compute a matching as both graphs are symmetric, so starting with G_2 would yield the same result. Note that, however, it is not mandatory that the heuristic finds the solution with an empty intersection, it can also find the optimal solution if the found matching in the first step is equal to the optimal matching.

As stated previously, the example can be sized up by duplicating G_1 and G_2. The duplicated graphs can even be connected as seen in Figure 3.2, just let the middle edge of the newly added three edges be matched in the heuristic and for the optimum solution. Summarized, we can create instances where the optimum solution is n and the heuristic returns zero. This means our heuristic provides us with no guarantee in theory. However, in Chapter 4 we will see that it performs quite well in practice.

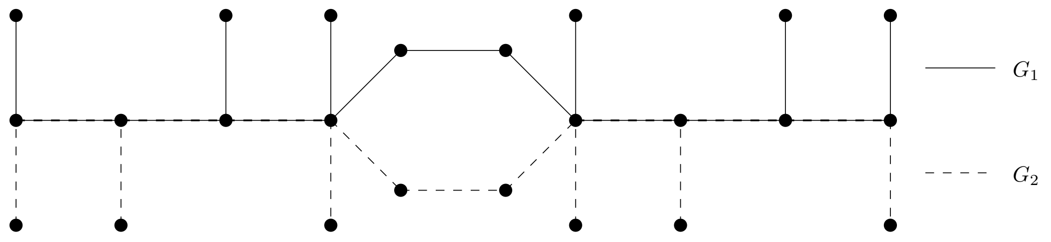


Figure 3.2: G_1 and G_2 sample doubled with connection

3.3 Polynomial Time Solvable Instances

In this section we persue the question, whether there are classes of instances that can be solved efficiently. We present and analyse a restriction where the problem can be solved in polynomial time. Section 2.1 contains a list of restrictions that leave SimMatch NP-hard.

SimMatch instances in which the number of possible maximum matchings in G_1 or G_2 is bounded polynomially can be solved in polynomial time. Only one of those graphs must have the property, the other one can be arbitrary. Graphs in which the number of maximum matchings is bounded polynomially include for example graphs consisting of a single path or a single cycle. Note that there are other trivial cases in which SimMatch is easy to solve, like both graphs do not share a common edge or share only a constant number of edges (just try to match every subset of the shared edges).
We first show that paths and cycles indeed only have a polynomial number of maximum matchings.
If the graph is is a path there are two cases: The path consists of an odd number or an even number of edges. If we have an odd number of edges there is only one possibility for a maximum matching: match the two outer edges and then every second edge resulting in a perfect matching. If we have an even number of edges one node will remain unmatched. This unmatched node determines the rest of the matching as all other nodes need to be matched. That means there exist as many maximum matchings as nodes.
If the graph is a cycle we have a similar situation. An even number of edges results in a perfect matching, there are two of them. An odd number of edges leaves one node unmatched, here we have got as many possible maximum matchings as nodes.

Now we assume that one of the graphs of our input has only a polynomial number of maximum matchings. In our algorithm we use this fact to iterate over all those maximum matchings and for each of them find the best fitting maximum matching in the other graph. Among all these maximum matching pairs, we will select the pair with the largest intersection. Before we present this algorithm, we need to show that it is possible to iterate over all maximum matchings in a graph in polynomial time if the number of maximum matchings is bounded polynomially. It has been shown that it is possible to enumerate all maximum matchings in *bipartite* graphs in O(n) time per matching [Uno97], but apparently there is no known algorithm for efficiently enumerating all maximum matching in *general* graphs.

We present now an algorithm, Enumerate-Maximum-Matchings, that enumerates all maximum matchings in a graph in $\mathrm{O}(\mathcal{MM}(G) \cdot \sqrt{n}m^2)$ time, where $\mathcal{MM}(G)$ denotes the number of maximum matchings in the graph G. This helper algorithm will be used in the Sim-Match-One-Limited-Graph algorithm which solves SimMatch in polynomial time for instances in which one graph has a polynomially bounded number of maximum matchings. Enumerate-Maximum-Matchings uses branch and bound techniques to efficiently enumerate all possible maximum matchings. It iterates over all edges in a fixed order while storing a set of edges which will be completed by remaining edges to form a maximum matching. If a maximum matching is still possible when adding the current observed edge to the set, it adds this edge and continues this path, if a maximum matching is still possible when omitting the current observed edge, it removes this edge and also continues this path (branching). If a maximum matching is not possible any more when adding or omitting the observed edge, this path is pruned and no longer followed. Whether or not a maximum matching is still possible with an already taken set of edges and a remaining set of possible edges can be checked easily with a general maximum matching algorithm, which can be computed in $O(\sqrt{n}m)$ time [MV80, Vaz94]. We simply check whether the size of a maximum matching in the set of possible edges (without edges adjacent to edges of the already taken set) plus the number of edges in the already taken set has as many edges as a maximum matching in the whole graph. We only follow a path if a maximum matching is possible within that branch, the length of a path is always m, where $m = |E|$. Hence our search includes only paths to leaves which represent maximum matchings, every step in a path needs at most $\mathrm{O}(\sqrt{n}m)$ time to check whether this path is still admissible, every path has length m which leads to an overall running time of $\mathrm{O}(\mathcal{MM}(G) \cdot \sqrt{n}m^2)$.

The algorithm Enumerate-Maximum-Matchings operates on a graph $G = (V, E)$ in which the edges are numerated from 1 to m, $E = \{1, \ldots, m\}$. The function MMStillPossible(M, k) in lines 3 and 5 returns true, iff a maximum matching in G is still possible by completing the set M with edges of $\{k, k + 1, \ldots, m\} \setminus \{\text{edges adjacent to } M\}$. Thus recursive branching in line 4 and 6 is only done if there is still at least one maximum matching solution possible in this path. Invoking the function with Enumerate-Maximum-Matchings($\emptyset, 1$) returns a set including all maximum matchings in the graph G.

```
ENUMERATE-MAXIMUM-MATCHINGS($set : M,\ int : k$)
1  if ( $k > m$ ) then return $\{M\}$
2  $res \leftarrow \emptyset$
3  if ( MMSTILLPOSSIBLE($M \cup \{e_k\}, k+1$) )
4      then $res \leftarrow res \cup$ ENUMERATE-MAXIMUM-MATCHINGS($M \cup \{e_k\}, k+1$)
5  if ( MMSTILLPOSSIBLE($M, k+1$) )
6      then $res \leftarrow res \cup$ ENUMERATE-MAXIMUM-MATCHINGS($M, k+1$)
7  return $res$
```

With this helper algorithm which enumerates all maximum matchings in polynomial time per matching, we can now describe the algorithm solving SIMMATCH in polynomial time in instances, in which the number of maximum matchings in one graph is bounded polynomially. Below, we list the pseudo code of the algorithm. Without loss of generality, G_1 has at most a polynomial number of maximum matchings.

```
SIM-MATCH-ONE-LIMITED-GRAPH($LimitedGraph : G_1, ArbitraryGraph : G_2$)
 1  $M_1, M_2 \leftarrow \emptyset$
 2  $mmset_1 \leftarrow$ ENUMERATE-MAXIMUM-MATCHINGS($\emptyset, 1$) in $G_1$
 3  for each matching $M_1'$ in $mmset_1$
 4        do for each edge $e$ in $E_2$
 5               do if $e \in M_1'$
 6                     then $w[e] \leftarrow |E_2| + 2$
 7                     else  $w[e] \leftarrow |E_2| + 1$
 8           $M_2' \leftarrow$ maximum weighted matching in $G_2$ with weights $w$
 9           if $|M_1' \cap M_2'| \geq |M_1 \cap M_2|$
10              then $M_1 \leftarrow M_1'$
11                   $M_2 \leftarrow M_2'$
12  return $(M_1, M_2)$
```

