Generic Approaches to Optimization

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, what items should he pack into the knapsack to maximize the profit from his next trip?

This problem, usually called the *knapsack problem*, has many other applications. The books [122, 109] describe many of them. 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 have 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 optimization problem can be described by a set $\mathcal{U}$ of *potential* solutions, a set $\mathcal{L}$ of *feasible* solutions, and an *objective function* $f : \mathcal{L} \to \mathbb{R}$. 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 look for a solution that minimizes the value of the objective. In an *existence problem*, f is arbitrary and the question is whether the set of feasible solutions is nonempty.

For example, in the case of the knapsack problem with n items, a potential solution is simply a vector $x = (x_1, \ldots, x_n)$ with $x_i \in \{0,1\}$. Here $x_i = 1$ indicates that "element i is put into the knapsack" and $x_i = 0$ indicates that "element i is left out". Thus $\mathcal{U} = \{0,1\}^n$. The profits and weights are specified by vectors $p = (p_1, \ldots, p_n)$ and $w = (w_1, \ldots, w_n)$. A potential solution x is feasible if its weight does not exceed

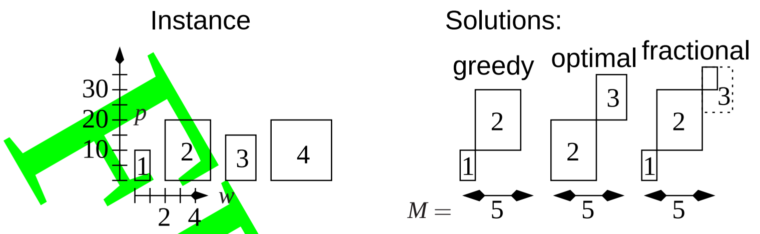


Fig. 12.1. The *left part* shows a knapsack instance with $p = (10,20,15,20)$, $w = (1,3,2,4)$, and $M = 5$. The items are indicated as rectangles whose width and height correspond to weight and profit, respectively. The *right part* shows three solutions: the one computed by the greedy algorithm from Sect. 12.2, an optimal solution computed by the dynamic programming algorithm from Sect. 12.3, and the solution of the linear relaxation (Sect. 12.1.1). The optimal solution has weight 5 and profit 35

the capacity of the knapsack, i.e., $\sum_{1\le i\le n} w_i x_i \le M$. The dot product $w \cdot x$ is a convenient shorthand for $\sum_{1\le i\le n} w_i x_i$. We can then say that $\mathscr{L} = \{x \in \mathscr{U} : w \cdot x \le M\}$ 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.[1]

We shall present seven generic approaches. We start out with black-box solvers that can be applied to any problem that can be formulated in the problem specification language of the solver. In this 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* that we have already seen 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 discuss 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.

12.1 Linear Programming – a Black-Box Solver

The easiest way to solve 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. Of course, the question is, for what

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

kinds of specification are general solvers available? Here, we introduce a particularly large class of problems for which efficient black-box solvers are available.

Definition 12.1. *A* linear program *(LP)*[2] *with n* variables *and m* constraints *is a maximization problem defined on a vector $x = (x_1, \dots, x_n)$ of real-valued variables. The* objective function *is a linear function f of x, i.e., $f : \mathbb{R}^n \to \mathbb{R}$ with $f(x) = c \cdot x$, where $c = (c_1, \dots, c_n)$ is called* cost *or* profit[3] vector. *The variables are constrained by m linear constraints of the form $a_i \cdot x \bowtie_i b_i$, where $\bowtie_i \in \{\leq, \geq, =\}$, $a_i = (a_{i1}, \dots, a_{in}) \in \mathbb{R}^n$, and $b_i \in \mathbb{R}$ for $i \in 1..m$. The set of feasible solutions is given by*

