A Comparison of Approximation Algorithms for the MaxCut–Problem[*]

Oliver Dolezal, Thomas Hofmeister, Hanno Lefmann[†]
Universität Dortmund, Informatik 2,
D–44221 Dortmund, Germany

Abstract

In this paper we compare, from a practical point of view, approximation algorithms for the problem MaxCut. For this problem, we are given an undirected graph $G = (V, E)$ with vertex set V and edge set E, and we are looking for a partition $V = V_1 \cup V_2$ with $V_1 \cap V_2 = \emptyset$ of the vertex set which maximizes the number of edges $e \in E$ which have one endpoint in V_1 and the other in V_2. The investigated algorithms include semidefinite programming, a random strategy, genetic algorithms, two combinatorial algorithms and a divide–and–conquer strategy.

1 Introduction

Consider an undirected, weighted graph $G = (V, E; w)$ with vertex set V, edge set E and non–negative weights $w_{i,j}$ of the edges $\{i, j\} \in E$ given by a mapping $w: E \to \mathcal{R}_0^+$. The MaxCut–problem consists of partitioning V into two disjoint subsets V_1 and V_2, such that the sum of the weights of edges with endpoints in different sets is maximized. Every partition $V = V_1 \uplus V_2$ is called a *cut* and the sum of the weights of edges running between the two sets is called the weight of the cut. In the following, we focus our investigation on the unweighted case. Therefore, we have:

$$w_{i,j} = \begin{cases} 1 & \text{if } \{i,j\} \in E, \\ 0 & \text{otherwise.} \end{cases}$$

As the decision variant of MaxCut is NP–complete [3], we cannot expect to compute the optimum efficiently, i.e., in polynomial time. Therefore we are faced with the problem of measuring the quality of the solutions found by the tested algorithms. To get an upper bound on the optimal value, we choose the solution found by the *semidefinite programming* (*SDP*) approach to a relaxation of the given MaxCut–problem. Goemans and Williamson showed in [4] that this method can be used to obtain an 0.878–approximation to the optimum and in addition an upper bound on the optimum. Let *SDPValDual* be the upper bound given by the semidefinite program and $w(cut_A)$ be the weight of the cut (number of edges between the two sets in the unweighted case) returned by some algorithm A for the same instance. Then the quotient $ratio_A = w(cut_A)/SDPValDual$ gives a measure for the quality of the weight of the cut returned by algorithm A. By the results of Goemans and Williamson, $ratio_{SDP} \geq 0.87$ holds.

[*] This research was supported by the Deutsche Forschungsgemeinschaft as part of the Collaborative Research Center "Computational Intelligence" (SFB531).

[†] {dolezal|hofmeist|lefmann}@ls2.informatik.uni-dortmund.de

If $ratio_A = 1$, the optimal cut size is returned by algorithm A. An introduction to semidefinite programming can be found in [6], a general survey on this topic is given in [1] and [11].
To get a reasonable lower bound on the optimum value of a solution of a MaxCut–problem, we choose a pure random strategy called *Random*. The idea is that *Random*, due to its simplicity, should be very fast but this strategy is not expected to give very good results. For this reason, other algorithms which run more slowly than *Random* and give worse results, can be considered to be not suitable to solve the MaxCut–problem, as long as they do not imply a reasonable guarantee for the size of the cut.
As another random approach, we applied a *genetic algorithm* (*GA*) to the MaxCut–problem. *GA*s are inspired by a model of natural evolution. They try to imitate natural optimization mechanisms involved in the model of biological evolution in order to solve mathematical problems. An overview on *GA*s can be found in [5].
Other candidates are a combinatorial algorithm and its variant using vertex–colorings, see [7]. These algorithms are investigated because one can guarantee a certain lower bound on the quality of the solutions. Although their solutions are worse than the one given by the semidefinite program (*SDP*), from a theoretical point of view, these algorithms are expected to be much faster than *SDP*.
The last algorithm investigated uses a *Divide and Conquer* strategy (*D&C*). *D&C*-type algorithms first divide a problem instance into subproblems. Then, these subproblems are solved recursively. Finally, the solution to the original problem is constructed using the solutions to the subproblems. In the case of MaxCut, the given input graph is divided into subgraphs and then MaxCut is solved for these subgraphs recursively.
The *D&C* strategy contains a local optimization at every stage. All the other algorithms are additionally locally optimized at the end with the result that the random algorithm after local optimization gives the best compromise between quality and running time.
Altogether six different types of algorithms are investigated:

1. *SDP:* Computes an approximate solution of MaxCut by means of *semidefinite programming*. The solution of the semidefinite program also provides us with an upper bound on the optimum, called *SDPValDual*. This upper bound is used to compare all algorithms tested here.

2. *Random:* Random strategy to solve the MaxCut–problem, which is used as some kind of lower bound on the optimum value of the MaxCut–problem. The output is used to compare the quality of all algorithms.

3. *Combinatorial:* Computation of an approximation to the optimum of a MaxCut–problem by a combinatorial approach from [7].

4. *CombColorings:* Variant of *Combinatorial* using vertex–colorings as in [7].

5. *GA:* Application of a *genetic algorithm* to the MaxCut–problem. Due to its relative independence of the structure of the problem [9], this algorithm can be quite easily adapted to other problems.

6. *D&C*: Approximation to the optimum of a MaxCut–problem using a *Divide and Conquer* strategy with local optimization.

This overview is followed by a detailed description of the tested algorithms and their implementation. In Section 3 the types of graphs used in the tests are described and in Section 4 the results of the test runs are presented. Finally a conclusion is drawn in the last section.

2 Description of the Algorithms

2.1 Semidefinite Programming

Semidefinite programming is similar to linear programming. For both types of problems, polynomial time algorithms are known which compute solutions which are arbitrarily close to the optimum. For a survey on semidefinite programming we refer to [6].
Let $G = (V, E)$ be an unweighted graph with vertex set $V = \{1, \ldots, n\}$. For solving the MaxCut–problem for G we formulate it as the following quadratic integer program:

$$\max \sum_{\{i,j\} \in E} \frac{1 - x_i \cdot x_j}{2}, \text{ such that } \quad x_i \in \{-1, 1\},\ i = 1, \ldots, n\ .$$

Notice that the term $(1 - x_i \cdot x_j)/2$ contributes 1 if $x_i \neq x_j$, and 0, otherwise. Thus, if we have a solution to this quadratic integer program, then by setting $V_1 = \{i \mid x_i = 1\}$ and $V_2 = \{i \mid x_i = -1\}$ we have found a partition of V, i.e., $V = V_1 \uplus V_2$. The value returned by the program yields the value of the cut, i.e., the number of edges $e \in E$, running between the two vertex sets V_1 and V_2.
Solving this quadratic integer program cannot be done efficiently, unless $\mathcal{P}$=$\mathcal{NP}$. Therefore, we transform it into a *semidefinite program.* First, we use some appropriate relaxation of this program, namely we replace the variables x_i by vectors $\vec{x}_i \in \mathcal{R}^n$ with unit length. This is a relaxation as we can view $x_i \in \{-1, +1\}$ as a vector $\vec{x}_i \in \mathcal{R}^n$, having as entries only zeros with the only exception that at position 1, we have the value of x_i. Then, this vector $\vec{x}_i$ has unit–length. Hence, we have for the relaxation of our MaxCut–problem the following formulation:

$$\max \sum_{\{i,j\} \in E} \frac{1 - \vec{x}_i \cdot \vec{x}_j}{2}, \text{ such that } \quad \|\vec{x}_i\| = 1, \quad \vec{x}_i \in \mathcal{R}^n, i = 1, \ldots, n,$$

where $\vec{x}_i \cdot \vec{x}_j$ is the usual component–wise scalar product. Now we are ready to transform this relaxed program into a semidefinite program by introducing new variables:

$$\mathit{SDPValDual} = \max \sum_{\{i,j\} \in E} \frac{1 - y_{i,j}}{2}, \begin{array}{l} \text{where } Y = (y_{i,j}) \text{ is a positive semidefinite} \\ n \times n \text{ matrix and } y_{i,i} = 1 \text{ for } i = 1, \ldots, n. \end{array}$$

Recall that an $n \times n$–matrix M is positive semidefinite iff $\vec{x}^T \cdot M \cdot \vec{x} \geq 0$ for all vectors $\vec{x} \in R^n$. In order to obtain from a (nearly) optimal solution to this semidefinite program a solution (of high quality) for our original MaxCut–problem, one proceeds as follows, compare [4], [6]:

1. Solve the semidefinite program as accurately as possible, where one obtains as a solution a matrix $Y = (y_{i,j})$ and the optimal (or nearly optimal) value *SDPValDual.*

2. Compute, using Cholesky–decomposition, a matrix B such that $Y = B^T \cdot B$. (Such a decomposition exists for positive semidefinite matrices Y.)

