Combinatorial Optimization

Jens Vygen

University of Bonn, Research Institute for Discrete Mathematics, Lennéstr. 2, 53113 Bonn, Germany

Combinatorial optimization problems arise in numerous applications. In general, we look for an optimal element of a finite set. However, this set is too large to be enumerated; it is implicitly given by its combinatorial structure. The goal is to develop efficient algorithms by understanding and exploiting this structure.

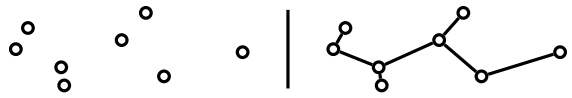
1 Some Important Problems

We first give some classical examples. We refer to the chapter on ⟶ *Graph Theory* for basic notation. In a digraph, we denote by $\delta^+(X)$ and $\delta^-(X)$ the set of edges leaving and entering X, respectively; here X can be a vertex or a set of vertices. In an undirected graph, $\delta(X)$ denotes the set of edges with exactly one endpoint in X.

1.1 Spanning trees

Here we are given a finite connected undirected graph (V, E) (so V is the set of vertices and E the set of edges) and weights on the edges, i.e., $c(e) \in \mathbb{R}$ for all $e \in E$. The task is to find a set $T \subseteq E$ such that (V, T) is a (spanning) tree and $\sum_{e \in T} c(e)$ is minimum. (Recall that a *tree* is a connected graph without cycles.)

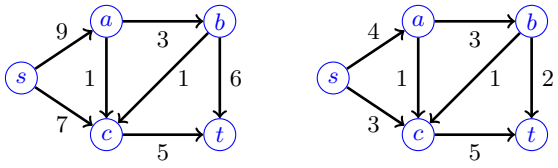
The figure below shows on the left a set V of eight points in the Euclidean plane. Assuming that (V, E) is the complete graph on these points and c is the Euclidean distances, the right-hand side shows an optimal solution.



1.2 Maximum flows

Given a finite directed graph (V, E), two vertices $s, t \in V$ (*source* and *sink*), and *capacities* $u(e) \in \mathbb{R}_{\geq 0}$ for all $e \in E$, we look for an *s-t-flow* $f : E \to \mathbb{R}_{\geq 0}$ with $f(e) \leq u(e)$ for all $e \in E$ and $f(\delta^-(v)) = f(\delta^+(v))$ for all $v \in V \setminus \{s, t\}$ (flow conservation: the total entering flow equals the total leaving flow at any vertex except s and t). The goal is to maximize $f(\delta^-(t)) - f(\delta^+(t))$, i.e., the total amount of flow shipped from s to t. This is called the *value* of f.

The figure below shows an example. The left-hand side displays an instance, the capacities are shown next to the edges. The right-hand side shows an s-t-flow of value 7. This is not optimal.



1.3 Matching

Given a finite undirected graph (V, E), find a matching $M \subseteq E$ that is as large as possible. (A *matching* is a set of edges whose endpoints are all distinct.)

1.4 Knapsack

Given $n \in \mathbb{N}$, positive integers a_i, b_i (profit and weight of item i, for $i = 1, \ldots, n$), and B (the knapsack's capacity), find a subset $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} b_i \leq B$, such that $\sum_{i \in I} a_i$ is as large as possible.

1.5 Traveling salesman

Given a finite set X with metric d, find a bijection $\pi : \{1, \ldots, n\} \to X$ such that the length of the corresponding tour,

$$\sum_{i=1}^{n-1} d(\pi(i), \pi(i+1)) + d(\pi(n), \pi(1)),$$

is as small as possible.

1.6 Set covering

Given a finite set U and subsets $S_1, \ldots, S_n$ of U, find the smallest collection of these subsets whose union is U, i.e., $I \subseteq \{1, \ldots, n\}$ with $\bigcup_{i \in I} S_i = U$ and $|I|$ minimum.

2 General Formulation and Goals

2.1 Instances and solutions

These problems have many common features.

In each case, there are infinitely many *instances*, each of which can be described (up to renaming) by a finite set of bits, and in some cases a finite set of real numbers.