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

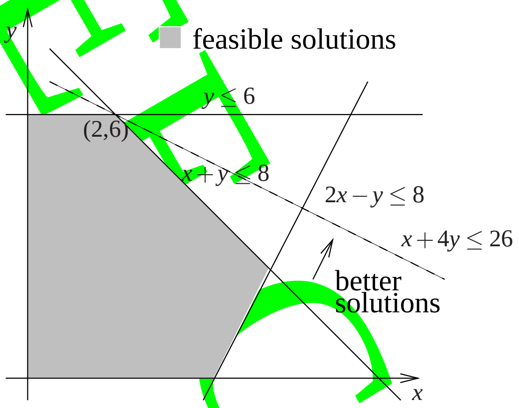


Fig. 12.2. A simple two-dimensional linear program in variables x and y, with three constraints and the objective "maximize $x + 4y$". The feasible region is shaded, and $(x, y) = (2, 6)$ is the optimal solution. Its objective value is 26. The vertex $(2, 6)$ is optimal because the half-plane $x + 4y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

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 a 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

[2] The term "linear program" stems from the 1940s [45] and has nothing to do with the modern meaning of "program" as in "computer program".

[3] It is common to use the term "profit" in maximization problems and "cost" in minimization problems.

farmer. The i-th 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.

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 0.01mg 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 or it does not. There is no possibility of adding an item partially. 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. 12.1.1) are the right tool for the knapsack problem.

We next connect linear programming to the problems that we have studied in previous chapters of the book. We shall show how to formulate the single-source shortest-path problem with nonnegative edge weights as a linear program. Let $G = (V,E)$ be a directed graph, let $s \in V$ be the source node, and let $c : E \to \mathbb{R}_{\geq 0}$ be the cost function on the edges of G. In our linear program, we have a variable d_v for every vertex of the graph. The intention is that d_v denotes the cost of the shortest path from s to v. Consider

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

Theorem 12.2. *Let $G = (V,E)$ be a directed graph, $s \in V$ a designated vertex, and $c : E \to \mathbb{R}_{\geq 0}$ 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 length of the shortest path from s to v. Then $\mu(v) \in \mathbb{R}_{\geq 0}$, since all nodes are reachable from s, and hence no vertex can have a distance $+\infty$ from s. We observe first that $d_v := \mu(v)$ 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 $s = v_0, v_1, \ldots, v_k = v$ 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 that $d_{v_j} \leq \sum_{0 \leq i < j} c(v_i, v_{i+1})$ by induction on j. 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 $d_v \le \mu(v)$ for all v for any feasible solution $(d_v)_{v\in V}$. 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 weights? Does it still work in the absence of negative cycles?

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

Exercise 12.3. Let $G = (V,E)$ be a directed graph and let s and t be two nodes. Let $cap : E \to \mathbb{R}_{\ge 0}$ and $c : E \to \mathbb{R}_{\ge 0}$ 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 $f : E \to \mathbb{R}_{\ge 0}$ with $0 \le f(e) \le cap(e)$ for all e and flow conservation at all nodes except s and t, i.e., for all $v \ne s,t$, 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. Show that this problem can be formulated as an LP.

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 this problem as an LP.

Linear programs are so important because they combine expressive power with efficient solution algorithms.

Theorem 12.3. *Linear programs can be solved in polynomial time [110, 106].*

The worst-case running time of the best algorithm known is $O\big(\max(m,n)^{7/2}L\big)$. In this bound, it is assumed that all coefficients c_j, a_{ij}, and b_i are integers with absolute value bounded by 2^L; n and m are the numbers of variables and constraints, respectively. Fortunately, the worst case rarely arises. Most linear programs can be solved relatively quickly by several procedures. One, the simplex algorithm, is briefly outlined in Sect. 12.5.1. For now, we should remember two facts: first, many problems can be formulated as linear programs, and second, there are efficient linear-program 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.

