10. Approximation Algorithms

10.1 Introduction

As we have seen, many combinatorial optimization problems are *NP*-hard and thus there is very little hope that we will be able to develop efficient algorithms for these problems. Nevertheless, many of these problems are fundamental and solving them is of great importance. There are various ways to cope with these hardness results:

1. *Exponential Algorithms*: Certainly, using an algorithm whose running time is exponential in the worst case might not be too bad after all if we only insist on solving instances of small to moderate size.
2. *Approximation Algorithms*: Approximation algorithms are efficient algorithms that compute suboptimal solutions with a provable approximation guarantee. That is, here we insist on polynomial-time computation but relax the condition that the algorithm has to find an optimal solution by requiring that it computes a feasible solution that is "close" to optimal.
3. *Heuristics*: Any approach that solves the problem without a formal guarantee on the quality of the solution can be considered as a heuristic for the problem. Some heuristics provide very good solutions in practice. An example of such an approach is *local search*: Start with an arbitrary solution and perform local improvement steps until no further improvement is possible. Moreover, heuristics are often practically appealing because they are simple and thus easy to implement.

We give some more remarks:

Some algorithms might perform very well in practice even though their worst-case running time is exponential. The simplex algorithm solving the *linear programming* problem is an example of such an algorithm. Most real-world instances do not correspond to worst-case instances and thus "typically" the algorithms' performance in practice is rather good. In a way, the worst-case running time viewpoint is overly pessimistic in this situation.

A very successful approach to attack optimization problems originating from practical applications is to formulate the problem as an *integer linear programming (ILP)* problem and to solve the program by *ILP*-solvers such as CPLEX. Such solvers are nowadays very efficient and are capable to solve large instances. Constructing the right *ILP*-method for solving a given problem is a matter of smart engineering. Some *ILP*-problems can be solved by just running an ILP-solver; others can only be solved with the help of more sophisticated methods such as branch-and-bound, cutting-plane, column generation, etc. Especially rostering problems, like classes of universities or schedules of personnel in hospitals, are notorious for being extremely hard to solve, already for small sizes. Solving ILP-problems is an art that can be learned only in practice.

Here we will focus on approximation algorithms in order to cope with *NP*-hardness of problems. We give a formal definition of these algorithms first.

Definition 10.1. An algorithm ALG for a minimization problem Π is an *α-approximation algorithm* with $\alpha \geq 1$ if it computes for every instance $I \in \mathcal{I}$ in polynomial time a feasible

solution $S \in \mathcal{F}$ whose cost $c(S)$ is at most α times the cost $\mathsf{OPT}(I)$ of an optimal solution for I, i.e., $c(S) \leq \alpha \cdot \mathsf{OPT}(I)$.

The definition is similar for maximization problems. Here it is more natural to assume that we want to maximize a weight (or profit) function $w : \mathcal{F} \to \mathbb{R}$ that maps every feasible solution $S \in \mathcal{F}$ of an instance $I \in \mathcal{I}$ to some real value.

Definition 10.2. An algorithm ALG for a maximization problem Π is an *α-approximation algorithm* with $\alpha \geq 1$ if it computes for every instance $I \in \mathcal{I}$ in polynomial time a feasible solution $S \in \mathcal{F}$ whose weight (or profit) $w(S)$ is at least $\frac{1}{\alpha}$ times the weight $\mathsf{OPT}(I)$ of an optimal (i.e., maximum weight) solution for I, i.e., $w(S) \geq \frac{1}{\alpha} \cdot \mathsf{OPT}(I)$.

Note that we would like to design approximation algorithms with the approximation ratio α being as small as possible. A lot of research in theoretical computer science and discrete mathematics is dedicated to the finding of "good" approximation algorithms for combinatorial optimization problems.

10.2 Approximation Algorithm for *Vertex Cover*

We start with an easy approximation algorithm for the *vertex cover* problem, which has been introduced before: Given a graph $G = (V,E)$, find a vertex cover $V' \subseteq V$ of smallest cardinality. Recall that we showed that the decision variant of *vertex cover* is *NP*-complete.

One of the major difficulties in the design of approximation algorithms is to come up with a good estimate for the optimal solution cost $\mathsf{OPT}(I)$. (We will omit I subsequently.) Recall that a matching M is a subset of the edges having the property that no two edges share a common endpoint. We call a matching M *maximum* if the cardinality of M is maximum; we call it *maximal* if it is *inclusion-wise* maximal, i.e., we cannot add another edge to M without rendering it infeasible. Note that a maximum matching is a maximal one but not vice versa.

Lemma 10.1. *Let $G = (V,E)$ be an undirected graph. If M is a matching of G then $OPT \geq |M|$.*

Proof. Consider an arbitrary vertex cover V' of G. Every matching edge $(u,v) \in M$ must be covered by at least one vertex in V', i.e., $\{u,v\} \cap V' \neq \emptyset$. Because the edges in M do not share any endpoints, we have $|V'| \geq |M|$. □

We conclude that we can derive an easy 2-approximation algorithm for *vertex cover* as follows:

Theorem 10.1. *Algorithm 14 is a 2-approximation algorithm for vertex cover.*

Proof. Clearly, the running time of Algorithm 14 is polynomial because we can find a maximal matching in time at most $O(n+m)$. The algorithm outputs a feasible vertex

Input: Undirected graph $G = (V,E)$.
Output: Vertex cover $V' \subseteq V$.