3. Choose a random vector $\vec{r}$, which is distributed according to the normal rotationally symmetric distribution.

4. Let $\vec{v_i}$, $i = 1, \ldots, n$, be the ith column vector of the matrix B. Determine the partition $V = V_1 \uplus V_2$ according to the following rule: We put vertex $i \in V$ into the set V_1 if $\vec{v_i} \cdot \vec{r} \geq 0$, and into V_2 otherwise. Compute the weight of the cut given by $V = V_1 \uplus V_2$.

5. Repeat steps 3 and 4 several times. Return the best solution $V = V_1 \uplus V_2$ found.

We remark that for the implementation for solving the semidefinite program we used the program solver *SDPA* by Fujisawa, Kojima and Nakata [2].

2.2 The Random Strategy

Given a graph $G = (V, E)$ with vertex set $V = \{1, \ldots, n\}$, the algorithm *Random* divides the vertex set V into two disjoint subsets V_1 and V_2 by going through all vertices of G successively and drawing a uniformly distributed random number $r_i \in [0, 1]$ for each vertex $i \in V$, $i = 1, \ldots, n$. Vertex i is inserted into one of the two sets V_1 or V_2 according to the following rule:
If $r_i \leq 0.5$, then insert i into V_1, otherwise insert it into V_2.
Obviously, the vertices of G are uniformly at random distributed among V_1 and V_2, each with probability 0.5. This distribution is done several times and each time the total weight $w(cut)$ of the corresponding cut is calculated.
Finally the cut with the biggest weight is returned as an approximate solution to MAXCUT. *Random* was not implemented with the hope to get very good solutions but to implement a very fast algorithm which gives reasonable solutions which can be used as lower bounds in the comparison with other algorithms.

2.3 Combinatorial

Combinatorial is based on a results from [7]. It was added to the group of investigated algorithms because it has the linear running time $O(n + m)$ in the uniform cost model for every given weighted input graph $G = (V, E; w)$ with $n = |V|$, $m = |E|$ and $w \colon E \to \mathcal{N}_0$. Moreover, the following lower bound on the weight of the computed cut $V = V_1 \uplus V_2$ was shown in [7]:

$$cut(V_1, V_2) \geq \frac{w(G) + w(M)}{2} \; .$$

In this inequality, $w(G) = \sum_{e \in E} w_e$ is the sum of the weights of all edges in G. Analogously, $w(M) = \sum_{e \in M} w_e$ is the weight of a matching M in the graph. A *matching* M in a graph $G = (V, E)$ is a set $\{e_1, \ldots, e_r\} \subseteq E$ of pairwise non–adjacent edges.
The main idea of *Combinatorial* is to find a matching M with large weight in G, and then to insert the vertices of the edges of M with endpoints in different sets V_1 and V_2 in such a way, that the weight of the corresponding cut is maximized. In order to decide whether for an edge $e = \{i, j\} \in M$ vertex i should belong to V_1 and vertex j to V_2 or vice versa, a potential function $VAL(G)$ is used which reflects the current achievable value of the desired cut. Here is the procedure in detail:

1. **Computing a large matching M:**

   We apply a well–known procedure for obtaining a "1–factorization" of a graph. Assume that the number n of vertices of G is even. Let w.l.o.g. G be the complete weighted graph K_n. Edges in G have weight 1, nonedges have weight 0. Imagine that (in an arbitrary order) the first $n-1$ vertices of a complete graph K_n are the points of a regular $(n-1)$-gon, where all diagonals are drawn. Using this as a base of a pyramid, we put above this the n'th vertex of K_n and draw all edges to the $(n-1)$ base points. If we now choose from the base one of the sides of the regular $(n-1)$-gon as well as all the parallel diagonals and that edge which goes from the missed base point to the n'th point, then we have a matching. Let $\mathcal{K}$ be the set of all $(n-1)$ matchings, which we obtain with such a procedure. No two distinct matchings in $\mathcal{K}$ have an edge in common. Let $M \in \mathcal{K}$ be a matching with largest weight, i.e., $w(M) \geq w(M')$ for each $M' \in \mathcal{K}$.

2. **Computing a large cut using the matching M:**

   (a) Initialize $V_1 := V_2 := \emptyset$.

(b) For every edge $e \in E$ the parameter $Val(e)$ is defined as follows:

$$Val(e) = \begin{cases} w_e/2 & \text{if } |e \cap (V_1 \cup V_2)| \leq 1, \\ w_e & \text{if } |e \cap V_1| = |e \cap V_2| = 1, \\ 0 & \text{otherwise.} \end{cases}$$

(c) Initialize a potential function $VAL(G)$ by $VAL(G) := \sum_{e \in E} Val(e) = w(G)/2$.

(d) For every edge $e = \{i, j\} \in M$ do successively:
$E' :=$ the set of all edges $e \in E$, where one endpoint is i or j, and the other is in $V_1 \cup V_2$. Put the vertices i and j into V_1 and V_2 in such a way that the sum $S = \sum_{e' \in E'} Val(e')$ is maximized and both vertices i and j lie in different sets. Update the values $Val(e')$ for all $e' \in E'$ and $Val(e)$.

(e) If there exist more vertices, join them into pairs in an arbitrary way and sort them into V_1 and V_2 as described in step (d). Update the corresponding values $Val(e)$. In case the number of vertices n is odd, add an additional 'isolated auxiliary vertex', i.e., all edges leaving this vertex to the other vertices have weight 0 and apply the algorithm as described above. Then delete the auxiliary vertex after this computation.

(f) *Combinatorial* returns a partition $V = V_1 \cup\!\!\!\cdot\, V_2$.

2.4 Combinatorial with Colorings

The algorithm *CombColorings* is a variant of *Combinatorial* which uses proper vertex–colorings. A vertex–coloring $\Delta\colon V \longrightarrow \mathcal{N}$ of a graph $G = (V, E)$ is *proper*, if for all edges $e = \{i, j\} \in E$ the colors of their endpoints are different, therefore $\Delta(i) \neq \Delta(j)$ holds. The algorithm can be described as follows:

1. Let $G = (V, E; w)$ be a weighted graph with weightfunction $w\colon E \longrightarrow \mathcal{N}_0$.

2. Color the vertices by positive integers as follows: For all vertices in V successively, assign, starting with color 1, every vertex the smallest color, which is not used by its adjacent vertices yet. We obtain a proper coloring $\Delta\colon V \longrightarrow \{1, \ldots, t\}$ using, say, t colors.

3. Let $G' = (V', E'; w')$ be the graph obtained by collecting all vertices of G with the same color to macro-vertices:

   (a) $V' = \{v'_1, v'_2, \ldots, v'_t\}$, $t =$ number of used colors.

   (b) $v'_i = \{k \in V \mid \Delta(k) = i\}$.

   (c) $E' = \{\{v'_i, v'_j\} \mid i \neq j, \exists\, k \in v'_i, l \in v'_j \text{ with } \{k, l\} \in E\}$.

   (d) $w'(\{v'_i, v'_j\}) = \sum\limits_{e \in A} w(e), \quad A = \{\{k, l\} \in E \mid k \in v'_i,\ l \in v'_j\}$.

4. Let *macro-cut* $V' = V'_1 \cup\!\!\!\cdot\, V'_2$ be the result when *Combinatorial* is applied to the graph G'.

5. Transform *macro-cut* into a cut of G:

   (a) Let V'_1 and V'_2 be the two sets constituting *macro-cut*.