For each instance, there is a set of *feasible solutions*. This set is finite in most cases. In the maximum flow problem it is actually infinite, but even here one can restrict w.l.o.g. to a finite set of solutions; see below.

Given an instance and a feasible solution, we can easily compute its *value*.

For example, in the matching problem, the instances are the finite undirected graphs; for each instance G the set of feasible solutions are the matchings in G; and for each matching, its value is simply its cardinality.

Even if the number of feasible solutions is finite, it cannot be bounded by a polynomial in the *instance size* (the number of bits that is needed to describe the instance). For example, there are n^{n-2} trees (V, T) with $V = \{1, \ldots, n\}$ (this is Cayley's formula). Similarly, the number of matchings on n vertices, subsets of an n-element set, and permutations on n elements, grow exponentially in n. One cannot enumerate all of them in reasonable time except for very small n.

Whenever an instance contains real numbers, we assume that we can do elementary operations with them, or we actually assume them to be rationals with binary encoding.

2.2 Algorithms

The main goal in combinatorial optimization is to devise efficient algorithms for solving such problems.

Efficient usually means *polynomial-time* (that is: the number of elementary steps can be bounded by a polynomial in the instance size). Of course, the faster, the better.

Solving a problem usually means always (for every given instance) computing a feasible solution with optimum value.

We give an example of an efficient algorithm solving the spanning tree problem in Section 3.

However, for NP-hard problems (like the last three examples in our list), this is impossible unless P=NP, and consequently one is satisfied with less (see Section 5).

2.3 Other Goals

Besides developing algorithms and proving their correctness and efficiency, combinatorial optimization (and related areas) also comprises other work:

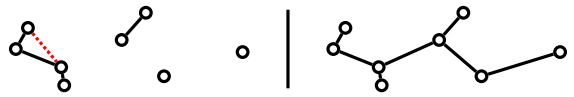
- analyze combinatorial structures, such as graphs, matroids, polyhedra, hypergraphs;
- establish relations between different combinatorial optimization problems: reductions, equivalence, bounds, relaxations;
- prove properties of optimal (or near-optimal) solutions;
- study the complexity of problems and establish hardness results;
- implement algorithms and analyze their practical performance;
- apply combinatorial optimization problems to real-world problems.

3 Greedy algorithm

The spanning tree problem has a very simple solution: the *greedy algorithm* does the job. We can start with the empty set and successively pick a cheapest edge that does not create a cycle, until our subgraph is connected. Formally:

1. Sort $E = \{e_1, \ldots, e_m\}$ so that $c(e_1) \leq \cdots \leq c(e_m)$.
2. Let T be the empty set.
3. For $i = 1, \ldots, m$ do:
   if $(V, T \cup \{e_i\})$ contains no cycle,
   then add e_i to T.

In our example, the first four steps would add the four shortest edges (shown in bold on the left-hand side below). Then the dotted edge is examined, but it is not added as it would create a cycle. The right-hand side shows the final output of the algorithm.



This algorithm can be easily implemented so that it runs in $O(nm)$ time, where $n = |V|$ and $m = |E|$. With a little more care, a running time of $O(m \log n)$ can be obtained. So this is a polynomial-time algorithm.

This algorithm computes a maximal set T such that (V, T) contains no cycle. In other words, (V, T) is a tree. It is not completely obvious that the output (V, T) is always an optimal solution, i.e., a tree with minimum weight. Let us give the nice and instructive proof of this fact:

3.1 Proof of correctness

Let (V, T^*) be an optimal tree, and choose T^* so that $|T^* \cap T|$ is as large as possible. Suppose $T^* \neq T$.

All spanning trees have exactly $|V| - 1$ edges, implying that $T^* \setminus T \neq \emptyset$. Let $j \in \{1, \ldots, m\}$ be the smallest index with $e_j \in T^* \setminus T$.

Since the greedy algorithm did not add e_j to T, there must be a cycle with edge set $C \subseteq \{e_j\} \cup (T \cap \{e_1, \ldots, e_{j-1}\})$ and $e_j \in C$.

