Chapter 3

Approximation

3.1 Introduction

Hard combinatorial optimization problems can not always be solved to optimality within a reasonable amount of time. Therefore, we sometimes resort to methods which generate provably good, but not necessarily optimal solutions efficiently, that is, in polynomial time. Such algorithms are called *approximation algorithms*. Two things are important in this informal definition: the fact that the algorithm runs in *polynomial time*, and the fact that the computed solution has to be a *provably good* solution. The latter means that the algorithms computes a solution that is in some sense close to an optimal solution for *any* given input instance. In other words, even for the worst possible input instance, the algorithm computes a solution that is close to the optimum. It is therefore also said that such an algorithm has a *performance guarantee*, since it can be guaranteed a priori, even without looking at the input instance, that the solution will be close to the optimum.

It is important to see the difference between this paradigm and the paradigm of local search algorithms: In a local search algorithm, we generally neither have a polynomial time bound on the computation time of the algorithm, nor do we generally have a performance guarantee for the quality of the solution (although there are also examples where such results can be proved also for local search algorithms; see, for instance, the results on the optimality of simulated annealing algorithms in [AL97]).

3.2 Worst-Case Ratio

To define the worst-case ratio WCR of an algorithm H for a minimization problem P we first define what it means that H is an α-approximation algorithm. Take an instance I of P. Let the objective value of the solution generated by the approximation algorithm H be $H(I)$ and the optimum value be $\mathrm{OPT}(I)$. Assuming that $\mathrm{OPT}(I) > 0$, the ratio

$$R(I) = \frac{H(I)}{\mathrm{OPT}(I)}$$

is a measure of the quality of the approximate solution. Note that $R(I) \geq 1$.

We say that H has a *performance guarantee* of α for some $\alpha \geq 1$ if $H(I) \leq \alpha \mathrm{OPT}(I)$ for all $I \in P$, that is, for *every* instance I of P the objective value computed by H is at most α times the optimal objective value. Assuming that $\mathrm{OPT}(I) > 0$, this is the same as to say that $R(I) \leq \alpha$ for all instances $I \in P$. Moreover, H is called an *α-approximation algorithm* if it has a performance guarantee of α and additionally has a polynomial running time, for any instance I of P.

Given an algorithm H we are of course interested in finding the smallest possible α that satisfies this condition. This gives rise to the definition of the worst-case ratio (WCR) for algorithm H; it is formally defined by

$$WCR = \inf\{\alpha \geq 1 \mid H(I) \leq \alpha \,\mathrm{OPT}(I) \text{ for all } I \in P\}\,.$$

If we assume that $\mathrm{OPT}(I) > 0$ for all instances I, we can alternatively write

$$WCR = \sup_{I \in P} \frac{H(I)}{\mathrm{OPT}(I)}.$$

Note that the worst-case ratio WCR is again at least 1. Note further that this definition includes two statements: first, that H has a performance guarantee of $\alpha = WCR$, second that for every $\beta < WCR$ there exists an instance $I \in P$ such that $H(I) > \beta \mathrm{OPT}(I)$.

For maximization problems we can define a similar quality measure.

$$WCR = \sup\{\alpha \leq 1 \mid H(I) \geq \alpha \,\mathrm{OPT}(I) \text{ for all } I \in P\}\,,$$

and whenever $\mathrm{OPT}(I) > 0$, we can alternatively write

$$WCR = \inf_{I \in P} \frac{H(I)}{\mathrm{OPT}(I)}\,.$$

This ratio is at most 1. Note that, due to symmetry reasons, in the literature the worst-case ratio for maximization problems is sometimes also defined as

$$WCR = \inf\{\alpha \geq 1 \mid H(I) \geq \frac{1}{\alpha} \,\mathrm{OPT}(I) \text{ for all } I \in P\}\,,$$

which has a value at least 1. However, we will use the first definition in the sequel. Moreover, unless stated otherwise let us assume for convenience that we consider only problems where $\mathrm{OPT}(I) > 0$.

Note that the WCR depends on the algorithm itself. It may be that for a particular problem P there are algorithms with a better worst-case ratio than others. We will generally be interested in worst-case ratios of polynomial time algorithms only. How do we prove that a certain value R is the WCR of a particular algorithm H for a particular minimization problem P? To show that R is a lower bound on the WCR of H, we must show that there are instances $I \in P$ for which $H(I)/\mathrm{OPT}(I)$ is arbitrarily close to R, ideally equal to R. So this part consists of finding instances where the algorithm performs badly. To show that R is an upper bound for the WCR, we have to prove that for any instance $I \in P$ we have $H(I) \leq R\,\mathrm{OPT}(I)$. But how should we do this unless we already know the optimal solution value $\mathrm{OPT}(I)$? This is generally the most challenging part of the proof, and it relies on a *lower bound* argument, also called a *dual* argument (for maximization problems, this would

be an upper bound argument) . So suppose we know that the the optimal value $\mathrm{OPT}(I)$ must at least be as large as $LB(I)$, for any instance I. (For example in the travelling salesman problem, any tour, including any optimal one, is at least as long as the edge lengths in a minimum spanning tree. Why?) Then if we can show that $H(I) \leq R\,LB(I)$ we are done. Since then we immediately obtain $H(I) \leq R\,LB(I) \leq R\,\mathrm{OPT}(I)$. We start with examples for the Node Cover problem and the Knapsack problem.