The algorithm iterates over all possible matchings in G_1. Enumerating all maximum matchings in line 2 requires only polynomial time since enumerating all maximum matchings is possible in $O(\sqrt{n}m^2)$ time per matching as shown previously. For each of those matchings M_1' it searches in G_2 for a maximum matching M_2' which shares a maximum number of edges with M_1'. This is done via weighting the edges. Maximum weighted matchings ($\sum_{e \in M} w(e)$ maximized) can be computed in O($nm + n^2$log n) time [Gab90]. Each edge in E_2 has weight $|E_2| + 1$, if the edge is also in matching M_1' it is weighted $|E_2| + 2$. This ensures that we get a maximum cardinality matching in G_2. Omitting edges can not compensated by taking fewer edges with higher weight as there are only $|E_2|$ of them and missing one decreases the weight by at least $|E_2| + 1$, taking the heavier weighted edges only gains one extra weight. Thus we get the matching with the highest value and therefore among all maximum matchings in G_2 the matching M_2' with the biggest intersection with M_1'. Among all possible maximum matchings M_1' we take the

pair (M_1', M_2') with the biggest value of $M_1' \cap M_2'$. This yields an optimal solution for SimMatch.
Each step of the for-loop in line 4 of Enumerate-Maximum-Matchings requires $O(nm + n^2 \log\, n)$ time due to computing the maximum weighted matching, resulting in a running time for the whole for-loop of $\mathrm{O}(\mathcal{MM}(G_1) \cdot (nm + n^2 \log\, n))$, where $\mathcal{MM}(G_1)$ denotes the number of maximum matchings in G_1. Enumerating all maximum matchings in line 2 dominates the running time of the algorithm, therefore the overall running time of Sim-Match-One-Limited-Graph is in $\mathrm{O}(\mathcal{MM}(G_1) \cdot \sqrt{n} m^2)$. Clearly, if the number of maximum matchings in G_1 is bounded polynomially, so is the running time of the algorithm.

4 Experimental Evaluation

In the previous chapter we presented some algorithmic approaches for SimMatch and analyzed the complexity of these approaches. However, the performance on practical instances is often quite different. In this chapter we will measure the practical performance on instances that are likely to appear in practical applications. We will test two approaches which have been introduced in the last chapter:

- ILP
- Heuristic

The ILP approach has been introduced in Section 3.1: First, we formulate the SimMatch instance as an ILP, then we solve the ILP which gives us an optimal solution for SimMatch. Since SimMatch is NP-hard we can not hope to obtain an optimal solution in polynomial time for all instances. This is also reflected in the experiments, for large instances the ILP reaches the timelimit and does not return within reasonable time. Here, the ILP does not provide us with an optimal solution and are not able to compare it with the heuristic.
The heuristic has been presented in Section 3.2, its running time is polynomial ($\mathrm{O}(n^2m + n^3\log\, n)$) but it does not provide us with a theoretical guarantee on the solution. So in some instances the objective value differs from the optimum obtained from the ILP.

Performing one test case consists of several steps:

1. Creating the instance
2. Running the algorithm
3. Analysing the result

An important and non-trivial part of the experiments is the creation of the instances, which consist of two graphs. The results heavily depend on the type of the input instance. As the SimMatch instances are quite specific, there does not exist a library with appropriate test cases (to the best of our knowledge), so we have to create them ourselves. In Section 4.1 we will discuss two graph generators. In Section 4.2 we will describe the rest of the framework for the experiments (running the algorithm), this includes the used tools, some hints on the implementation and parameters measured during the test.
Finally, in Section 4.3, we analyse the results of the experiments and give some interpretation. The results include running time of the heuristic and ILP, the quality (size) of the solution and some parameters of the heuristic.

4.1 Graph Generator

When testing the ILP and the heuristic, the sample generation is an important part. For some problems there exist collections of instances which are used for benchmarking. This makes it possible to compare different algorithms more objectively. Unfortunately, we could not find any collection of instances for matchings in graphs in general and we believe there are no appropriate instances available. Thus, we create our own samples. We describe the generation as clear as possible so that the reconstruction of equivalent samples is possible. Every SimMatch instance consists of two graphs. Our sample generation can be split into two different steps. The first step is creating one of the two graphs with a random algorithm described later. The second step is creating the other graph. The other graph could be created independently from the first graph (with the same random algorithm) with a random identification between the nodes of both graphs.

However, considering real problems and application, it is more likely that the second graph depends somehow on the first graph, e.g. a network that changes over time. We will model the dependency by modifying the first graph, resulting in the second graph. That means, we add edges and/or remove edges. The modification can vary from changing only a small amount (e.g. constant) of edges up to a large amount (e.g. half of the edges). This will result in more interesting and more difficult SimMatch instances compared to independent generation of the graphs. If the graphs in the SimMatch instance have a certain amount of similar edges, the instance is distinct from a pure maximum matching problem. To get this amount of similar edges using two independently created graphs, the graphs need to be quite dense. A high density allows taking a number of different edges, which satisfy the maximum matching condition, making it easier to select edges increasing the intersection of matched edges between both graphs. In this setting our heuristic almost always finds the optimal solution as we will see in the results.

First Graph: We use two different methods to create the first graph.

- Erdős-Renyi-Generation

- Planar-Graph-Generation

We use the Erdős-Renyi-Generator from the *Boost Graph Library* and the Planar-Graph-Generator from the *LEDA*® library. The Erdős-Renyi-Generator works as follows: Given n and m describing the number of nodes and edges, the generator creates a graph $G = (V, \emptyset)$ with n nodes. Then, m edges are inserted, equally likely for every pair or nodes $u, v \in V$ with $u \neq v$. The generation listed in pseudo code looks as follows:

```
Erdős-Renyi-Generator($n, m$)
1 $V \leftarrow \{1, 2, \ldots, n\}$
2 $E \leftarrow \emptyset$
3 while $|E| < m$
4       do $u, v \leftarrow$ random nodes of $V$
5          if $u \neq v$ and $\{u, v\} \notin E$
6               do $E \leftarrow E \cup \{\{u, v\}\}$
7 return $G = (V, E)$
```

Creating planar graphs and meeting certain requirements is a difficult problem on its own. As for our test cases the quality and the distribution of the generated planar graphs is not crucial, it is sufficient to use the planar-graph-generator from the *LEDA*® library. We just want to note that the *LEDA*® planar graph generator is not capable of creating every possible planar graph. Also, it prefers certain types of planar graphs. The generator is listed in pseudo code Planar-Graph-Generator.

```
Planar-Graph-Generator($n, m$)
 1  $V \leftarrow \{1, 2, \ldots, n\}$
 2 if $n = 1$
 3     then $E \leftarrow \emptyset$
 4 if $n = 2$
 5     then $E \leftarrow \{\{1, 2\}\}$
 6 if $n \geq 3$
 7     then create a triangle and compute embedding
 8           for $i \leftarrow 4$ to $n$
 9                do choose one face in $G = (E, V)$ randomly and insert $i$
                      into this facet, triangulate $G$ afterwards (add 3 edges)
10 while $|E| > m$
11       do $E \leftarrow E \setminus \{$random edge of $E\}$
12 return $G = (V, E)$
```

The generator first creates a maximal planar graph with n nodes by iteratively inserting a node into an already existing random facet and adds three edges to this node for triangulation afterwards (Lines 1-9). After this, it randomly deletes edges until m edges remain and return the graph.