$(V, T^* \setminus \{e_j\})$ is not connected, so there is a set $X \subset V$ with $\delta(X) \cap T^* = \{e_j\}$. (Recall that $\delta(X)$ denotes the set of edges with exactly one endpoint in X.)

Now $|C \cap \delta(X)|$ is even, so at least two. Let $e_i \in (C \cap \delta(X)) \setminus \{e_j\}$. Note that $i < j$ and thus $c(e_i) \leq c(e_j)$.

Let $T^{**} := (T^* \setminus \{e_j\}) \cup \{e_i\}$. Then (V, T^{**}) is a tree with $c(T^{**}) = c(T^*) - c(e_j) + c(e_i) \leq c(T^*)$. So T^{**} is also optimal. But T^{**} has one edge more in common with T (the edge e_i) than T^*, contradicting the choice of T^*.

3.2 Generalizations

In general (and for any of the other problems above), no simple "greedy" algorithm will always find an optimal solution.

The reason that it works for spanning trees is that here the feasible solutions form the bases of a *matroid*. Matroids are a well-understood combinatorial structure that can in fact be characterized by the optimality of the greedy algorithm.

Generalizations like optimization over the intersection of two matroids or minimization of submodular functions (given by an oracle) can also be solved in polynomial time, with more complicated combinatorial algorithms.

4 Duality and Min-Max Equations

Relations between different problems can lead to many important insights and algorithms. We give some well-known examples.

4.1 Max-Flow Min-Cut Theorem

We begin with the maximum flow problem and its relation to s-t-cuts. An s-t-cut is the set of edges leaving X (denoted by $\delta^+(X)$) for a set $X \subset V$ with $s \in X$ and $t \notin X$.

The total capacity of the edges in such an s-t-cut, denoted by $u(\delta^+(X))$, is an upper bound on the value of any s-t-flow f in (G, u). This is because this value is precisely $f(\delta^+(X)) - f(\delta^-(X))$ for every set X containing s but not t, and $0 \leq f(e) \leq u(e)$ for all $e \in E$.

The famous *max-flow min-cut theorem* says that the upper bound is tight: the maximum value of an s-t-flow equals the minimum capacity of an s-t-cut.

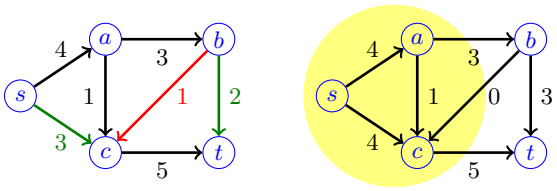
In other words: if f is any s-t-flow with maximum value, then there is a set $X \subset V$ with $s \in X$, $t \notin X$, $f(e) = u(e)$ for all $e \in \delta^+(X)$, and $f(e) = 0$ for all $e \in \delta^-(X)$.

Indeed, if no such set exists, we can find a directed path P from s to t in which each edge $e = (v, w)$ is either an edge of G with $f(e) < u(e)$, or the reverse $e' := (w, v)$ is an edge of G with $f(e') > 0$. (This follows from letting X be the set of vertices that are reachable from s along such paths.)

Such paths are called *augmenting paths* because along such a path we can augment the flow by increasing the flow on forward edges and decreasing it on backward edges. Some flow algorithms (but generally not the most efficient ones) start with the all-zero flow and successively find an augmenting path.

The figure below shows how to augment the flow shown in Section 1.2 by one unit along the path $a - c - b - t$. The resulting flow with value 8,

shown on the right, is optimal, as is proved by the s-t-cut $\delta^+(\{s,a,c\}) = \{(a,b),(c,t)\}$ of capacity 8.



The above relation also shows that for finding an s-t-cut with minimum capacity, it suffices to solve the maximum flow problem. This can also be used to compute a minimum cut in an undirected graph or to compute the connectivity of a given graph.

Any s-t-flow can be decomposed into flows on s-t-paths, and possibly on cycles (but cyclic flow is redundant as it does not contribute to the value). This decomposition can be done greedily, and then the list of paths is sufficient to recover the flow. This shows that one can restrict to a finite number of feasible solutions without loss of generality.