   (b) Set *ColCut*:=(V_1,V_2) with $V_1 = \{v \in v' \mid v' \in V'_1\}$, $V_2 = \{v \in v' \mid v' \in V'_2\}$.

6. Let *ColCut* be the result of *CombColorings*.

CombColorings guarantees the following lower bound on the size of the returned cut [7]:

$$cut(V_1, V_2) \geq \frac{w(G)}{2} \cdot \left(1 + \frac{1}{t-1}\right), \ t = \text{number of colors},$$

provided t is even, otherwise, for t odd we have: $cut(V_1, V_2) \geq \frac{w(G)}{2} \cdot \left(1 + \frac{1}{t}\right)$.

2.5 Genetic Algorithms

Genetic Algorithms (*GAs*) are random search algorithms inspired by the model of natural evolution. Potential solutions are coded in a simple chromosome-like data structure. According to the model of biological evolution, the single potential solutions are called *individuals* and the set of individuals is called *population*. In order to solve an optimization problem, *GAs* successively create k populations developed with the help of the random operators *selection*, *crossover* and *mutation* with the goal to increase the qualities of the corresponding solutions coded by the individuals:

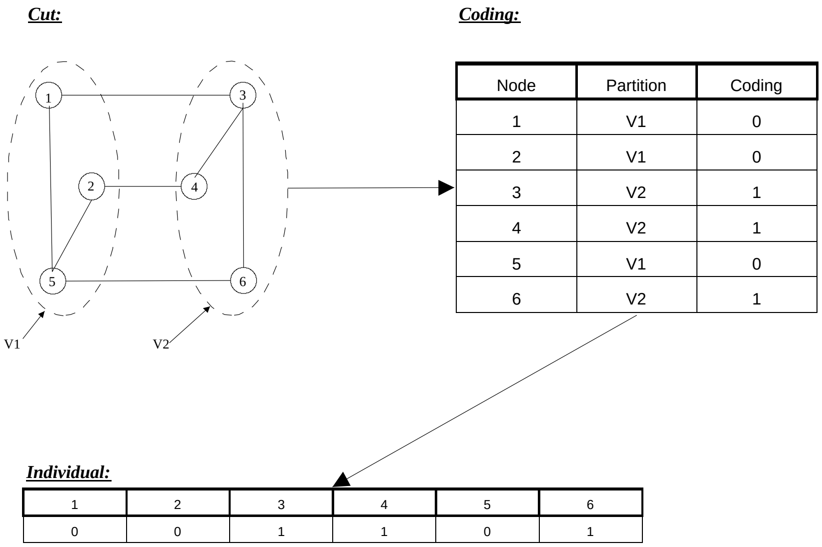
```
Algorithm GA:
begin
  $t := 0$;
  initialize $P(t)$; evaluate $P(t)$;
  while not $t \geq k$
    $t := t + 1$;
    $P(t) :=$ select $P(t-1)$;
    recombine $P(t)$; mutate $P(t)$; evaluate $P(t)$;
  endwhile
end
```

The procedure *initialize* creates randomly an initial population. In the evaluation step, all individuals of the current population $P(t)$ are assigned some *fitness value* which quantifies the quality of the solution they code. These *fitness values* are used by the *selection operator* which chooses the individuals from the population $P(t-1)$ which are used to build the new population $P(t)$. The higher the fitness of an individual is, the higher is the chance for this individual to be selected. The chosen individuals are either copied into the new population as they are or — with some probability p_c — they are subject to a *recombination* to form new individuals which are inserted into the new population. At the end of the loop, all individuals of the new population are mutated with the probability p_m.

In order to use a *GA* to solve a MaxCut–problem, we first have to code the potential solutions, i.e., cuts of the given graph G. These will be the individuals. Several cuts (individuals) will form the population. For this purpose, for graphs on n vertices the cuts are coded into bit-strings $(x_1, \ldots, x_n)$ of length n. Let $V = \{1, \ldots, n\}$ be the vertex set of G. The value x_j at position $j \in \{1, \ldots, n\}$ of the string has the following meaning:

$$x_j = \begin{cases} 0 & \Longleftrightarrow \quad \textit{vertex } j \in V_1, \\ 1 & \Longleftrightarrow \quad \textit{vertex } j \in V_2. \end{cases}$$

Figure 1 illustrates this coding. The *fitness value* of each individual (bitstring) i is given by the size of the cut it codes: $\textit{Fitness}(i) :=$ size of the cut coded by individual i.



Node	Partition	Coding
1	V1	0
2	V1	0
3	V2	1
4	V2	1
5	V1	0
6	V2	1

1	2	3	4	5	6
0	0	1	1	0	1

Figure 1: Coding of an individual.

As the selection method, we used *roulette-wheel-selection*. We chose this method because it favours individuals with high fitness values without totally suppressing the selection of individuals with low fitness values. This heuristically avoids a premature convergence of the population.

```
Procedure selection (in: Population; out: Individual)
begin
  $TotalFitness := \sum_{i=1}^{n} Fitness(i)$;
  $RelFitness(i) := Fitness(i)/TotalFitness$;
  $CumulativeFitness(i) := \sum_{j=1}^{i} RelFitness(j)$;
  $r :=$ uniformly distributed random number in $[0, 1]$ ;
  $i := 1$;
  while $CumulativeFitness(i) < r$
    $i := i + 1$;
  endwhile
  return($Individual[i]$);
end
```

The last individual of the population is assigned the cumulative fitness 1 and, as $r \in [0, 1]$, there is always an individual i with $CumulativeFitness(i) \geq r$. Obviously, those individuals with a higher fitness than the others are selected more often to form the new population. Therefore the increase of fitness is encouraged.

To implement crossover, two different methods were used, namely *Single-point crossover* and *Fixed crossover*. The first method is a standard crossover method for *GA*s independent of the structure of the individuals which are crossed [9]. The second method is dependent on the structure of the crossed individuals and was implemented to investigate whether the use of structural information is an advantage. For single-point crossover a natural number $j \in \{1, \ldots, n\}$ is chosen uniformly at random. Each of the two bit-strings which represent the parents, is divided into two parts: the first j bits and the last $(n - j)$ bits. Then we form the first new individual by concatenating the first j bits of individual 1 and the last $(n - j)$ bits of individual 2. Vice versa, the second new individual is created by concatenating the first j bits of individual 2 and the last $(n-j)$ bits of individual 1. This process is shown in Figure 2.

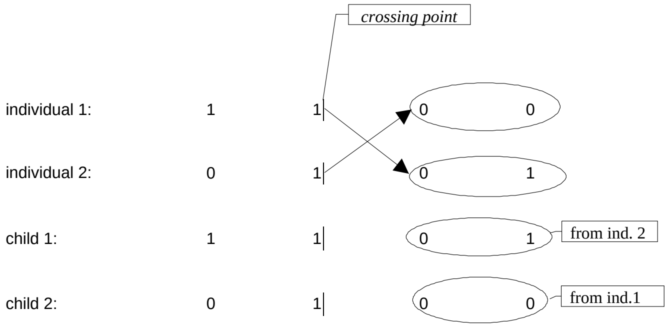


Figure 2: single-point crossover.

Fixed crossover is based on the following random function $f\colon \{0,1\}^2 \to \{0,1\}$ with $f(0,0){=}0$, $f(0,1){=}Random_1$, $f(1,0){=}Random_2$, and $f(1,1){=}1$. Here, $Random_1$ and $Random_2$ are random numbers from {0,1}, which are drawn at the beginning of the crossover and then are constant during the execution of crossover. With the help of f, the new individual *NewInd* is created from the old individuals *OldInd* in the following way:

$$\begin{aligned} NewInd[j] &:= f(OldInd_1[j], OldInd_2[j]), \\ OldInd_i[j] &= \text{value of } j\text{'th bit of individual } i. \end{aligned}$$

Thus, if two individuals 1 and 2 have at some position j the same entry, then the new individual has at position j also this entry. Otherwise, the entry of the new individual is given by $Random_1$ or $Random_2$, dependent on the old entries. Figure 3 gives an example for fixed crossover with $Random_1 = 0$ and $Random_2 = 1$. In contrast to single-point crossover, the operation fixed crossover returns only one child as offspring.