Second Graph: As stated previously, beside creating the second graph with the same method as the first one, we create it by editing the first graph. We use the following modifications:

add k edges *Adding k edges* to a graph is done by choosing a random edge equally under all edges which are not yet present in the graph and add it until we have added k edges.

add k edges and delete k edges *Add and delete edges* is done similar, first add k edges, then delete edges by picking random edges from the present edges in

Table 4.1: Varied Parameters For Creating The Instances

First Graph			*Second Graph*	
Planar Erdős-Renyi	n	m	same method as first graph	
			delete k edges add and delete k edges k-distance	k

Table 4.2: Parameters n, m and k

n	m	k
$\{20\}$	Planar: $\{10, \ldots, 54\}$ Erdős-Renyi: $\{10, \ldots, 190\}$	$\{10, \log m, \sqrt{m}, \lfloor m/2 \rfloor\}$
$\{50, 100, 150, \ldots, 10^4\}$	Planar: $\{n, 2n, \lfloor 5n/2 \rfloor\}$ Erdős-Renyi: $\{2n, \lfloor n\log n \rfloor, \lfloor n\sqrt{n} \rfloor\}$	

the graph. The number k will be chosen dependent on the number of edges present in the graph, it will vary from a constant up to $n\sqrt{n}$.

k-distance, preserving planarity on planar graph The *k-distance* modification is a little different: Randomly choose k-times two nodes u, v with $u \neq v$ of V. If $\{u, v\} \in E$ remove $\{u, v\}$ from E, otherwise add $\{u, v\}$ to E. If the first graph is planar, only add $\{u, v\}$ if it does not violate planarity, if it does, do nothing and retry. Note that this can be very expensive as it often fails to insert an edge. We check whether the graph remains planar when adding $\{u, v\}$ by using the *boyer-myrvold-planarity-test* of the *Boost Graph Library* which runs in linear time with respect to the number of nodes.

independent creation The second graph is created with the same method as the first graph. The identification of the nodes is done randomly.

In Table 4.1 you see a summary of all varied parameters for the creation of the instances. The values for the parameters n, m and k are listed in the additional Table 4.2, we also describe them here: We created test cases with $n \in \{20\} \cup \{50, 100, 150, \ldots, 10'000\}$. We set m depending on n and the graph type of the first graph (note that for planar graphs: $n \geq 3 \Rightarrow m \leq 3n - 6$). If $n = 20$ we set m to $m \in \{10, 11, 12, \ldots, 190\}$, if the first graph is planar only to $m \in \{10, 11, 12, \ldots, 54\}$. If $n \neq 20$, we used $m \in \{n, 2n, \lfloor 5n/2 \rfloor\}$ for planar graphs, for Erdős-Renyi graphs we used $m \in \{2n, \lfloor n\log n \rfloor, \lfloor n\sqrt{n} \rfloor\}$. The size k for the modification of the second graph was varied dependent on m: $k \in \{10, \log m, \sqrt{m}, \lfloor m/2 \rfloor\}$.

4.2 Framework

Our framework is used to test the performance of the ILP-approach and the Heuristic-approach on SimMatch instances. To this extent we measure the running time and the quality for an implementation of the ILP and the heuristic on an appropriate set of random instances. The creation of the instances is described in Section 4.1. In this section we describe the rest of the framework.
The whole framework runs on a *Linux* platform (all used tools are also available for *Windows*). For the implementation we use the programming language *C++* and the *C++ standard library*. Beside the basics provided by the *C++ standard library*, we use the following more specific tools:

1. *Boost Graph Library (1.35.0)* for handling graphs, see [SL02]
2. *C++* algorithm library *LEDA® (5.0.1)*, see [MN99]
3. *ILOG CPLEX® (11.2.0)* ILP-solver [1]

Within the program the graphs are stored in the *boost* graph format to support reusability of the code and to allow the use of some already existing graph manipulation tools without the need of a wrapper. The use of the *boost* graph format has some overhead, but in our case this is negligible and the benefits are worth it. The *Boost Graph Library* includes many useful algorithms like Dijkstra, maximum matching or planarity test. *Boost* also supports handling the graphs in the *DOT language*, we use this as our primary graph format. A number of graph visualization software supports the *DOT language* format which is practical when we want to analyze the graphs manually. The *DOT language* allows setting attributes for nodes, edges or the whole graph. For our graphs, we give each edge a string attribute "color", which could be either "black" or "red". This is used to store the computed matching directly within the *DOT*-file. If the color of an edge is red, this means the edges is matched, if it is "black", the edge is unmatched. The color attribute is recognized by most graph visualization software resulting in colored edges. This provides also a fast way for manually analysing the given solutions. Figure 4.1 shows a small example graph extracted from one of our test instances. The red edges represent matched edges, the matchings is maximal and every node is insident to one matched edge.

Beside the *Boost Graph Library*, we additionally use the library *LEDA*. *LEDA* provides some algorithms which are missing in the *Boost Graph Library.* In our case we required an algorithm solving weighted matchings and creating planar graphs, which are currently not supported by *Boost.* The worst case running time for the weighted matching algorithm of *LEDA* is in O(nmlog n). This is slightly worse than the best known algorithm ($O(nm + n^2\log n)$, [Gab90]) but will be

[1]The *ILOG CPLEX®* ILP-solver is a commercial application. There exists also a free open source tool, *lpsolve*, for solving the ILPs

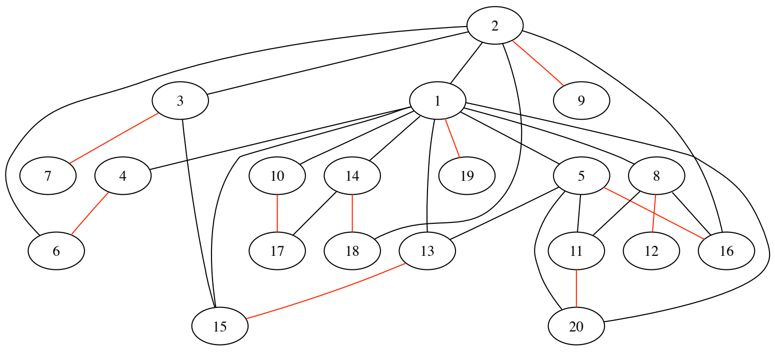


Figure 4.1: Example Graph From Our Test Cases, $n = 20, m = 30$

sufficient for our purpose.

The third tool is the *ILOG CPLEX* ILP-solver which is used to solve ILPs. This solver is able to solve integer linear programs for many instances very efficiently. It is possible to use the ILP solver with text input files describing the ILP, but we will use the *C++* application interface provided by the solver, invoking the solver within our application.