12.1.1 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

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

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 you know an upper bound U on the value of any variable in the optimal solution. Hint: replace any variable of the original ILP by a set of $\mathrm{O}(\log U)$ 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 $x_i \in \{0,1\}$ by the constraint $x_i \in [0,1]$.

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

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

This has a natural interpretation. It is no longer required to add items completely to the knapsack; 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* 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 j such that $\sum_{i=1}^{j} w_i > M$ (if there is no such index, we can take all knapsack items). Now we set

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

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 might 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. 5.5.

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_j to zero. 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 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.

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, and a decision about an item, for example, whether to include it in the solution or not, is made when the item is considered. 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 Chap. 10 and for minimum spanning trees in Chap. 11 also follow the greedy strategy. For the single-source shortest-path problem with nonnegative weights, we considered the edges in order of the tentative distance of their source nodes. For these problems, the greedy approach led to an optimal solution.

Usually, greedy algorithms yield only suboptimal solutions. Let us consider the knapsack problem again. A typical greedy approach would be to scan the items in

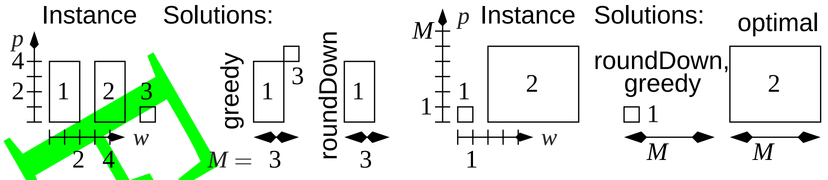


Fig. 12.3. Two instances of the knapsack problem. *Left*: for $p = (4,4,1)$, $w = (2,2,1)$, and $M = 3$, *greedy* performs better than *roundDown*. *Right*: for $p = (1, M-1)$ and $w = (1, M)$, both *greedy* and *roundDown* are far from optimal

order of decreasing profit density and to include items that 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* 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, 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 optimum. 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 high profit density 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_j of the fractional solution.[4] It then returns the better of the two.

We can give an interesting performance guarantee. The algorithm *round* 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. Let x^* denote any optimal solution, and let x^f be the optimal solution to the fractional knapsack problem. Then $p \cdot x^* \leq p \cdot x^f$. The value of the objective function is increased further by setting $x_j = 1$ in the fractional solution. We obtain

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

$\square$

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

There are many ways to refine the 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 come 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* $L_{\max} = \max_{1 \le j \le m} \ell_j$ of the schedule.

One application scenario is as follows. We have a video game processor with several identical processor cores. The jobs would be 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.

We give next a simple greedy algorithm for the problem above [80] that has the additional property that it does not need to know the sizes of the jobs in advance. We assign jobs in the order they arrive. 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 algorithm is frequently referred to as the *shortest-queue algorithm*. It does not guarantee optimal solutions, but always computes nearly optimal solutions.

Theorem 12.5. *The shortest-queue algorithm ensures that*

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

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

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

Solving this for $L_{\max}$ yields

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

□

We are almost finished. We now observe that $\sum_i t_i/m$ and $\max_i t_i$ are lower bounds on the makespan of any schedule and hence also 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 $L_1 = \sum_i t_i/m$ and $L_2 = \max_i t_i$. The makespan of the optimal solution is at least $\max(L_1, L_2)$. The makespan of the shortest-queue solution is bounded by

$$L_1 + \frac{m-1}{m}L_2 \le \frac{mL_1 + (m-1)L_2}{m} \le \frac{(2m-1)\max(L_1,L_2)}{m}$$
$$= (2-\frac{1}{m})\cdot\max(L_1,L_2)\ .$$

□

The shortest-queue algorithm is no better than claimed above. Consider an instance with $n = m(m-1)+1$, $t_n = m$, and $t_i = 1$ for $i < n$. The optimal solution has a makespan $L_{\max}^{opt} = m$, 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 12.7.** Show that the shortest-queue algorithm achieves an approximation ratio of $4/3$ if the jobs are sorted by decreasing size.