3.2.1 Node Cover.

In a graph $G = (V, E)$ a node cover is a subset of the nodes $W \subseteq V$ with the property that each edge is incident with at least one of the nodes of W. The objective of the node cover problem (NC) is to find a node cover with a minimum number of nodes. This problem is NP-hard.

Notice: The corresponding recognition problem is NP-complete, and the recognition problem reduces to the optimization problem, so in the literature it is often said that the optimization problem is therefore NP-hard. Since the class NP itself is only defined in terms of recognition problems, however, the optimization problem cannot be part of the class NP. Therefore, a problem X (recognition or optimization) is often called NP-hard if any problem in NP polynomially *reduces* to X. Alternatively, one can require a polynomial *transformation* here; this would result in a smaller class of NP-hard problems. The advantage of the definition with a *reduction*, however, is that the class then also contains all optimization problems, and not only recognition problems.

Consider the following two algorithms to solve the node cover problem.

Algorithm 1
Input: A graph $G = (V, E)$
Output: A node cover W
$W = \emptyset$
while $E \neq \emptyset$
do
    Choose a node $v \in V$ with the largest degree;
    $W = W \cup \{v\}$;
    Remove v and all edges incident to v;
od

Algorithm 2
Input: A graph $G = (V, E)$
Output: A node cover W
$W = \emptyset$
while $E \neq \emptyset$
do
    Choose an arbitrary edge $\{v, w\} \in E$;
    $W = W \cup \{v, w\}$;
    Remove v and w and all edges incident to them;
od

Algorithm 1 is a greedy type of algorithm; the hope is that always selecting the largest degree node will 'cover' as many edges as possible, and this results in a small overall node cover. At first glance it seems to more reasonable than Algorithm 2. Indeed in practical applications, we see that Algorithm 1 outperforms Algorithm 2 on average. However, we will see that the WCR of Algorithm 2 is much better than the WCR of algorithm 1.

Consider the instance shown in Figure 3.1 and apply algorithm 1 to it.

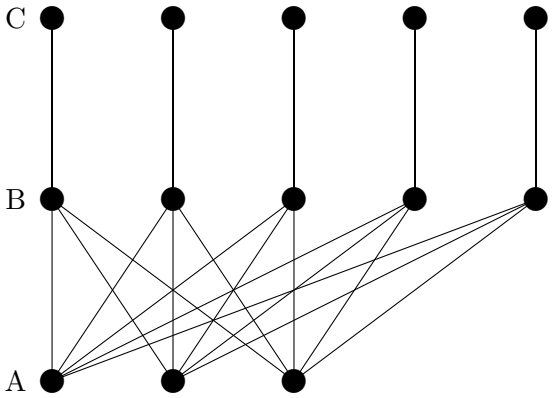

Figure 3.1: Algorithm 1: Instance 1.

In this instance we break ties by choosing nodes in the lowest possible layer first. So first nodes of the A-layer, then nodes of the B-layer, and then nodes of the C-layer. Doing so we see that all nodes in the lowest two layers are chosen by algorithm 1, whereas the nodes in the middle layer form an optimum node cover. This example can easily be extended with more nodes in each layer. However, if the number of nodes in the middle layer and the top layer is n, then the number of nodes in the bottom layer is at most $n-1$. Thus, the node cover found by algorithm 1 is, in this example, a little less than twice the optimum node cover in the worst case. This example shows that the WCR of algorithm 1 is at least 2. Does this show that the WCR of algorithm 1 is 2? No, since there may be instances on which algorithm 1 performs even worse. Such instances are defined next.

In Figure 3.2, we see an example of an instance where the node cover constructed by algorithm 1 consists of the nodes in the two bottom layers, and the optimum node cover consists of the middle layer, as for instance 1. Now, however the number of nodes in the lowest layer is larger than the number of nodes in the middle layer. To be exact this number is $\sum_{i=2}^{\lfloor \frac{n}{2} \rfloor} \lfloor \frac{n}{i} \rfloor + \lceil \frac{n}{2} \rceil$ which is of the order $n\log\, n$, where n is the number of nodes in the middle layer. The idea is that there are $\lfloor \frac{n}{2} \rfloor$ nodes in the bottom layer that are connected to two different nodes from the middle layer each, $\lfloor \frac{n}{3} \rfloor$ nodes in the bottom layer that are connected to three different nodes from the middle layer each, and so forth up to one node in the bottom layer that is connected to all nodes from the middle layer. It is easy to verify that algorithm 1, in the worst case, chooses all nodes from the bottom layer, and then all nodes from the middle layer.

This implies that we have instances I_n, $n \in I\!N$, for which algorithm 1 finds a node cover which consists of $\Omega(n \log n)$ nodes, whereas the optimum node cover consists of only n nodes. In other words, we have found a family of instances where the ratio of algorithm 1's performance relative to the optimal solution can be larger than any constant factor of the optimal node

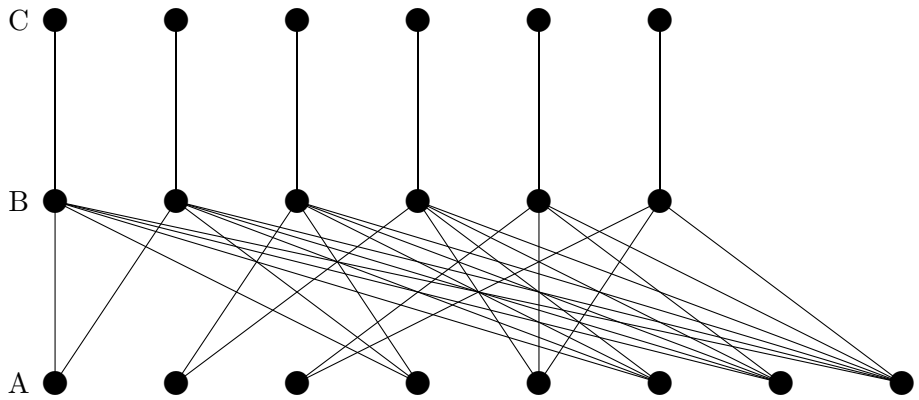

Figure 3.2: Algorithm 1: Instance 2.

