12

Generic Approaches to Optimization

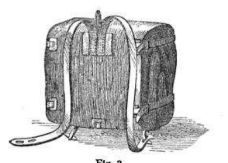


Fig. 3.

A smuggler in the mountainous region of Profitania has n items in his cellar. If he sells an item i across the border, he makes a profit p_i. However, the smuggler's trade union only allows him to carry knapsacks with a maximum weight of M. If item i has weight w_i, which of the items should he pack into the knapsack to maximize the profit from his next trip[1]*?*

This problem, usually called the *knapsack problem*, has numerous other applications, many of which are described in the books [179, 206]. For example, an investment bank might have an amount M of capital to invest and a set of possible investments. Each investment i has an expected profit p_i for an investment of cost w_i. In this chapter, we use the knapsack problem as an example to illustrate several generic approaches to optimization. These approaches are quite flexible and can be adapted to complicated situations that are ubiquitous in practical applications.

In the previous chapters we considered very efficient specific solutions for frequently occurring simple problems such as finding shortest paths or minimum spanning trees. Now we look at generic solution methods that work for a much larger range of applications. Of course, the generic methods do not usually achieve the same efficiency as specific solutions. However, they save development time.

Formally, an instance I of an optimization problem can be described by a set $\mathcal{U}_I$ of *potential* solutions, a set $\mathcal{L}_I$ of *feasible* solutions, and an *objective function* f_I with $f_I\colon \mathcal{L}_I \to \mathbb{R}$. For simplicity, we will mainly drop the subscript I and write $\mathcal{U}$, $\mathcal{L}$, and f. In a *maximization* problem, we are looking for a feasible solution $x^* \in \mathcal{L}$ that maximizes the value of the objective function over all feasible solutions. In a *minimization* problem, we are looking for a solution that minimizes the value of the objective function. In a *search* problem, the objective function is irrelevant; the task is to find any feasible solution $x \in \mathcal{L}$, if this set is not empty. Similarly, in an *existence problem*, the objective function is irrelevant and the question is whether $\mathcal{L}$ is nonempty.

For example, in the knapsack problem an instance I specifies the maximum weight M, the number n of objects as well as profits and weights of these objects, as vectors $p = (p_1, \ldots, p_n)$ and $w = (w_1, \ldots, w_n)$. Figure 12.1 gives an example in-

[1] The illustration above shows a 19th century American knapsack `commons.wikimedia.org/wiki/File:19th_century_knowledge_hiking_and_camping_sheepskin_knapsack_sleeping_bag_rolled_up.jpg`

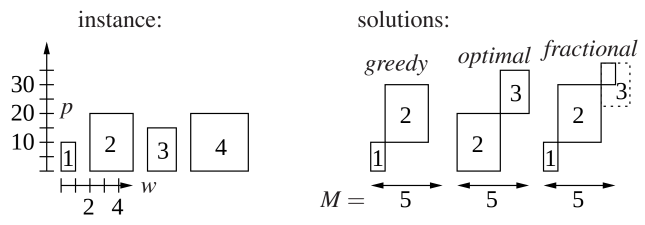


Fig. 12.1. The *left part* shows a knapsack instance with $p = (10,20,15,20)$, $w = (1,3,2,4)$, and $M = 5$. The items are indicated by rectangles whose width and height correspond to the weight and profit, respectively. The *right part* shows three solutions: the one computed by the *greedy* algorithm in Sect. 12.2, an *optimal* solution computed by the dynamic programming algorithm in Sect. 12.3, and the *fractional* solution of the linear relaxation (Sect. 12.1.2). The optimal solution has weight 5 and profit 35.

stance. A potential solution for I is simply a vector $x = (x_1,\ldots,x_n)$ with $x_i \in \{0,1\}$. Here $x_i = 1$ indicates that item i is put into the knapsack and $x_i = 0$ indicates that item i is left out. Thus $\mathscr{U} = \{0,1\}^n$. A potential solution x is feasible if its total weight does not exceed the capacity of the knapsack, i.e., if $\sum_{1\leq i\leq n} w_i x_i \leq M$. The dot product $w \cdot x$ is a convenient shorthand for $\sum_{1\leq i\leq n} w_i x_i$. We can then say that $\mathscr{L} = \{x \in \mathscr{U} : w \cdot x \leq M\}$ is the set of feasible solutions, and $f(x) = p \cdot x$ is the objective function.

The distinction between minimization and maximization problems is not essential because setting $f = -f$ converts a maximization problem into a minimization problem and vice versa. We shall use maximization as our default simply because our example problem is more naturally viewed as a maximization problem.[2]

In this chapter we shall present seven generic approaches to solving optimization problems. We start out with black-box solvers that can be applied to any optimization problem whose instances can be expressed in the problem specification language of the solver. In such a case, the only task of the user is to formulate the given problem in the language of the black-box solver. Section 12.1 introduces this approach using *linear programming* and *integer linear programming* as examples. The *greedy approach*, which we have already met in Chap. 11, is reviewed in Sect. 12.2. The approach of *dynamic programming* discussed in Sect. 12.3 is a more flexible way to construct solutions. We can also systematically explore the entire set of potential solutions, as described in Sect. 12.4. *Constraint programming*, *SAT solvers*, and *ILP solvers* are special cases of *systematic search*. Finally, we consider two very flexible approaches to exploring only a subset of the solution space. *Local search*, discussed in Sect. 12.5, modifies a single solution until it has the desired quality. *Evolutionary algorithms*, described in Sect. 12.6, simulate a population of candidate solutions. Most of the methods described above can be parallelized. We outline basic approaches within each section.

[2] Be aware that most of the literature uses minimization as the default.

12.1 Linear Programming – Use a Black-Box Solver

The easiest way to solve an instance of an optimization problem is to write down a specification of the space of feasible solutions and of the objective function and then use an existing software package to find an optimal solution. Such software packages are often called “black-box solvers” since the user only needs to know their interface and not the methods used for finding an optimal solution. Of course, the question is for which kinds of specification general solvers are available. In this section, we introduce a particularly large class of problem instances for which such black-box solvers are available. In *Linear Programming* one specifies the set of feasible solutions as a set of vectors in $\mathbb{R}^m$ by linear inequalities and the objective function as a linear function with values in $\mathbb{R}$. There are software packages that solve such instances, which can be termed “efficient” in a theoretical or in a practical sense. In *(Mixed) Integer Linear Programming* (some or) all the components of the feasible solutions are restricted to be integers. Black-box software packages are available for this type of specification as well, which can solve such instances quickly in many cases, although the algorithms are not guaranteed to run in polynomial time.

12.1.1 Linear Programming

Definition 12.1. *A* linear program[3] *(LP) with n* variables *and m* constraints *is an instance of a maximization problem that is specified in the following way. The possible solutions are vectors $x = (x_1, \ldots, x_n)$ with real components x_j, $j \in 1..n$, which are called* variables*. The objective function is a linear function f of x, i.e., $f\colon \mathbb{R}^n \to \mathbb{R}$ with $f(x) = c \cdot x$, where $c = (c_1, \ldots, c_n)$ is called the* cost *or* profit vector[4]. *The variables are constrained by m linear constraints of the form $a_i \cdot x \bowtie_i b_i$, where $\bowtie_i \in \{\leq, \geq, =\}$, $a_i = (a_{i1}, \ldots, a_{in}) \in \mathbb{R}^n$, and $b_i \in \mathbb{R}$ for $i \in 1..m$. The set of feasible solutions is given by*

$$\mathcal{L} = \left\{ x \in \mathbb{R}^n : \forall i \in 1..m : a_i \cdot x \bowtie_i b_i \text{ and } \forall j \in 1..n : x_j \geq 0 \right\}.$$

Figure 12.2 shows a simple example. A classical application of linear programming is the *diet problem*. A farmer wants to mix food for his cows. There are n different kinds of food on the market, say, corn, soya, fish meal, ... One kilogram of food j costs c_j euros. There are m requirements for healthy nutrition. For example, the cows should get enough calories, protein, vitamin C, and so on. One kilogram of food j contains a_{ij} percent of a cow’s daily requirement with respect to requirement i. A solution to the following linear program gives a cost-optimal diet that satisfies the health constraints: Let x_j denote the amount (in kilogram) of food j used by the farmer. The ith nutritional requirement is modeled by the inequality $\sum_j a_{ij} x_j \geq 100$. The cost of the diet is given by $\sum_j c_j x_j$. The goal is to minimize the cost of the diet.

[3] The term “linear programming” stems from the 1940s [81] and has nothing to do with the modern meaning of “program” as in “computer program”.

[4] If all c_j are positive, it is common to use the term “profit” in maximization problems and “cost” in minimization problems.

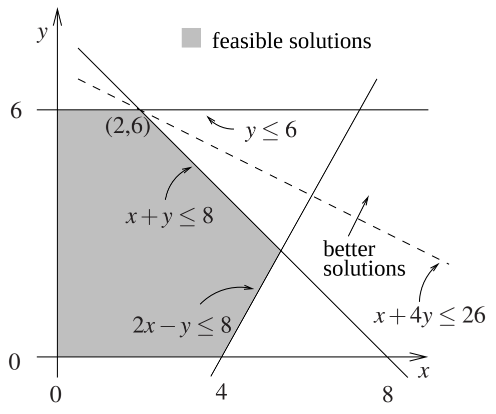


Fig. 12.2. A simple two-dimensional linear program in variables x and y, with three constraints and the objective function "maximize $x+4y$". The feasible region is shaded, and $(x,y)=(2,6)$ is the optimal solution. Its objective value is 26. The vertex $(2,6)$ is optimal because the half-plane described by $x+4y \le 26$ contains the entire feasible region and has $(2,6)$ in its boundary.

Exercise 12.1. How do you model supplies that are available only in limited amounts, for example food produced by the farmer himself? Also, explain how to specify additional constraints such as "no more than 0.01 mg cadmium contamination per cow per day".

Can the knapsack problem be formulated as a linear program? Probably not. Each item either goes into the knapsack as a whole or not at all. There is no possibility of adding only a part of an item. In contrast, it is assumed in the diet problem that any arbitrary amount of any food can be purchased, for example 3.7245 kg and not just 3 kg or 4 kg. Integer linear programs (see Sect. 12.1.2) are the suitable method for formulating the knapsack problem.

We next connect linear programming to a problem that was studied earlier in the book. We show how to formulate the single-source shortest-path problem with nonnegative edge weights in the language of linear programming. Let $G=(V,E)$ be a directed graph, let $s \in V$ be the source node, and let $c\colon E \to \mathbb{R}_{\ge 0}$ be the cost function on the edges of G. In the linear program that corresponds to this instance there is a variable d_v for each vertex v in G. The intention is that in an optimal solution d_v denotes the cost of the shortest path from s to v. Somewhat surprisingly, we formulate the *shortest* path problem as a *maximization* problem. Consider

$$\begin{array}{ll} \text{maximize} & \sum_{v \in V} d_v \\ \text{subject to} & d_s = 0, \\ & d_w \le d_v + c(e) \quad \text{for all } e=(v,w) \in E. \end{array}$$

In order to gain intuition why this might be a suitable formulation, the reader should recall the string model for the single-source shortest path problem that we discussed at the beginning of Chap. 10. In this model every node sits *as far below* the start node s *as possible*, without any edge being overstretched. That is, for vertices v and w that form an edge $e = (v,w)$ and their distances d_v and d_w from s we must have the relation $d_w \leq d_v + c(e)$. Maximizing the sum is a weak version of expressing the goal that each d_v should be as large as possible. We prove that solving the LP above is equivalent to solving the given shortest-path instance.

Theorem 12.2. *Let $G = (V,E)$ be a directed graph, $s \in V$ a designated vertex, and $c\colon E \to \mathbb{R}_{\geq 0}$ a nonnegative cost function. If all vertices of G are reachable from s, the shortest-path distances in G are the unique optimal solution to the linear program above.*

Proof. Let $\mu(v)$ be the distance from s to v. Then $\mu(v) \in \mathbb{R}_{\geq 0}$, since edge costs are nonnegative and all nodes are reachable from s, and hence no vertex can have a distance $-\infty$ or $+\infty$ from s. We observe first that the choice $d_v := \mu(v)$ for all v satisfies the constraints of the LP. Indeed, $\mu(s) = 0$ and $\mu(w) \leq \mu(v) + c(e)$ for any edge $e = (v,w)$.