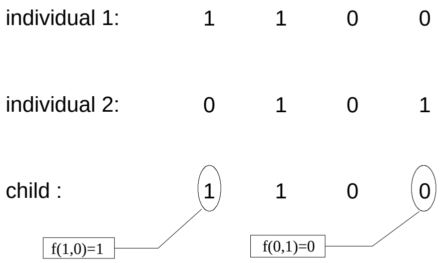


Figure 3: Example for fixed crossover.

In order to apply mutation to a population, a random number $r \in [0,1]$ is drawn for every bit of every individual's bitstring and, if r is less than the given probability for mutation, the corresponding bit is inverted. We used $p_m = 1/n$ (n = length of bitstring) as the probability for mutation [9]. Other values for this probability were also tested, but gave no better results. As a variant, we additionally tested a mutation method, where one chooses randomly the bits to be inverted in the same manner like in the previous described method, but a bit is only inverted if this inversion yields a bigger cut value than the former bitstring.

```
Algorithm GA:
input:
  $G$ = an undirected Graph,
  $MaxGen$ = maximal number of generations which will be created,
  $p_c$ = probability for crossover, $p_m$ = probability for mutation.
output: A partition $V_1 \cup V_2 = V$.
begin
  createInitPopulation();
  evaluatePopulation(); // compute for all individuals the sizes of the represented cuts.
  BestSoFar := individual of the current generation which represents the biggest cut;
  for $Gen = 1$ to $MaxGen$  // create a new population
    $i := 0$;
    while $i < n$
      selection($Population$,$Individual_1$);
      $r$ := uniformly distributed random number in [0,1];
      if $r \leq p_c$
        Choose uniformly at random another individual from the old population.
        Apply single-point crossover to the two chosen individuals as described above.
        Insert both new created individuals into the new population.
        $i := i + 2$;
      else
        Insert $Individual_1$ into the new population;
        $i := i + 1$;
      endif
    endwhile
    if $i < n$
      selection($Population$,$AdditionalInd$);
      Insert $AdditionalInd$ into the new population;
    endif
    Apply mutation to the whole new population with probability $p_m$ as described above.
    evaluate();
    $BestSoFar$ := individual of the current generation which represents the biggest cut;
  endfor
  return($BestSoFar$);
end
```

```
Procedure: createInitPopulation()
begin
  for $i = 1$ to $n$ // go through all individuals successively
    for $BitIndex = 1$ to $Graph_{Size}$// go through all bits of $individual_i$
      $InitVal$ := 0 or 1, each chosen with the probability 1/2.
      $individual_i[BitIndex] := InitVal$.
    endfor
  endfor
end
```

If one uses fixed crossover instead of single-point crossover, the counter i must only be incremented by 1 instead of 2 after crossover, because fixed crossover produces only one new individual.

2.6 D & C

D&C is a *divide-and-conquer* approach to solve the MaxCut–problem. Given an input graph $G = (V, E)$, the vertex set V is partitioned recursively into two subsets V_1 and V_2. In order to partition V in the beginning, all vertices in V successively are inserted randomly into V_1 or V_2, each with probability 1/2 and independently of each other. Every time the algorithm returns from a recursive branch it delivers a cut for those two vertex subsets V_1 and V_2 of V that it was working on. Since a single cut consists of two vertex sets we receive four subsets of vertices altogether. These are combined to a cut, such that the weight of the created cut is maximized. At each stage of the algorithm the result is locally optimized. We describe the algorithm in detail as follows:

1. Let $G = (V, E)$ be an undirected graph.
2. Compute a partition (V_1, V_2)=*DivideAndConquer(G, V)*.
3. Apply local optimization to the resulting cut: Go through all vertices successively and move it to the other vertex set if this step increases the weight of the cut. Every time a vertex is moved, start the local optimization once again from the beginning until the last vertex is reached without moving any vertex to the other vertex set.
4. Repeat steps 2. and 3. several times. As the result return the best solution $V = V_1 \cupdot V_2$ found.

The procedure *DivideAndConquer* is defined as follows:

```
Procedure DivideAndConquer(G,V)
input: graph $G = (V, E)$.
output: vertex sets $V_1$ and $V_2$ with $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$.
begin
  $n := |V|$;
  if $n \leq 2$ then return (first element of $V$, second element of $V$) else $Q_1 := Q_2 := \emptyset$;
  for $i = 1$ to $n$
      $r :=$ uniformly distributed random number in $[0, 1]$;
      if $r \leq 0.5$ then $Q_1 := Q_1 \cup \{v_i\}$; else $Q_2 := Q_2 \cup \{v_i\}$;
      $(T_1, T_2) :=$ DivideAndConquer(G,$Q_1$); Apply local optimization;
      $(T_3, T_4) :=$ DivideAndConquer(G,$Q_2$); Apply local optimization;
      $Cut_1 := (T_1 \cup T_2, T_3 \cup T_4)$; $Cut_2 := (T_1 \cup T_3, T_2 \cup T_4)$; $Cut_3 := (T_1 \cup T_4, T_2 \cup T_3)$;
      return(biggest of the cuts $Cut_1$, $Cut_2$, $Cut_3$);
  end
end
```

2.7 Local Optimization of the Algorithms

In order to improve the quality of the solutions, we implemented a second version with local optimization for all algorithms except for *D&C*, as there local optimization is already included

in the algorithm. The local optimization was achieved by going successively through all vertices that constitute the cut. For each vertex the new cut size is computed which would result from moving the current vertex into the other vertex set. If the new cut size is bigger than the old one, the vertex is moved and the local optimization starts from the beginning. The local optimization stops when the last vertex is reached and no vertex is moved to the other vertex set.
The described local optimization was applied to *SDP* and *Random* after every trial constitution of a cut. Recall that both algorithms build several cuts and choose the best of them as the result. Concerning *Combinatorial*, the local optimization is used at the very end. At the same place *CombColorings*, that uses *Combinatorial* as a sub-procedure, is optimized locally, too. But it is also locally optimized after the cut consisting of macro vertices is inflated to the original graph, as this also improves the result. In the algorithm *GA*, every new individual created by crossover is locally optimized immediately after its creation and in *D&C* local optimization is implicitly.
We also tested a second version of the local optimization. This second version differs from the above one, as it does not start immediately at the beginning whenever a vertex has changed its position. It continues until the last vertex is reached and then starts at the beginning. The condition for the stopping of the algorithm remains. This second version stops, when it reaches the last vertex without having moved any vertex to the other vertex set. It turns out that the qualities of the solutions produced by both methods do not vary significantly. For this reason we do not distinguish the two methods in the following, however, we distinguish the test runs which were started with local optimization and without this optimization.

3 Formal Aspects of the Tests

3.1 The Tested Graphs

In the test runs each of the different algorithms was applied to undirected, unweighted graphs with 100 vertices which were instances of 6 different classes of graphs. Though most of the algorithms are also able to cope with weighted graphs, we only tested the unweighted case here. We used different classes of graphs with the purpose to investigate whether the quality of the solutions, which the algorithms produce, depends on the class of the graphs. The 6 classes can be divided into two groups:

Simple random graphs $G_{Rand}(n;p)$:

These graphs on n vertices are created by inserting each edge with probability p, independently of the others.

Bipartite random graphs $G_{BiRand}(n;p,q)$:

The vertex set is divided into two parts:

$$V' = \{v_i \mid v_i \in V \text{ and } i \text{ is odd}\} \quad V'' = \{v_i \mid v_i \in V \text{ and } i \text{ is even}\}.$$

Then, for all pairs (v_i, v_j) of vertices with $v_i, v_j \in V$ and $i < j$ a uniformly distributed random number $r \in [0,1]$ is drawn and the corresponding edge $\{v_i, v_j\}$ is inserted if r is less than a given probability *Prob*. This probability is either p or q depending on the numbers of the vertices:

$$Prob = \begin{cases} q & \text{if } i \text{ and } j \text{ are both odd or are both even,} \\ p & \text{otherwise.} \end{cases}$$

This method of building a graph gives us the possibility of creating bipartite graphs only, by choosing $0 \leq p \leq 1$ and $q = 0$ as probabilities. A graph is called *bipartite* if its vertex set can be divided into two subsets such that there only exist edges between vertices in different sets. The advantage of using bipartite graphs is that the optimal cut is given by the two subsets by definition. Additionally, we can achieve "almost bipartite" graphs by using small values for the probability q.