cover, since

$$\text{Algorithm\_1}(I_n) \geq \log n \ \text{OPT}(I_n)\,.$$

This, however, means that the worst-case ratio of algorithm 1 is unbounded.

For algorithm 2 it is also easy to establish a lower bound of 2 on the WCR. Simply take a graph consisting of $2n$ nodes and n edges forming a perfect matching. All nodes will be chosen in the node cover, where only n nodes suffice. However, surprisingly enough, this lower bound turns out to be an upper bound as well; in other words, we can show that algorithm 2 has a performance guarantee of 2.

To see this we use a lower bound argument as explained already above. In the case of algorithm 2, the lower bound we are going to use is the cardinality of a matching: for any matching M, every node cover has to select at least one of the two nodes from every edge $e \in M$. Observe now that the edges chosen by algorithm 2 form a matching M. Suppose that this matching consists of k edges. Then the node cover constructed by algorithm 2 consists of $2k$ nodes. But according to our argument above, any node cover must consist of at least k nodes. Thus $k \leq \text{OPT}(I)$. But this yields $H(I) = 2k \leq 2\,\text{OPT}(I)$. Together with the fact that there is an instance such that $H(I) = 2\,\text{OPT}(I)$, we have proven that $WCR = 2$ for algorithm 2. Since the algorithm obviously runs in polynomial time (even linear time), it is in fact a 2-approximation algorithm for the node cover problem.

3.2.2 Knapsack.

We are given a set of n items, each with a positive size s_j $(j = 1, \ldots, n)$ and a positive value v_j $(j = 1, \ldots, n)$. The feasible solutions are the subsets of the set of items with cumulative weight at most S, the knapsack size. The objective is to find a feasible solution of maximum value. Formulated as a $0-1$ integer program, we use binary variables x_j, $j = 1, \ldots, n$, with the intended meaning that $x_j = 1$ if item j is selected to be in the knapsack, $x_j = 0$ otherwise.

The problem reads:

$$\begin{array}{ll} max & \sum_{j=1\ldots n} v_j\, x_j \\ s.t. & \sum_{j=1\ldots n} s_j\, x_j \leq S \\ & x_j \in \{0,1\} \text{ for all } j = 1,\ldots,n \end{array}$$

We consider the following two algorithms for finding good solutions to the knapsack problem.

Algorithm 1: Sort the items by non-increasing relative value, i.e., by non-increasing ratios v_j/s_j. Choose in this order greedily items until choosing another item would exceed the capacity S.

Algorithm 2: Take the item with the highest value.

Algorithm 1 works fine if the number of items in the knapsack will not be too small. The solution constructed by this algorithm is in fact related to an optimal solution of the linear programming relaxation of the above $0-1$ integer programming formulation (see below) .

Algorithm 2, however, works fine if among the items there is one with a size close to S. We will show that both algorithms have a bad WCR, but because they handle different instances well we can simply run both algorithms and take the best solution.

The following instance is an example on which algorithm 1 performs badly.

$S = 10$

i	1	2
v_i	2	10
s_i	1	10

The greedy solution is $(x_1, x_2) = (1, 0)$ with value 2, whereas the optimal value is 10 for solution $(x_1, x_2) = (0, 1)$. This gives a ratio of $\frac{1}{5}$. In fact, it is not difficult to see that a value for the ratio arbitrarily close to 0 can be obtained.

The following instance is an example on which algorithm 2 performs badly.

$S = 8$

i	1	2	3	4	5	6	7	8
v_i	1	1	1	1	1	1	1	1
s_i	1	1	1	1	1	1	1	1

An item with the highest value is item 1 (for instance). Thus algorithm 2 gives value 1, whereas the optimal value is 8. Thus, the ratio of algorithm 2 for this instance is $\frac{1}{8}$. Again, we can define instances for algorithm 2 with ratio arbitrarily close to 0.

It is not hard to check that algorithm 2 performs well on the first instance, and that algorithm 1 performs well on the second instance. This suggests that the following algorithm might have a better WCR than the singleton algorithms 1 and 2:

Algorithm H: take the best solution of the two solutions generated with algorithms 1 and 2. For example, consider the following instance.

$C = 20$

i	1	2	3
v_i	12	10	10
s_i	11	10	10

Algorithm H gives a value of 12, whereas the optimum value is 20.

Theorem 3.1. *Algorithm H has a WCR*$= \frac{1}{2}$.