4.2 Disjoint paths

If all capacities are integral (i.e., are integers), one can find a maximum flow by always augmenting by 1 along an augmenting path, until none exists anymore. This is not a polynomial-time algorithm (because the number of iterations can grow exponentially in the instance size), but it shows that in this case there is always an optimal flow that is integral. An integral flow can be decomposed into integral flows on paths (and possibly cycles).

Hence, in the special case of unit capacities, an integral flow can be regarded as a set of pairwise edge-disjoint s-t-paths. Therefore, the max-flow min-cut theorem implies the following theorem, due to Karl Menger:

Let (V,E) be a directed graph and $s,t \in V$. Then the maximum number of paths from s to t that are pairwise edge-disjoint equals the minimum number of edges in an s-t-cut.

Other versions of Menger's theorem exist, for instance, for undirected graphs and for (internally) vertex-disjoint paths.

In general, finding disjoint paths with prescribed endpoints is difficult: For example, it is NP-complete to decide whether in a given directed graph with vertices s and t there is a path P from s to t and a path Q from t to s such that P and Q are edge-disjoint.

4.3 LP duality

The maximum flow problem (and also generalizations like minimum-cost flows and multicommodity flows) can be formulated as linear programs in a straightforward way.

Most other combinatorial optimization problems involve binary decisions and can be formulated naturally as (mixed-)integer linear programs. We give an example for the matching problem.

The matching problem can be written as *integer linear program*

$$\max\{\mathbb{1}^\top x : Ax \le \mathbb{1},\, x_e \in \{0,1\}\ \forall e \in E\}$$

where A is the vertex-edge-incidence matrix of the given graph $G = (V,E)$, $\mathbb{1} = (1,1,\dots,1)^\top$ denotes an appropriate all-one vector (so $\mathbb{1}^\top x$ is just an abbreviation of $\sum_{e\in E} x_e$), and $\le$ is meant component-wise. The feasible solutions to this integer linear program are exactly the incidence vectors of matchings in G.

Solving integer linear programs is NP-hard in general (cf. Section 5.2), but linear programs (without integrality constraints) can be solved in polynomial time ($\longrightarrow$ *Continuous Optimization*). This is one reason why it is often useful to consider the linear relaxation, which here is:

$$\max\{\mathbb{1}^\top x : Ax \le \mathbb{1},\, x \ge 0\},$$

where 0 and $\mathbb{1}$ denote appropriate all-zero and all-one vectors, respectively. Now the entries of x can be any real numbers between 0 and 1.

The dual LP is:

$$\min\{y^\top \mathbb{1} : y^\top A \ge \mathbb{1},\, y \ge 0\}.$$

By weak duality, every dual feasible vector y yields an upper bound on the optimum. (Indeed, if x is the incidence vector of a matching M and $y \ge 0$ with $y^\top A \ge \mathbb{1}$, then $|M| = \mathbb{1}^\top x \le y^\top Ax \le y^\top \mathbb{1}$.)

If G is bipartite, it turns out that these two LPs actually have integral optimal solutions. The

minimal integral feasible solutions of the dual LP are exactly the incidence vectors of *vertex covers* (sets $X \subseteq V$ such that every edge has at least one endpoint in X).

In other words, in any bipartite graph G, the maximum size of a matching equals the minimum size of a vertex cover. This is a theorem of Dénes Kőnig. It can also be deduced from the max-flow min-cut theorem.

For general graphs, this is not the case, as for example the triangle (complete graph on three vertices) shows. Nevertheless, the convex hull of incidence vectors of matchings in general graphs can also be described well: it is

$$\Big\{x : Ax \leq \mathbb{1},\, x \geq 0,\, \sum_{e \in E[A]} x_e \leq \big\lfloor \tfrac{|A|}{2} \big\rfloor \;\forall A \subseteq V\Big\},$$

where $E[A]$ denotes the set of edges whose endpoints both belong to A. This was shown by Jack Edmonds (1965), who also found a polynomial-time algorithm. In contrast, the problem of finding a minimum vertex cover in a given graph is NP-hard.

