Roberto Battiti · Mauro Brunato

The LION Way

Machine Learning *plus* Intelligent Optimization

Version 3.0 — December 2017

intelligent-optimization.org/LIONbook

second case one moves freely in input space looking for locally optimal points.

This holds also for SSL. For example the work in [53] shows how to combine constraints and metric learning into semi-supervised clustering.

Constraint-based clustering approaches start from pairwise *must-link* or *cannot-link* constraints (requests that two points do or do not belong to the same cluster) and insert into the objective function to be minimized a penalty for violating constraints. By the way, constraints can be derived from the labels but also from other sources of information. For example, the Euclidean K-means algorithm partitions the points into k sets so that the function:

$$\sum_i \|\mathbf{x}_i - \boldsymbol{\mu}_{l_i}\|^2$$

is locally minimized. In it, the vector $\boldsymbol{\mu}_{l_i}$ is the winning centroid associated with point $\mathbf{x}_i$, the one minimizing the distance.

If two sets of must-link pairs $\mathcal{M}$ and cannot-link pairs $\mathcal{C}$ are available, one can encourage a placement of the centroids in order to satisfy the constraints by adding a penalty w_{ij} for a single violation of a constraint in $\mathcal{M}$ and a penalty $\overline{w}_{ij}$ for a single violation of a constraint in $\mathcal{C}$, obtaining the following function to be minimized ("**pairwise constrained K-means**"):

$$E_{\text{pckmeans}} = \sum_i \|\mathbf{x}_i - \mu_{l_i}\|^2 + \sum_{(\mathbf{x}_i,\mathbf{x}_j)\in\mathcal{M} \text{ and } l_i\neq l_j} w_{ij} + \sum_{(\mathbf{x}_i,\mathbf{x}_j)\in\mathcal{C} \text{ and } l_i=l_j} \overline{w}_{ij}. \tag{23.8}$$

Pairwise constraints can also be used for **metric learning**. If the metric is parametrized with a symmetric positive-definite matrix $\mathbf{A}$ as follows,

$$\|\mathbf{x}_i - \mathbf{x}_j\|_{\mathbf{A}} = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^T \mathbf{A} (\mathbf{x}_i - \mathbf{x}_j)},$$

the problem amounts to determining appropriate values of the matrix coefficients. If the matrix is diagonal, the problem becomes that of *weighing* the different features.

The constraints represent the user's view of similarities: they can be used to change the metric to reflect this view, by minimizing the distance between must-link instances and at the same time maximizing the distance between cannot-link instances. After the metric is modified, one can use a traditional clustering algorithm like K-means.

Asking for a single metric for the entire space can be inappropriate and a different metric $\mathbf{A}_h$ can be used for each k-means cluster h. The MPCK-MEANS algorithm in [53] uses the method of Expectation-Maximization, see also Section 17.3, and alternates between cluster assignment in the E-step, and centroid estimation and metric learning in the M-step.

Constraints are used during cluster initialization and when assigning points to clusters. The distance metric is adapted by re-estimating $\mathbf{A}_h$ during each iteration based on the current cluster assignment and constraint violations. An interesting paper dealing with metric learning for text documents is [248]. More details about semi-supervised learning are present in [79] and [391].

Gist

In many cases labeled examples are scarce and costly to obtain, while tons of unlabeled cases are available, usually sleeping in business databases or in the web.

Semi-supervised learning schemes use *both* the available labeled examples *and* the unlabeled ones to improve the overall classification accuracy.

The distribution of all examples can be used to encourage ML classification schemes to create boundaries between classes passing through **low-density areas** (transductive SVM).

If the problem is modeled as a **graph** (entities and relationships labeled with distances) smoothing operations on graphs can be used to transfer the information of some labeled nodes to the neighboring nodes (graph Laplacian).

The distribution of examples can be used to **learn a metric**, a crucial component to proceed with supervised learning.

A space alien arriving on Earth could combine the zillions of information bits in web pages plus some labeled information obtained by a human pen pal (or by a Yahoo-like directory) for an intensive course to understand human civilization before conquering us. Terrestrial businesses use similar techniques to mine data and conquer more customers.