The proof of this theorem relies on an upper bound argument (recall that Knapsack is a maximization problem). The upper bound on the optimal solution is obtained by considering the *linear programming relaxation* of the above $0-1$ integer programming formulation of the problem:

$$\begin{array}{ll} max & \sum_{j=1\ldots n} v_j\, x_j \\ s.t. & \sum_{j=1\ldots n} s_j\, x_j \leq S \\ & 0 \leq x_j \leq 1 \text{ for all } j = 1, \ldots, n\,. \end{array}$$

The optimal solution of this LP relaxation is an upper bound on the optimal solution of the Knapsack problem: any (optimal) solution of the exact $0-1$ formulation of the Knapsack problem is also a solution of the LP relaxation, therefore the optimum value for the LP relaxation cannot be smaller. (In the LP relaxation we are even allowed to take fractions of items, whereas in the original Knapsack problem we are not allowed to do so.) In fact, the optimal solution of this LP relaxation has a very simple structure: Assume that the order of relative values of the items is $v_1/s_1 \geq v_2/s_2 \geq \cdots \geq v_n/s_n$, and let k be the smallest index such that $\sum_{j=1}^{k} s_j > S$. Then an optimal LP solution is given by $x_j = 1$ for all $j = 1, \ldots, k-1$, $x_k = (S - \sum_{j=1}^{k-1} s_j)/s_k$, and $x_j = 0$ for all $j = k+1, \ldots, n$. This can be proved by LP duality, giving a dual feasible solution with the same objective function value. It is, however, also intuitively clear: If allowed to take also fractions of items, one would fill the knapsack with the items with the largest relative values, until the first item does not fit anymore. From this item, one would take a fraction as large as possible such that it still fits into the knapsack. So an optimal LP solution is very closely related to the solution of the greedy algorithm 1. The further details of the proof are left as an exercise.

3.2.3 LP-based approximation.

There exist several ways of using an LP relaxation of a combinatorial optimization problem to design good approximation algorithms.

We have seen in the knapsack example that the greedy solution (algorithm 1) is closely related to an optimal LP solution (this was shown in an exercise). In this case we were dealing with a maximization problem, and the solution of the LP relaxation was an upper bound on the optimal solution for the Knapsack problem. The greedy solution computed by algorithm 1 can be interpreted as follows: We could have considered an optimal LP solution first, and from this 'fractional solution' we construct a feasible integral solution that is not 'too far away' from the LP solution. To do so, we just remove the (only) fractional item k from the fractional solution (this corresponds to the solution computed by the greedy algorithm 1). The loss in value by doing this is at most the value v_k of that item. This might have led to loosing more than half of the value of the fractional solution, but the combination of algorithm 1 with algorithm 2 protected us against this risk: we considered alternatively to choose the most valuable item. The basic underlying idea, however, is:

- Formulate the problem as an integer linear program
- Solve the corresponding LP relaxation (that is, relaxing the integrality condition)

- From this 'fractional solution', somehow construct a feasible solution for the original problem
- Try to prove a performance guarantee, using the fact that the optimal LP solution value is a bound for the value of any optimal integral solution

Note that for maximization problems the LP relaxation gives an upper bound, and for minimization problems a lower bound on the optimal integral solution value.

Instead of using the LP relaxation as an upper bound we might use as well the dual of the LP relaxation, or even the integer dual. Let us consider this for a minimization problem, where LP relaxations provide us with lower bounds. Let $D(I)$ be the optimum of the dual of the linear relaxation, $ID(I)$ the optimum of the integer dual program. We have then

$$ID(I) \leq D(I) = LP(I) \leq \mathrm{OPT}(I) \leq H(I)$$

for every algorithm H. Whenever $H(I)$ is at most a constant factor larger then one of the values left to $\mathrm{OPT}(I)$, for any instance I, we have proved a performance guarantee for H.

The second algorithm for the minimum node cover problem has an interpretation within linear programming as well, based on the idea of 'rounding' the LP solution. We can thus prove a $WCR \leq 2$ for this algorithm even for the more general *weighted* node cover problem. This will be discussed in an exercise.

3.3 Randomized Algorithms.

Randomization is a relatively simple, yet successful technique for developing approximative algorithms with good expected behavior. The basic idea is that in worst case analysis often only few or even artificial instances are responsible for a bad worst case performance guarantee of a particular algorithm. By using randomization within the algorithm itself, it is often possible to obtain better performance guarantees on average (that is, in expectation), since the worst case instances are not able to 'fool the algorithm' in all of the cases. Quite often, we can even derandomize such algorithms (in polynomial time) and are thus able to find a deterministic algorithm with a finite performance guarantee. We illustrate this idea on the MAXSAT problem.

The problem MAXSAT can be described as follows. There are n binary variables $x_1, \ldots, x_n$. Each variable must be assigned one of two values, namely *true* or *false*. These variables are present in a set of m so-called clauses $C_1, \ldots, C_m$. A clause is a disjunction of variables, possibly some of them in negated form, for example $(x_1 \vee \overline{x}_2 \vee x_3)$, where $\overline{x}_2$ is the negation of x_2 ($\overline{x}_2$ = true if and only if x_2 = false). The (possibly negated) variables that appear in any clause are called *literals*. A clause evaluates to true if at least one of its literals is true, otherwise it is false. An instance of MAXSAT is defined by n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$. The problem is to find a truth assignment for the variables such that a maximum number of clauses is satisfied, that is, evaluates to true.