Now we will describe the implementation of the algorithms, starting with the ILP algorithm.

ILP Algorithm We implemented the simplified version of the ILP as described in Section 3.1. Our used tools considerably reduce the required work for the implemention the ILP is (and the heuristic). The ILP algorithm is implemented as follows. The graphs are read from file and stored in our graph format compatible with the *Boost Graph Library*. Then we invoke the maximum matching algorithm from the *Boost Graph Library* for each graph, which runs in $O(nm\log n)$. The maximum size of a matching is needed to create the simplified version of the ILP. The simplified version needs significantly less time to be solved by the solver. After creating the ILP we solve it with *ILOG CPLEX*. The resulting solution for SimMatch can be easily induced from the solution of the ILP. Note that it is useful to set a maximum time limit for solving the ILP as sometimes the *ILOG CPLEX* solver will not return in appropriate time. Also note that we only accept an optimal solution for the ILP.

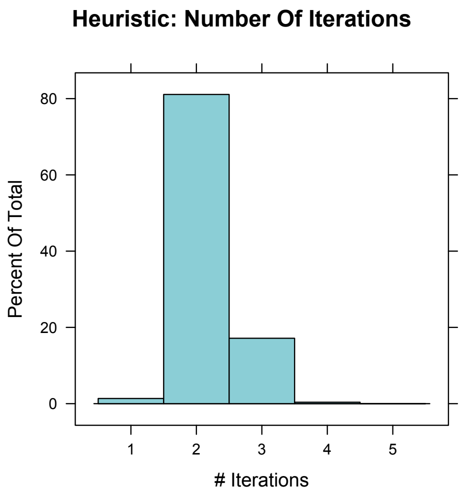
Heuristic The heuristic is implemented as in the pseudo code in Algorithm Sim-Match-Heuristic in Section 3.2. We use the maximum weighted matching algorithm from *LEDA* which has a worst-case running time of $O(nm\log n)$ time. Note that the graphs need to be stored in the *LEDA* graph format to use the weighted matching algorithm. The limitations for this implementation is not the running time but the correctness of the *LEDA* maximum weighted matching algorithm. If there are too many nodes and edges ($n + m > 10^7$) the algorithm will not work because of precision problems (rounding issues). This also depends on the edge weights. There is no danger of unnoticed false answers as the maximum weighted matching solutions are always checked per certificate at the end.

Finally, we discuss the overall process of performing a test case and the measured parameters. The different steps of a test case are as follows: First we create the instance, then we execute once the heuristic and once the ILP algorithm on the instance. Then, we analyze the solution of the algorithms. For both algorithms, we measured the running time and the size of the solution value. For the heuristic, we measured additional parameters such as the number of iterations and whether it is useful to start the iterations once with G_1 and once with G_2.
As solving the ILP sometimes requires a large amount of time, we set a time limit of 600 seconds for solving the ILP. This time limit was reached for many instances. For these instances we have no result of the ILP algorithm, and since only the ILP guarantees returning an optimal solution, we have no information of the optimal solution for these instances.

All the test cases where run under a machine with an *Intel Xeon CPU E5430* 2.66 GHz processor and eight cores computing with 64 bit. The machine had 32 GB of main memory and had the *Linux* operating system *SuSE 11.0-64* installed. Note that we did not make the full use of the eight cores as our implementation uses only one processor at a time.

4.3 Test Results

In Section 3.1 and 3.2 we presented an ILP approach and a heuristic to solve SimMatch. With the experiments we want to test how the ILP algorithm and the heuristic perform on instances and whether they are applicable.
First, we present the measured data with charts and data tables and discuss them, in the end we will give a short summary and an overall interpretation. As described in Section 4.1, the input instances vary on several orthogonal parameters, such as the type of the graph (planar or not), the number of nodes (n), the number of edges (m) and the type of generator used for the second graph. A summary can be found in Table 4.1 and 4.2. Often we do not discuss the different types of graphs separately if the results do not show significant differences in the graph types. However, if there is a noteable dependency between the analyzed data and the type of the instances, we will discuss the results seperately.
Before we compare the Heuristic and the ILP, we first present some observations of the heuristic run. To understand these results, it may be necessary to take a look back to the Sim-Match-Heuristic algorithm in Section 3.2. As described, the heuristic repeats improving the matching M_1 and M_2 for G_1 and G_2, respectively,

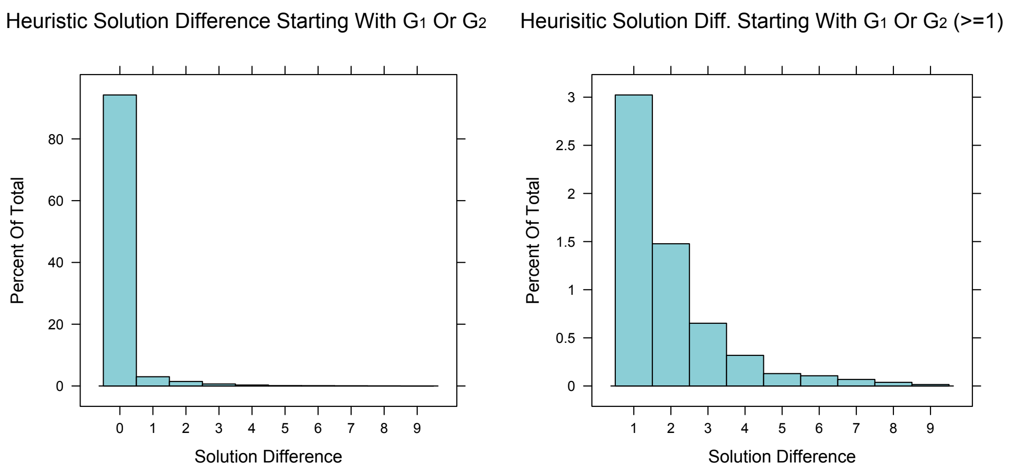


# Iterations	1	2	3	4	5	∑
# Test Cases	361	21599	4568	100	8	26636

Figure 4.2: Heuristic Number Of Iterations

as long as $|M_1 \cap M_2|$ increases (repeat loop starting in line 6). In every round M_1 and M_2 are recomputed. We measure the number of elapsed rounds: First with G_1 as the first graph and then with G_2 as the first graph yielding (possibly) two different values. In Figure 4.2 the number of iterations is visualized with a histogram and a table. As we can see, the number of iterations does not exceed five for all test cases. Most of the test cases (≈80%) have two iterations, about 20% have three iterations, and only very few have one, four or five iterations. It is clear, that there are not many test cases with only one iteration as the iteration continues if $|M_1 \cap M_2|$ increases and for most test cases $|M_1 \cap M_2|$ is larger than 0 after the first iteration. So most test cases have only 2 or 3 iterations, only very few have 4 or 5 iterations even if the size of the graphs goes up to 10^4 nodes and 10^6 edges. Also taking into consideration that we measured $26 \cdot 10^3$ cases let us suspect that the number of iterations is far less than the worst case assumption n. The number of iterations seems to be constant or at least very slowly growing for our test cases. If the number of iterations was indeed constant in the average case, the average case running time complexity would be $\mathrm{O}(nm + n^2 \log n)$, which is the same complexity as for weighted maximum matchings. Whether this is true might be an interesting question for further analysis. Especially when the heuristic or the used maximum weighted matching algorithm is provided with some randomness, it might be possible to find a tighter bound for the average case for the number of iterations than n.