We next show that if $(d_v)_{v \in V}$ satisfies all constraints of the LP above, then $d_v \leq \mu(v)$ for all v. Consider any v, and let $\langle s = v_0, v_1, \ldots, v_k = v\rangle$ be a shortest path from s to v. Then $\mu(v) = \sum_{0 \leq i < k} c(v_i, v_{i+1})$. We shall show by induction on j that $d_{v_j} \leq \sum_{0 \leq i < j} c(v_i, v_{i+1})$, for $j = 0, \ldots, k$. For $j = 0$, this follows from $d_s = 0$ by the first constraint. For $j > 0$, we have

$$d_{v_j} \leq d_{v_{j-1}} + c(v_{j-1}, v_j) \leq \sum_{0 \leq i < j-1} c(v_i, v_{i+1}) + c(v_{j-1}, v_j) = \sum_{0 \leq i < j} c(v_i, v_{i+1}),$$

where the first inequality follows from the second set of constraints of the LP and the second inequality comes from the induction hypothesis.

We have now shown that $(\mu(v))_{v \in V}$ is a feasible solution, and that $d_v \leq \mu(v)$ for all v for all feasible solutions $(d_v)_{v \in V}$. Since the objective of the LP is to maximize the sum of the d_v's, we must have $d_v = \mu(v)$ for all v in the optimal solution to the LP. □

Exercise 12.2. Where does the proof above fail when not all nodes are reachable from s or when there are negative edge costs? Does it still work in the absence of negative cycles?

The proof that the LP above actually captures the given instance of the shortest-path problem is nontrivial. When you formulate a problem instance as an LP, you should always prove that the LP is indeed a correct description of the instance that you are trying to solve.

Exercise 12.3. Let $G = (V,E)$ be a directed graph and let s ("source") and t ("sink") be two nodes. Let $cap\colon E \to \mathbb{R}_{\geq 0}$ and $c\colon E \to \mathbb{R}_{\geq 0}$ be nonnegative functions on the edges of G. For an edge e, we call $cap(e)$ and $c(e)$ the capacity and cost, respectively, of e. A *flow* is a function $f\colon E \to \mathbb{R}_{\geq 0}$ with $0 \leq f(e) \leq cap(e)$ for all e and flow conservation at all nodes except s and t, i.e., for all $v \neq s,t$, we have

$$\text{flow into } v = \sum_{e=(u,v)} f(e) = \sum_{e=(v,w)} f(e) = \text{flow out of } v.$$

The *value* of the flow is the net flow out of s, i.e., $\sum_{e=(s,v)} f(e) - \sum_{e=(u,s)} f(e)$. The *maximum-flow problem* asks for a flow of maximum value, given G, s, t, and *cap*. Show that such an instance of the flow problem can be formulated as an LP.

If also c is given, the *cost* of a flow is $\sum_e f(e)c(e)$. The *minimum-cost maximum-flow problem* asks for a maximum flow of minimum cost. Show how to formulate instances of this problem as an LP.

Linear programs are of central importance because they combine expressive power with efficient solution algorithms. For discussing efficiency, we have to explain how to measure the size of the input. The coefficients a_{ij}, b_i, and c_j of the linear inequalities and the objective function are assumed to be rational numbers. The size of an LP is then specified by m, n, and L, where L is an upper bound on the number of bits needed to write (in binary) the numerators and denominators of these coefficients. We say an algorithm *solves* Linear Programming if, when presented with an arbitrary LP, it either (1) returns an optimal feasible solution or (2) correctly asserts that there is no feasible solution or (3) correctly asserts that there are feasible solutions with arbitrarily large values of the objective function.

Theorem 12.3. *Linear programs can be solved in polynomial time [176, 180].*

The two polynomial-time algorithms are the Ellipsoid method [180] and the interior point method [176]. A compact account of the interior point method can be found in [221]. The worst-case running time of the best interior point algorithm is $\mathrm{O}(\max(m,n)^{7/2}L)$. Fortunately, the worst case rarely arises. Most linear programs can be solved relatively quickly by any one of several procedures. One, the simplex algorithm, is briefly outlined in Sect. 12.5.1. For now, the reader should remember two facts: First, many optimization problems can be formulated as linear programs, and second, there are efficient LP solvers that can be used as black boxes. In fact, although LP solvers are used on a routine basis, very few people in the world know exactly how to implement a highly efficient LP solver.

The simplex algorithm is notoriously difficult to parallelize, since an efficient implementation performs very little work in each step. The polynomial-time algorithms perform fewer steps with more work in each step and have been parallelized successfully in state-of-the-art LP solvers.

12.1.2 Integer Linear Programming

The expressive power of linear programming grows when some or all of the variables can be designated to be integral. Such variables can then take on only integer values, and not arbitrary real values. If all variables are constrained to be integral, the formulation of the problem is called an *integer linear program* (ILP). If some but not all variables are constrained to be integral, the formulation is called a *mixed integer linear program* (MILP). For example, our knapsack problem is tantamount to the following 0-1 integer linear program:

$$\text{maximize } p \cdot x$$

subject to

$$w \cdot x \leq M \quad \text{and} \quad x_j \in \{0,1\} \text{ for } j \in 1..n.$$

In a 0-1 integer linear program, the variables are constrained to the values 0 and 1.

Exercise 12.4. Explain how to replace any ILP by a 0-1 ILP, assuming that an upper bound U on the value of any variable in the optimal solution is known. Hint: Replace each variable of the original ILP by a set of $\mathrm{O}(\log U)$ many 0-1 variables.

Unfortunately, solving ILPs and MILPs is **NP**-hard. Indeed, even the knapsack problem is **NP**-hard. Nevertheless, ILPs can often be solved in practice using linear-programming packages. In Sect. 12.4, we shall outline how this is done. When an exact solution would be too time-consuming, linear programming can help to find approximate solutions. The *linear-program relaxation* of an ILP is the LP obtained by omitting the integrality constraints on the variables. For example, in the knapsack problem we would replace the constraint $x_j \in \{0,1\}$ by the constraint $x_j \in [0,1]$.

An LP relaxation can be solved by an LP solver. In many cases, the solution to the relaxation teaches us something about the underlying ILP. One observation always holds true (for maximization problems): The objective value of the relaxation is at least as large as the objective value of the underlying ILP. This claim is trivial, because any feasible solution to the ILP is also a feasible solution to the relaxation. The optimal solution to the LP relaxation will in general be *fractional*, i.e., variables will take on rational values that are not integral. However, it might be the case that only a few variables have nonintegral values. By appropriate rounding of fractional variables to integer values, we can often obtain good integer feasible solutions.

We shall give an example. The linear relaxation of the knapsack problem is given by

$$\text{maximize } p \cdot x$$

subject to

$$w \cdot x \leq M \quad \text{and} \quad x_j \in [0,1] \text{ for } j \in 1..n.$$

This has a natural interpretation. It is no longer required to add items to the knapsack as a whole; one can now take any fraction of an item. In our smuggling scenario, the *fractional knapsack problem* corresponds to a situation involving divisible goods such as liquids or powders.

The fractional knapsack problem is easy to solve in time $\mathrm{O}(n \log n)$; there is no need to use a general-purpose LP solver. We renumber (sort) the items by *profit density* p_j/w_j such that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}.$$

We find the smallest index ℓ such that $\sum_{j=1}^{\ell} w_j > M$ (if there is no such index, we can take all knapsack items). The item ℓ with fractional value x_ℓ is also called *critical item*. Now we set

$$x_1 = \cdots = x_{\ell-1} = 1, \quad x_\ell = \left(M - \sum_{j=1}^{\ell-1} w_j \right) / w_\ell, \quad \text{and } x_{\ell+1} = \cdots = x_n = 0.$$

Figure 12.1 gives an example. The fractional solution above is the starting point for many good algorithms for the knapsack problem. We shall see more of this later.

Exercise 12.5 (linear relaxation of the knapsack problem).

(a) Prove that the above routine computes an optimal solution. Hint: You may want to use an *exchange argument* similar to the one used to prove the cut property of minimum spanning trees in Sect. 11.1.
(b) Outline an algorithm that computes an optimal solution in linear expected time. Hint: Use a variant of *quickSelect*, described in Sect. 5.8.
(c) Parallelize your algorithm from (b).

A solution to the fractional knapsack problem is easily converted to a feasible solution to the knapsack problem. We simply take the fractional solution and round the sole fractional variable x_ℓ to 0. We call this algorithm *roundDown*.

Exercise 12.6. Formulate the following *set-covering* problem as an ILP. Given a set M, subsets $M_i \subseteq M$ for $i \in 1..n$ with $\bigcup_{i=1}^{n} M_i = M$, and a cost $c_i \in \mathbb{N}$ for each M_i, select $F \subseteq 1..n$ such that $\bigcup_{i \in F} M_i = M$ and $\sum_{i \in F} c_i$ is minimized.

Boolean formulae. These provide another powerful description language for search and decision problems. Here, variables range over the Boolean values `1` and `0`, and the connectors $\wedge$, $\vee$, and $\neg$ are used to build formulae. A Boolean formula is *satisfiable* if there is an assignment of Boolean values to the variables such that the formula evaluates to `1`. As an example, we consider the *pigeonhole principle*: It is impossible to pack $n+1$ items into n bins such that every bin contains at most one item. This principle can be formulated as the statement that certain formulae are unsatisfiable. Fix n. We have variables x_{ij} for $1 \le i \le n+1$ and $1 \le j \le n$. So i ranges over items and j ranges over bins. Variable x_{ij} represents the statement "item i is in bin j". The constraint that every item must be put into (at least) one bin is formulated as $x_{i1} \vee \cdots \vee x_{in}$, for $1 \le i \le n+1$. The constraint that no bin should hold more than one item is expressed by the subformulas $\neg(\bigvee_{1 \le i < h \le n+1} x_{ij} \wedge x_{hj})$, for $1 \le j \le n$. The conjunction of these $n+m+1$ formulae is unsatisfiable, since from a satisfying assignment for the variables one could calculate a way of distributing the $n+1$ items into the n bins, which does not exist. SAT solvers decide if a given Boolean formula is *sat*isfiable or not, and in the positive case calculate a satisfying assignment. Although the satisfiability problem is **NP**-complete, there are now solvers that can solve real-world instances that involve hundreds of thousands of variables.[5]

Exercise 12.7. Formulate the pigeonhole principle for $n+1$ items and n bins as an integer linear program.

[5] See `www.satcompetition.org/`.

SAT instances can be solved in parallel using the *portfolio approach* – run p SAT solvers in parallel and (almost) independently. Two measures are needed to make this successful. *Diversification* tries to make these SAT solvers to behave differently. For example, we can randomize decisions and vary tuning parameters. Furthermore, information “learned” during the search has to be *exchanged* in a judicious way. The resulting speedups fluctuate wildly – between occasional slowdown and huge superlinear speedup. This naturally leads to some debate about how effective this is. However, several natural ways to average speedups seem to indicate that the portfolio approach scales surprisingly well up to at least hundreds of PEs [28].

The portfolio approach can also be used for other systematic-search problems (see also Sect. 12.4); for example for solving ILPs or for other logical inference problems.

12.2 Greedy Algorithms – Never Look Back

The term *greedy algorithm* is used for a problem-solving strategy where the items under consideration are inspected in some order, usually some carefully chosen order. When an item is considered, a decision about this item is made; for example, whether it is included into the solution. Decisions are never reversed. The algorithm for the fractional knapsack problem given in the preceding section follows the greedy strategy; we consider the items in decreasing order of profit density. The algorithms for shortest paths in acyclic graphs and for the case of nonnegative edge weights in Sects. 10.2 and 10.3 and those for minimum spanning trees in Chap. 11 also follow the greedy strategy. For the single-source shortest-path problem with nonnegative edge weights, we considered the edges in the order of the tentative distances of their source nodes. For the latter two problems, the greedy approach led to an optimal solution.