***Exercise 12.8 (bin packing).** Suppose a smuggler boss 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 they can all carry a weight M. Try to obtain an approximation ratio for your *bin-packing* algorithm.

Boolean formulae provide another powerful description language. 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 now formulate the *pigeonhole principle* as a satisfiability problem: it is impossible to pack $n+1$ items into n bins such that every bin contains one item at most. 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. Every item must be put into (at least) one bin, i.e., $x_{i1} \vee \ldots \vee x_{in}$ for $1 \le i \le n+1$. No bin should receive more than one item, i.e., $\neg(\bigvee_{1\le i<h\le n+1} x_{ij}x_{hj})$ for $1 \le j \le n$. The conjunction of these formulae is unsatisfiable. SAT solvers decide the satisfiability of Boolean formulae. Although the satisfiability problem is **NP**-complete, there are now solvers that can solve real-world problems that involve hundreds of thousands of variables.[5]

Exercise 12.9. Formulate the pigeonhole principle as an integer linear program.

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

12.3 Dynamic Programming – Building It Piece by Piece

For many optimization problems, the following *principle of optimality* holds: *an optimal solution is composed of optimal solutions to subproblems. If a subproblem has several optimal solutions, it does not matter which one is used.*

The idea behind dynamic programming is to build an exhaustive table of optimal solutions. We start with trivial subproblems. We build optimal solutions for increasingly larger problems by constructing them from the tabulated solutions to smaller problems.

Again, we shall use the knapsack problem as an example. We define $P(i,C)$ as 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 start with trivial cases and work our way up. The trivial cases are "no items" and "total weight zero". In both of these cases, the maximum profit is zero. So

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

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 a solution of total weight at most $C-w_i$ for the first $i-1$ items. Of course, the former alternative is only feasible if $C \geq w_i$. We summarize this discussion in the following recurrence for $P(i,C)$:

$$P(i,C) = \begin{cases} \max(P(i-1,C), P(i-1,C-w_i)+p_i) & \text{if } w_i \leq C \\ P(i-1,C) & \text{if } w_i > C \end{cases}$$

Exercise 12.10. Show that the case distinction 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 $P(i,C)$ with one column for each possible capacity C and one row for each item i. Table 12.1 gives an example. There are many ways to fill this table, for example row by row. In order to reconstruct a solution from this table, we work our way backwards, starting 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 continue to row $i-1$ and column $C-w_i$. We continue with this procedure until we arrive at row 0, by which time the solution $(x_1,\ldots,x_n)$ has been completed.

Exercise 12.11. Dynamic programming, as described above, needs to store a table containing $\Theta(nM)$ integers. Give a more space-efficient solution that stores only a single bit in each table entry except for two rows of $P(i,C)$ values at any given time. What information is stored in this bit? How is it used to reconstruct a solution? How can you get down to *one* row of stored values? Hint: exploit your freedom 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 $M = 5$. **Bold-face** entries contribute to the optimal solution

$i \backslash C$	0	1	2	3	4	5
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**

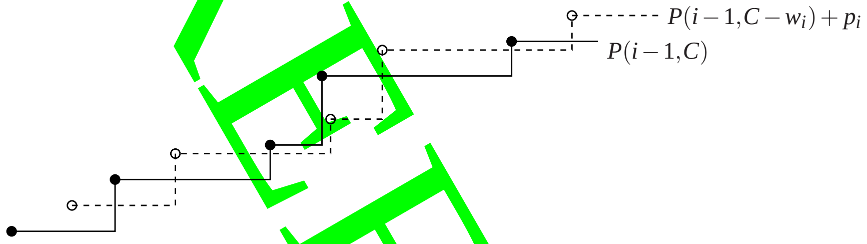


Fig. 12.4. The solid step function shows $C \mapsto P(i-1,C)$, and the dashed step function shows $C \mapsto P(i-1,C-w_i)+p_i$. $P(i,C)$ is the pointwise maximum of the two functions. The solid step function is stored as the sequence of solid points. The representation of the dashed step function is 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* [146, 17] 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 C value. 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 $P(i,C) = p_1$ 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 $|L_{i-1}|$. Conclude that the running time of the algorithm is proportional to the number of Pareto-optimal solutions.