Obviously, the algorithm that simply sets all variables to true can have an arbitrarily bad performance. In fact, any algorithm that chooses a fixed truth assignment without having looked at the particular instance before may be arbitrarily bad. However, this simple algorithm can be improved upon very easily by determining a random truth assignment of the variables: Each variable is assigned true or false, each with probability $\frac{1}{2}$, and for each variable independently. The probability that a clause is satisfied can now be computed as follows: if a clause contains k literals, the probability that all literals evaluate to false is $(\frac{1}{2})^k$. Thus, the probability that at least one of the literals is true is $1-(\frac{1}{2})^k$. This is the probability that the clause is evaluated to true. This is also the contribution of the clause to the expected objective function value of the problem. So if we denote by $E(x)$ the expected number of fulfilled clauses, any clause C_i, say with k_i literals, contributes with $1-(\frac{1}{2})^{k_i}$ to $E(x)$. Note that each clause therefore contributes with at least $\frac{1}{2}$ (k_i may be 1) to the expected objective function value $E(x)$, and that therefore this algorithm has an expected objective function value of at least $\frac{1}{2}m$. Clearly, since m, the total number of clauses, is an upper bound on the optimum, we have that $E(x) \geq \frac{1}{2}\mathrm{OPT}$, an expected performance guarantee of 1/2. In general, if we know that the minimum number of literals in each clause is at least k, then the expected performance guarantee is $\alpha_k := 1-(\frac{1}{2})^k$.

We can derandomize this algorithm as follows. Suppose that we have computed the expected number of true clauses under the condition that x_1 is true: $E(x \mid x_1 = \text{true})$, and under the condition that x_1 is false: $E(x \mid x_1 = \text{false})$. We can compute these conditional probabilities similar to the way we compute $E(x)$. Then $E(x) = \frac{1}{2}E(x \mid x_1 = \text{true}) + \frac{1}{2}E(x \mid x_1 = \text{false})$. Thus, at least one of the conditional probabilities is greater than or equal to $E(x)$. Let us fix the corresponding value for x_1, and then repeat the process for the other variables. Then the final (deterministic!) truth assignment has a value of at least $E(x)$. Thus, the performance guarantee of this algorithm is at least $\frac{1}{2}$.

To illustrate this, consider the following instance of MAXSAT with 4 variables x_1, x_2, x_3, and x_4, and the following 10 clauses.

$$\begin{array}{lllll} (x_1) & (x_3) & (x_1, x_2) & (\overline{x_1}, x_3) & (\overline{x_1}, \overline{x_2}, x_4) \\ (\overline{x_2}) & (\overline{x_4}) & (\overline{x_3}, \overline{x_4}) & (x_1, \overline{x_2}, x_3) & (\overline{x_1}, x_2, \overline{x_3}, x_4) \end{array}$$

Since each literal that is not fixed has probability $\frac{1}{2}$ to be true, we get the following expected values for the number of clauses that evaluate to true (the first column shows the two alternatives for x_1, the second the two alternatives for x_2, given that x_1 is fixed true.

Clauses	nothing fixed	$x_1$false	$x_1$true	$x_1$true $x_2$false	$x_1$true $x_2$true
(x_1)	0.5	0.	1.	1.	1.
$(\overline{x_2})$	0.5	0.5	0.5	1.	0.
(x_3)	0.5	0.5	0.5	0.5	0.5
$(\overline{x_4})$	0.5	0.5	0.5	0.5	0.5
(x_1, x_2)	0.75	0.5	1.	1.	1.
$(\overline{x_3}, \overline{x_4})$	0.75	0.75	0.75	0.75	0.75
$(\overline{x_1}, x_3)$	0.75	1.	0.5	0.5	0.5
$(x_1, \overline{x_2}, x_3)$	0.875	0.75	1.	1.	1.
$(\overline{x_1}, \overline{x_2}, x_4)$	0.875	1.	0.75	1.	0.5
$(\overline{x_1}, x_2, \overline{x_3}, x_4)$	0.9375	1.	0.875	0.75	1.
Expected value	6.9375	6.5	7.325	8.	6.75

...

The continuation of this process is illustrated in the tree of Figure 3.3. The final truth assignment of the derandomized algorithm will therefore be $x_1 = \text{true}$, $x_2 = \text{false}$, $x_3 = \text{true}$, and $x_4 = \text{false}$. Notice that the final output of the derandomized algorithm depends on the order in which the variables are treated in the derandomization process. However, irrespective of this order we know that the above stated performance bound of α_k holds if k is the minimum number of literals per clause. Notice also that this derandomization process is a polynomial time algorithm.

3.4 Lower bounds on Approximations

In the previous subsections we gave examples of problems for which a specific approximation algorithm had a constant WCR, or at least a constant performance guarantee. However, this does not give any guarantee that the given algorithm is best possible in this sense, that is, we can not be sure whether there are algorithms around the corner with a better performance guarantee or WCR. It is an interesting question what the best WCR is that we can expect over all algorithms for a specific problem. It is a trivial fact that for a minimization problem each algorithm has a WCR of at least 1, and at most 1 for a maximization problem. Sometimes we can even reach this bound. On the other hand, sometimes we can prove that there is no algorithm with a better WCR than a certain $r > 1$, unless P=NP. Note that this is a statement on all possible algorithms that can be devised for a certain problem. For such problems there is even a complexity class, introduced in the early nineties, named APX.