Usually, greedy algorithms yield only suboptimal solutions. Let us consider the knapsack problem again. The greedy approach from above scans the items in order of decreasing profit density and includes items that will still fit into the knapsack. We shall give this algorithm the name *greedy*. Figures 12.1 and 12.3 give examples. Observe that *greedy* always gives solutions at least as good as *roundDown*

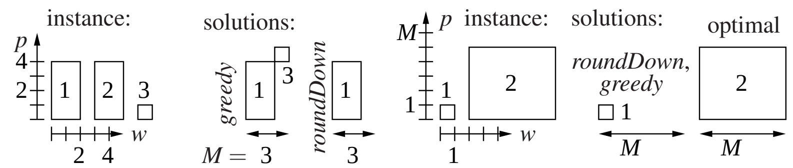


Fig. 12.3. Two instances of the knapsack problem. *Left*: For $p = (4,4,1)$, $w = (2,2,1)$, and $M = 3$, *greedy* performs better than *roundDown*. *Right*: For $p = (1, M-1)$ and $w = (1, M)$, both *greedy* and *roundDown* are far from optimal.

gives. Once *roundDown* encounters an item that it cannot include, it stops. However, *greedy* keeps on looking and often succeeds in including additional items of less weight. Although the example in Fig. 12.1 gives the same result for both *greedy* and *roundDown*, the results generally *are* different. For example, consider the two instances in Fig. 12.3. With profits $p = (4,4,1)$, weights $w = (2,2,1)$, and $M = 3$, *greedy* includes the first and third items, yielding a profit of 5, whereas *roundDown* includes just the first item and obtains only a profit of 4. Both algorithms may produce solutions that are far from optimal. For example, for any capacity M, consider the two-item instance with profits $p = (1, M-1)$ and weights $w = (1, M)$. Both *greedy* and *roundDown* include only the first item, which has a profit density slightly higher than that of the second item, but a very small absolute profit. In this case it would be much better to include just the second item.

We can turn this observation into an algorithm, which we call *round*. This computes two solutions: the solution x^d proposed by *roundDown* and the solution x^c obtained by choosing exactly the critical item x_ℓ.[6] It then returns the better of the two.

We can give an interesting performance guarantee for algorithm *round*. It always achieves at least 50% of the profit of the optimal solution. More generally, we say that an algorithm achieves an *approximation ratio* of α if, for all inputs, its solution is at most a factor α worse than the optimal solution.

Theorem 12.4. *The algorithm round achieves an approximation ratio of 2.*

Proof. Consider an instance p, w, x. Let x^* denote any optimal solution, and let x^f be the optimal solution for the same input when fractional solutions are admitted. Then $p \cdot x^* \leq p \cdot x^f$. The value of the objective function is increased by including the critical item, i.e., by setting $x_\ell = 1$ in the fractional solution. We obtain

$$p \cdot x^* \leq p \cdot x^f \leq p \cdot x^d + p \cdot x^c \leq 2 \max\left\{p \cdot x^d, p \cdot x^c\right\},$$

and hence the profit achieved by algorithm *round* is at least half the optimum. □

There are many ways to refine algorithm *round* without sacrificing this approximation guarantee. We can replace x^d by the greedy solution. We can similarly augment x^c with any greedy solution for a smaller instance where item j is removed and the capacity is reduced by w_j.

We now turn to another important class of optimization problems, called *scheduling problems*. Consider the following scenario, known as the *scheduling problem for independent weighted jobs on identical machines*. We are given m identical machines, on which we want to process n jobs; the execution of job j takes t_j time units. An assignment $x : 1..n \to 1..m$ of jobs to machines is called a *schedule*. Thus the *load* ℓ_j assigned to machine j is $\sum_{i:\, x(i)=j} t_i$. The goal is to minimize the *makespan* $L_{\max} = \max_{1 \leq j \leq m} \ell_j$ of the schedule.

[6] We assume here that "unreasonably large" items with $w_i > M$ have been removed from the problem in a preprocessing step.

This is a fundamental load-balancing problem for parallel processing and we shall consider it further in Chap. 14. One application scenario is as follows. We have a video game processor with several identical processor cores. The jobs are the tasks executed in a video game such as audio processing, preparing graphics objects for the image-processing unit, simulating physical effects, and simulating the intelligence of the game. The makespan will then determine the required time between two time steps of the game and hence the frame rate at which changes in the game can be displayed. Users of a game expect a frame rate which guarantees pleasant viewing.

We give next a simple greedy algorithm for the problem above [136] that has the additional property that it does not need to know the sizes of the jobs in advance. We can even assume that the jobs are presented one after the other, and we assign them on the basis of the knowledge we have so far. Algorithms with this property ("unknown future") are called *online* algorithms. When job i arrives, we assign it to the machine with the smallest load. Formally, we compute the loads $\ell_j = \sum_{h<i \wedge x(h)=j} t_h$ of all machines j and assign the new job to the least loaded machine, i.e., $x(i) := j_i$, where j_i is such that $\ell_{j_i} = \min_{1 \le j \le m} \ell_j$. This *shortest-queue algorithm* does not guarantee optimal solutions, but always computes nearly optimal solutions.

Theorem 12.5. *The list-scheduling algorithm ensures*

$$L_{\max} \le \frac{1}{m}\sum_{i=1}^{n} t_i + \frac{m-1}{m} \max_{1 \le i \le n} t_i.$$

Proof. In the schedule generated by the shortest-queue algorithm, some machine j^* has a load $L_{\max}$. We focus on the last job i^* that is assigned to machine j^*. When job i^* is assigned to j^*, all m machines have a load of at least $L_{\max} - t_{i^*}$, i.e.,

$$\sum_{i \ne i^*} t_i \ge (L_{\max} - t_{i^*}) \cdot m.$$

Solving this for $L_{\max}$ yields

$$L_{\max} \le \frac{1}{m}\sum_{i \ne i^*} t_i + t_{i^*} = \frac{1}{m}\sum_{i} t_i + \frac{m-1}{m} t_{i^*} \le \frac{1}{m}\sum_{i=1}^{n} t_i + \frac{m-1}{m} \max_{1 \le i \le n} t_i. \qquad \square$$

We next observe that $\frac{1}{m}\sum_i t_i/m$ and $\max_i t_i$ are lower bounds on the makespan of any schedule and hence also of the optimal schedule. We obtain the following corollary.

Corollary 12.6. *The approximation ratio of the shortest-queue algorithm is* $2 - 1/m$.

Proof. Let $L_1 = \frac{1}{m}\sum_i t_i$ and $L_2 = \max_i t_i$. The makespan L^* of the optimal solution is at least $\max(L_1, L_2)$. The makespan of the shortest-queue solution is bounded by

$$L_1 + \frac{m-1}{m} L_2 \le L^* + \frac{m-1}{m} L^* = \left(2 - \frac{1}{m}\right) L^*. \qquad \square$$

The shortest-queue algorithm is not better than claimed above. Consider an instance with $n = m(m-1)+1$, $t_i = 1$ for $i = 1,\dots,n-1$, and $t_n = m$. The optimal solution has a makespan $L^* = m$, whereas the shortest-queue algorithm produces a solution with a makespan $L_{\max} = 2m-1$. The shortest-queue algorithm is an online algorithm. It produces a solution which is at most a factor $2-1/m$ worse than the solution produced by an algorithm that knows the entire input. In such a situation, we say that the online algorithm has a *competitive ratio* of $\alpha = 2-1/m$.

***Exercise 12.8.** Show that the shortest-queue algorithm achieves an approximation ratio of $4/3$ if the jobs are sorted by decreasing size.

***Exercise 12.9 (bin packing).** Suppose a smuggler has perishable goods in her cellar. She has to hire enough porters to ship all items tonight. Develop a greedy algorithm that tries to minimize the number of people she needs to hire, assuming that each one can carry a weight M. Try to obtain an approximation ratio for your *bin packing* algorithm.

Parallel Greedy Algorithms

In general, greedy algorithms are difficult to parallelize. However, we may be able to exploit special properties of the application. When the priorities of the items can be computed up front, we can order them using parallel sorting. Perhaps we can even compute the decisions in parallel, for example using a prefix sum. This was the case for the greedy knapsack algorithm, where the decision just depends on the total weight of the preceding objects.

Sometimes at least a certain subset of items can be processed independently. We saw an example in the parallel DAG traversal algorithm in Sect. 9.4 and for parallel shortest paths in Sect. 10.9. If all else fails, we can also relax the ordering of the objects, approximating a greedy algorithm. We have seen an example in Sect. 10.9 for the Δ-stepping algorithm for shortest paths.

12.3 Dynamic Programming – Build It Piece by Piece

The first idea in dynamic programming is to expand a given problem instance into a system of auxiliary instances, which are called *subproblems*. These subproblems are then solved systematically. When solving a subproblem, one assumes that the solutions for all its subproblems have been computed before. The second idea is that for many optimization problems the following *principle of optimality* holds: An *optimal* solution for a subproblem is composed of *optimal* solutions for some of its subproblems. If a subproblem has several optimal solutions, it does not matter which one is used. A recurring phenomenon is that there is a choice among the possible sets of subproblems that are to be combined to obtain an optimal solution. An algorithm that follows the principle of dynamic programming uses the optimality principle to systematically build up a table of optimal solutions for all subproblems. Finally, an optimal solution for the original problem instance is constructed from the solutions for the subproblems.

Again, we shall use the knapsack problem as an example. Consider an instance I, consisting of weight vector w, profit vector c, and capacity M. For each $i \in 0..n$ and each C between 0 and M we define a *subproblem*, as follows: $P(i,C)$ is the maximum profit possible when only items 1 to i can be put in the knapsack and the total weight is at most C. Our goal is to compute $P(n,M)$. (We shall see below that solutions for all these subproblems can be used for calculating an optimal selection.) We start with trivial cases and work our way up. The trivial cases are "no items" and "total weight 0". In both of these cases, the maximum profit is 0. So

$$P(0,C) = 0 \text{ for all } C \quad \text{and} \quad P(i,0) = 0 \text{ for all } i.$$

Consider next the case $i > 0$ and $C > 0$. In the solution that maximizes the profit, we either use item i or do not use it. In the latter case, the maximum achievable profit is $P(i-1,C)$. In the former case, the maximum achievable profit is $P(i-1,C-w_i)+p_i$, since we obtain a profit of p_i for item i and must use an optimal solution for the first $i-1$ items under the constraint that the total weight is at most $C-w_i$. Of course, the former alternative is only feasible if $C \geq w_i$. We summarize this discussion in the following recurrence for $P(i,C)$:

$$P(i,C) = \begin{cases} 0, & \text{if } i = 0 \text{ or } C = 0, \\ \max(P(i-1,C), P(i-1,C-w_i)+p_i) & \text{if } i \geq 1 \text{ and } w_i \leq C, \\ P(i-1,C) & \text{if } i \geq 1 \text{ and } w_i > C. \end{cases} \tag{12.1}$$

Exercise 12.10. Show that the case distinction for w_i in the definition of $P(i,C)$ can be avoided by defining $P(i,C) = -\infty$ for $C < 0$.

Using the above recurrence, we can compute $P(n,M)$ by filling a table P with one column for each possible capacity C and one row for each item set $1..i$. Table 12.1 gives an example. There are many possible orders in which to fill out the table, for example row by row. In order to construct an optimal solution for the original instance from the table, we work our way backwards, starting with entry $P(n,M)$ at the bottom right-hand corner of the table. We set $i = n$ and $C = M$. If $P(i,C) = P(i-1,C)$, we set $x_i = 0$ and continue to row $i-1$ and column C. Otherwise, we set $x_i = 1$. We have $P(i,C) = P(i-1,C-w_i)+p_i$, and therefore we continue to row $i-1$ and column $C-w_i$. We continue in this fashion until we arrive at row 0. At this point $(x_1,\ldots,x_n)$ is an optimal solution for the original knapsack instance.

Exercise 12.11. The dynamic programming algorithm for the knapsack problem, as just described, needs to store a table containing $\Theta(nM)$ integers. Give a more space-efficient solution that at any given time stores only a single bit in each table entry as well as two full rows of table P. What information is stored in the bit? How is the information in the bits used to construct a solution? Can you get down to storing the bits and only *one* full row of the table? Hint: Exploit the freedom you have in the order of filling in table values.

Table 12.1. A dynamic-programming table for the knapsack instance with $p = (10, 20, 15, 20)$, $w = (1, 3, 2, 4)$, and $M = 5$. Entries that are inspected when the optimal solution is constructed are in **boldface**.