5 Dealing with NP-Hard Problems

The other three problems mentioned above (see Sections 1.4, 1.5, and 1.6) are NP-hard: they have a polynomial-time algorithm if and only if P=NP.

Since most researchers believe that P≠NP, they gave up looking for polynomial-time algorithms for NP-hard problems. Weaker goals strive for algorithms that, for instance,

- solve interesting special cases in polynomial time;
- run in exponential time but faster than trivial enumeration;
- always compute a feasible solution whose value is at most k times worse than the optimum (so-called *k-approximation algorithms*; see Section 5.1);
- are efficient or compute good solutions for most instances, in some probabilistic model;
- are *randomized* (use random bits in their computation) and have a good expected behaviour; or
- run fast and produce good results in practice although there is no proof (*heuristics*).

5.1 Approximation algorithms

From a theoretical point of view, the notion of approximation algorithms has proved to be most fruitful. For example, for the knapsack problem (cf. Section 1.4) there is an algorithm that for any given instance and any given number $\epsilon > 0$ computes a solution at most $1 + \epsilon$ times worse than the optimum, and whose running time is proportional to $\frac{n^2}{\epsilon}$. For the traveling salesman problem (cf. Section 1.5 and ⟶ *Traveling Salesman Problem*), there is a $\frac{3}{2}$-approximation algorithm.

For set covering (cf, Section 1.6) there is no constant factor approximation algorithm unless P=NP. But consider the special case where we ask for a minimum vertex cover in a given graph G: here U is the edge set of G and $S_i = \delta(v_i)$ for $i = 1, \ldots, n$, where $V = \{v_1, \ldots, v_n\}$ is the vertex set of G.

Here we can use the above-mentioned fact that the size of any matching in G is a lower bound. Indeed, take any (inclusion-wise) maximal matching M (e.g., found by the greedy algorithm), then the $2|M|$ endpoints of the edges in M form a vertex cover. As $|M|$ is a lower bound on the optimum, this is a simple 2-approximation algorithm.

5.2 Integer linear optimization

Most classical combinatorial optimization problems can be formulated as integer linear programs

$$\min\big\{c^\top x : Ax \leq b,\, x \in \mathbb{Z}^n\big\}.$$

This includes all problems discussed in this chapter, except the maximum flow problem, which is in fact a linear program (⟶ *Continuous Optimization*). Often the variables are restricted to 0 or 1. Sometimes, some variables are continuous, and others are discrete:

$$\min\big\{c^\top x : Ax + By \leq d,\, x \in \mathbb{R}^m,\, y \in \mathbb{Z}^n\big\}.$$

Such problems are called mixed-integer linear programs.

Discrete optimization comprises combinatorial optimization but also general (mixed-)integer optimization problems with no special combinatorial structure.

For general (mixed-)integer linear optimization, all known algorithms have exponential worst-case running time. The most successful algorithms in practice use a combination of cutting planes and branch-and-bound (see Sections 6.2 and 6.7). These are implemented in advanced commercial software. Since many practical problems (including almost all classical combinatorial optimization problems) can be described as (mixed-)integer linear programs, such software is routinely used in practice to solve small and medium-size instances of such problems. However, combinatorial algorithms that exploit the specific structure of the given problem are normally superior, and often the only choice for very large instances.

6 Techniques

Since good algorithms have to exploit the structure of the problem, every problem requires different techniques. Some techniques are quite general and can be applied for a large variety of problems, but in many cases they will not work well. Nevertheless we list the most important techniques that have been applied successfully to several combinatorial optimization problems.

6.1 Reductions

Reducing an unknown problem to a known (and solved) problem is the most important technique of course. To prove hardness, one proceeds the other way round: we reduce a problem that we know to be hard to a new problem (that then also must be hard). If reductions work in both ways, problems can actually regarded to be equivalent.