1 Find a maximal matching M of G.
2 Output the set V' of matched vertices.

Algorithm 14: Approximation algorithm for *vertex cover*.

cover because of the maximality of M. To see this, suppose that the resulting set V' is not a vertex cover. Then there is an edge (u,v) with $u,v \notin V'$ and thus both u and v are unmatched in M. We can then add the edge (u,v) to M and obtain a feasible matching, which contradicts the maximality of M. Finally, observe that $|V'| = 2|M| \leq 2\mathsf{OPT}$ by Lemma 10.1. □

Note that it suffices to compute a maximal (not necessarily maximum) matching in Algorithm 14, which can be done in linear time $O(m)$.

An immediate question that comes to ones mind is whether the approximation ratio is best possible. This indeed involves two kinds of questions in general:

1. Is the approximation ratio α of the algorithm tight?
2. Is the approximation ratio α of the algorithm best possible for *vertex cover*?

The first question essentially asks whether the analysis of the approximation ratio is tight. This is usually answered by exhibiting an example instance for which the algorithm computes a solution whose cost is α times the optimal one. The second one asks for much more: Can one show that there is no approximation algorithm with approximation ratio $\alpha - \varepsilon$ for every $\varepsilon > 0$? Such an *inapproximability result* usually relies on some conjecture such as that $P \neq NP$.

Lets first argue that the approximation ratio of Algorithm 14 is indeed tight.

Example 10.1. Consider a complete bipartite graph with n vertices on each side. The above algorithm will pick all $2n$ vertices, while picking one side of the bipartition constitutes an optimal solution of cardinality n. The approximation ratio of 2 is therefore tight.

The answer to the second question is not clear, despite intensive research. The currently best known lower bound on the inapproximability of *vertex cover* is as follows (stated without proof).

Theorem 10.2. *Vertex cover cannot be approximated within a factor of* 1.3606, *unless* $P = NP$.

10.3 Approximation Algorithms for *TSP*

As introduced before, the *traveling salesman problem* asks for the computation of a shortest tour in a given graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$.

We first show the following inapproximability result.

Theorem 10.3. *For any polynomial-time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $P = NP$.*

Proof. Suppose we have an algorithm ALG that approximates *TSP* within a factor $\alpha(n)$. We show that we can use ALG to decide in polynomial time whether a given graph has a Hamiltonian cycle or not, which is impossible unless $P = NP$.

Let $G = (V,E)$ be a given graph on n vertices. We extend G to a complete graph and assign each original edge a cost of 1 and every other edge a cost of $n\alpha(n)$. Run the $\alpha(n)$-approximation algorithm ALG on the resulting instance. We claim that G contains a Hamiltonian cycle if and only if the TSP tour computed by ALG has cost less than or equal to $n\alpha(n)$.

Suppose G has a Hamiltonian cycle. Then the optimal TSP tour in the extended graph has cost n. The approximate TSP tour computed by ALG must therefore have cost less than or equal to $n\alpha(n)$. Suppose G does not contain a Hamiltonian cycle. Then every feasible TSP tour in the extended graph must use at least one edge of cost $n\alpha(n)$, i.e., the cost of the tour is greater than $n\alpha(n)$ (assuming that G has at least $n \geq 2$ vertices). Thus, the cost of the approximate TSP tour computed by ALG is greater than $n\alpha(n)$. The claim follows. □

The above inapproximability result is extremely bad news. The situation changes if we consider the *metric TSP* problem.

Metric Traveling Salesman Problem (Metric TSP):

Given:	An undirected complete graph $G = (V,E)$ with non-negative costs $c : E \to \mathbb{R}^+$ satisfying the *triangle inequality*, i.e., for every $u,v,w \in V$, $c_{uw} \leq c_{uv} + c_{vw}$.
Goal:	Compute a tour in G that minimizes the total cost.

The *metric TSP* problem remains *NP*-complete: Recall that we showed that the *TSP* problem is *NP*-complete by reducing *Hamiltonian Cycle* to this problem. The reduction only used edge costs 1 and 2. Note that such edge costs always constitute a metric. Thus, the same proof shows that *metric TSP* is *NP*-complete.

We next derive two constant factor approximation algorithms for this problem.

Given a subset $Q \subseteq E$ of the edges, we define $c(Q)$ as the total cost of all edges in Q, i.e., $c(Q) = \sum_{e \in Q} c_e$.

The following lemma establishes a lower bound on the optimal cost:

Lemma 10.2. *Let T be a minimum spanning tree of G. Then $\mathsf{OPT} \geq c(T)$.*

Proof. Consider an optimal *TSP* tour and remove an arbitrary edge from this tour. We obtain a spanning tree of G whose cost is at most OPT. The cost of T is thus at most OPT. □

This lemma leads to the following idea:[5]

Input: Complete graph $G = (V, E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$ satisfying the triangle inequality.
Output: *TSP* tour of G.