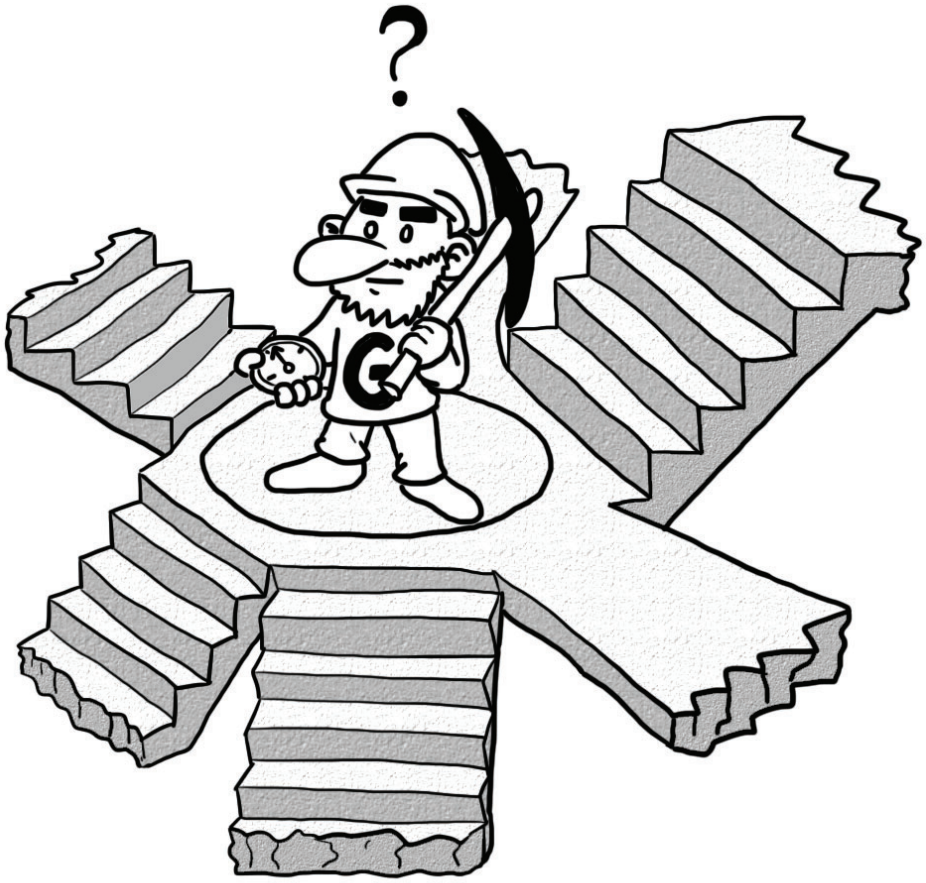
Part III

Optimization: basics

Chapter 24

Greedy and Local Search

Ognuno ha sulle proprie spalle la responsabilità delle proprie scelte. È un bel peso.
Everybody carries on his shoulders the responsibility of his choices. It is a heavy weight.
(Romano Battiti)



Many issues in everyday's life are related to solving optimization problems, to improve solutions, or to find solutions which are so good that no other solution is better (called "global optima").

For example, a salesman is given a list of cities and their mutual distances, and wants to find the *shortest possible tour* that visits each of them. This is called the **Traveling Salesman Problem** (TSP) and its relevance is obvious, to reduce travel costs and carbon dioxide emissions. TSP was first formulated in 1930 and it still is an extremely difficult problem, one of the most intensively studied in optimization. An **instance** of a problem is a specific case to be solved. In TSP a possible instance is to find a tour visiting Trento, Bologna, Napoli, Milano and Manarola. Even though finding optimal solutions for large TSP instances is computationally difficult, in most cases practically impossible, a large number of **heuristics** are known, so that even instances with tens of thousands of cities can be effectively solved in practice. Heuristics are algorithms without formal proof of convergence or guarantees of approximation but often effective in practice.

In abstract and general terms, in optimization one is given a function f defined on a set of possible input values $\mathcal{X}$. In TSP, f is the tour length, and $\mathcal{X}$ is the set of all possible tours visiting the cities, i.e., of their permutations. The function $f(\mathcal{X})$ to be optimized is called with more poetic names in some communities: *fitness* function, *goodness* function, *objective* function. If $\mathcal{X}$ is defined by a discrete set of possibilities (like binary values, permutations, integers) one speaks about **discrete optimization**. On the contrary, **continuous optimization** considers real-valued inputs.

One aims at finding the input configuration leading to the least possible value of the function f. Often a set of **constraints** on $\mathcal{X}$ have to be satisfied for a solution to be considered **admissible**. If one wants to pick quantities for possible foods in order to minimize the daily cost of a diet, useful constraints are to be placed on the minimum amount of calories and of vitamins. The solution corresponding to fasting costs zero but will lead to starvation, and is thefore not admissible.