	$C =$					
	0	1	2	3	4	5
$i = 0$	0	0	0	0	0	0
1	**0**	10	10	10	10	10
2	0	10	10	**20**	30	30
3	0	10	15	25	30	**35**
4	0	10	15	25	30	**35**

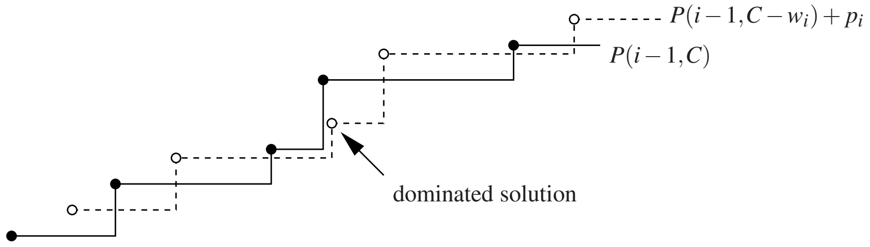


Fig. 12.4. The step function $C \mapsto P(i-1, C)$ is drawn with a solid line, and the step function $C \mapsto P(i-1, C-w_i) + p_i$ with a dashed line. The function $P(i, C)$ is the pointwise maximum of these two functions. The solid step function is stored as the sequence of solid points. The representation of the dashed step function is obtained by adding (w_i, p_i) to every solid point. The representation of $C \mapsto P(i, C)$ is obtained by merging the two representations and deleting all dominated elements.

We shall next describe an important improvement with respect to space consumption and speed. Instead of computing $P(i, C)$ for all i and all C, the *Nemhauser–Ullmann algorithm* [36, 238] computes only *Pareto-optimal* solutions. A solution x is Pareto-optimal if there is no solution that *dominates* it, i.e., has a greater profit and no greater cost or the same profit and less cost. In other words, since $P(i, C)$ is an increasing function of C, only the pairs $(C, P(i, C))$ with $P(i, C) > P(i, C-1)$ are needed for an optimal solution. We store these pairs in a list L_i sorted by the value of C. So $L_0 = \langle(0, 0)\rangle$, indicating that $P(0, C) = 0$ for all $C \geq 0$, and $L_1 = \langle(0, 0), (w_1, p_1)\rangle$, indicating that $P(1, C) = 0$ for $0 \leq C < w_1$ and $P(1, C) = p_1$ for $C \geq w_1$.

How can we go from L_{i-1} to L_i? The recurrence for $P(i, C)$ paves the way; see Fig. 12.4. We have the list representation L_{i-1} for the function $C \mapsto P(i-1, C)$. We obtain the representation L'_{i-1} for $C \mapsto P(i-1, C-w_i) + p_i$ by shifting every point in L_{i-1} by (w_i, p_i). We merge L_{i-1} and L'_{i-1} into a single list by order of first component and delete all elements that are dominated by another value, i.e., we delete all elements that are preceded by an element with a higher second component, and, for each fixed value of C, we keep only the element with the largest second component.

Exercise 12.12. Give pseudocode for the above merge. Show that the merge can be carried out in time $\mathrm{O}(|L_{i-1}|)$. Conclude that the running time of the Nemhauser–Ullmann algorithm is proportional to the number of all Pareto-optimal solutions produced in the course of the algorithm for all i taken together.

Both the basic dynamic-programming algorithm for the knapsack problem and its improved (Nemhauser–Ullmann) version require $\Theta(nM)$ worst-case time. This is quite good if M is not too large. Since the running time is polynomial in n and M, the algorithm is called *pseudo-polynomial*. The "pseudo" means that it is not necessarily polynomial in the *input size* measured in bits; however, it is polynomial in the natural parameters n and M. There is, however, an important difference between the basic and the refined approach. The basic approach has best-case running time $\Theta(nM)$. The best case for the refined approach is $\mathrm{O}(n)$. The *average-case* complexity of the refined algorithm is polynomial in n, independent of M. This holds even if the averaging is done only over perturbations of an arbitrary instance by a small amount of random noise. We refer the reader to [36] for details.

Exercise 12.13 (dynamic programming by profit). Assume an instance of the knapsack problem is given. Explore the following alternative way of defining subproblems: Let $W(i,P)$ be the smallest weight bound that makes it possible to achieve a profit of at least P, using knapsack items $1..i$. Obviously we have $W(i,P) = 0$ for $1 \leq i \leq n$ and all $P \leq 0$. Let $W(0,P) = \infty$ for all $P > 0$.

(a) Show that $W(i,P) = \min\{W(i-1,P), W(i-1,P-p_i)+w_i\}$, for $1 \leq i \leq n$ and $P \geq 0$.
(b) Develop a table-based dynamic-programming algorithm using the above recurrence that computes optimal solutions for the given instance in time $\mathrm{O}(np^*)$, where p^* is the profit of an optimal solution. Hint: Assume first that p^*, or at least a good upper bound for it, is known. Then explain how to achieve the goal without this assumption.

Exercise 12.14 (making change). Suppose you have to program a vending machine that should give exact change using a minimum number of coins.

(a) Develop an optimal greedy algorithm that works in the Euro zone with coins worth 1, 2, 5, 10, 20, 50, 100, and 200 cents and in Canada with coins worth (1,) 5, 10, 25, (50,) 100, and 200 cents[7].
(b) Show that this algorithm would not be optimal if there were also a 4 cent coin.
(c) Develop a dynamic-programming algorithm that gives optimal change for any currency system.

Exercise 12.15 (chained matrix products). We want to compute the matrix product $M_1 M_2 \cdots M_n$, where M_i is a $k_{i-1} \times k_i$ matrix. Assume that a pairwise matrix product is computed in the straightforward way using mks element multiplications to obtain the product of an $m \times k$ matrix with a $k \times s$ matrix. Exploit the associativity

[7] In Canada, the 50 cent coin is legal tender, but rarely used. Production of 1 cent coins was discontinued in 2012.

of matrix products to minimize the number of element multiplications needed. Use dynamic programming to find an optimal evaluation order in time $O(n^3)$. For example, the product of a 4×5 matrix M_1, a 5×2 matrix M_2, and a 2×8 matrix M_3 can be computed in two ways. Computing $M_1(M_2M_3)$ takes $5 \cdot 2 \cdot 8 + 4 \cdot 5 \cdot 8 = 240$ multiplications, whereas computing $(M_1M_2)M_3$ takes only $4 \cdot 5 \cdot 2 + 4 \cdot 2 \cdot 8 = 104$ multiplications.

Exercise 12.16 (edit distance). The *edit distance* (or *Levenshtein distance*) $L(s,t)$ between two strings s and t is the minimum number of character deletions, insertions, and replacements ("editing steps") one has to apply to s to produce the string t. For example, $L(\texttt{graph}, \texttt{group}) = 3$ (delete `h`, replace `a` by `o`, insert `u` before `p`). Let n be the length of s and m be the length of t. We define subproblems as follows: $d(i,j) = L(\langle s_1,\ldots,s_i\rangle, \langle t_1,\ldots,t_j\rangle)$, for $0 \le i \le n$, $0 \le j \le m$. Show that

$$d(i,j) = \min\left\{d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)+[s_i \ne t_j]\right\},$$

where $[s_i \ne t_j]$ is 1 if s_i and t_j are different and is 0 otherwise. Use these optimality equations to formulate a dynamic-programming algorithm to calculate the edit distance of $L(s,t)$. What is the running time? Explain how one subsequently finds an optimal sequence of editing steps to transform s into t.

Exercise 12.17. Does the principle of optimality hold for minimum spanning trees? Check the following three possibilities for definitions of subproblems: subsets of nodes, arbitrary subsets of edges, and prefixes of the sorted sequence of edges.

Exercise 12.18 (constrained shortest path). Consider a directed graph $G = (V,E)$ where edges $e \in E$ have a *length* $\ell(e)$ and a *cost* $c(e)$. We want to find a path from node s to node t that minimizes the total length subject to the constraint that the total cost of the path is at most C. Show that subpaths from s' to t' of optimal solutions are *not* necessarily shortest paths from s' to t'.

Parallel Dynamic Programming

Roughly speaking, dynamic-programming algorithms spend most of their time filling table entries. If several of these entries can be computed independently, this can be done in parallel. For example, for the knapsack problem, (12.1) tells us that all entries in row i depend only on row $i-1$ and hence can be computed in parallel.

Exercise 12.19. Explain how to parallelize one step of the Nemhauser–Ullmann algorithm for the knapsack problem using linear work and polylogarithmic time. Hint: You can use parallel merging and prefix sums.

Sometimes, in order to expose parallelism, we need to fill the tables in an order that we may not think of in the sequential algorithm. For example, when computing the edit distance in Exercise 12.16, we get some parallelism if we fill the table $d(\cdot,\cdot)$ along the diagonals, i.e., for fixed k, the entries $d(i,k-i)$ are independent and depend only on the preceding two diagonals.

12.4 Systematic Search – When in Doubt, Use Brute Force

In many optimization problems, for each given instance I the universe $\mathscr{U}$ of possible solutions is finite, so that we can in principle solve the optimization problem by trying all possibilities.[8] Applying this idea in the naive way does not lead very far. For many problems, the "search space" $\mathscr{U}$ grows rapidly with the input size $|I|$. But we can frequently restrict the search to *promising* candidates, and then the concept carries a lot further.

We shall explain the concept of systematic search using the knapsack problem and a specific approach to systematic search known as *branch-and-bound*. In Exercises 12.22 and 12.21, we outline systematic-search routines following a somewhat different pattern.

Branch-and-bound can be used when (feasible) solutions can be represented as vectors (or, more generally, sequences) whose components attain only finitely many values. The set of all these vectors (the *search space*) is searched systematically. Systematic search views the search space as a tree. Figure 12.5 gives pseudocode for a systematic-search routine *bbKnapsack* for the knapsack problem. The tree representing the search space is generated and traversed using a recursive procedure. Fig. 12.6 shows a sample run. A tree node corresponds to a partial solution in which some components have been fixed and the others are still free. Such a node can be regarded as a subinstance. The root is the vector in which all components are free. *Branching* is the elementary step in a systematic-search routine. The children of some node, or subinstance, are generated by inserting all sensible values for some free component of the partial solution. For each of the resulting new nodes the procedure is called recursively. Within the recursive call, the chosen value is fixed. In case of *bbKnapsack* the potential solutions are vectors in $\{0,1\}^n$, and in partial solutions some of the components are fixed, which means that for some objects it has been decided whether to include them or not. In principle an arbitrary component can be chosen to be fixed next. The routine *bbKnapsack* fixes the components x_i one after another in order of decreasing profit density. When treating x_i, it first tries including item i by setting $x_i := 1$, and then excluding it by setting $x_i := 0$. In both cases, components $x_{i+1}, \ldots, x_n$ are treated by recursion. The assignment $x_i := 1$ is not feasible and is not considered if the total weight of the included objects exceeds M. To organize this, a "remaining capacity" M' is carried along, which is reduced by the weight of any object when it is included. So the choice $x_i := 1$ is left out if $w_i > M'$. With these definitions, after all variables have been set, in the nth level of the recursion, *bbKnapsack* will have found a feasible solution. Indeed, without the *bounding rule*, which is discussed below, the algorithm would systematically explore a tree of partial solutions whose leaves correspond to all feasible solutions. *Branching* occurs at nodes at which both subtrees corresponding to the choices $x_i := 0$ and $x_i := 1$ are explored. Because of the order in which the components are fixed, the first feasible solution encountered would be the solution found by the algorithm *greedy*.