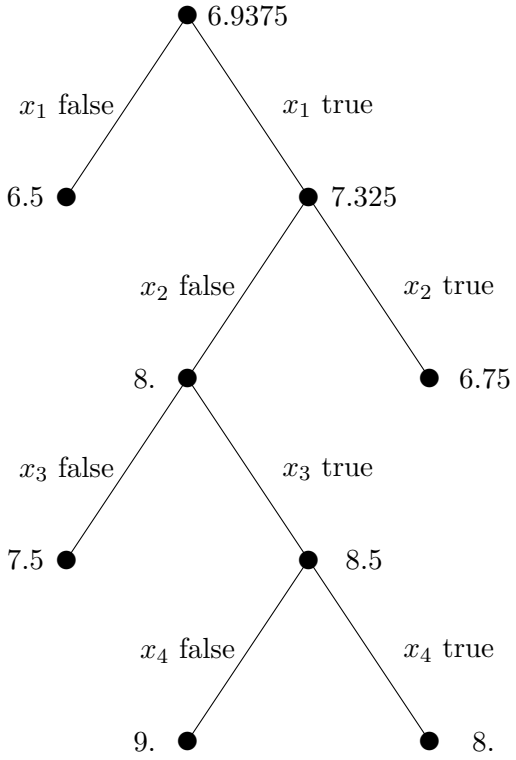

Figure 3.3: Derandomization for a MAXSAT instance

Loosely said, for any APX-hard minimization (maximization) problem P, there exists an $r > 1$ ($r < 1$) such that there cannot be an r-approximation algorithm for P, unless P=NP.

We are, however, not going into the details of the APX complexity class and the associated approximation-preserving reductions. In many cases also simple proofs can be obtained that show that that an optimization problem P cannot be approximated by a certain constant ratio, unless P=NP. These so-called *gap-technique* proofs work typically as follows. We choose some NP-complete problem P'. The choice for P' is often the decision version of the problem P, but maybe another problem as well. We then show that we can polynomially reduce the NP-complete problem P' to the problem P in such a way that an approximation algorithm for P would yield a polynomial time algorithm to solve P'. The latter would then yield P=NP.

As an example we consider the graph coloring problem. It is well-known that the recognition problem whether the nodes of a graph $G = (V, E)$ can be feasibly colored with k colors is NP-complete, for any fixed $k \geq 3$. (Note that for $k = 2$ the problem is equivalent to the recognition problem for bipartite graphs, since a graph is 2-colorable if and only if it is bipartite; this is a polynomial time solvable problem.) Now suppose that you have an α-approximation algorithm for the problem to find a feasible coloring with as few colors as possible, with some constant $\alpha < \frac{4}{3}$. In other words, if $G = (V, E)$ has an optimal coloring with l colors, your algorithm will feasibly color the graph with at most αl colors, in polynomial time. Now suppose a graph $G = (V, E)$ is given. If $G = (V, E)$ is 3-colorable, then your algorithm would be able to color it with at most $x = \alpha\, 3$ colors, and since $\alpha < \frac{4}{3}$, $x = 3$. If $G = (V, E)$ is not 3-colorable, in the optimal solution at least 4 colors are required,

so your algorithm would also require at least 4 colors. But this yields the following: If your algorithm finds a coloring with 3 colors, then we know that we have a yes-instance of the 3-coloring problem. On the other hand, if your algorithm finds a coloring with 4 or more colors then we know that we do NOT have a yes-instance of the 3-coloring problem. In other words your algorithm solves, in polynomial time, the recognition problem whether a graph is 3-colorable, which is NP-complete. Thus, we have proved the following theorem.

Theorem 3.2. *Unless P=NP, there is no approximation algorithm for the node coloring problem with a performance guarantee strictly less than* $\frac{4}{3}$.

For each NP-hard minimization problem there is a $\rho \in [1, \infty]$ such that for any number r larger that ρ there is an algorithm with WCR $\leq r$, and for any number r less than ρ there is no such algorithm, unless P=NP. This threshold ρ is simply the infimum of the performance guarantees α of all possible α-approximation algorithms, assuming P$\neq$NP. We have just shown that for the coloring problem, this threshold $\rho \geq \frac{4}{3}$, assuming P$\neq$NP. The exact value of this value ρ is known in some cases. For most problems, however, the gap between the best known α-approximation algorithm and the best known β-inapproximability result for some value $\beta \leq \alpha$ is quite large, sometimes not even bounded by a constant. A whole branch of researchers within the field uses and invents sophisticated techniques to narrow this gap for the usual suspect problems in discrete optimization. The definition of the complexity class APX and subsequent developments, including the famous PCP-theorem, are examples of the gradual success that is achieved in this area...

3.5 The TSP

For TSP instances we can in general not derive a finite WCR, unless P=NP, i.e., $\rho = \infty$. This result is derived from the NP-completeness of HAMILTON CYCLE (HC).

What we still can do is to restrict our instances. We can limit the classes of graphs we consider, or we can (and will) restrict the objective functions, i.e., the distances. In fact, in the sequel we restrict ourselves to *metric* TSP instances for which the distances satisfy the *triangle inequality*, that is, we have that $d_{ik} \leq d_{ij} + d_{jk}$ for all i, j, k in V.

The double tree algorithm (TT).
Algorithm TT consists of four phases. In the first three phases, we construct an Euler-cycle that we convert into a Hamilton-circuit in Phase 4.

Phase 1. Construct a minimum spanning tree with respect to the distance function d.
Phase 2. Double all edges in the tree to obtain an Eulerian graph.
Phase 3. Determine an Euler-cycle in the Eulerian graph determined in Phase 2. Since the Eulerian graph is connected (it contains the minimum spanning tree as a subgraph), the cycle contains each vertex at least once.

Example Consider the following 7-city TSP instance.