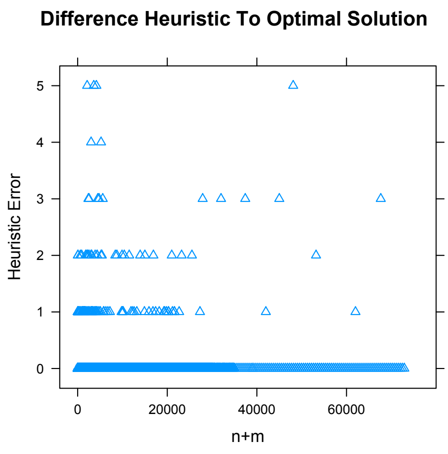
Another interesting part of the heuristic is the question, whether it is worth computing a SimMatch solution iteratively *two* times. The iterative recomputation



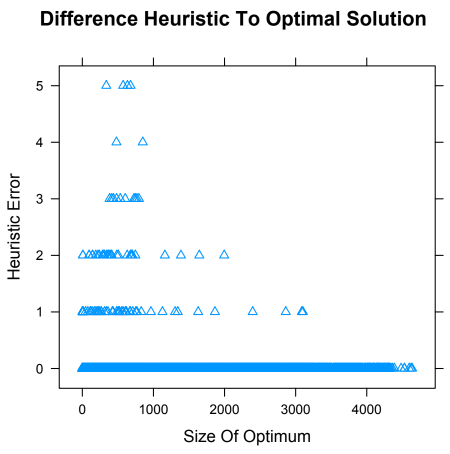
Δ Start $G_1 \mid G_2$	0	1	2	3	4	5	6	7	8	9	$\sum$
# Test Cases	12549	399	195	86	42	17	14	9	5	2	13318

Figure 4.3: Difference Between Starting With G_1 Or G_2 In The Heuristic

of the matchings in both graphs is done twice, first with G_1 as the first graph for which a matching is computed, then with G_2 as the first graph (line 3 in the pseudo code). If the size of the objective value does not differ between those two runs it would be sufficient to perform only one computation which decreases the running time by almost the a factor of 2. In Figure 4.3 we visualized the difference of the size of SimMatch solution between the two iterative runs, starting once with G_1 and once with G_2. The right histogram shows only the test cases in which a difference between both runs exists in order to get a higher y-scale resolution. In most cases (94%) both runs yield a solution with an equal value, 3% have a difference of 1, about 1.5% a difference of 2 and than the number of cases decreases by half when the difference increases by one, i.e. it seems that the number of test cases decreases exponentially with the size of the difference.
The mean relative difference from the regular heuristic solution would be only 0.174% if we always choose the graph which yields a lower or equal objective value when starting with it the one iterative run. On the other hand, as we will see, the mean relative error of the heuristic to the optimal solution is also only 0.113%. That means performing both runs can decrease the heuristic error noticeable. Therfore it is worth performing both runs if it is important to be very close to the optimal solution. Also note that the part of performing both runs can be easily parallelized in order to run at the same time on two different processors. However, our implementation uses only one thread.
The most interesting experiments are: How good are that the heuristic solutions, i.e. how far away are they from an optimal solution and how fast can they be computed (also in comparison to the ILP approach)? First, we address the former case, that means we analyze the difference of an optimal solution to the heuristic



(a)



(b)

Heuristic Error	0	1	2	3	4	5	$\sum$
# Test Cases	8432	87	31	10	2	4	8566

Figure 4.4: Heuristic vs Optimum

solution for our instances. If the ILP algorithm runs properly it returns an optimal solution. However, the ILP algorithm does not always returns successfully as during the solving of the ILP with the *ILOG CPLEX*® solver may exceed our time limit. We set the time limit to 600 seconds for merely solving the ILP without counting anything else like reading in the graph files. When the graph files exceed a certain size, the ILP normally does not finish within the time limit. In our experiments we could get optimal ILP solutions for graphs with $n + m$ up to $80'000$.

For all instances for which the ILP algorithm does return an optimal solution, we can compare this solution to the heuristic solution. As expected, if the ILP algorithm runs properly, its solution is always greater or equal to the heuristic solution as it is an optimal solution. In Figure 4.4 we visualized the difference between the optimal solution obtained by the ILP algorithm and the heuristic solution. This is the absolute error of the heuristic for single instances. In Figure 4.4(a) the heuristic error is plotted versus the size of the graph instances, which here is the mean of $n + m$ of both graphs. Most errors take place with smaller instances ($n + m < 10'000$) but also note that most of the instances for which an optimal solution exists are of smaller size. In Figure 4.4(b) the heuristic error is plotted versus the size of the optimal solution. Most of the errors occur for instances with an optimum value between 0 and $1'000$. For exactly which type of instances the errors occurs will be discussed later in more detail.

In Figure 4.5 you can see the relative approximation factor of the heuristic, on the right side a subset with a higher resolution. The worst quality for all instances

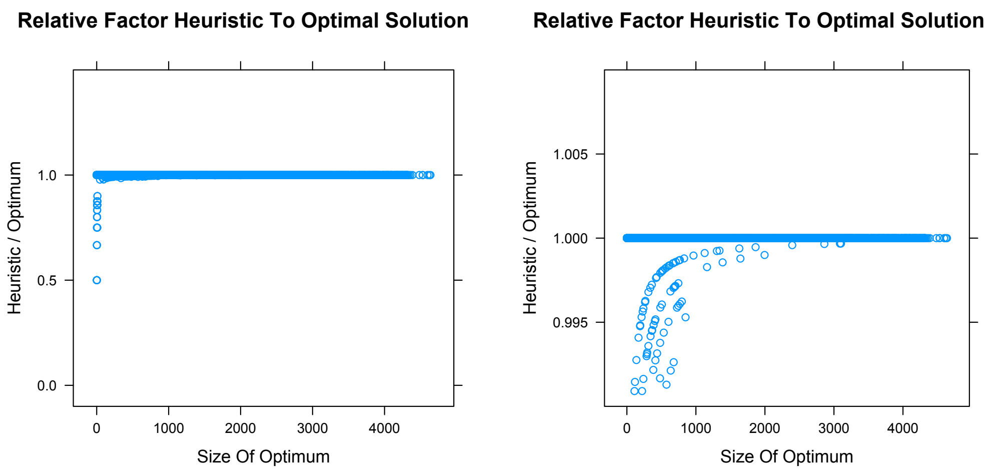
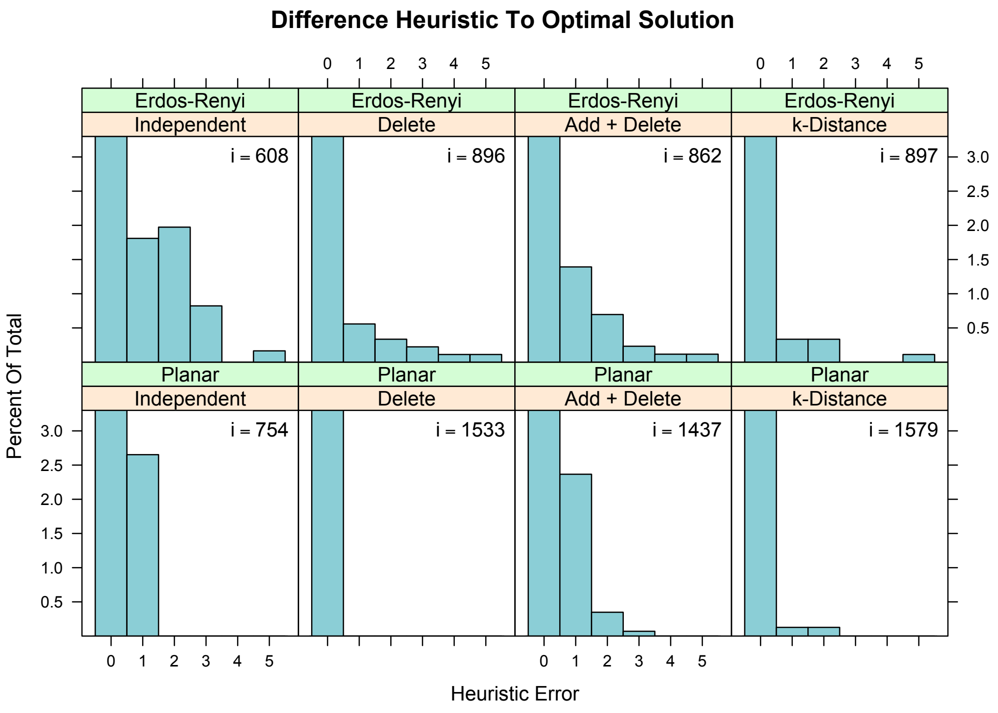