[8] The sentence "When in doubt, use brute force." is attributed to Ken Thompson.

```
**Function** $bbKnapsack((p_1,\dots,p_n),(w_1,\dots,w_n),M) : \{0,1\}^n$
    **assert** $p_1/w_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n$    // assume input sorted by profit density
    $\hat{x} = heuristicKnapsack((p_1,\dots,p_n),(w_1,\dots,w_n),M) : \{0,1\}^n$    // best solution so far
    $x : \{0,1\}^n$    // current partial solution $(x_1,\dots,x_n)$
    $recurse(1,M,0)$
    **return** $\hat{x}$

    // $recurse$ finds solutions assuming $x_1,\dots,x_{i-1}$ are fixed, $M' = M - \sum_{j<i} x_j w_j$, $P = \sum_{j<i} x_j p_j$.
    **Procedure** $recurse(i,M',P : \mathbb{N})$
        $u := P + upperBound((p_i,\dots,p_n),(w_i,\dots,w_n),M')$
        **if** $u > p \cdot \hat{x}$ **then**    // possibly better solution than $\hat{x}$
            **if** $i > n$ **then** $\hat{x} := x$
            **else**    // branch on variable $x_i$
                **if** $w_i \leq M'$ **then** $x_i := 1$; $recurse(i+1, M'-w_i, P+p_i)$
                **if** $u > p \cdot \hat{x}$ **then** $x_i := 0$; $recurse(i+1, M', P)$
```

Fig. 12.5. A branch-and-bound algorithm for the knapsack problem. An initial feasible solution is constructed by the function *heuristicKnapsack* using some heuristic algorithm. The function *upperBound* computes an upper bound for the possible profit.

Bounding is a method for pruning subtrees that cannot contain better solutions than those already known. A branch-and-bound algorithm keeps the best feasible solution found so far in a global variable, in our case the variable $\hat{x}$; this solution is often called the *incumbent* solution. It is initialized to a trivial solution or one determined by a heuristic routine. At all times, the value $p \cdot \hat{x}$ provides a lower bound on the value of the objective function that can be obtained. Whenever a partial solution x is processed, this lower bound is complemented by an upper bound u for the value of the objective function obtainable by extending x to a full feasible solution. In our example, the upper bound could be the the sum of the profit of the partial solution and the profit for the fractional knapsack problem with items $i..n$ and capacity $M' = M - \sum_{j<i} x_j w_j$. Branch-and-bound stops expanding the current branch of the search tree when $u \leq p \cdot \hat{x}$, i.e., when there is no hope of finding a better solution in the subtree rooted at the current node.

Exercise 12.20. Explain how to implement the function *upperBound* in Fig. 12.5 so that it runs in time $O(\log n)$. Hint: Precompute the prefix sums $\sum_{j \leq i} w_i$ and $\sum_{j \leq i} p_i$, for $1 \leq i \leq n$, and use binary search.

Exercise 12.21 (constraint programming and the eight-queens problem). Consider a chessboard. The task is to place eight queens on the board so that they do not attack each other, i.e., no two queens should be placed in the same row, column, diagonal, or antidiagonal. So each row contains exactly one queen. Let x_i be the position of the queen in row i. Then $x_i \in 1..8$. The solution must satisfy the following constraints: $x_i \neq x_j$, $i + x_i \neq j + x_j$, and $x_i - i \neq x_j - j$ for $1 \leq i < j \leq 8$. What do these conditions express? Show that they are sufficient. A systematic search can use

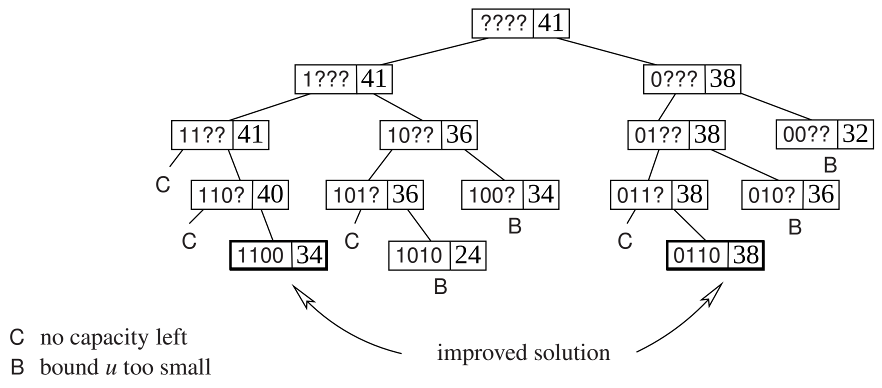


Fig. 12.6. The part of the search space explored by *bbKnapsack* for a knapsack instance with $p = (10, 24, 14, 24)$, $w = (1, 3, 2, 4)$, and $M = 5$ is processed. As initial solution we use the empty $\hat{x} = (0, 0, 0, 0)$ with profit 0. The function *upperBound* is computed by rounding down the optimal value of the objective function for the fractional knapsack problem. The nodes of the search tree contain the components $x_1 \cdots x_{i-1}$ fixed so far and the upper bound u. Left subtrees are explored first; they correspond to setting $x_i := 1$. There are two for not exploring the subtrees of a nonleaf node: Either there is not enough capacity left for the choice $x_i := 1$ (indicated by "C") or it turns out that a feasible solution with a profit at least as large as the upper bound is already known (indicated by "B", the recursive call does not have an effect).

the following optimization: When a variable x_i is fixed at some value, this excludes some values for variables that are still free. Modify the systematic search so that it keeps track of the values that are still available for free variables. Stop exploration as soon as there is a free variable that has no value available to it anymore. This technique of eliminating values is basic to *constraint programming*.

Exercise 12.22 (the 15-puzzle).

The 15-puzzle is a popular sliding-block puzzle. Fifteen square tiles marked with numbers 1, 2, ..., 15, sit in a 4×4 frame. This leaves one "hole". For an example see the top of the figure on the left. You are supposed to move the tiles into the right order, by performing *moves*. A move is defined as the action of interchanging a square and the hole in the array of tiles. Design an algorithm that finds a shortest sequence of moves from a given starting configuration to the ordered configuration shown at the bottom of the figure on the left. Assume that there is a solution. Use *iterative deepening depth-first search* [188]: Try all one-move sequences first, then all two-move sequences, and so on. This should work for the simpler 8-puzzle, with eight tiles in a 3×3 frame. For the 15-puzzle, use the following optimizations: Never undo the immediately preceding move. Use the number of moves that would be needed if all pieces could move freely to their target position as a lower bound, and stop exploring a subtree if this bound proves that the current search depth is too small. Decide beforehand whether the number of moves is odd or even. Implement your algorithm to run in constant time per move tried.

12.4.1 Solving Integer Linear Programs

In Sect. 12.1.2, we have seen how to formulate the knapsack problem as a 0-1 ILP. We shall now indicate how the branch-and-bound procedure developed for the knapsack problem can be applied to any 0-1 ILP. Recall that in a 0-1 ILP the values of the variables are constrained to 0 and 1. Our discussion will be brief, and we refer the reader to a textbook on integer linear programming [239, 286] for more information.

The main change is that the function *upperBound* now solves a general linear program that has variables $x_i, \ldots, x_n$ with range $[0,1]$. The constraints for this LP come from the input ILP, with the variables x_1 to x_{i-1} replaced by their values. In the remainder of this section, we shall simply refer to this linear program as "the LP".

If the LP has a feasible solution, *upperBound* returns the optimal value for the LP. If the LP has no feasible solution, *upperBound* returns $-\infty$ so that the ILP solver will stop exploring this branch of the search space. We shall next describe several generalizations of the basic branch-and-bound procedure that sometimes lead to considerable improvements:

Branch selection. We may pick any unfixed variable x_j for branching. In particular, we can make the choice depend on the solution of the LP. A commonly used rule is to branch on a variable whose fractional value in the LP is closest to $1/2$.

Order of search tree traversal. In the knapsack example, the search tree was traversed depth-first, and the 1-branch was tried first. In general, we are free to choose any order of tree traversal. There are at least two considerations influencing the choice of strategy. If no good feasible solution is known, it is good to use a depth-first strategy so that complete solutions are explored quickly. Otherwise, it is better to use a *best-first* strategy that explores those search tree nodes that are most likely to contain good solutions. Search tree nodes are kept in a priority queue, and the next node to be explored is the most promising node in the queue. The priority could be the upper bound returned by the LP. However, since the LP is expensive to evaluate, one sometimes settles for an approximation.

Finding solutions. We may be lucky in that the solution of the LP turns out to assign integer values to all variables. In this case there is no need for further branching. Application-specific heuristics can additionally help to find good solutions quickly.

Branch-and-cut. When an ILP solver branches too often, the size of the search tree explodes and it becomes too expensive to find an optimal solution. One way to avoid branching is to add constraints to the linear program that *cut* away solutions with fractional values for the variables without changing the solutions with integer values.

12.4.2 *Parallel Systematic Search

Systematic search is time-consuming and hence parallelization is desirable. Since the search procedure tries many solution options, there is considerable potential for parallelization. However, there are also challenges:

- The computations can be very fine-grained. This means that we may not be able to afford communication or other PE interactions for every individual computation.
- It is difficult to predict the total execution time involved in a subtree of the search. Hence, load balancing has to be dynamic.
- The computations are not completely independent. For example, in Fig. 12.5, when a new best solution is found, this may prune subtrees currently being explored by other PEs. Hence, this information has to be disseminated quickly. This also implies that parallel processing may explore more candidates than the sequential code does.

We first outline how to parallelize the branch-and-bound framework for ILP presented in Sect. 12.4.1 on a shared-memory system. Then we describe in more detail a distributed-memory parallelization of the branch-and-bound solver for the knapsack problem shown in Fig. 12.5. The basic idea is simple. When branching on a variable x, we can spawn several tasks – one for each value tried. This gives only very limited parallelism for a single variable, but by recursively applying the same idea to the spawned tasks, we quickly get a large set of tasks that can all be executed in parallel. An attractive way to manage the tasks is to use a bulk parallel priority queue (see Sect. 6.4). We may want to mix this with a depth-first strategy – occasionally, a task switches to depth-first search in order to obtain concrete solutions. The best solution value found so far should be a global variable that can be used to prune tasks that cannot possibly yield an improvement.

For the knapsack solver in Fig. 12.5, this best-first approach will not work well – even creating a task already costs time $\Omega(n)$, whereas we have seen in Exercise 12.20 that exploring a single node in the depth-first search approach can be done in logarithmic time. On the other hand, strict depth-first search is inherently sequential. The work-stealing approach described in Sect. 14.5 offers a way out. Working on a problem sequentially means depth-first exploration using backtracking. Splitting a problem means splitting off a part of the explored tree. This is made possible by managing the stack for backtracking explicitly so that splitting amounts to manipulating entries of the stack. A simple and effective splitting strategy is to search the stack top-down for the first item i where the branch $x_i = 0$ has not been tried yet. One of the resulting parts is the old stack, except that the branch $x_i = 0$ is now marked as already tried. The split-off part is a stack which copies the old stack for positions $1..i$ and initializes the search to explore the corresponding subtree with $x_i = 0$. Since each split sets one x_i to 0, the splitting depth from Sect. 14.5 can become at most n. If the search really requires work exponential in n, this means that the splitting depth is only logarithmic in the work to be done.

In order to do bounding effectively, the value $p \cdot \hat{x}$ has to be globally known. On a shared-memory machine we simply would use a global variable that is atomically updated whenever a larger value is found. On a distributed-memory machine, we need to emulate that approach in a scalable way. This is possible by embedding a spanning tree of bounded degree (e.g., maximum degree 3) and diameter $O(\log p)$ into the processor network. A PE that finds or receives an improved value forwards it to those neighbors in the tree which may not have seen that value.[9] This approach ensures that a new bound spreads over all PEs in time $O(\log p)$ without causing significant contention anywhere.

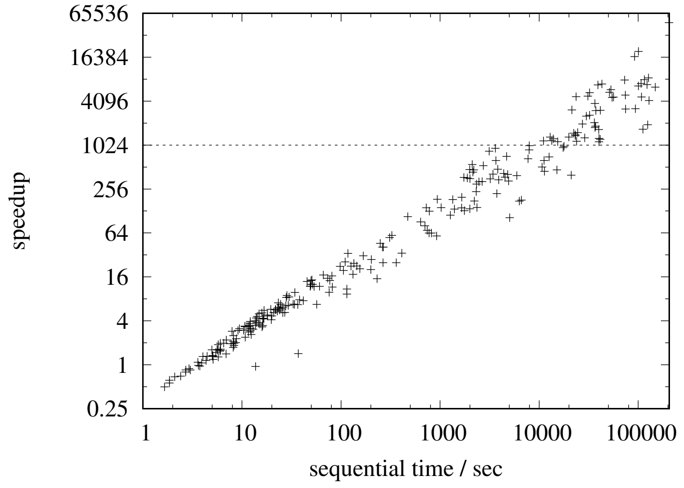


Fig. 12.7. Speedup as function of sequential execution time for 256 random instances with $p = 1024$, $n = 2000$, $w_i \in [0.01, 1.01]$, $p_i \in [w_i + 0.125,]$, and $M = \sum_i w_i/2$. See [267] for more details.