This chapter presents the basic building blocks of **greedy and local search**, the more advanced Reactive Search Optimization (RSO) will be presented in Chapter 27. To avoid confusion, let's note that the term "local search" has nothing to do with the search for information or web pages, like in Google or similar services. To avoid confusions, remember to make it clear if you speak to normal people, who may think about local search of restaurants in the neighborhood of their current GPS position. In optimization one searches here for actions, decisions, effective innovations, aiming at improving solutions to problems, optimal solutions when possible, or at least approximations thereof.

Our optimization part starts with a chapter about greedy and local search for discrete optimization for many reasons. First, most real-world problems have to do with **choices among a discrete set of alternatives**. If one uses a computer even real numbers are actually approximated by a representation with a small number of bits. Understanding the main methods is much simpler with discrete local search than with methods based on real variables and derivatives.

Second, improving situations by a sequence of small greedy and local steps is deeply rooted in our human nature, as reflected in many quotes and proverbs across cultures: "little things make big things happen," "it does not matter how slowly you go so long as you do not stop" (Confucius), "the drop carves the stone" (Latin), "bean by bean the sack gets full" (Greek)... In passing, let's note that everybody's life can be seen as a running optimization algorithm: most of the changes are localized, dramatic changes do happen, but not so frequently. Imagine that you just found a partner, a possible companion for your life. Local changes are frequent in the initial part of a relationship. For example, you may convince your partner to dress in a better way, to avoid eating garlic, or to change opinions about various issues. You may have to stand worsening changes, like having your partner watching football games in the weekend, to eventually obtain improvements, in the form of a more relaxed partner. Or you may finally decide that small changes are not sufficient and that the way out is a drastic *diversification* or *restart* (finding a better partner).

Although the terminology can vary among different authors, there is a deep **connection between the greedy and local aspects**. In both cases the focus is limited to **profiting from a current tentative solution**, in a **short-sighted** manner.

In some cases one searches in the neighborhood of a complete admissible solution by small perturbations (a.k.a. **perturbative local search**), in other cases one gradually modifies and extends a partial solution (**constructive greedy search**).

The examples in this chapter are mostly for discrete optimization problems because of their simplicity, but we will encounter similar local search principles also for continuous optimization in Chapter 25.

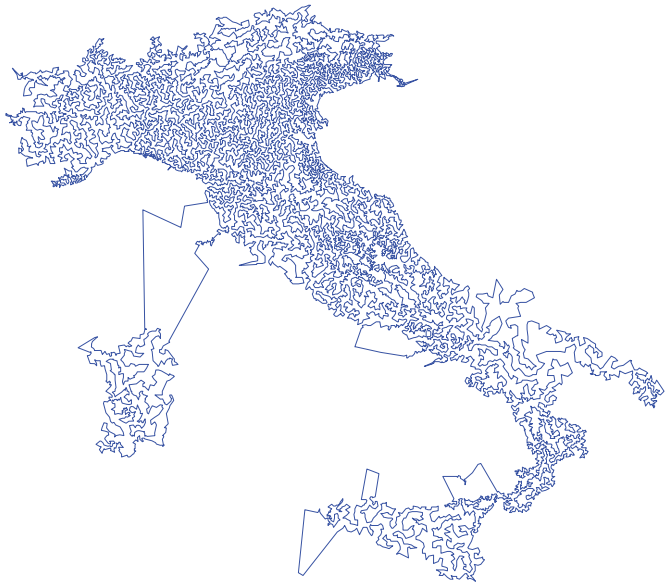


Figure 24.1: The Traveling Salesman Problem: the shortest tour of 16862 cities in Italy.

24.0.1 Case study: the Traveling Salesman Problem

The **Traveling Salesman Problem**, is paradigmatic and very relevant for both practical applications and theoretical reasons. The practical application can be read out from the name: imagine a salesman who has to visit a certain number of customers in different cities, and wants to minimize the fuel consumption, or the time dedicated to travel. Variations with more constraints and complexity consider multiple travelers, trucks with different capacities, time requirements for pickup and delivery of goods, etc. (vehicle routing, pickup and delivery with time windows [122]).

An illustration is in Fig. 24.1: a tourist wants to tour all Italian cities while minimizing time, or fuel consumption in case he travels with an SUV.