1 Compute a minimum spanning tree T of G.
2 Double all edges of T to obtain a Eulerian graph G'.
3 Extract a Eulerian tour C' from G'.
4 Traverse C' and short-cut previously visited vertices.
5 Output the resulting tour C.

Algorithm 15: Approximation algorithm for *metric TSP*.

Theorem 10.4. *Algorithm 15 is a 2-approximation algorithm for metric TSP.*

Proof. Note that the algorithm has polynomial running time. Also, the returned tour is a TSP tour by construction. Because edge costs satisfy the triangle inequality, the tour C resulting from short-cutting the Eulerian tour C' in Algorithm 15 has cost at most $2c(T)$, where T is the minimum spanning tree computed in Step 1. By Lemma 10.2, the cost of C is thus at most $2\mathsf{OPT}$. □

We can actually derive a better approximation algorithm by refining the idea of Algorithm 15. Note that the reason for doubling the edges of a minimum spanning tree T was that we would like to obtain a Eulerian graph from which we can then extract a Eulerian tour. Are there better ways to construct a Eulerian graph starting with a minimum spanning tree T? Certainly, we only have to take care of the odd degree vertices, say V', of T. Note that in a tree there must be an even number of odd degree vertices.

So one way of making these odd degree vertices become even degree vertices is to add the edges of a perfect matching on V' to T. Intuitively, we would like to keep the total cost of the augmented tree small and thus compute a minimum cost perfect matching. As the following lemma shows, the cost of this matching can be related to the optimal cost.

Lemma 10.3. *Let $V' \subseteq V$ be a subset containing an even number of vertices. Let M be a minimum cost perfect matching on V'. Then $\mathsf{OPT} \geq 2c(M)$.*

Proof. Consider an optimal *TSP* tour C of length OPT. Traverse this tour and short-cut all vertices in $V \setminus V'$. Because of the triangle inequality, the resulting tour C' on V' has length at most OPT. C' can be seen as the union of two perfect matchings on V'. The cheaper matching of these two must have cost at most $\frac{1}{2}\mathsf{OPT}$. We conclude that a minimum cost perfect matching M on V' has cost at most $\frac{1}{2}\mathsf{OPT}$. □

We combine the above observations in the following algorithm, which is also known as *Christofides' algorithm*.

[5] Recall that a *Eulerian graph* is a connected graph that has no vertices of odd degree. A *Eulerian tour* is a cycle that visits every edge of the graph exactly once. Given a Eulerian graph, we can always find a Eulerian tour.

Input: Complete graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$ satisfying the triangle inequality.
Output: *TSP* tour of G.

1 Compute a minimum spanning tree T of G.
2 Compute a perfect matching M on the odd degree vertices V' of T.
3 Combine T and M to obtain a Eulerian graph G'.
4 Extract a Eulerian tour C' from G'.
5 Traverse C' and short-cut previously visited vertices.
6 Output the resulting tour C.

Algorithm 16: Approximation algorithm for *metric TSP*.

Theorem 10.5. *Algorithm 16 is a $\frac{3}{2}$-approximation algorithm for metric TSP.*

Proof. Note that the algorithm can be implemented to run in polynomial time (computing a perfect matching in an undirected graph can be done in polynomial time). The proof follows because the Eulerian graph G' has total cost $c(T)+c(M)$. Because of the triangle-inequality, short-cutting the Eulerian tour C' does not increase the cost. The resulting tour C has thus cost at most $c(T)+c(M)$, which by Lemmas 10.2 and 10.3 is at most $\frac{3}{2}\mathsf{OPT}$. □

The algorithm is tight (example omitted). Despite intensive research efforts, this is still the best known approximation algorithm for the *metric TSP* problem.

10.4 Approximation Algorithm for *Steiner Tree*

We next consider a fundamental network design problem, namely the *Steiner tree problem*. It naturally generalizes the *minimum spanning tree problem*:

Steiner Tree Problem:

Given:	An undirected graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$ and a set of terminal nodes $R \subseteq V$.
Goal:	Compute a minimum cost tree T in G that connects all terminals in R.

The nodes in R are usually called *terminals*; those in $V \setminus R$ are called *Steiner* nodes. The *Steiner tree* problem thus asks for the computation of a minimum cost tree, also called *Steiner tree*, that spans all terminals in R and possibly some Steiner nodes. The decision variant of the problem is *NP*-complete. Note that if we knew the set $S \subseteq V \setminus R$ of Steiner nodes that are included in an optimal solution, then we could simply compute an optimal Steiner tree by computing a minimum spanning tree on the vertex set $R \cup S$ in G. Thus, the difficulty of the problems is that we do not know which Steiner nodes to include.

We first show that we can restrict our attention without loss of generality to the so-called *metric Steiner tree problem*. In the metric version of the problem, we are given a *complete* graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$ that satisfy the *triangle*