The basic dynamic-programming algorithm for the knapsack problem and also its optimization 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 $O(n)$. 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 [17] for details.

Exercise 12.13 (dynamic programming by profit). Define $W(i,P)$ to be the smallest weight needed to achieve a profit of at least P using knapsack items $1..i$.

(a) Show that $W(i,P) = \min\{W(i-1,P), W(i-1,P-p_i)+w_i\}$.
(b) Develop a table-based dynamic-programming algorithm using the above recurrence that computes optimal solutions to the knapsack problem in time $\mathrm{O}(np^*)$, where p^* is the profit of the optimal solution. Hint: assume first that p^*, or at least a good upper bound for it, is known. Then remove 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 the dollar zone with coins worth 1, 5, 10, 25, 50, and 100 cents.
(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 $M_1M_2\cdots M_n$, 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 of matrix products to minimize the number of element multiplications needed. Use dynamic programming to find an optimal evaluation order in time $\mathrm{O}(n^3)$. 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 (minimum edit distance). The *minimum edit distance* (or *Levenshtein distance*) $L(s,t)$ between two strings s and t is the minimum number of character deletions, insertions, and replacements applied to s that produces the string t. For example, $L(\mathtt{graph},\mathtt{group})=3$ (delete h, replace a by o, insert u before p). Define $d(i,j)=L(\langle s_1,\dots,s_i\rangle,\langle t_1,\dots,t_j\rangle)$. Show that

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

where $[s_i \neq t_j]$ is one if s_i and t_j are different and is zero otherwise.

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 $\langle s', t' \rangle$ of optimal solutions are *not* necessarily shortest paths from s' to t'.

12.4 Systematic Search – When in Doubt, Use Brute Force

In many optimization problems, the universe $\mathcal{U}$ of possible solutions is finite, so that we can in principle solve the optimization problem by trying all possibilities. Naive application of this idea does not lead very far, however, 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 12.20 and 12.21, we outline systematic-search routines following a somewhat different pattern.

Figure 12.5 gives pseudocode for a systematic-search routine *bbKnapsack* for the knapsack problem and Figure 12.6 shows a sample run. *Branching* is the most fundamental ingredient of systematic-search routines. All sensible values for some piece of the solution are tried. For each of these values, the resulting problem is solved recursively. Within the recursive call, the chosen value is fixed. The routine *bbKnapsack* first tries including an item by setting $x_i := 1$, and then excluding it by setting $x_i := 0$. The variables are fixed one after another in order of decreasing profit density. The assignment $x_i := 1$ is not tried if this would exceed the remaining knapsack capacity M'. With these definitions, after all variables have been set, in the n-th level of recursion, *bbKnapsack* will have found a feasible solution. Indeed, without the bounding rule below, the algorithm would systematically explore *all* possible solutions and the *first* feasible solution encountered would be the solution found by the algorithm *greedy*. The (partial) solutions explored by the algorithm form a tree. Branching happens at internal nodes of this tree.