In an instance of TSP, one is given a set of n cities and the matrix of distances $d_{ij} > 0$ between couples of cities. A *tour* is a closed path visiting every city exactly once. The problem is to find a tour with minimum length. We can represent a tour with a cyclic permutation π of $\{1, ..., n\}$, where $\pi(i)$ is the city visited in the i-th place.

One can easily generalize the problem to any graph (V, E) not necessarily related to geography. The **decision version** (with "yes" or "no" answer) of the generalized TSP is: given a complete graph $(V, V \times V)$, a "cost" function $c : V \times V \to \mathbb{Z}$, and a maximum cost $k \in \mathbb{Z}$, is there a tour with total cost at most k? The requirement of complete connectivity means that an admissible tour can be found immediately, just generate a random permutation. If the graph is not complete, even determining if there is a tour, also called Hamiltonian path, is NP-complete (extremely unlikely to be solved in reasonable CPU times).

A fast algorithms for TSP is unlikely to exist, in fact TSP also belongs to the dreadful family of NP-complete problems. Of course, if one presents us with a solution we can easily check that the distance is k in polynomial time (actually this is the "meaning" of NP).

A brute-force **exhaustive search** algorithm is obvious: generate all possible tours and calculate the corresponding distances. Unfortunately, we need to examine $(n-1)! = (n-1) \times (n-2) \times (n-3)... \times 1$ permutations (not $n!$

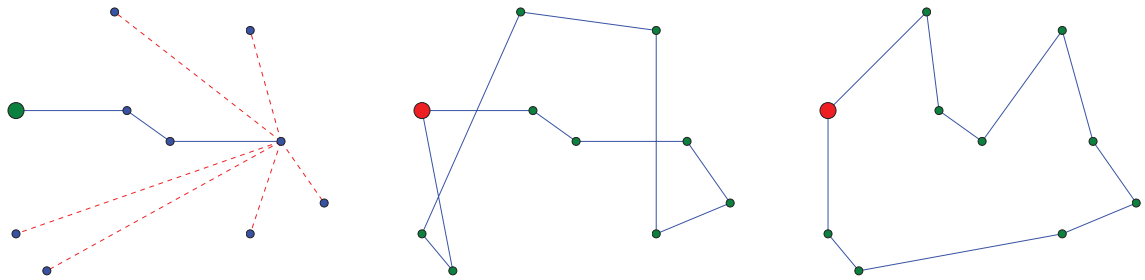


Figure 24.2: A 10 city TSP instance. Left: greedy construction after three steps from the initial city (larger dot); the algorithm will now choose the shortest among the dashed edges; center: the completed greedy solution; right: the optimal solution.

because we can change the starting point of the tour and still get the same tour). As you know, the factorial is growing *very* rapidly. If you do not know, calculate 3000! to get a couple of pages full of digits.

There is a lesson here: brute-force algorithms can solve small instances (10 cities are OK) and make a nice demo, but going to the real application will rapidly lead to unacceptable CPU times.

As a small digression, the TSP is an interesting problem for which **special versions** have pleasant properties. In particular, if the cost function satisfies the **triangle inequality** $c(u, v) \leq c(u, w) + c(w, v)$ (moving directly from u to v is always cheaper than passing through an intermediate point w), a **polynomial-time approximation algorithm** exists at less than twice the optimal cost. Without triangle inequality a ρ-approximated algorithm cannot be found unless $P = NP$. Real-world salesmen are lucky, because the Euclidean distance, as well as distances on the Earth surface, satisfies the triangle inequality. **Approximation algorithms with guaranteed approximation ratios** are a growing area of research.

24.1 Greedy constructions

In greedy algorithms "better an egg today than a hen tomorrow".

In some cases a solution can be built through a sequence of decisions, each step involving a small part of a solution. In the Traveling Salesman Problem a tour can be built by joining together partial tour segments. A **greedy algorithm** always makes the choice that looks best at the moment. For example, a greedy strategy for the TSP starts from a random initial city, and moves to the nearest city that hasn't been visited yet. This greedy choice is repeated until all cities have been visited (Fig. 24.2).

A greedy algorithm is therefore **shortsighted**: it can make commitments to certain choices too early, which prevent it from finding the best overall solution later. A choice cannot be undone during future steps. An example is shown in Fig. 24.2: ten cities are in such positions that a short-sighted salesman takes a much longer tour than the optimal one.

We will later explore other constructive strategies such as dynamic programming (see Sec. 34.2), where a solution is built by combining optimal solutions to subproblems. The simpler greedy algorithms, instead, skip the exhaustive examination of possibilities to select only the (locally) most appealing combination.