	1	2	3	4	5	6	7
1		1	2	2	3	4	5
2			1	1	3	3	4
3				2	3	2	3
4					1	2	3
5						1	4
6							1
7							

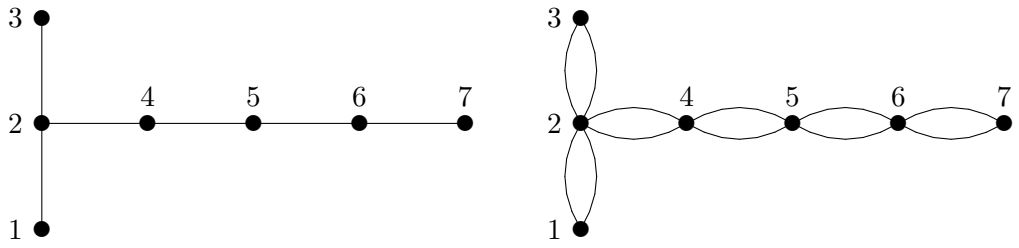

Figure 3.4: Phases 1 and 2.

The Euler-cycle generated in phase 3 is 1232456765421.

Phase 4. Convert the Euler-cycle into a Hamilton-cycle by applying *short-cuts*, i.e., replace a pair of edges $\{i, j\}$, and $\{j, k\}$ by $\{i, k\}$. We can only do this if j appears somewhere else in the Euler circuit.

This operation can be performed on any cycle. Its effect is that a new cycle is constructed with one edge less; in this cycle, there is one vertex that is visited one time less in comparison

with the previous cycle. Due to the assumption that the length function satisfies the triangle inequality, we have that applying a short-cut does not increase the length of the cycle.

Lemma 3.3. *Let $G = (V, E)$ be a complete graph, and let $d : E \to R^+$ be a distance function on the edges, which satisfies the triangle inequality. Let C be a cycle in this graph with total length $d(C)$. If C' is a cycle constructed from C by applying short-cuts, then we have that the total length of C' is no more than $d(C)$.*

We apply a short-cut on each second appearance of a vertex j. Therefore, each vertex remains in the cycle at least once. After the short-cut has been applied to all second appearances of the vertices, the cycle contains each vertex exactly once, that is, we have constructed a Hamilton-circuit.

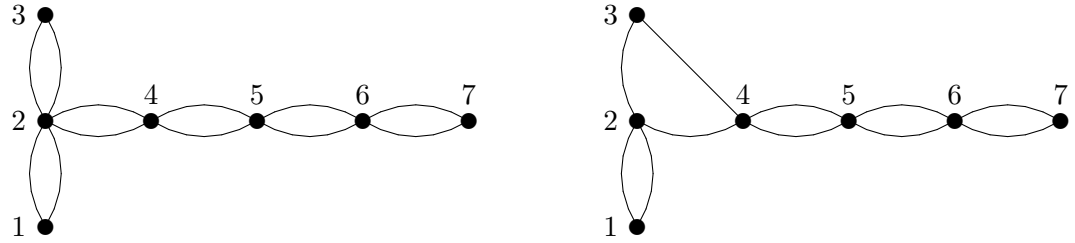

Figure 3.5: Phase 4: replacing 324 by 34

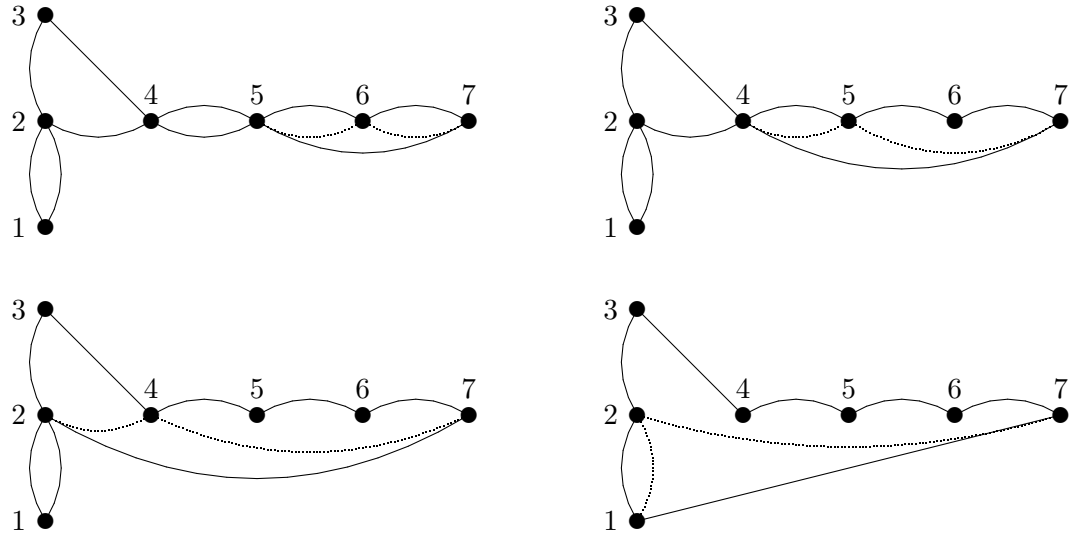

Figure 3.6: Phase 4: further replacements

We now show that Algorithm TT will never produce a solution with length more than twice the length of an optimal solution. Hence, we show that Algorithm TT has *worst-case ratio* equal to 2.

Let z_{opt} denote the length of an optimal tour, and let z_{TT} denote the length of the tour constructed by algorithm TT. Finally, let z_T denote the length of a minimum spanning tree. We first prove that $z_T \leq z_{\text{opt}}$; we then complete the proof by showing that $z_{TT} \leq 2 * z_T$.