3.2 The Tests

All algorithms were tested on 20 random graphs from each of the following classes: $G_{Rand}(100; 0.7)$, $G_{Rand}(100; 0.5)$, $G_{Rand}(100; 0.3)$, $G_{BiRand}(100; 0.7, 0.0)$, $G_{BiRand}(100; 0.3, 0.0)$ and $G_{BiRand}(100; 0.7, 0.3)$. Recall that the graphs for $G_{BiRand}(n; p, 0.0)$ are bipartite. We used 20 graphs from each of the classes to avoid that some of the algorithms behave very well or very bad on certain graphs. Therefore we applied every algorithm to 20 random graphs of each of the 6 classes that gives a total of 120 test runs per algorithm. For a better comparison of the results of the algorithms, all algorithms are applied to the same 120 graphs.

3.3 Presentation of the Test Runs

Let us first explain the presentation of the test runs using Figure 4.

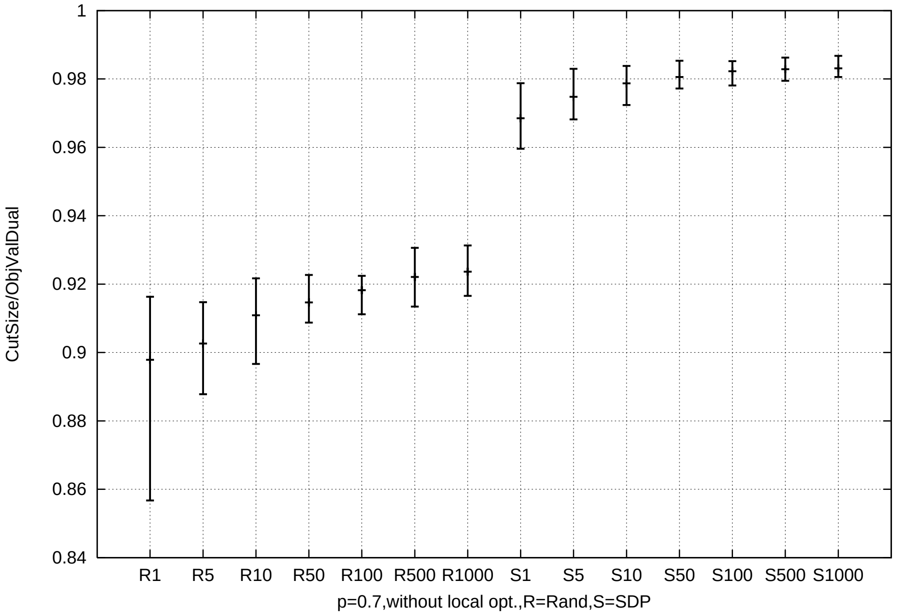


Figure 4: Relative cut sizes computed by *Random* and *SDP* for different numbers of random trials.

As described above, we applied the algorithms to 20 different random graphs of each class. The results of these 20 runs per class were compressed to the biggest and the smallest value and the median each algorithm achieved. These three values per algorithm and graph class are indicated at each x position in the figure. The middle marks the median and the upper and lower mark the biggest and smallest value returned by the presented algorithm. Accordingly, one can find the type of the tested algorithm on the x-axis (e.g. *R100* for algorithm *Random* and 100 trials) and the achieved values, that is the cut size or the running time, on the y-axis. The running time is given in seconds and the cut size is given as the quotient of the absolute cut size the algorithm computed and the optimal cut size *SDPValDual* estimated by *SDP*:

$$RelCutSize = \frac{AbsCutSize}{SDPValDual}.$$

All figures refer only to one graph class which is indicated by the caption of the figure or mentioned in the text referring to them.

4 Results of the Test Runs

Some of the algorithms, namely *SDP*, *Random* and *GA*, require the adjustment of their parameters before they are compared to the others. This is done by testing different values for the parameters and choosing the ones that lead to the best results on average. The parameters that have to be adjusted and the actual values chosen are described in the next two sections.
Then we tested those versions of the algorithms that are not locally optimized. The purpose of these test runs was to investigate the algorithms in their pure form without local optimization. Then we compared the versions with local optimization. These versions were investigated, because the local optimization improved the results of all algorithms without increasing the needed running times significantly and therefore these versions should be used in practice.

4.1 Adjusting ***Random***, ***SDP*** and ***GA***

In the algorithms *SDP* and *Random* the number R_{num} of cuts that are created randomly is the essential parameter, as the best one of these cuts is returned as the result.
The algorithm *GA* offers a large amount of parameters that can be adjusted. The parameters we adjusted are the population size, the number of generations and some variants of building the initial population, the crossover method and the mutation method. We used the value $p_c = 0.6$ as the probability for crossover. We tested some other values of p_c, but these tests gave no essentially different results. The same holds for the probability p_m of mutation. We used $p_m = 1/n$ for the latter, where n is the number of vertices of the graph G. Thus, on average one bit per individual is inverted. Other tested values for the probabilities did not really improve the results. As variations of the last two parameters gave no different results, we do not distinguish them from now on.
In order to determine a value for R_{num} which leads to good cut sizes and also keeps the running times reasonable, we applied *Random* and *SDP* to all test graphs with the following numbers of trials: 1, 5, 10, 50, 100, 500 and 1000. These tests showed that for both algorithms and for input graphs of type G_{Rand} the difference between the best and worst result of the test runs became smaller as the number of trials grew. At the same time the qualities of the calculated cuts were improved, when the number of trials was increased.
Figure 4 shows typical results reached by *SDP* and *Random* without local optimization, as they were applied on a random graph of the class G_{Rand} with 100 vertices and with an insertion probability of $p = 0.7$. On the y-axis one can see the relative cut sizes computed with the number of trials given on the x-axis. Recall that the relative cut size was defined as the returned cut size divided by the value *SDPValDual*. Hereby the label R on the x-axis indicates that the values for *Random* are given and the label S stands for *SDP*, e.g., *S100* = 100 trials with algorithm *SDP*.

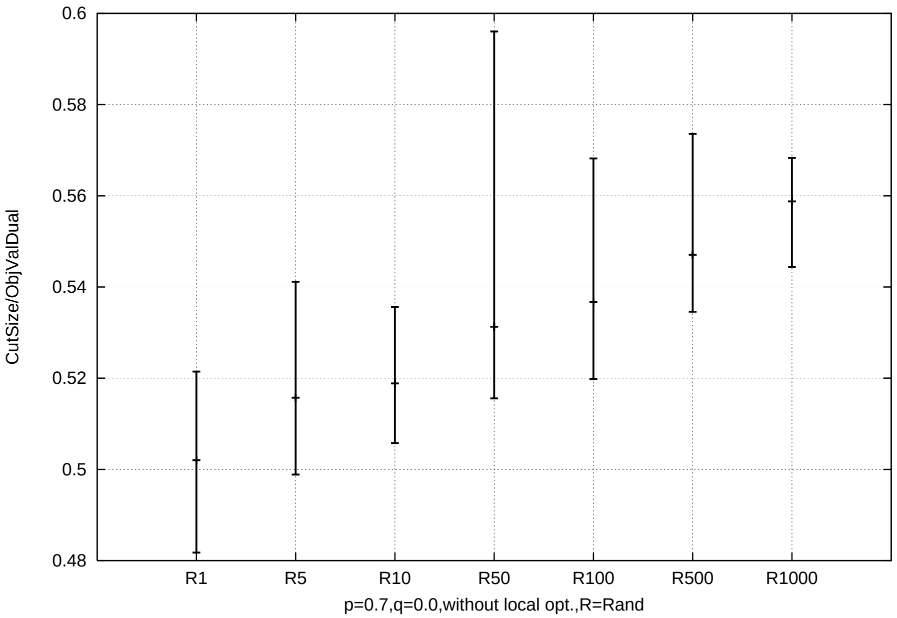


Figure 5: *Random* applied to a bipartite graph without local optimization.