24.1.1 Greedy algorithms for minimum spanning trees

Greedy algorithms rarely find optimal solutions to problems, but there are happy exceptions.

A general way to prove the optimality of a greedy algorithm is as follows:

1. Formulate the problem in an *inductive* way: one makes a choice, and is left with a subproblem to solve.

2. Prove that there is always an optimal solution that *starts with the greedy choice*. In other words, prove that at least *a* globally optimal solution can be reached be completing the current partial solution given by greedy choice (at least one optimal solution is preserved).

3. Demonstrate that, after the greedy choice, one is left with a subproblem so that combining the optimal solution of the subproblem with the greedy choice, one arrives at an optimal solution of the original problem.

Examples of greedy algorithms are Kruskal's and Prim's algorithms for finding *minimum spanning trees* in graphs, and the algorithm for finding optimum Huffman trees for compressed codes. We also encountered greedy algorithms in this book when dealing with trees (Chapter 6).

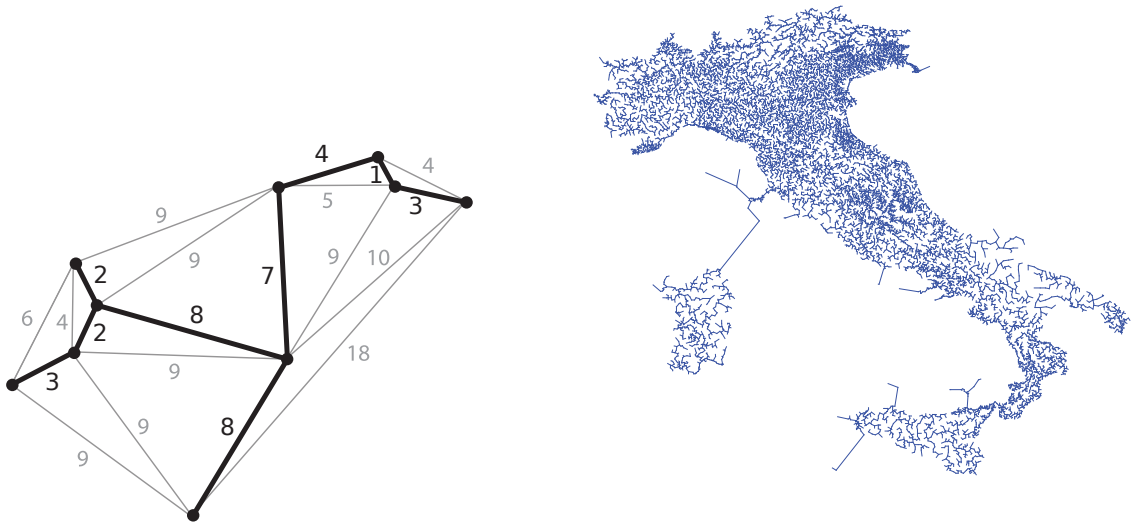


Figure 24.3: Minimum spanning trees. Left: a graph with varying edge costs; right: 16862 Italian cities.

Let's consider the **minimum spanning tree (MST) problem** [102]. The objective is to build a tree connecting all vertices of a connected, undirected graph, with the minimal total cost of the selected edges. Note that some edges can be missing, although there must be a way to pass from two arbitrary nodes by following selected edges (this is the meaning of connectedness).

For a concrete application, consider a company in telecommunications or electricity distribution in need to wire a new area. The company has to serve a number of existing or planned locations (houses, factories: the vertices of the graph) and to follow existing infrastructures (roads, tunnels, sewers: the edges of the graph), each associated to a different cost due to their length and to other features (need to open trenches in roads, put poles). Costs depend on many factors other than distance, so a direct connection between two nodes might be more expensive than a detour through multiple intermediate nodes, and the triangle inequality no longer applies. The minimum-cost connection of all nodes won't contain any cycles (otherwise we could spare one connection), therefore it is a tree. The practical "meaning" of a tree (useful to fix ideas and remember abstract concepts) is therefore: "connect all nodes without waste." Cycles are a waste of resources because they imply at least two ways to reach a node, going in the two directions of the cycle. Trees are born to be greedy.

The main difference between MST and TSP is precisely that one does not look for a tour, but for a tree. This "small" differences make MST a very friendly problem, which can be solved to optimality in polynomial CPU time.