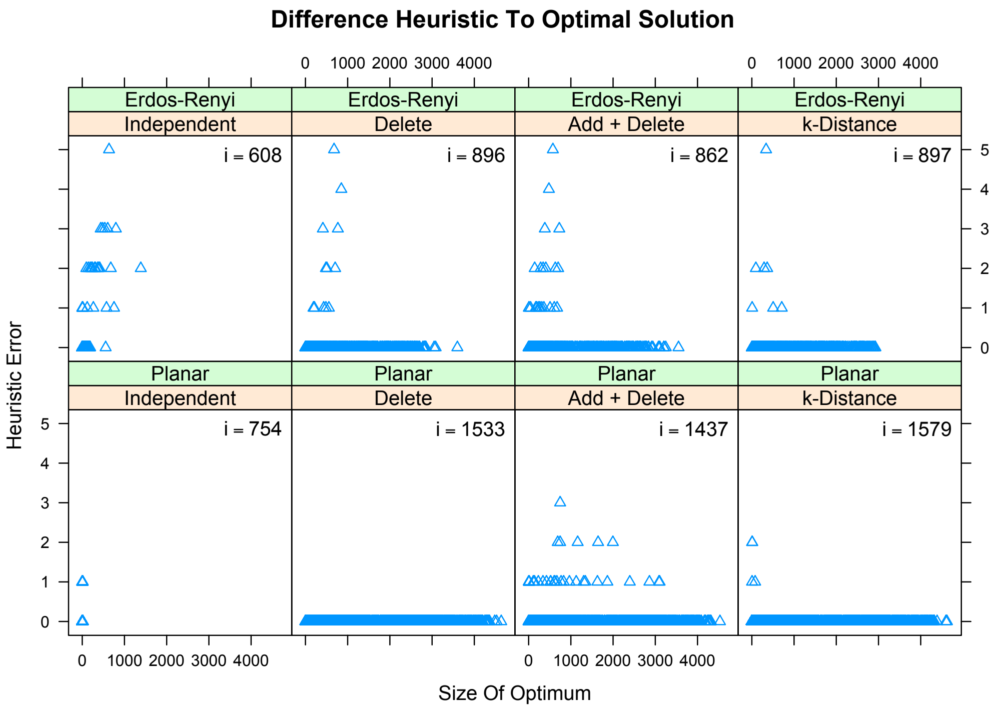
Figure 4.5: Relative Error Of The Heuristic

was a 0.5 approximation, i.e. the solution value of the heuristic was only half of the optimal value. However, a 0 approximation would have been possible as seen in Figure 3.1. On the average, the approximation of our heuristic is very good. The mean relative error is only 0.113%. The heuristic finds almost always an optimal solution, that is in more than 98% of the test cases. When the heuristic is not optimal, it is mostly off by one. Beneath this low error rate, we found another interesting fact: If both graphs have enough edges, then there exist a maximum matching in both graphs for which the intersection covers all nodes of one of the graph (possibly except one node), the heuristic did always return the optimal solution in our test cases. More precise: If the optimum value is at least $\lfloor min(n_1, n_2)/2 \rfloor$ (which is an upper bound for the optimum solution), the heuristic did return an optimal solution for all of our test cases. 3075 of our test cases meet the condition for the optimum value. This could be interpreted as follows: If the heuristic can choose between several combinations of maximum matchings which intersection size is maximum, it is easy for the heuristic to find an optimal solution.

We will now analyze for which type of graphs the heuristic is most error prone. We distinguish several different parameters of the type of instance generation listed in Table 4.1. We could observe the most interesting differences if we differentiate the type of creation of the first graph, i.e. planar or Erdős-Renyi creation, and the type of creation of the second graph, i.e independent of the first, delete k edges, add and delete k edges or k-distance. In Figure 4.6 we have visualized the heuristic error differentiated by the type of the first graph and the type of the second graph. On the top right of the graphics is listed the number of observations. Figure 4.6(a) consists of histograms showing the percentage of the heuristic error for different graph types. Erdős-Renyi graph generation with the second graph being created independently is most error prone, thereafter, when the second graph is created by



(a) Heuristic Error Histogram



(b) Heuristic Error Versus Size Of Optimum

Figure 4.6: Heuristic Error Distinguished By Graph Type

adding and deleting k edges from the first graph, there is also a higher error rate. `k-distance` and `delete` are quite similar modifications since if the first graph is sparse, it is likely most of the steps of the k-distance is just adding one edge. Note, that k-distance, if the first graph is planar, is a little bit more complicated as inserting edges must not violate planarity. An interesting fact is that we have not a single instance with an error if the first graph is planar and the second was created by deleting k edges of the first graph. In general it seems that there is not a high error rate if one graph is a subgraph of the other, i.e. one graph differs from the other only by additional edges. This is the case if the second graph is created by deleting edges and it is *likely* if the second graph is created by k-distance and the first graph is sparse or very dense.
In Figure 4.6(b) you can see the heuristic error plotted versus the size of the optimum differentiated by graph types. Focusing only on the distribution of the size of the optimum, we see that if both graphs are generated independently and planar, the size of the optima is very low. As a planar graph is always sparse, there is not a high chance of having many edges present in both graphs. For most graph types, most errors occur when the size of the optimum is below $1'000$. Only for planar graphs with the second graph created by adding and deleting edges, the errors are spread more widely. This may be due to the fact that our planar graphs with a lot of nodes never can not have much edges (if $n \geq 3$ then $m \leq 3n - 6$ in planar graphs) and thus adding and deleting edges may have a bigger influence. Remember that despite some differences of the absolute errors between the different type of instances, the mean relative error of the heuristic is still small.