As already shown in [8], we saw that when *SDP* was applied to graphs of the class G_{BiRand}, it always computed the optimum in terms of *SDPValDual*. The number of trials did not influence the result in this case. On the other hand, the results which were computed by *Random* were worse than the ones given by *SDP*. Therefore, the number of trials is important, as far as *Random* is concerned. Figure 5 shows the results of *Random* for a graph of the type G_{BiRand} and different numbers of trials.

The running times of both algorithms were quite similar in all tests. As an example, Figures 6 and 7 show the running times for a random graph of type G_{Rand} with an insertion probability of $p = 0.7$. On the x-axis one can see the type of the tested algorithm and the number of random trials (i.e., *R50* means: algorithm *Random* and 50 trials) and on the y-axis one can see the time in seconds which was needed for the computation.

Obviously, the running times of *Random* grow much faster than the ones of *SDP* as the number of trials is increased. This is due to the fact that *SDP* needs most of its running time to solve the semidefinite program and the time which is used for the random creation of cuts is relatively small. As a compromise between the quality of the cut and a small running time we chose 100 random trials for *Random* and *SDP* in the comparison of all algorithms.

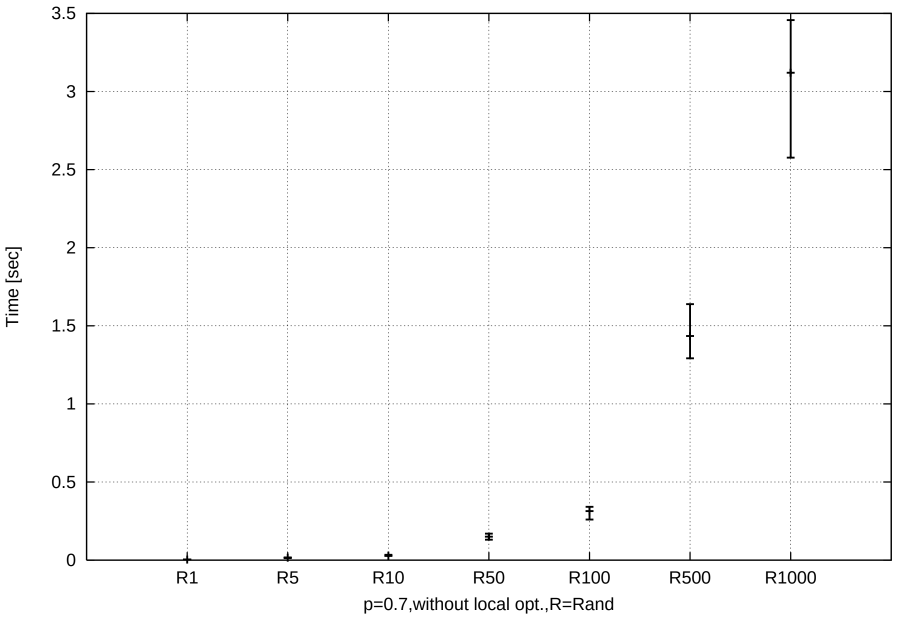


Figure 6: Running times of *Random* for different numbers of random trials.

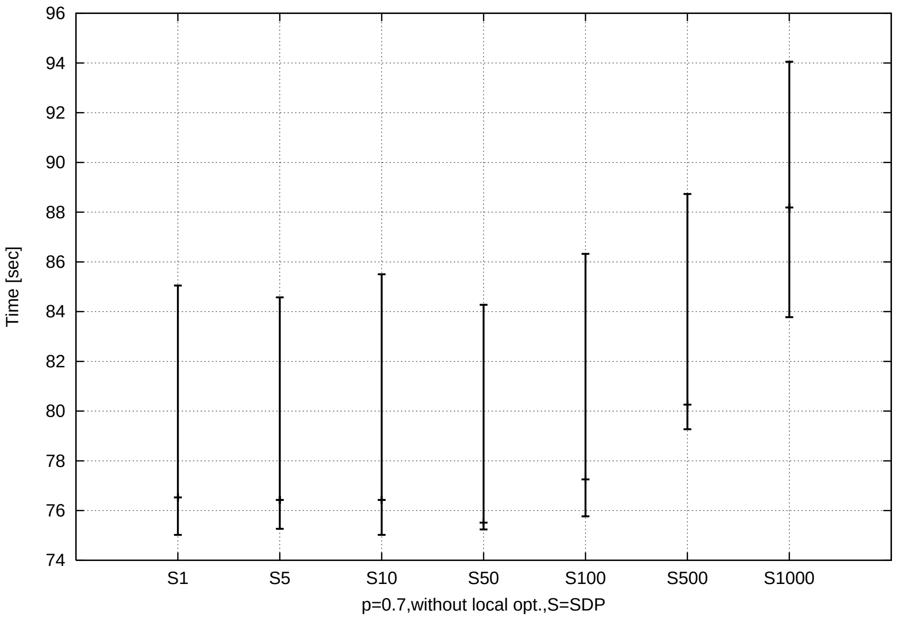


Figure 7: Running times of *SDP* for different numbers of random trials.

4.1.1 Parameters of *GA*

The algorithm *GA* uses the following basic settings for the involved parameters:

- size of the population: 50
- number of generations: 100
- insertion probability for the initial population: 0.5
- probability for crossover: 0.6
- method for crossover: single-point crossover
- probability for mutation: 0.01
- method for mutation: simple mutation (no local optimization).

These basic settings were varied as described below in order to see which settings yield the best results. Every not explicitly mentioned parameter was set to its basic setting.
First of all we tested *GA* for 50, 100 and 200 generations. As can be seen typically in Figure 8 (G50-G100), the relative cut sizes were increased a little, when the number of generations was increased. The same held when the size of the population was increased from 50 to 100 individuals (P100). Figure 9 shows the running times for the test runs from Figure 8.

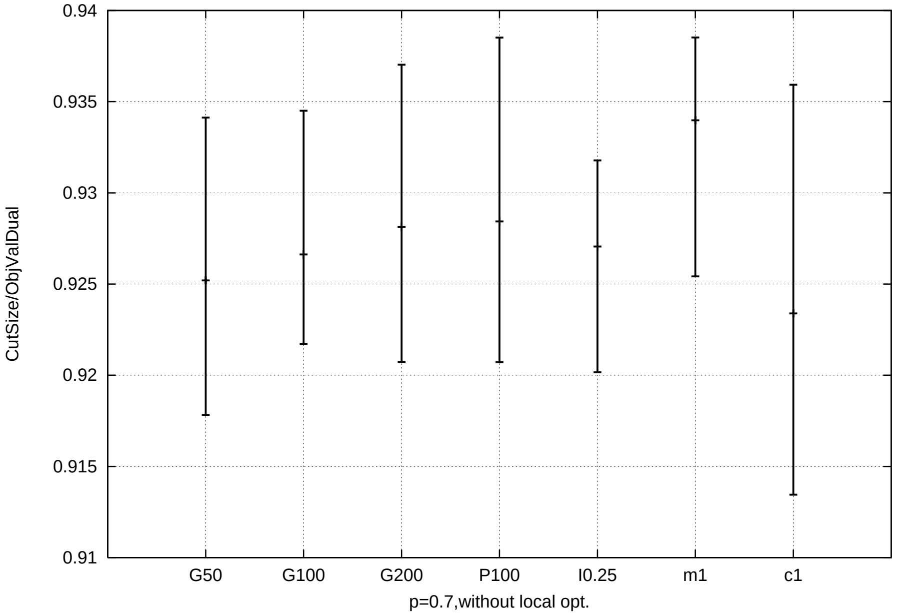


Figure 8: Relative cut sizes of *GA* for $G_{Rand}(0.7)$ without local optimization.