1. Consider any optimal tour. If we delete an arbitrary edge from it, then we obtain a

Hamiltonian path with length at most z_{opt}. Since a Hamiltonian path is a special case of a spanning tree, we have that its length amounts to at least z_T. Concluding, we have that $z_T \leq z_{\text{opt}}$.

2. The total length of the edges in the Eulerian graph that is constructed by doubling the minimum spanning tree is equal to $2 * z_T$. From Lemma 5.2, it follows that the length z_{TT} of the Hamiltonian circuit that is obtained by applying short-cuts amounts to no more than the length of the Eulerian cycle, which is equal to $2 * z_T$.

Combining both results, we get $z_{TT} \leq 2 * z_T \leq 2 * z_{\text{opt}}$.

The tree-matching algorithm (TM).

From the analysis above, it follows that, if we want to improve on our worst-case ratio, then we can try to decrease the length of the Eulerian cycle. In the sequel we will do so. This means that we use the same structure of algorithm TT. In fact, the Phases 1, 3, and 4 in Algorithm TM are identical to the corresponding phases in Algorithm TT. Thus, we only change the phase in which the Eulerian Circuit is constructed. Recall that a graph is Eulerian if and only if it is connected, and each vertex has even degree. It seems obvious to start with a minimum spanning tree to make sure that the graph is connected. The only problem left is to take care of the vertices with odd degree denoted by V_0; note that the number of vertices with odd degree is even. We see that we can get even degree in each of these vertices by adding a perfect matching M on these vertices with minimum length. This is exactly what happens in phase 2 of algorithm TM.

Consider the example again. The initial minimum spanning tree contains 4 vertices of odd degree, namely 1, 2, 3, and 7. The minimum weight perfect matching of these vertices consists of the edges $\{1, 2\}$ and $\{3, 7\}$. Thus, we get the following extension of the minimum spanning tree.

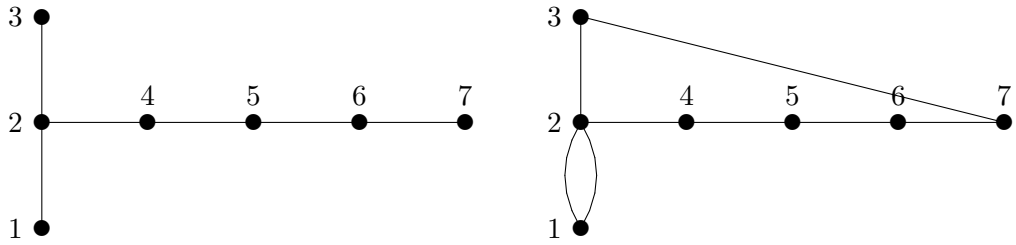

Figure 3.7: Phases 2 of algorithm TM.

An Euler-cycle in this graph is 123765421, where 2 is the only vertex that appears more than once, and therefore we remove one of its occurences.

This small change with respect to algorithm TT results in a better worst-case behaviour. For any instance of the TSP (with triangle-inequality), algorithm TM constructs a tour with length no more than $\frac{3}{2}$ times the length of an optimal tour. To prove this, it suffices to show that the length z_M of the matching M is no more than $\frac{1}{2}$ times z_{opt}. Namely, then $z_{TM} \leq z_T + z_M \leq z_{\text{opt}} + \frac{1}{2} z_{\text{opt}} = \frac{3}{2} * z_{\text{opt}}$.

The proof makes use of the shrinking algorithm again. Consider any optimal tour with value z_{opt}. Apply short-cuts such that only the vertices in V_0 remain in the cycle; this yields a

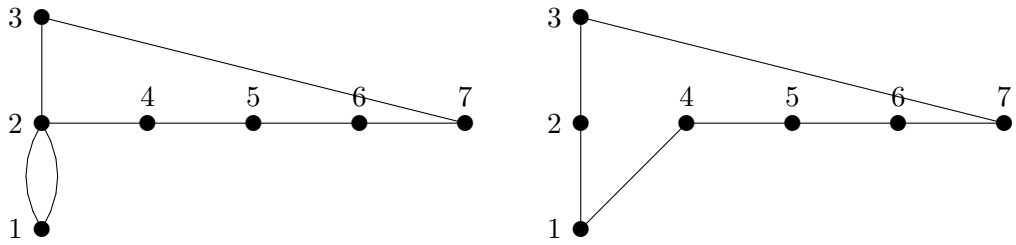

Figure 3.8: Replacing 421 by 41.

circuit C on the vertices in V_0 with length no more than z_{opt}. C can be partitioned into two complete matchings M_1 and M_2 on V_0 by 'walking' along the circuit and putting the first edge in M_1, the second edge in M_2, the third edge in M_1, etc. Note that the circuit contains an even number of edges, because the cardinality of V_0 is even.

Since M_1 and M_2 are complete matchings on V_0, we have that $d(M) \leq d(M_1)$ and $d(M) \leq d(M_2)$. Hence, we have that $2 * d(M) \leq d(M_1) + d(M_2) = d(C) \leq z_{\text{opt}}$, which was to be proven.

Note that for both algorithms our analysis of the worst-case ratio depends heavily on the assumption that the triangle inequality holds. We have shown that, in case the triangle inequality fails to hold, the worst-case ratio is no longer fixed, that is, for any constant c an instance of the TSP can be constructed such that the length of the tour constructed by the algorithm is more than c times as long as the length of the optimal tour.