Bounding is a method for pruning subtrees that cannot contain optimal solutions. A branch-and-bound algorithm keeps the best feasible solution found in a global variable $\hat{x}$; this solution is often called the *incumbent* solution. It is initialized to a solution determined by a heuristic routine and, at all times, provides a lower bound $p \cdot \hat{x}$ on the value of the objective function that can be obtained. This lower bound is complemented by an upper bound u for the value of the objective function obtainable by extending the current partial solution x to a full feasible solution. In our example,

```
Function bbKnapsack($(p_1,\dots,p_n),(w_1,\dots,w_n),M)$ : $\mathcal{L}$
    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($(p_1,\dots,p_n),(w_1,\dots,w_n),M$) : $\mathcal{L}$     // best solution so far
    $x : \mathcal{L}$                                              // current partial solution
    recurse$(1, M, 0)$
    return $\hat{x}$

    // Find solutions assuming $x_1,\dots,x_{i-1}$ are fixed, $M' = M - \sum_{k<i} x_i w_i$, $P = \sum_{k<i} x_i p_i$.
    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                                // not bounded
            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

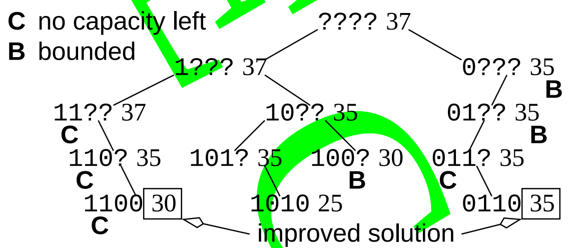


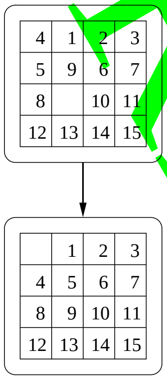
Fig. 12.6. The search space explored by *knapsackBB* for a knapsack instance with $p = (10,20,15,20)$, $w = (1,3,2,4)$, and $M = 5$, and an empty initial solution $\hat{x} = (0,0,0,0)$. The function *upperBound* is computed by rounding down the optimal value of the objective function for the fractional knapsack problem. The nodes of the search tree contain $x_1 \cdots x_{i-1}$ and the upper bound u. Left children are explored first and correspond to setting $x_i := 1$. There are two reasons for not exploring a child: either there is not enough capacity left to include an element (indicated by C), or a feasible solution with a profit equal to the upper bound is already known (indicated by B)

the upper bound could be the profit for the fractional knapsack problem with items $i..n$ and capacity $M' = M - \sum_{j<i} x_i w_i$.

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 an improved solution in the current subtree of the search space. We test $u > p\cdot\hat{x}$ again before exploring the case $x_i = 0$ because $\hat{x}$ might change when the case $x_i = 1$ is explored.

Exercise 12.19. 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 $\sum_{k \le i} w_i$ and $\sum_{k \le i} p_i$ and use binary search.

Exercise 12.20 (the 15-puzzle). The 15-puzzle is a popular sliding-block puzzle. You have to move 15 square tiles in a 4×4 frame into the right order. Define a move as the action of interchanging a square and the hole in the array of tiles.



Design an algorithm that finds a shortest-move sequence from a given starting configuration to the ordered configuration shown at the bottom of the figure on the left. Use *iterative deepening depth-first search* [114]: try all one-move sequences first, then all two-move sequences, and so on. This should work for the simpler 8-puzzle. 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 be moved freely 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.

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 $1 \le i < j \le 8$. What do these conditions express? Show that they are sufficient. A systematic search can use 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*.

12.4.1 Solving Integer Linear Programs

In Sect. 12.1.1, we have seen how to formulate the knapsack problem as a 0–1 integer linear program. We shall now indicate how the branch-and-bound procedure developed for the knapsack problem can be applied to any 0–1 integer linear program. Recall that in a 0–1 integer linear program 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 [147, 172] 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 describe next 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.

12.5 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 12.7 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 from the neighborhood is chosen. Finally, some heuristic decides when to 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.

Our first example is the traveling salesman problem described in Sect. 11.6.2. 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. 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 12.22. 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

```
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. 12.7. Local search

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 12.8 shows an example 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 maximal objective value.* In the examples in Figs. 12.2 and 12.8, 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 on 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