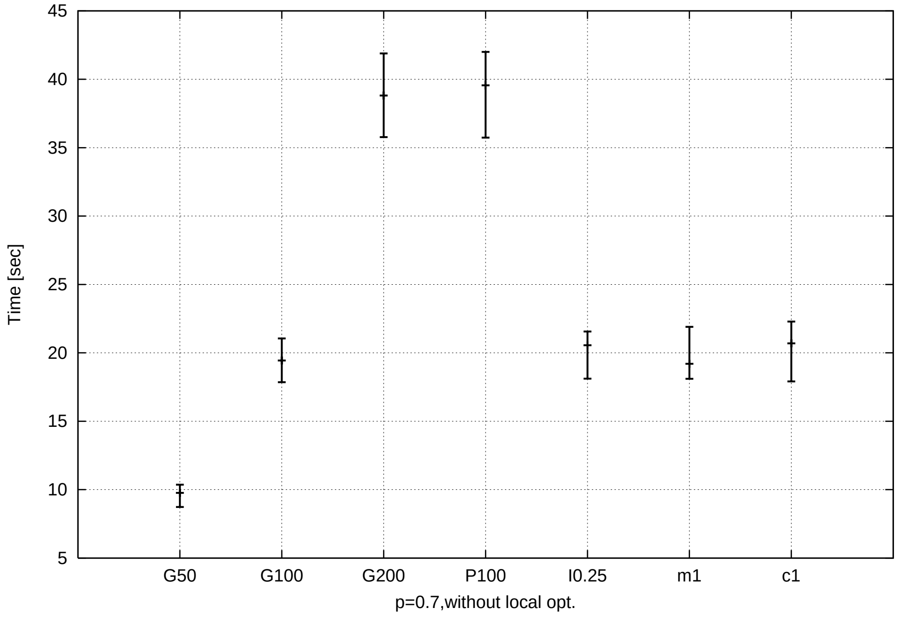


Figure 9: Running times of GA for $G_{Rand}(0.7)$ without local optimization.

The use of different insertion probabilities, say 0.25 instead of 0.5, did not improve the results and also did not change the running times essentially.
When fixed crossover (c1) is used instead of single-point crossover, the running times stay the same. However, the quality of the best solution is improved, but the variance grows which leads to a worse median. The use of local optimization in the mutation operator (m1) improves the size of the computed cut significantly, without increasing the variance and without increasing the running times significantly.
All observations described above hold for graphs of type G_{BiRand} accordingly. Altogether one may say, that different settings of the parameters influence the running times and partially improve the quality of the cut a little, but there seems to be no real best input independent setting. For this reason the basic settings described in the beginning of this section were used in the comparison of all algorithms.

4.2 Comparison of the Algorithms without Local Optimization

As mentioned above, we used 100 random trials for the algorithms *Random* and *SDP*. Concerning *SDP*, the running time is increased, if we use a larger number of random trials, though the quality of the cut is hardly improved, if the number of random trials is increased beyond 100. Therefore, we chose 100 random trials for *SDP*. The choice of an appropriate number of random trials for *Random* is more difficult. On the one hand, the quality of the cut is improved, when the number of random trials is increased, but also the running time is increased, accordingly. Remember that we implemented *Random* due to its simplicity and small running time in order to get some kind of lower bound on the quality of the cuts, the choice of 100 random trials seemed appropriate. For the *genetic algorithm* a population size of 50, 100 generations, single-point crossover, mutation without local optimization, and a probability of 0.5 for building the initial population were used. Altogether, 20 test runs for each of the 6 graph types and every

algorithm except for *D&C* were executed using these settings. The algorithm *D&C* was left out in this comparison because it optimizes locally implicitly. Hence, *D&C* is only compared to the algorithms with local optimization discussed in the next section. Tables 1 and 2 show the results of the test runs. Hereby we present the relative cut sizes $\frac{CutSize}{SDPValDual}$ in Table 1 in order to make the results comparable. Table 1 shows the maximal and minimal relative cut size returned per algorithm and the type of graph. In Table 2 the running times (in seconds) are listed which were used for the calculations presented in Table 1.
Comparing the results, we see that *SDP* always returned the results with the highest quality. Next came *Combinatorial* followed by *CombColorings*, *GA* and *Random*, respectively. Moreover, the relative cut sizes obtained with *SDP* are significantly bigger than the theoretical 0.87-guarantee for the cut size given in [4].

graph type	Random	SDP	Combinatorial	CombColorings	GA
$G_{Rand}(0.3)$ max	0.85343	0.97367	0.92096	0.90675	0.87331
$G_{Rand}(0.3)$ min	0.82443	0.9553	0.88573	0.86645	0.83607
$G_{Rand}(0.5)$ max	0.89087	0.97654	0.94319	0.92867	0.90788
$G_{Rand}(0.5)$ min	0.86927	0.96697	0.90591	0.90578	0.88323
$G_{Rand}(0.7)$ max	0.92244	0.98523	0.96269	0.95323	0.9345
$G_{Rand}(0.7)$ min	0.91119	0.97812	0.9372	0.93819	0.92171
$G_{BiRand}(0.3, 0.0)$ max	0.56383	1	1	1	0.5992
$G_{BiRand}(0.3, 0.0)$ min	0.532	1	0.72583	0.6555	0.55285
$G_{BiRand}(0.7, 0.0)$ max	0.56819	1	1	1	0.58886
$G_{BiRand}(0.7, 0.0)$ min	0.51979	1	1	0.78786	0.5441
$G_{BiRand}(0.7, 0.3)$ max	0.75472	1	0.98706	0.95287	0.78399
$G_{BiRand}(0.7, 0.3)$ min	0.72328	1	0.86947	0.80645	0.7351

Table 1: Relative cut sizes [CutSize/SDPValDual] without local optimization.

graph type	Random	SDP	Combinatorial	CombColorings	GA
$G_{Rand}(0.3)$ max	0.187	76.964	0.008	0.009	8.4
$G_{Rand}(0.3)$ min	0.148	77.289	0.008	0.009	8.172
$G_{Rand}(0.5)$ max	0.239	77.757	0.014	0.017	14.537
$G_{Rand}(0.5)$ min	0.193	78.73	0.013	0.016	14.023
$G_{Rand}(0.7)$ max	0.328	77.254	0.02	0.025	18.806
$G_{Rand}(0.7)$ min	0.34	77.257	0.019	0.026	19.217
$G_{BiRand}(0.3, 0.0)$ max	0.09	86.625	0.004	0.004	5.489
$G_{BiRand}(0.3, 0.0)$ min	0.113	86.625	0.004	0.004	4.618
$G_{BiRand}(0.7, 0.0)$ max	0.18	81.241	0.009	0.009	9.71
$G_{BiRand}(0.7, 0.0)$ min	0.179	81.241	0.009	0.009	9.693
$G_{BiRand}(0.7, 0.3)$ max	0.25	115.715	0.013	0.015	13.436
$G_{BiRand}(0.7, 0.3)$ min	0.25	115.715	0.013	0.015	14.345

Table 2: Running times [sec] without local optimization.

The quality of the cut sizes which were computed by *CombColorings* is slightly worse than the ones calculated by *Combinatorial* though *CombColorings* has bigger running times. It seems that the time spent for finding a vertex coloring is useless because it returns no better results. Compared to these three algorithms, the results of *GA* are quite bad but better than the ones of *Random*. According to our test runs, the fastest algorithm was *Combinatorial*, followed by *CombColorings*, *Random*, *GA* and *SDP*, in this order. The last two ones, namely *GA* and *SDP* are significantly slower than the others.