6.2 Enumeration techniques

Some problems can be solved by skillful enumeration. Dynamic programming is such a technique. It works if optimal solutions arise from optimal solutions to "smaller" problems by simple operations. Dijkstra's shortest path algorithm is a good example. Many algorithms on trees use dynamic programming.

Another well-known enumeration technique is *branch-and-bound*. Here one enumerates only parts of a decision tree because lower and upper bounds tell us that the unvisited parts cannot contain a better solution. How well this works mainly depends on how good the available bounds are.

6.3 Reducing or decomposing the instance

Often, an instance can be pre-processed by removing irrelevant parts. In other cases, one can compute a smaller instance or an instance with a certain structure, whose solution implies a solution of the original instance.

Another well-known technique is divide-and-conquer. In some problems, instances can be decomposed/partitioned into smaller instances, whose solutions can then be combined in some way.

6.4 Combinatorial or algebraic structures

If the instances have a certain structure (like planarity or certain connectivity or sparsity properties of graphs, cross-free set families, matroid structures, submodular functions, etc.), this must usually be exploited.

Also, optimal solutions (of relaxations or the original problem) often have a useful structure. Sometimes (e.g., by sparsification or uncrossing techniques) such a structure can be obtained even if it is not there originally.

Many algorithms compute and use a combinatorial structure as a main tool. This is often a graph structure, but sometimes an algebraic view can reveal certain properties. For instance, the Laplacian matrix of a graph has many useful properties. Sometimes simple properties, like parity, can be extremely useful and elegant.

6.5 Primal-dual relations

We discussed LP duality, a key tool for many algorithms, above. Lagrangian duality can also be useful for nonlinear problems. Sometimes other kinds of duality, like planar duality or dual matroids are very useful.

6.6 Improvement techniques

It is natural to start with some solution and iteratively improve it. The greedy algorithm and finding augmenting paths can be considered as special cases. In general, some way of measuring progress is needed so that the algorithm will terminate.

The general principle of starting with any feasible solution and iteratively improving it by small local changes is called *local search*. Local search heuristics are often quite successful in practice, but in many cases no reasonable performance guarantees can be given.

6.7 Relaxation and rounding

Relaxations can arise combinatorially (by allowing solutions that do not have a certain property that was originally required for feasible solutions), or by omitting integrality constraints of a description as an optimization problem over variables in $\mathbb{R}^n$.

Linear programming formulations can imply polynomial-time algorithms even if they have exponentially many variables or constraints (by the equivalence of optimization and separation). Linear relaxations can be strengthened by adding further linear constraints, called *cutting planes*.

One can also consider non-linear relaxations. In particular, *semidefinite relaxations* have been used for some approximation algorithms.

Of course, after solving a relaxation, the originally required property must be restored somehow. If a fractional solution is made integral, this is often called *rounding*. Sophisticated rounding algorithms for various purposes have been developed.

6.8 Scaling and rounding

Often, a problem becomes easier if the numbers in the instance are small integers. This can be achieved by scaling and rounding, of course at a loss of accuracy. The knapsack problem (cf. Section 1.4) is a good example: the best algorithms use scaling and rounding and then solve the rounded instance by dynamic programming.

In some cases, a solution of the rounded instance can be used in subsequent iterations to obtain more accurate, or even exact, solutions of the original instance faster.

6.9 Geometric techniques

Geometric techniques are also playing an increasing role. Describing (the convex hull of) feasible solutions by a polyhedron is a standard technique. Planar embeddings of graphs (if existent) can often be exploited in algorithms. Approximating a certain metric space by a simpler one is an important technique in the design of approximation algorithms.

6.10 Probabilistic techniques

Sometimes, a probabilistic view makes problems much easier. For example, a fractional solution can be viewed as a convex combination of extreme points, or as a probability distribution. Arguing over the expectation of some random variables can lead to simple algorithms and proofs. Many randomized algorithms can be derandomized, but this often complicates matters.

Further Reading

1. Korte, B., Vygen, J. 2012. Combinatorial Optimization: Theory and Algorithms. Berlin: Springer; 5th edition
2. Schrijver, A. 2003. Combinatorial Optimization: Polyhedra and Efficiency. Berlin: Springer