For the abstract formulation of the **minimum spanning tree** problem, let $G = (V, E)$ be an undirected graph, V being the set of vertices, E the set of edges (connections between vertices). For each edge $(u, v) \in E$, the cost $c(u, v)$

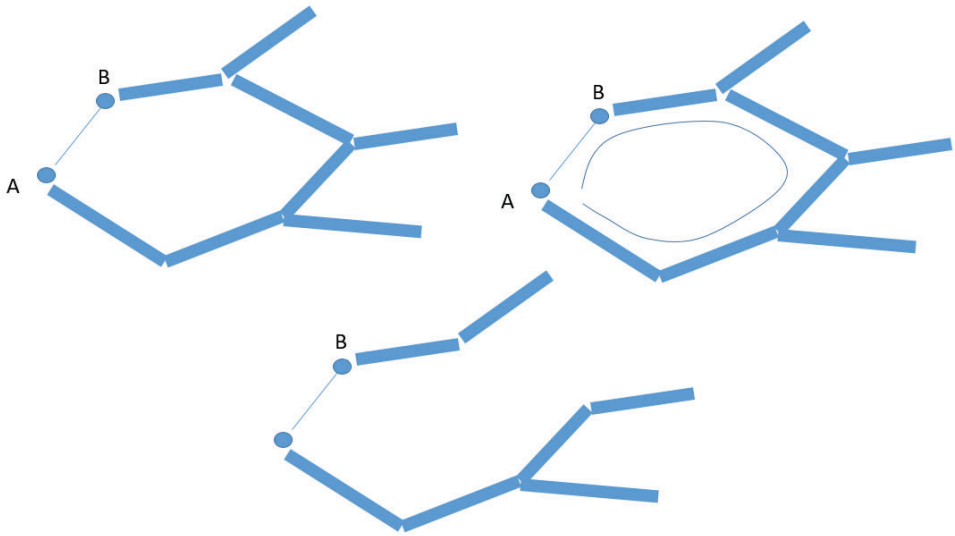


Figure 24.4: Adding the lightest edge is safe for MST. If an optimal solution does not contain it, we can do a substitution. Add the lightest edge to the optimal tree, and obtain a *cycle*, because A and B are already connected in an optimal tree. Then cut an edge of the cycle different from our lightest edge (with cost equal or larger!) and get a tree with the same or lower cost. Actually the cost cannot be lower because our tree was optimal but it can have the same cost (one can have more optimal solutions).

to connect u and v is given. One aims at finding a subset $T \subseteq E$ that connects all vertices without cycles and with the minimum possible total cost:

$$c(T) = \sum_{(u,v)\in T} c(u,v)$$

Let's concentrate on a core spanning-tree algorithm. The starting idea of the greedy construction is to grow the tree by adding one edge at a time to a set of edges A. Let's build some intuition. One is already thinking "greedily" and is therefore tempted to start from the least costly edge (the lightest edge). Is it safe to add it to the initial tree? Or can one miss an optimal solution? The fact that it is safe is demonstrated in Fig. 24.4 with the "substitution" argument.

In the analysis of algorithms one often considers **invariants** (logical assertions that are always true during the execution) and **progress properties** (changing in time).

A useful invariant here is:

Prior to each iteration, A is a subset of some minimum spanning tree.

If we manage to keep the invariant and increase the size of A (progress) we are done.

At a certain step, a **safe edge** for A is one which can be added to A while maintaining the invariant.

The generic algorithm is therefore as follows:

In the beginning, when A is the empty set, the invariant is true. We also demonstrated that the property holds if the first edge in A is one of the lightest edges. If the invariant holds at the beginning of a new iteration, and the tree is not complete, at least a safe edge must exist (picture in your mind the complete minimum spanning tree of which A is, by definition, a subset). Let's see how we can find it.

A **cut** $(S, V - S)$ of a graph is a partition of its vertices (imagine cutting some edges to separate two parts of the graph); an edge **crosses** a cut if its two endpoints belong to different partitions. A cut is said to respect an edge subset A if none of A's edges crosses it. An edge crossing the cut is **light** if its cost is minimum among all crossing edges.