What speedup can we expect from this parallel knapsack searcher? When we use it just to validate an already known optimal solution $\hat{x}$, the analysis in Sect. 14.5 with a splitting depth and communication cost of $O(n)$ yields $O\left(T_{\text{seq}}/p + n^2\right)$, i.e., linear speedup for sufficiently difficult instances. For finding a solution, no such analysis is known. One might expect that the efficiency would be somewhat lower than for validating a solution since the delay in propagating new bounds will cause superfluous computations. However, superlinear speedup over the Algorithm in Fig. 12.5 for particular instances is also possible: Some PE might find the optimal solution very early, causing parts of the search space of the sequential algorithm to be pruned. Figure 12.7 shows speedups for 256 random difficult instances on 1024 PEs (on a rather slow machine even for the mid-1990s). As is to be expected, for instances solved quickly by the sequential solver, little speedup is obtained. However, for the most difficult instances, considerable superlinear speedup is observed. The sum of the sequential execution times is 1410 times larger than the sum of the parallel execution times, i.e., in some sense, the overall speedup is superlinear. This is surprising but not uncommon in parallel exploration of search spaces. Parallel search is more robust

[9] A similar, somewhat more complicated mechanism involving rooted trees is described in [171, 267].

than sequential search, since the sequential exploration of the search space might get bogged down in a large fruitless part of the search space.

For an even simpler approach to parallel systematic search that does not explicitly split the search space, refer to the discussion of portfolio-based SAT solvers (Page 365).

12.5 Local Search – Think Globally, Act Locally

The optimization algorithms we have seen so far are applicable only in special circumstances. Dynamic programming needs a special structure of the problem and may require a lot of space and time. Systematic search is usually too slow for large inputs. Greedy algorithms are fast but often yield only low-quality solutions. *Local search* is a widely applicable iterative procedure. It starts with some feasible solution and then moves from feasible solution to feasible solution by local modifications. Figure 12.8 gives the basic framework. We shall refine it later.

Local search maintains a current feasible solution x and the best solution $\hat{x}$ seen so far. In each step, local search moves from the current solution to a neighboring solution. What are neighboring solutions? Any solution that can be obtained from the current solution by making small changes to it. For example, in the case of the knapsack problem, we might remove up to two items from the knapsack and replace them by up to two other items. The precise definition of the neighborhood depends on the application and the algorithm designer. We use $\mathcal{N}(x)$ to denote the *neighborhood* of x. The second important design decision is which solution is chosen from the neighborhood. Finally, some heuristic decides when the search should stop.

In the rest of this section, we shall tell you more about local search.

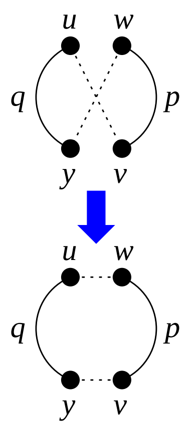
12.5.1 Hill Climbing

Hill climbing is the greedy version of local search. It moves only to neighbors that are better than the currently best solution. This restriction further simplifies the local search. The variables $\hat{x}$ and x are the same, and we stop when there are no improved solutions in the neighborhood $\mathcal{N}$. The only nontrivial aspect of hill climbing is the choice of the neighborhood. We shall give two examples where hill climbing works quite well, followed by an example where it fails badly.

```
find some feasible solution $x \in \mathcal{L}$
$\hat{x} := x$                                        // $\hat{x}$ is best solution found so far
while not satisfied with $\hat{x}$ do
    $x :=$ some heuristically chosen element from $\mathcal{N}(x) \cap \mathcal{L}$
    if $f(x) > f(\hat{x})$ then $\hat{x} := x$
```

Fig. 12.8. Local search

Our first example is the traveling salesman problem described in Sect. 11.7.2. Given an undirected graph and a distance function on the edges satisfying the triangle inequality, the goal is to find a shortest tour that visits all nodes of the graph. We define the neighbors of a tour as follows. Let (u, v) and (w, y) be two edges of the tour, i.e., the tour has the form $(u, v), p, (w, y), q$, where p is a path from v to w and q is a path from y to u. We remove these two edges from the tour, and replace them by the edges (u, w) and (v, y). The new tour first traverses (u, w), then uses the reversal of p back to v, then uses (v, y), and finally traverses q back to u; see the Figure on the right for an illustration. This move is known as a 2-exchange, and a tour that cannot be improved by a 2-exchange is said to be 2-optimal. In many instances of the traveling salesman problem, 2-optimal tours come quite close to optimal tours.



Exercise 12.23. Describe a scheme where three edges are removed and replaced by new edges.

An interesting example of hill climbing with a clever choice of the neighborhood function is the *simplex algorithm* for linear programming (see Sect. 12.1). This is the most widely used algorithm for linear programming. The set of feasible solutions $\mathcal{L}$ of a linear program is defined by a set of linear equalities and inequalities $a_i \cdot x \bowtie b_i$, $1 \le i \le m$. The points satisfying a linear equality $a_i \cdot x = b_i$ form a *hyperplane* in R^n, and the points satisfying a linear inequality $a_i \cdot x \le b_i$ or $a_i \cdot x \ge b_i$ form a *half-space*. Hyperplanes are the n-dimensional analogues of planes and half-spaces are the analogues of half-planes. The set of feasible solutions is an intersection of m half-spaces and hyperplanes and forms a *convex polytope*. We have already seen an example in two-dimensional space in Fig. 12.2. Figure 12.9 shows an example

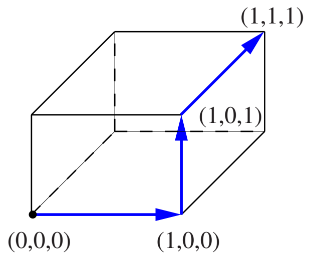


Fig. 12.9. The three-dimensional unit cube is defined by the inequalities $x \ge 0$, $x \le 1$, $y \ge 0$, $y \le 1$, $z \ge 0$, and $z \le 1$. At the vertices $(1,1,1)$ and $(1,0,1)$, three inequalities are tight, and on the edge connecting these vertices, the inequalities $x \le 1$ and $z \le 1$ are tight. For the objective "maximize $x + y + z$", the simplex algorithm starting at $(0,0,0)$ may move along the path indicated by the arrows. The vertex $(1,1,1)$ is optimal, since the half-space $x + y + z \le 3$ contains the entire feasible region and has $(1,1,1)$ in its boundary.

in three-dimensional space. Convex polytopes are the n-dimensional analogues of convex polygons. In the interior of the polytope, all inequalities are strict (= satisfied with inequality); on the boundary, some inequalities are tight (= satisfied with equality). The vertices and edges of the polytope are particularly important parts of the boundary. We shall now sketch how the simplex algorithm works. We assume that there are no equality constraints. Observe that an equality constraint c can be solved for any one of its variables; this variable can then be removed by substituting into the other equalities and inequalities. Afterwards, the constraint c is redundant and can be dropped.

The simplex algorithm starts at an arbitrary vertex of the feasible region. In each step, it moves to a neighboring vertex, i.e., a vertex reachable via an edge, with a larger objective value. If there is more than one such neighbor, a common strategy is to move to the neighbor with the largest objective value. If there is no neighbor with a larger objective value, the algorithm stops. *At this point, the algorithm has found the vertex with the maximum objective value.* In the examples in Figs. 12.2 and 12.9, the captions argue why this is true. The general argument is as follows. Let x^* be the vertex at which the simplex algorithm stops. The feasible region is contained in a cone with apex x^* and spanned by the edges incident to x^*. All these edges go to vertices with smaller objective values and hence the entire cone is contained in the half-space $\{x : c \cdot x \leq c \cdot x^*\}$. Thus no feasible point can have an objective value larger than x^*. We have described the simplex algorithm as a walk on the boundary of a convex polytope, i.e., in geometric language. It can be described equivalently using the language of linear algebra. Actual implementations use the linear-algebra description.

In the case of linear programming, hill climbing leads to an optimal solution. In general, however, hill climbing will not find an optimal solution. In fact, it will not even find a near-optimal solution. Consider the following example. Our task is to find the highest point on earth, i.e., Mount Everest. A feasible solution is any point on earth. The local neighborhood of a point is any point within a distance of 10 km. So the algorithm would start at some point on earth, then go to the highest point within a distance of 10 km, then go again to the highest point within a distance of 10 km, and so on. If one were to start from the second author's home (altitude 206 meters), the first step would lead to an altitude of 350 m, and there the algorithm would stop, because there is no higher hill within 10 km of that point. There are very few places in the world where the algorithm would continue for long, and even fewer places where it would find Mount Everest.

Why does hill climbing work so nicely for linear programming, but fail to find Mount Everest? The reason is that the earth has many local optima, hills that are the highest point within a range of 10 km. In contrast, a linear program has only one local optimum (which then, of course, is also a global optimum). For a problem with many local optima, we should expect *any* generic method to have difficulties. Observe that increasing the size of the neighborhoods in the search for Mount Everest does not really solve the problem, except if the neighborhoods are made to cover the entire earth. But finding the optimum in a neighborhood is then as hard as the full problem.

12.5.2 Simulated Annealing – Learning from Nature

If we want to ban the bane of local optima in local search, we must find a way to escape from them. This means that we sometimes have to accept moves that decrease the objective value. What could "sometimes" mean in this context? We have contradictory goals. On the one hand, we must be willing to take many downhill steps so that we can escape from wide local optima. On the other hand, we must be sufficiently target-oriented in order to find a global optimum at the end of a long, narrow ridge. A very popular and successful approach to reconciling these contradictory goals is *simulated annealing*; see Fig. 12.10. This works in phases that are controlled by a parameter T, called the *temperature* of the process. We shall explain below why the language of physics is used in the description of simulated annealing. In each phase, a number of moves are made. In each move, a neighbor $x' \in \mathcal{N}(x) \cap \mathcal{L}$ is chosen uniformly at random, and the move from x to x' is made with a certain probability. This probability is 1 if x' improves upon x. It is less than 1 if the move is to an inferior solution. The trick is to make the probability depend on T. If T is large, we make the move to an inferior solution relatively likely; if T is close to 0, we make such a move relatively unlikely. The hope is that, in this way, the process zeroes in on a region containing a good local optimum in phases of high temperature and then actually finds a near-optimal solution in the phases of low temperature. The exact choice of the transition probability in the case where x' is an inferior solution is given by $\exp((f(x') - f(x))/T)$. Observe that T is in the denominator and that $f(x') - f(x)$ is negative. So the probability decreases with T and also with the absolute loss in objective value.

```
find some feasible solution $x \in \mathcal{L}$
$T$:=some positive value                                  // initial temperature of the system
while $T$ is still sufficiently large do
    perform a number of steps of the following form
        pick $x'$ from $\mathcal{N}(x) \cap \mathcal{L}$ uniformly at random
        with probability $\min(1, \exp(\frac{f(x')-f(x)}{T})$ do $x := x'$
    decrease $T$                                 // make moves to inferior solutions less likely
```

Fig. 12.10. Simulated annealing

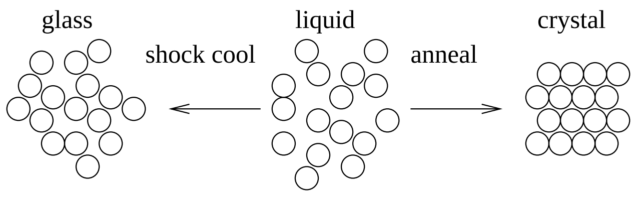


Fig. 12.11. Annealing versus shock cooling

Why is the language of physics used, and why this apparently strange choice of transition probabilities? Simulated annealing is inspired by the physical process of *annealing*, which can be used to minimize[10] the global energy of a physical system. For example, consider a pot of molten silica (SiO_2); see Fig. 12.11. If we cool it very quickly, we obtain a glass – an amorphous substance in which every molecule is in a local minimum of energy. This process of shock cooling has a certain similarity to hill climbing. Every molecule simply drops into a state of locally minimum energy; in hill climbing, we accept a local modification of the state if it leads to a smaller value of the objective function. However, a glass is not a state of global minimum energy. A state of much lower energy is reached by a quartz crystal, in which all molecules are arranged in a regular way. This state can be reached (or approximated) by cooling the melt very slowly. This process is called *annealing*. How can it be that molecules arrange themselves into a perfect shape over a distance of billions of molecular diameters although they feel only local forces extending over a few molecular diameters?