The next interesting part of the experiments is the running time of the heuristic and the ILP algorithm. The running time strongly depends on the size of the graphs in the instance, but especially for the ILP algorithm some instances need far more time to be solved in comparison to other instances of the same size. In Figure 4.7 we plotted the running time in seconds versus the size of the graphs (mean of $n + m$) for the ILP algorithm and the heuristic. Notice that we did not include the running time of the ILP algorithm when it exceeded the time limit, which was set to 600 seconds. As we can see, the heuristic uses far less running time than the ILP. Also, for a lot of instances (probably "difficult" ones) the ILP uses considerably more running time than other instances of the same size. For some instances with $n + m \approx 5'000$ the ILP uses up to 600 seconds while the heuristic uses only about 1 second. The running time of the ILP is very high for some instances and appears to be not practical any more for instances exceeding a certain size.
In Figure 4.8 we visualized only the running time of the heuristic to get a higher resolution. In Figure 4.8(a) we use the same scale for the x-axes as in Figure 4.7 to allow better comparison. In contrast to the ILP, the heuristic was able to solve instances with $n + m > 80'000$ without exceeding the time limit of 600 seconds. The running time of all instances the heuristic solved can be seen in Figure 4.8(b). Those instances go up to $1'000'000$ for $n + m$.

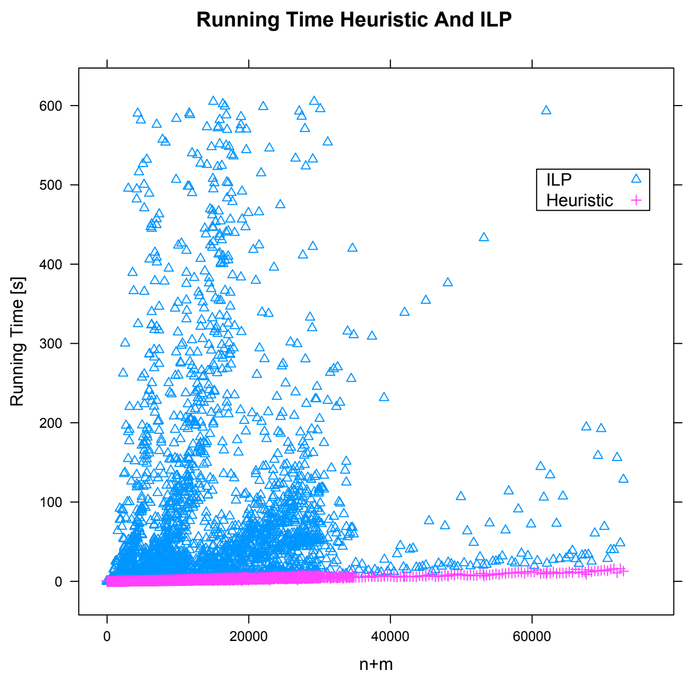


Figure 4.7: Running Time

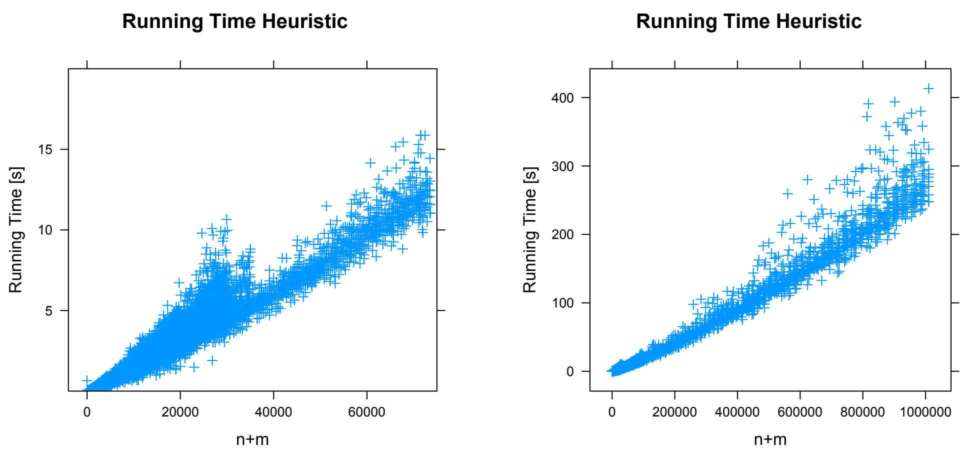


Figure 4.8: Running Time Heuristic

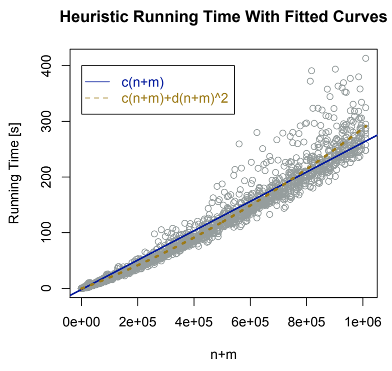


Figure 4.9: Fitting Heuristic Running Time

In Figure 4.9 we fit the running time of the heuristic with the linear function $c(n+m)$ and the quadratic function $c(n+m)^2$ by the least squares method. When we fit the running time with a linear function, the running time grows too fast. When we fit it with a quadratic function, it grows too slow. So the growth of the running time with $n+m$ seems to be subquadratic but also superlinear. The running time of the heuristic on our instances seems to lie between $c(n+m)$ and $c(n+m)^2$. When we compared this to our theoretical complexity $\mathrm{O}(n^2m+n^3\log\, n)$ the practical running time appears to be far less. As discussed during the first presented results of the experiments, it might be admissible to assume that the heuristic only needs a constant number of iterations to improve the maximum matchings in both graphs. Taking also into account that the maximum weighted matching algorithm can run faster than its worse case complexity $\mathrm{O}(nm+n^2\log\, n)$ for practical instances explains the conservative growth of the running time of the heuristic.

We now want to give a short summary of our experimental results. At the beginning we saw that the number of iterations for improving the maximum matchings in both graphs does not exceed 5. For over 98% of the instances the heuristic performs only 2 or 3 iterations so it appears that the number of iterations is growing much slower, if not being constant, than the theoretical upper bound n for practical instances.

After that we checked whether it is worth to start the iterative process of improving the maximum matchings twice (once starting with G_1, then with G_2). On the average it saves about half of the relative heuristic error, so it does pay off unless it is very important to decrease the running time.

Then we analysed the error of the heuristic, that is, the value of the ILP solution, which is optimal, minus the value of the heuristic solution. The mean relative error is very low (0.113%), in more than 98% of the test cases the heuristic found an optimal solution. The biggest absolute error was 5 while the size of the computed optimal solutions reached up to $10'000$. Most of the errors occurred when each of the two graphs has edges the other one has not, or the other way around, when one graph is subgraph of the other the heuristic did produce less errors.

Finally, we analyzed the running time of the ILP algorithm and the heuristic. The running time of the ILP is higher than the one of the heuristic and appears to be very fast growing. While the ILP was able to solve instances up to $10'000$ for $n+m$, the heuristic solved instances up to $1'000'000$ for $n+m$. The running time of the heuristic appears to grow by $c(n{+}m)^k$, with c being constant and $1 < k < 2$.

The next chapter concludes this work and summarises the results made so far, it will also point out some ideas for future work.