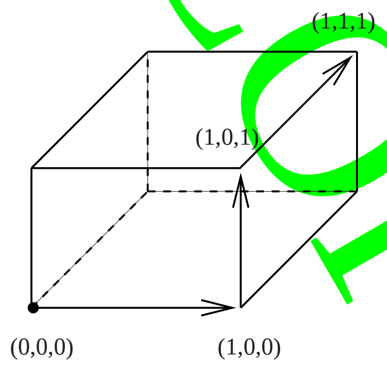


Fig. 12.8. The three-dimensional unit cube is defined by the inequalities $x \geq 0$, $x \leq 1$, $y \geq 0$, $y \leq 1$, $z \geq 0$, and $z \leq 1$. At the vertices $(1,1,1)$ and $(1,0,1)$, three inequalities are tight, and on the edge connecting these vertices, the inequalities $x \leq 1$ and $z \leq 1$ are tight. For the objective "maximize $x+y+z$", the simplex algorithm starting at $(0,0,0)$ may move along the path indicated by arrows. The vertex $(1,1,1)$ is optimal, since the half-space $x+y+z \leq 3$ contains the entire feasible region and has $(1,1,1)$ in its boundary

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 first 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 make many downhill steps so that we can escape from wide local optima. On the other hand, we must be sufficiently target-oriented so that we find a global optimum at the end of a long narrow ridge. A very popular and successful approach for reconciling these contradictory goals is *simulated annealing*; see Fig. 12.9. 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 one if x' improves upon x. It is less than one 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 zero, we make such a move relatively unlikely. The hope is that, in this way, the process zeros 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.

```
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. 12.9. Simulated annealing

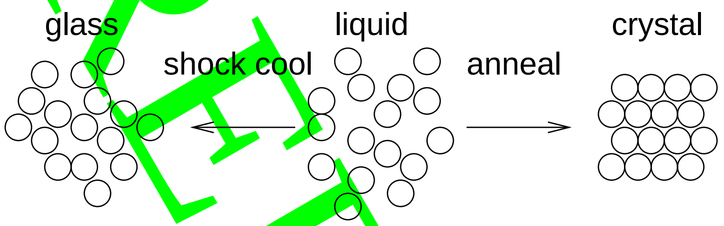


Fig. 12.10. Annealing versus shock cooling

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.

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[6] the global energy of a physical system. For example, consider a pot of molten silica (SiO_2); see Fig. 12.10. 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 minimal 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

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

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

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

where T is the temperature of the system and $\mathscr{L}$ is the set of states of the system. This energy distribution is called the *Boltzmann distribution*. When T decreases, the probability of states with a minimal energy grows. Actually, in the limit $T \to 0$, the probability of states with a minimal energy approaches one.

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

The physical analogy gives some idea of why simulated annealing might work,[7] 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 do that for infinitely many temperatures. Simulated-annealing algorithms therefore 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 so 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 one. 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 one. 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 prematurely forced 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

[7] Note that we have written “might work” and not “works”.

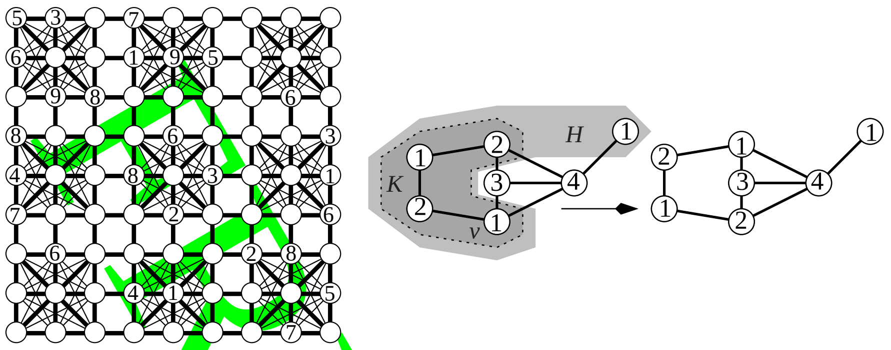


Fig. 12.11. The figure on the *left* shows a partial coloring of the graph underlying sudoku puzzles. The **bold** straight-line segments indicate cliques consisting of all nodes touched by the line. The figure on the *right* shows a step of Kempe chain annealing using colors 1 and 2 and a node v

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 12.23. 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.

Graph Coloring

We shall now exemplify simulated annealing on the *graph-coloring problem* already mentioned in Sect. 2.10. 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 these countries have a common border, and thus these 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 [162]. 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. 12.11 with the digits 1..9. We shall present two simulated-annealing approaches to graph coloring; many more have been tried.

Kempe Chain Annealing

Of course, 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 12.11 gives an example. Kempe chain annealing starts with any feasible coloring.

***Exercise 12.24.** 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., $x(v) = j$. 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 i-th 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 12.25. 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 minimal number of colors.

Experimental Results

Johnson et al. [101] performed a detailed study of algorithms for graph coloring, with particular emphasis on simulated annealing. We shall briefly 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 for distinct edges are independent. 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 1 000 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 12.12

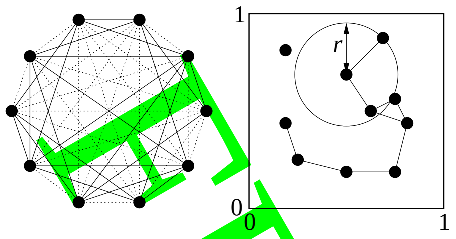


Fig. 12.12. *Left*: a random graph with 10 nodes and $p = 0.5$. The edges chosen are drawn solid, and the edges rejected are drawn dashed. *Right*: a random geometric graph with 10 nodes and range $r = 0.27$

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, far away 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 Kosciusko (altitude 2229 m), a solution far from optimum. 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. Several independent runs are also an easy source of parallelism: just run the program on several different workstations concurrently.

12.6 Evolutionary Algorithms

Living beings are ingeniously adaptive to their environment, and master the problems encountered in daily life with great ease. Can we somehow use the principles of life for developing 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.