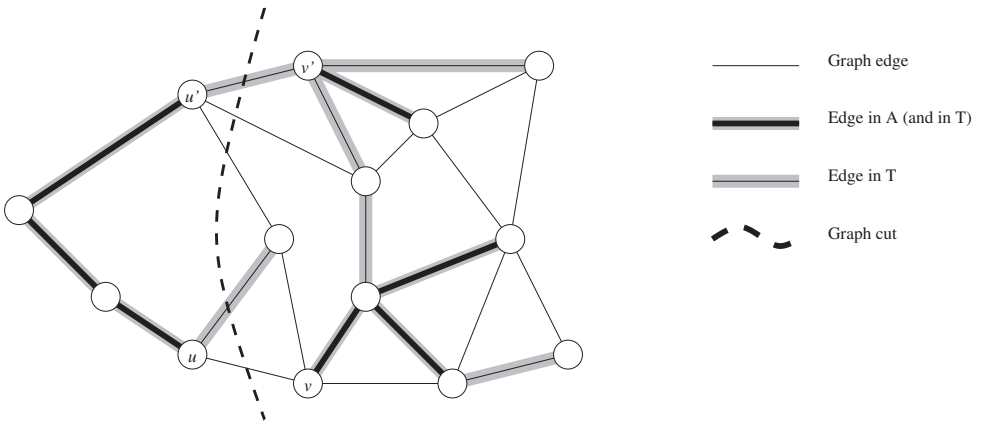


Figure 24.5: Minimum spanning tree construction. Dark thick edges are the partial MST A, gray edges are a complete MST T. The cut respects A. If (u, v) is light in the cut, then we can take any MST T and, if it doesn't contain (u, v), find an edge $(u', v') \in T$ with equal cost and replace it.

Now, let A be the current subset of E included in some minimum spanning tree, and let $(S, V - S)$ be any cut of G that respects A. A light edge (u, v) crossing the cut is safe for A. Suppose in fact that a minimum spanning tree T contains A, but not the light edge (u, v), as in Fig. 24.5. Then $T \cup \{(u, v)\}$ contains a cycle, and at least one other edge (u', v') in the cycle crosses the cut. Let $T' = T \cup \{(u, v)\} \setminus \{(u', v')\}$. Then T' is also a tree (we just cut the cycle in a different place), and since T is minimum and (u, v) is light, then (u', v') is light too (otherwise T' would have a total cost less than T). Therefore T' is a minimum spanning tree, A is a subset of T', and (u, v) is safe for A because it also belongs to T'.

This simple fact allows us to greedily find an edge to extend a partial MST to completion: just **cut the graph without crossing any edge of the current solution, and take one of the lowest-cost crossing edges**. The greedy property is apparent if you imagine that the algorithms has to pay money corresponding to the cost of the new edge to add at each iteration.

To summarize, we demonstrated that a greedy action maintains the invariant of producing a subset of an optimal spanning tree. Because the tree size is limited, the algorithm will converge producing an optimal solution.

Different MST greedy algorithms make different choices as to the cut to consider:

- Prim's algorithm maintains A as a connected tree whose vertices define the cut S. Every time, therefore, a least-cost edge is added going from the current tree to a new node, which will be added to the cut S for the next iteration.
- Kruskal's algorithm maintains A as a forest of partial trees (initially all disconnected nodes in V) and repeatedly adds a least-cost edge between two different trees.

Both algorithms can be made to run in $O(E \log V)$ time by using *binary heap* data structures [102]. With *Fibonacci heaps*, Prim's algorithms runs in time $O(E + V \log V)$ an improvement for dense graphs with $|E|$ much larger than $|V|$.

24.2 Local search based on perturbations

Aside from a small family of lucky problems, the greedy construction approach often leads to suboptimal solutions. Indeed, many problems have been shown to possess "pathological" instances for which a greedy algorithm produces a very bad solution, although in many cases better than a random solution.