5 Conclusion and Outlook

This work deals with the problem of finding two maximum matchings in two similar graphs, so that the cardinality of the intersection of the matchings is maximum. We name the problem SimMatch. In this chapter we give a short summary of our work. After that, we present some ideas for future work. For future work, we think about extending certain parts of our work related to SimMatch and we also propose to extend the idea of simultaneous problems to other areas than matchings.

5.1 Summary

The main idea of this work has been to analyze the problem of finding matchings simultaneously in graphs that share some edges and nodes, so that the matchings are similar. We formulated this as the problem SimMatch, that is, find two maximum matchings in two graphs, so that the intersection of the matchings is maximum.
Our theses about SimMatch consists of three parts: The complexity of SimMatch, approaches to solve SimMatch and the experimental part, in which we tested some approaches to solve SimMatch.
In the complexity analysis, we proved that SimMatch is NP-hard by reducing the problem MaxCut to it. We also showed that for the following restrictions on the graph structure the problem remains NP-hard: both graphs consist solely of cycles, both graphs are planar, both graphs are connected, both graphs consist solely of simple paths and both graphs consist of a single tree. We also found an approximation gap for SimMatch: It is NP-hard to approximate SimMatch closer than 50/51 with respect to its optimal solution. The approximation gap remains for the restrictions on the graph structure listed above, merely, for graphs consisting of a single paths or single trees, we can only show a gap of 288/289.
In the second part, we presented some approaches for solving SimMatch. We presented two versions of an ILP-formulation, for which the optimal solution corresponds to an optimal solution of SimMatch. For the ILP version that can be solved faster by our ILP-solver in the experimental part, the size of a maximum matching in the graphs must be used. The running time for solving those ILPs is not polynomially, unless P=NP. We also presented a heuristic for SimMatch which runs in polynomial time ($\mathrm{O}(n^2m + n^3\log n)$). Unfortunately, the heuristic leaves us in general with no relative guarantee on the solution.
Instances, in which one graph has a polynomially bounded number of maximum matchings can be solved in polynomial time, or more precisely: We presented an

algorithm which solves SimMatch in $O(\min(\mathcal{MM}(G_1), \mathcal{MM}(G_2)) \cdot \sqrt{n}m^2)$ time, where $\mathcal{MM}(G)$ denotes the number of maximum matchings in graph G.
The last part of this work consists of experiments. We implemented the ILP approach and the heuristic and tested them on random instances. The graphs in our instances varied in several parameters. Although the heuristic does not give any theoretical guarantees on the optimal solution, it performed very well in the experiments. The mean relative error rate compared to the optimum is only 0.113% and in more than 98% of the test cases the heuristic found an optimal solution. The ILP approach always returns an optimal solution, however, this approach is only feasible for instances with graphs with $10'000 \geq n + m$, as for larger instances, and even for some instances with $n + m \leq 10'000$, it exceeded our time limit. So for small instances the ILP approach is applicable. In contrast, the running time of the heuristic is growing quite conservatively, the heuristic did solve instances with up to $1'000'000$ for $n + m$.
In the next section we give an outlook on possible future work.

5.2 Future Work

By reducing MaxCut to SimMatch we have shown the NP-hardness of SimMatch. We also used the reduction to prove the inapproximability of SimMatch. With the work of [TSSW00], which proves an approximation gap of 16/17 for MaxCut, we proved an approximation gap of 50/51 for SimMatch in Section 2.2. We think that this bound is not necessarily tight. It may be improved by using another reduction (maybe by reducing it from a completely different problem with an approximation gap) or by using a method not based on [TSSW00].
Or the other way around: What is the tightest bound one could find, i.e. what is a lower bound for an approximation? At the moment it is unclear where the lower bound lies, it can lie between 0 and 50/51. Getting a lower bound constructional can be achieved by finding an approximation for SimMatch with a relative guarantee on the solution. In Section 3.2 we presented a heuristic, which, unfortunately, does not provide us with a relative guarantee on the solution. We propose two ways which may result in an approximation for SimMatch:
One way would be the relaxation of the ILP-formulation we have given in Section 3.1. That means, relaxing the integrality constraints on the variables in a way, that the solution of the LP indicates a solution of the ILP which still guarantees us a relative factor for the solution in comparison to the optimal solution. Examples of this technique can be found in Vazirani's book [Vaz03].
Our other proposal to get an approximation is using random algorithms. Maybe it is possible to provide our heuristic with some randomness when searching for maximum matchings, which could lead to an approximation with an expected relative guarantee on the solution.
Beside improving the knowledge of the complexity of SimMatch and finding an approximation, it would also be interesting to see SimMatch used in some ap-

plications. For example including the heuristic in the graph drawing algorithm of [Wal03], as described in the introduction, Section 1.

These are our ideas for future work directly related to SimMatch. Now, we discuss more general possible future work. In Section 1.1 we defined the problem SimMatch, but beside that, we gave a more general version of the problem. During our work, we focused on the specific problem SimMatch, not on the general version. So another problem is extending this work to the more general version, some result may be adapted while others must be modified or developed newly from scratch.
Aside from the problem of finding *matchings* simultaneously, there are other simultaneous problems, which have not been analysed yet. Simultaneous matchings are only one part of the framework of simultaneous problems, so there remain a lot of problems for analysis. Concluding, we give three examples of additional possible simultaneous problems:

- *Simultaneous Cut:* Find a minimum (maximum) cut in several graphs, so that the intersection of the cuts is maximized.
- *Simultaneous Covering Set:* Find a covering set in several graphs, so that the intersection of the covering sets is maximized.
- *Simultaneous Coloring:* Find a coloring with k (=4?) colors in several graphs, so that the number of nodes, having the same color in the graphs, is maximized

Bibliography

[Edm65] J. Edmonds. Paths, trees, and flowers. *Can. J. Math.*, 17:449–467, 1965.

[Gab90] 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 (SODA '90)*, pages 434–443, Philadelphia, 1990. Society for Industrial and Applied Mathematics.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractability.* Freeman, 1979.

[GJS74] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing (STOC '74)*, pages 47–63, New York, NY, USA, 1974. ACM.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA.* Cambridge Univ. Press, 1999.

[MS04] Marcin Mucha and Piotr Sankowski. Maximum matchings via gaussian elimination. In *FOCS*, pages 248–255. IEEE Computer Society, 2004.

[MV80] Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{V}E)$ algorithm for finding maximum matching in general graphs. *Foundations of Computer Science*, pages 17–27, Oct. 1980.

[SL02] Jeremy G Siek and Lie-Quan Lee. *The Boost graph library.* Addison-Wesley, 2002.

[TSSW00] L. Trevisan, G.B. Sorkin, M. Sudan, and D.P. Williamson. Gadgets, approximation, and linear programming. *SIAM Journal on Computing*, 29(6):2074–2097, 2000.

[Uno97] Takeaki Uno. chapter Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs, pages 92–101. 1997.

[Vaz94] Vijay V. Vazirani. A theory of alternating paths and blossoms for proving correctness of the general graph maximum matching algorithm. *Combinatorica*, 14(1):71–109, Mar 1994.

[Vaz03] Vijay V. Vazirani. *Approximation algorithms.* Springer, corr. 2. print. edition, 2003.

[Wal03] Chris Walshaw. A multilevel algorithm for force-directed graph-drawing. *Journal of Graph Algorithms and Application*, 7(3):253–258, 2003.