Qualitatively, the explanation is that local energy minima have enough time to dissolve in favor of globally more efficient structures. For example, assume that a cluster of a dozen molecules approaches a small perfect crystal that already consists of thousands of molecules. Then, with enough time, the cluster will dissolve and its molecules can attach to the crystal. Here is a more formal description of this process, which can be shown to hold for a reasonable model of the system: If cooling is sufficiently slow, the system reaches *thermal equilibrium* at every temperature. Equilibrium at temperature T means that a state x of the system with energy E_x is assumed with probability

$$\frac{\exp(-E_x/T)}{\sum_{y\in\mathcal{L}}\exp(-E_y/T)},$$

where T is the temperature of the system and $\mathcal{L}$ is the set of states of the system. This energy distribution is called the *Boltzmann distribution*. When T decreases, the probability of states with minimum energy grows. In fact, in the limit $T \to 0$, the probability of states with minimum energy approaches 1.

The same mathematics works for abstract systems corresponding to a maximization problem. We identify the cost function f with the energy of the system, and a feasible solution with the state of the system. It can be shown that the system approaches a Boltzmann distribution for a quite general class of neighborhoods and the following rules for choosing the next state:

```
pick $x'$ from $\mathcal{N}(x)\cap\mathcal{L}$ uniformly at random;
with probability $\min\left(1,\exp((f(x')-f(x))/T)\right)$ **do** $x := x'$.
```

The physical analogy gives some idea of why simulated annealing might work,[11] but it does not provide an implementable algorithm. We have to get rid of two infinities: For every temperature, we wait infinitely long to reach equilibrium, and we do that for infinitely many temperatures. Simulated-annealing algorithms therefore

[10] Note that we are talking about *minimization* now.

[11] Note that we have written "might work" and not "works".

have to decide on a *cooling schedule*, i.e., how the temperature T should be varied over time. A simple schedule chooses a starting temperature T_0 that is supposed to be just large enough that all neighbors are accepted. Furthermore, for a given problem instance, there is a fixed number N of iterations to be used at each temperature. The idea is that N should be as small as possible but still allow the system to get close to equilibrium. After every N iterations, T is decreased by multiplying it by a constant α less than 1. Typically, α is between 0.8 and 0.99. When T has become so small that moves to inferior solutions have become highly unlikely (this is the case when T is comparable to the smallest difference in objective value between any two feasible solutions), T is finally set to 0, i.e., the annealing process concludes with a hill-climbing search.

Better performance can be obtained with *dynamic schedules*. For example, the initial temperature can be determined by starting with a low temperature and increasing it quickly until the fraction of transitions accepted approaches 1. Dynamic schedules base their decision about how much T should be lowered on the actually observed variation in $f(x)$ during the local search. If the temperature change is tiny compared with the variation, it has too little effect. If the change is too close to or even larger than the variation observed, there is a danger that the system will be forced prematurely into a local optimum. The number of steps to be made until the temperature is lowered can be made dependent on the actual number of moves accepted. Furthermore, one can use a simplified statistical model of the process to estimate when the system is approaching equilibrium. The details of dynamic schedules are beyond the scope of this exposition. Readers are referred to [1] for more details on simulated annealing.

Exercise 12.24. Design a simulated-annealing algorithm for the knapsack problem. The local neighborhood of a feasible solution is all solutions that can be obtained by removing up to two elements and then adding up to two elements.

12.5.2.1 Graph Coloring

We shall now exemplify simulated annealing using the *graph-coloring problem* already mentioned in Sect. 2.13. Recall that we are given an undirected graph $G = (V, E)$ and are looking for an assignment $c : V \to 1..k$ such that no two adjacent nodes are given the same color, i.e., $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$. There is always a solution with $k = |V|$ colors; we simply give each node its own color. The goal is to minimize k. There are many applications of graph coloring and related problems. The most "classical" one is map coloring – the nodes are countries and edges indicate that countries have a common border, and thus those countries should not be rendered in the same color. A famous theorem of graph theory states that all maps (i.e., planar graphs) can be colored with at most four colors [264]. Sudoku puzzles are a well-known instance of the graph-coloring problem, where the player is asked to complete a partial coloring of the graph shown in Fig. 12.12 with the digits 1..9. We shall present two simulated-annealing approaches to graph coloring; many more have been tried.

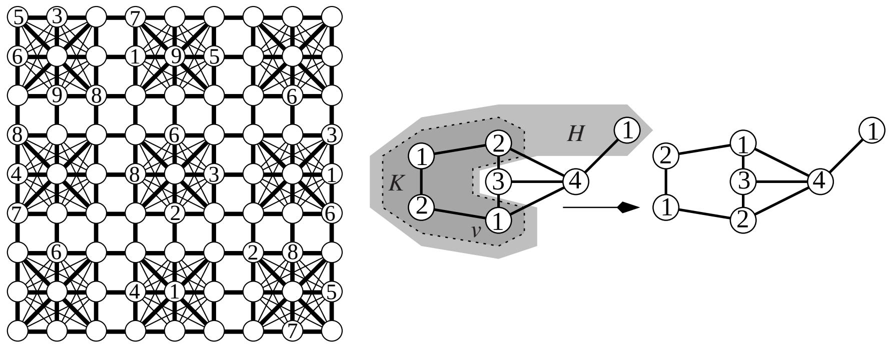


Fig. 12.12. The figure on the *left* shows a partial coloring of the graph underlying sudoku puzzles. The **bold** straight-line segments indicate cliques consisting of all nodes touched by the line. The figure on the *right* shows a step of Kempe chain annealing using colors 1 and 2 and a node v.

Kempe chain annealing. The obvious objective function for graph coloring is the number of colors used. However, this choice of objective function is too simplistic in a local-search framework, since a typical local move will not change the number of colors used. We need an objective function that rewards local changes that are "on a good way" towards using fewer colors. One such function is the sum of the squared sizes of the color classes. Formally, let $C_i = \{v \in V : c(v) = i\}$ be the set of nodes that are colored i. Then

$$f(c) = \sum_i |C_i|^2.$$

This objective function is to be maximized. Observe that the objective function increases when a large color class is enlarged further at the cost of a small color class. Thus local improvements will eventually empty some color classes, i.e., the number of colors decreases.

Having settled the objective function, we come to the definition of a local change or a neighborhood. A trivial definition is as follows: A local change consists in recoloring a single vertex; it can be given any color not used on one of its neighbors. Kempe chain annealing uses a more liberal definition of "local recoloring". Alfred Bray Kempe (1849–1922) was one of the early investigators of the four-color problem; he invented Kempe chains in his futile attempts at a proof. Suppose that we want to change the color $c(v)$ of node v from i to j. In order to maintain feasibility, we have to change some other node colors too: Node v might be connected to nodes currently colored j. So we color these nodes with color i. These nodes might, in turn, be connected to other nodes of color j, and so on. More formally, consider the node-induced subgraph H of G which contains all nodes with colors i and j. The connected component of H that contains v is the *Kempe chain* K we are interested in. We maintain feasibility by swapping colors i and j in K. Figure 12.12 gives an example. Kempe chain annealing starts with any feasible coloring.

***Exercise 12.25.** Use Kempe chains to prove that any planar graph G can be colored with five colors. Hint: Use the fact that a planar graph is guaranteed to have a node of degree five or less. Let v be any such node. Remove it from G, and color $G-v$ recursively. Put v back in. If at most four different colors are used on the neighbors of v, there is a free color for v. So assume otherwise. Assume, without loss of generality, that the neighbors of v are colored with colors 1 to 5 in clockwise order. Consider the subgraph of nodes colored 1 and 3. If the neighbors of v with colors 1 and 3 are in distinct connected components of this subgraph, a Kempe chain can be used to recolor the node colored 1 with color 3. If they are in the same component, consider the subgraph of nodes colored 2 and 4. Argue that the neighbors of v with colors 2 and 4 must be in distinct components of this subgraph.

The penalty function approach. A generally useful idea for local search is to relax some of the constraints on feasible solutions in order to make the search more flexible and to ease the discovery of a starting solution. Observe that we have assumed so far that we somehow have a feasible solution available to us. However, in some situations, finding any feasible solution is already a hard problem; the eight-queens problem of Exercise 12.21 is an example. In order to obtain a feasible solution at the end of the process, the objective function is modified to penalize infeasible solutions. The constraints are effectively moved into the objective function.

In the graph-coloring example, we now also allow illegal colorings, i.e., colorings in which neighboring nodes may have the same color. An initial solution is generated by guessing the number of colors needed and coloring the nodes randomly. A neighbor of the current coloring c is generated by picking a random color j and a random node v colored j, i.e., $c(v) = j$. Then, a random new color for node v is chosen from all the colors already in use plus one fresh, previously unused color.

As above, let C_i be the set of nodes colored i and let $E_i = E \cap (C_i \times C_i)$ be the set of edges connecting two nodes in C_i. The objective is to minimize

$$f(c) = 2\sum_i |C_i| \cdot |E_i| - \sum_i |C_i|^2 .$$

The first term penalizes illegal edges; each illegal edge connecting two nodes of color i contributes the size of the ith color class. The second term favors large color classes, as we have already seen above. The objective function does not necessarily have its global minimum at an optimal coloring. However, local minima are legal colorings. Hence, the penalty version of simulated annealing is guaranteed to find a legal coloring even if it starts with an illegal coloring.

Exercise 12.26. Show that the objective function above has its local minima at legal colorings. Hint: Consider the change in $f(c)$ if one end of a legally colored edge is recolored with a fresh color. Prove that the objective function above does not necessarily have its global optimum at a solution using the minimum number of colors.

Experimental results. Johnson et al. [168] performed a detailed study of algorithms for graph coloring, with particular emphasis on simulated annealing. We shall briefly

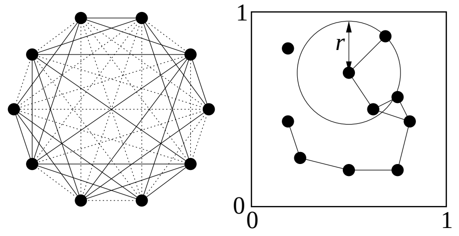


Fig. 12.13. *Left*: a random graph with 10 nodes and $p = 0.5$. The edges chosen are drawn solid, and the edges rejected are drawn dashed. *Right*: a random geometric graph with 10 nodes and range $r = 0.27$.

report on their findings and then draw some conclusions. Most of their experiments were performed on random graphs in the $G_{n,p}$ model or on random geometric graphs.

In the $G_{n,p}$ model, where p is a parameter in $[0,1]$, an undirected random graph with n nodes is built by adding each of the $n(n-1)/2$ candidate edges with probability p. The random choices are independent for distinct edges. In this way, the expected degree of every node is $p(n-1)$ and the expected number of edges is $pn(n-1)/2$. For random graphs with 1000 nodes and edge probability 0.5, Kempe chain annealing produced very good colorings, given enough time. However, a sophisticated and expensive greedy algorithm, XRLF, produced even better solutions in less time. For very dense random graphs with $p = 0.9$, Kempe chain annealing performed better than XRLF. For sparser random graphs with edge probability 0.1, penalty function annealing outperformed Kempe chain annealing and could sometimes compete with XRLF.

Another interesting class of random inputs is *random geometric graphs*. Here, we choose n random, uniformly distributed points in the unit square $[0,1] \times [0,1]$. These points represent the nodes of the graph. We connect two points by an edge if their Euclidean distance is less than or equal to some given range r. Figure 12.13 gives an example. Such instances are frequently used to model situations where the nodes represent radio transmitters and colors represent frequency bands. Nodes that lie within a distance r from one another must not use the same frequency, to avoid interference. For this model, Kempe chain annealing performed well, but was outperformed by a third annealing strategy, called *fixed-K annealing*.