4.3 Comparison of the Algorithms with Local Optimization

In the comparison of the algorithms with local optimization we used the same parameters as before. For *Random* and *SDP* we used 100 random trials and for *GA* the population size 100 and 50 generations, single-point crossover, mutation without local optimization and a probability of 0.5 for building the initial population.
The local optimization was, except in the case of *GA*, always applied after the computation of the cuts. This means that the basic structures of the algorithms were not changed. They only were extended by local optimization. In the case of *GA*, each individual was locally optimized right after its creation. For all algorithms, local optimization was used.
The results of the experiments with local optimization are given in Table 3 and 4. Table 3 shows the computed minimal and maximal relative cut sizes and Table 4 shows the corresponding running times. Compared to the case without local optimization, *GA* and *Random* became significantly slower as these two algorithms spend a lot of their running times for local optimization and hence the solutions of these two algorithms before local optimization have quite a big distance to the nearest local optimum. The situation is different with *Combinatorial* and *CombColorings*. The running times of these two algorithms are not increased significantly by local optimization which indicates that their solutions are quite near some local optima.

type of graph	Random	SDP	Combinatorial	CombColorings	GA	$D\&C$
$G_{Rand}(0.3)$ max	0.97259	0.97581	0.95692	0.90675	0.97581	0.97474
$G_{Rand}(0.3)$ min	0.96164	0.96479	0.9309	0.86659	0.96542	0.96542
$G_{Rand}(0.5)$ max	0.98052	0.98083	0.9734	0.93968	0.98119	0.98119
$G_{Rand}(0.5)$ min	0.9722	0.97497	0.94953	0.90646	0.97387	0.97566
$G_{Rand}(0.7)$ max	0.98686	0.98736	0.98104	0.96197	0.98729	0.98729
$G_{Rand}(0.7)$ min	0.97996	0.98253	0.96206	0.94106	0.98253	0.98353
$G_{BiRand}(0.3, 0.0)$ max	1	1	1	1	1	1
$G_{BiRand}(0.3, 0.0)$ min	1	1	1	0.6555	1	1
$G_{BiRand}(0.7, 0.0)$ max	1	1	1	1	1	1
$G_{BiRand}(0.7, 0.0)$ min	1	1	1	0.93139	1	1
$G_{BiRand}(0.7, 0.3)$ max	1	1	1	0.95586	1	1
$G_{BiRand}(0.7, 0.3)$ min	1	1	1	0.86678	1	1

Table 3: Relative cut sizes [CutSize/SDPValDual] with local optimization.

type of graph	Random	SDP	Combinatorial	CombColorings	GA	$D\&C$
$G_{Rand}(0.3)$ max	3.065	76.34	0.024	0.009	36.31	99.878
$G_{Rand}(0.3)$ min	2.994	78.609	0.016	0.009	33.323	99.919
$G_{Rand}(0.5)$ max	5.262	79.019	0.04	0.016	54.434	111.02
$G_{Rand}(0.5)$ min	4.725	80.265	0.039	0.017	60.44	105.321
$G_{Rand}(0.7)$ max	7.543	82.894	0.051	0.027	83.741	113.861
$G_{Rand}(0.7)$ min	7.164	81.912	0.041	0.027	84.041	110.141
$G_{BiRand}(0.3, 0.0)$ max	1.572	86.816	0.008	0.004	16.914	96.408
$G_{BiRand}(0.3, 0.0)$ min	1.572	86.816	0.008	0.004	16.914	96.408
$G_{BiRand}(0.7, 0.0)$ max	3.279	84.997	0.011	0.01	35.064	95.08
$G_{BiRand}(0.7, 0.0)$ min	3.279	84.997	0.011	0.009	35.064	95.08
$G_{BiRand}(0.7, 0.3)$ max	5.172	92.506	0.022	0.016	46.234	116.922
$G_{BiRand}(0.7, 0.3)$ min	5.172	92.506	0.022	0.016	46.234	116.922

Table 4: Running times [sec] of the algorithms that refer to Table 3.

Such conclusions based on the increase of running time can not be made for *SDP* and *D&C*,

because *SDP* has quite a big basic running time and so the time needed for the local optimization increases the total running time only marginally. Comparing the relative cut sizes returned by *SDP* in both cases, one can see that local optimization does not improve the relative cut sizes very much. The algorithms which were most improved by local optimization are *GA* and *Random*. Together with the small running time compared to *SDP*, the algorithm *Random* with local optimization offers for the tested graphs the best compromise between small running times and high qualities of the cuts, as long as no lower bound on the cut size is needed. If such a lower bound on the cut size is important for an application, *Random* is not applicable. In these cases the application of *SDP* should be considered.

Apart from the running time and only considering the quality of the cuts the tables show that all algorithms except *Combinatorial* and *CombColorings* return similar results. The results of *Combinatorial* and *CombColorings* are a little bit worse than the others when only the qualities of the cut sizes are considered.

5 Conclusion

As we pointed out in the previous two sections, the algorithm *Random* with local optimization gives the best compromise between small running time and quality of the relative cut size. Comparing the locally optimized versions of *Random* and *GA* one can see that the qualities of their relative cut sizes are quite similar, but *GA* needs a significantly bigger running time. Therefore *GA* is considered to be less applicable than *Random*. Even the advantage that *GA* is quite independent of the structure of a problem and therefore can be easily adapted to similar problems fades compared to the high quality results presented by the pure random strategy. Though *SDP* is one of the slowest algorithms, it has the advantage of a good theoretical lower bound on the minimal cut size proved by Goemans and Williamson [4]. Furthermore, in all of our experiments the theoretically lower bound of achieving 0.87... of the optimum was exceeded. Therefore it would be interesting to put effort on improving the running times of this algorithm. Both combinatorial algorithms, *Combinatorial* and *CombColorings*, returned quite similar results. Without local optimization they obtained better results than *Random* but with local optimization their results were slightly worse than the ones presented by the pure random strategy. The last tested algorithm, *D&C*, returned results similar or even better than *SDP*, but it also had the largest running time.

Altogether we observed that except for *SDP*, local optimization is an important tool to achieve large relative cut sizes. Therefore the development of new algorithms based on local optimization seems to be desirable.

References

[1] F. Alizadeh, *Interior Point Methods in Semidefinite Programming with Applications to Combinatorial Optimization*, SIAM Journal on Optimization 5, 13-51, 1995.

[2] K. Fujisawa, M. Kojima and K. Nakata, *SDPA (Semidefinite Programming Algorithm) — User's Manual*, Research Reports on Mathematical and Computing Sciences, Series B: Operations Research, B-308, Department of Mathematical and Computing Science, Tokyo Institute of Technology, 2-12-1 Oh-Okayama, Meguro-ku, Tokyo 152, Japan, 1996. `(ftp.is.titech.ac.jp/pub/OpRes/software/SDPA)`

[3] M. R. Garey, D. S. Johnson and L. Stockmeyer, *Some Simplified NP-complete Graph Problems*, Theoretical Computer Science 1, 257-267, 1976.

[4] M. X. Goemans and D. P. Williamson, *Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming*, Journal of the ACM 42, 1115-1145, 1995.

[5] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison Wesley, 1989.

[6] T. Hofmeister and M. Hühne, *Semidefinite Programming and its Applications to Approximation Algorithms*, in: E.W. Mayr, H. J. Prömel and A. Steger (eds.), *Lectures on Proof Verification and Approximation Algorithms,* 263-298, Lecture Notes in Computer Science Tutorial 1367, Springer, Berlin, 1998.

[7] T. Hofmeister and H. Lefmann, *A Combinatorial Design Approach to* MaxCut, Random Structures & Algorithms 9, 163-175, 1996.

[8] H. J. Karloff, *How good is the Goemans-Williamson* MaxCut *algorithm?*, Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC 96), 427-434, 1996.

[9] S. Khuri, Th. Bäck and J. Heitkötter, *An Evolutionary Approach to Combinatorial Optimization Problems*, Proceedings of the 22nd Annual ACM Computer Science Conference, ed. D. Cizmar, 66-73, ACM Press, New York, 1994.

[10] W. M. Spears, K. A. De Jong, Th. Bäck, D. B. Fogel and H. de Garis, *An Overview of Evolutionary Computation*, Machine Learning: ECML-93, in: P. B. Brazdil (ed.), Lecture Notes in Artificial Intelligence 667, Springer, Berlin, 442-459, 1993.

[11] L. Vandenberghe and S. Boyd, *Semidefinite Programming*, SIAM Review 38, 49-95, 1996.