```
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 $x' = mutate(x^1)$
    $population := population \cup \{x'\}$
    $population := \{x \in population : x$ is sufficiently fit$\}$
```

Fig. 12.13. A generic evolutionary algorithm

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. 12.13. 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.

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 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 12.14 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

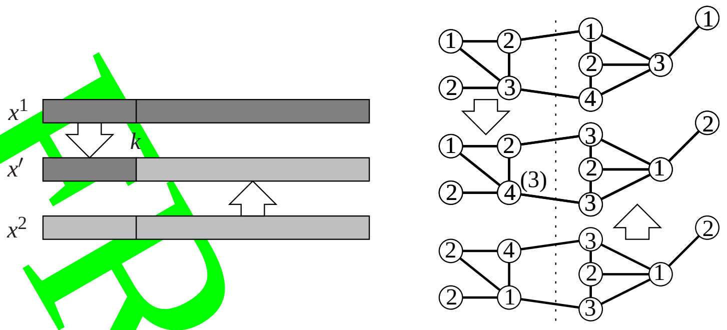


Fig. 12.14. Mating using crossover (*left*) and by stitching together pieces of a graph coloring (*right*)

now connect nodes with the same color. This could be repaired using some heuristics, for example choosing the smallest legal color for miscolored nodes in the part corresponding to the first parent. Figure 12.14 gives an example.

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

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.

12.7 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 even 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 blackbox 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 on “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 on “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 [45]. Although this algorithm works well in practice, some of its variants take exponential time in the worst case. It is a famous 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 [184]. 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 [110] and Karmakar [106] found polynomial-time algorithms for linear programming. There are many good textbooks on linear programming (e.g. [23, 58, 73, 147, 172, 199]).

Another interesting blackbox solver is *constraint programming* [90, 121]. 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.