What should we learn from this? The relative performance of the simulated-annealing approaches depends strongly on the class of inputs and the available computing time. Moreover, it is impossible to make predictions about their performance on any given instance class on the basis of experience from other instance classes. So, be warned. Simulated annealing is a heuristic and, as for any other heuristic, you should not make claims about its performance on an instance class before you have tested it extensively on that class.

12.5.3 More on Local Search

We close our treatment of local search with a discussion of three refinements that can be used to modify or replace the approaches presented so far.

Threshold Acceptance. There seems to be nothing magic about the particular form of the acceptance rule used in simulated annealing. For example, a simpler yet also successful rule uses the parameter T as a threshold. New states with a value $f(x)$ below the threshold are accepted, whereas others are not.

Tabu Lists. Local-search algorithms sometimes return to the same suboptimal solution again and again – they cycle. For example, simulated annealing might have reached the top of a steep hill. Randomization will steer the search away from the optimum, but the state may remain on the hill for a long time. *Tabu search* steers the search away from local optima by keeping a *tabu list* of "solution elements" that should be "avoided" in new solutions for the time being. For example, in graph coloring, a search step could change the color of a node v from i to j and then store the tuple (v,i) in the tabu list to indicate that color i is forbidden for v as long as (v,i) is in the tabu list. Usually, this tabu condition is not applied if an improved solution is obtained by coloring node v with color i. Tabu lists are so successful that they can be used as the core technique of an independent variant of local search called *tabu search*.

Restarts. The typical behavior of a well-tuned local-search algorithm is that it moves to an area with good feasible solutions and then explores this area, trying to find better and better local optima. However, it might be that there are other, faraway areas with much better solutions. The search for Mount Everest illustrates this point. If we start in Australia, the best we can hope for is to end up at Mount Kosciuszko (altitude 2229 m), a solution far from optimal. It therefore makes sense to run the algorithm multiple times with different random starting solutions because it is likely that different starting points will explore different areas of good solutions. Starting the search for Mount Everest at multiple locations and in all continents will certainly lead to a better solution than just starting in Australia. Even if these restarts do not improve the average performance of the algorithm, they may make it more robust in the sense that it will be less likely to produce grossly suboptimal solutions.

12.5.4 Parallel Local Search

Local search is difficult to parallelize since, by definition, it performs one step after another. Of course, we can parallelize the operations in one step, for example by evaluating several members in the neighborhood $\mathcal{N}(x)$ in parallel. In Kempe chain annealing, several processors might explore different Kempe chains starting from different nodes. We can also try to perform more work in each step to reduce the number of steps, for example by using a larger neighborhood. We can also perform several independent local searches and then take the best result. In some sense, this is a parallelization of the restart strategy described in Sect. 12.5.3. A successful parallelization of a local search strategy might use parallelism on multiple levels. For example a few threads on each processor chip could evaluate an objective function in parallel, several of these thread groups working on different members of a large neighborhood, and multiple nodes of a distributed-memory machine work on independent local searches. However, for more scalable solutions, one should consider more advanced interactions between parallel solvers such as in the evolutionary algorithms described next.

12.6 Evolutionary Algorithms

Living beings are ingeniously adaptive to their environment, and master the problems encountered in their daily life with great ease. Can we somehow use the principles of life to develop good algorithms? The theory of evolution tells us that the mechanisms leading to this performance are *mutation*, *recombination*, and *survival of the fittest*. What could an evolutionary approach mean for optimization problems?

The genome describing an individual corresponds to the description of a feasible solution. We can also interpret infeasible solutions as dead or ill individuals. In nature, it is important that there is a sufficiently large *population* of genomes; otherwise, recombination deteriorates to incest, and survival of the fittest cannot demonstrate its benefits. So, instead of one solution as in local search, we now work with a pool of feasible solutions.

The individuals in a population produce offspring. Because resources are limited, individuals better adapted to the environment are more likely to survive and to produce more offspring. In analogy, feasible solutions are evaluated using a fitness function f, and fitter solutions are more likely to survive and to produce offspring. Evolutionary algorithms usually work with a solution pool of limited size, say N. Survival of the fittest can then be implemented as keeping only the N best solutions.

Even in bacteria, which reproduce by cell division, no offspring is identical to its parent. The reason is *mutation*. When a genome is copied, small errors happen. Although mutations usually have an adverse effect on fitness, some also improve fitness. Local changes in a solution are the analogy of mutations.

An even more important ingredient in evolution is *recombination*. Offspring contain genetic information from both parents. The importance of recombination is easy to understand if one considers how rare useful mutations are. Therefore it takes much longer to obtain an individual with two new useful mutations than it takes to combine two individuals with two different useful mutations.

We now have all the ingredients needed for a generic evolutionary algorithm; see Fig. 12.14. As with the other approaches presented in this chapter, many details need to be filled in before one can obtain an algorithm for a specific problem. The algorithm starts by creating an initial population of size N. This process should involve randomness, but it is also useful to use heuristics that produce good initial solutions.

```
Create an initial population $population = \{x^1, \ldots, x^N\}$
**while** not finished **do**
    **if** matingStep **then**
        select individuals $x^1$, $x^2$ with high fitness and produce $x' := mate(x^1, x^2)$
    **else** select an individual $x^1$ with high fitness and produce $x' = mutate(x^1)$
    $population := population \cup \{x'\}$
    $population := \{x \in population : x \text{ is sufficiently fit}\}$
```

Fig. 12.14. A generic evolutionary algorithm

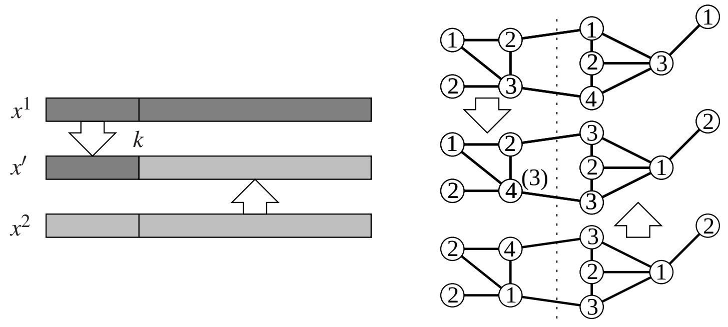


Fig. 12.15. Mating using crossover (*left*) and by stitching together pieces of a graph coloring (*right*).

In the loop, it is first decided whether an offspring should be produced by mutation or by recombination. This is a probabilistic decision. Then, one or two individuals are chosen for reproduction. To put selection pressure on the population, it is important to base reproductive success on the fitness of the individuals. However, it is usually not desirable to draw a hard line and use only the fittest individuals, because this might lead to too uniform a population and hence incest. For example, one can instead choose reproduction candidates randomly, giving a higher selection probability to fitter individuals. An important design decision is how to fix these probabilities. One choice is to sort the individuals by fitness and then to define the reproduction probability as some decreasing function of rank. This indirect approach has the advantage that it is independent of the objective function f and the absolute fitness differences between individuals, which are likely to decrease during the course of evolution.

The most critical operation is *mate*, which produces new offspring from two ancestors. The "canonical" mating operation is called *crossover*. Here, individuals are assumed to be represented by a string of n bits. An integer k is chosen. The new individual takes its first k bits from one parent and its last $n-k$ bits from the other parent. Figure 12.15 shows this procedure. Alternatively, one may choose k random positions from the first parent and the remaining bits from the other parent. For our knapsack example, crossover is a quite natural choice. Each bit decides whether the corresponding item is in the knapsack or not. In other cases, crossover is less natural or would require a very careful encoding. For example, for graph coloring, it would seem more natural to cut the graph into two pieces such that only a few edges are cut. Now one piece inherits its colors from the first parent, and the other piece inherits its colors from the other parent. Some of the edges running between the pieces might now connect nodes with the same color. This could be repaired using some heuristic, for example choosing the smallest legal color for miscolored nodes in the part corresponding to the first parent. Figure 12.15 gives an example.

Mutations are realized as in local search. In fact, local search is nothing but an evolutionary algorithm with population size 1.

The simplest way to limit the size of the population is to keep it fixed by removing the least fit individual in each iteration. Other approaches that provide room for different "ecological niches" can also be used. For example, for the knapsack problem, one could keep all Pareto-optimal solutions. The evolutionary algorithm would then resemble the optimized dynamic-programming algorithm.

12.6.1 Parallel Evolutionary Algorithms

In principle, evolutionary algorithms are easy to parallelize [14]. We simply perform multiple mating and mutation steps in parallel. This may imply that we have to work with a larger population than in a sequential algorithm. Rather than using a single large population, it then makes sense to work with multiple subpopulations. Each PE (or group of PEs) generally works on the local subpopulation. Subpopulations not only reduce communication and synchronization overhead but may also allow better solution quality. This is an effect also observed in nature – multiple subpopulations, for example on different islands, lead to higher biological diversity. This was already observed by Charles Darwin [82].

To avoid "incest", fit individuals are occasionally exchanged between the subpopulations. For example, each subpopulation can occasionally send one of its fittest individuals (or a mutation thereof) to a random other subpopulation. This approach guarantees a good balance between communication overhead and spreading of good solutions. It has been used very successfully for graph partitioning [276].

12.7 Implementation Notes

We have seen several generic approaches to optimization that are applicable to a wide variety of problems. When you face a new application, you are therefore likely to have a choice from among more approaches than you can realistically implement. In a commercial environment, you may have to home in on a single approach quickly. Here are some rules of thumb that may help:

- Study the problem, relate it to problems you are familiar with, and search for it on the web.
- Look for approaches that have worked on related problems.
- Consider black-box solvers.
- If the problem instances are small, systematic search or dynamic programming may allow you to find optimal solutions.
- If none of the above looks promising, implement a simple prototype solver using a greedy approach or some other simple, fast heuristic; the prototype will help you to understand the problem and might be useful as a component of a more sophisticated algorithm.
- Develop a local-search algorithm. Focus on a good representation of solutions and how to incorporate application-specific knowledge into the searcher. If you have a promising idea for a mating operator, you can also consider evolutionary algorithms. Use randomization and restarts to make the results more robust.

There are many implementations of linear-programming solvers. Since a good implementation is *very* complicated, you should definitely use one of these packages except in very special circumstances. The Wikipedia page “Linear programming” is a good starting point. Some systems for linear programming also support integer linear programming.

There are also many frameworks that simplify the implementation of local-search or evolutionary algorithms. Since these algorithms are fairly simple, the use of these frameworks is not as widespread as for linear programming. Nevertheless, the implementations available might have nontrivial built-in algorithms for dynamic setting of search parameters, and they might support parallel processing. The Wikipedia page “Evolutionary algorithm” contains pointers.

12.8 Historical Notes and Further Findings

We have only scratched the surface of (integer) linear programming. Implementing solvers, clever modeling of problems, and handling huge input instances have led to thousands of scientific papers. In the late 1940s, Dantzig invented the simplex algorithm [81]. Although this algorithm works well in practice, some of its variants take exponential time in the worst case. It is a well-known open problem whether some variant runs in polynomial time in the worst case. It is known, though, that even slightly perturbing the coefficients of the constraints leads to polynomial expected execution time [301]. Sometimes, even problem instances with an exponential number of constraints or variables can be solved efficiently. The trick is to handle explicitly only those constraints that may be violated and those variables that may be nonzero in an optimal solution. This works if we can efficiently find violated constraints or possibly nonzero variables and if the total number of constraints and variables generated remains small. Khachiyan [180] and Karmarkar [176] found polynomial-time algorithms for linear programming. There are many good textbooks on linear programming (e.g., [44, 98, 122, 239, 286, 320]).

Another interesting black-box solver is *constraint programming* [149, 203]. We hinted at the technique in Exercise 12.21. Here, we are again dealing with variables and constraints. However, now the variables come from discrete sets (usually small finite sets). Constraints come in a much wider variety. There are equalities and inequalities, possibly involving arithmetic expressions, but also higher-level constraints. For example, $\mathit{allDifferent}(x_1,\ldots,x_k)$ requires that $x_1,\ldots,x_k$ all receive different values. Constraint programs are solved using a cleverly pruned systematic search. Constraint programming is more flexible than linear programming, but restricted to smaller problem instances. Wikipedia is a good starting point for learning more about constraint programming.