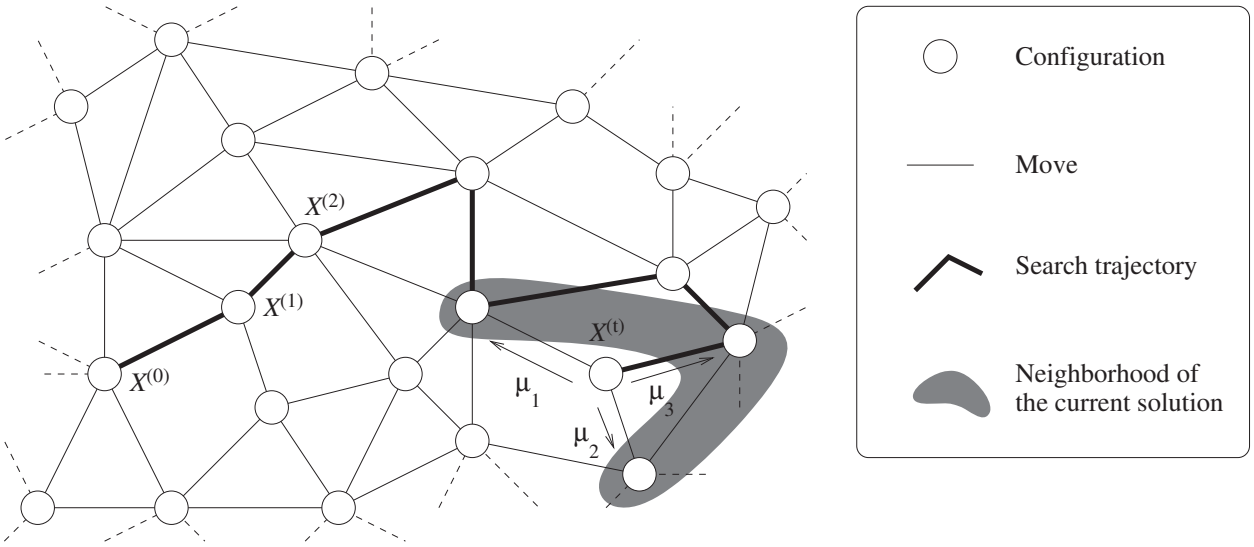


Figure 24.6: The local search heuristics in action.

This Section discusses what to do next. Suppose that you have an initial, suboptimal solution to a problem, be it obtained by a greedy algorithm or by any other means (e.g., randomly). A basic problem-solving strategy consists of starting from such initial tentative solution and trying to improve it through **repeated local (small) changes**. At each repetition, the current configuration is slightly modified (*perturbed*) and the function to be optimized is calculated. The change is kept if the new solution is better, otherwise another change is tried.

Let's define the notation. $\mathcal{X}$ is the search space, i.e., the set of all solutions, also known as "configurations," to a given problem instance; for example, in TSP $\mathcal{X}$ is the set of all possible city permutations. Given a configuration $X \in \mathcal{X}$, we can apply to it set of "perturbations" to transform it into other configurations; let us call such perturbations, or "moves", $\mu_0, \ldots, \mu_M$. Since we are considering an iterative process, let us call the initial configuration $X^{(0)}$, while the configuration at step ("time") t will be $X^{(t)}$. Every configuration after the first is generated by perturbing the previous one. The **neighborhood** $N(X^{(t)})$ of point $X^{(t)}$ is the set of configurations that can be obtained by perturbing the current one:

$$N(X^{(t)}) = \{\mu_i(X^{(t)}), i = 0, \ldots, M\}.$$

If the search space is given by binary strings with a given length L: $\mathcal{X} = \{0,1\}^L$, the moves can be those changing (or complementing or *flipping*) the individual bits, and therefore M is equal to the string length L.

We can abstractly represent the search space as a graph as in Fig. 24.6: configurations are nodes in the graph, while moves between configurations are represented by edges. The *search trajectory* $(X^{(0)}, X^{(1)}, X^{(2)}, \ldots, X^{(t)})$ is a path in this graph. The neighborhood $N(X^{(t)})$ is the set of first neighbors of $X^{(t)}$ within the graph. In the example, the current configuration can be perturbed in three different ways (μ_1, μ_2 and μ_3); two perturbations, μ_1 and μ_3, lead to previously visited configurations, while μ_3 generates a new one.

Local search starts from an admissible configuration $X^{(0)}$ and builds a **trajectory** $X^{(0)}, \ldots, X^{(t+1)}$. The successor of the current point is a point in the neighborhood with a lower value of the function f to be minimized:

$$Y \leftarrow \text{IMPROVING-NEIGHBOR}(\ N(X^{(t)})\ ) \tag{24.1}$$

$$X^{(t+1)} = \begin{cases} Y & \text{if } f(Y) < f(X^{(t)}) \\ X^{(t)} & \text{otherwise (search stops).} \end{cases} \tag{24.2}$$

IMPROVING-NEIGHBOR returns an improving element in the neighborhood. In a simple case this is the element with the lowest f value, but other possibilities exist, for example the *first improving* neighbor encountered.

If no neighbor has a better f value, i.e., if the configuration is a **local minimizer**, the search stops. Let's note that maximization of a function f is the same problem as minimization of $-f$. Like all symmetric situations, this fact can create some confusion of terminology. For example, *steepest descent* assumes a minimizing point of view, while *hill climbing* assumes the opposite point of view. In most of the book we will base the discussion on minimization, and *local minima* will be the points which cannot be improved by moving to one of their neighbors. Local optimum is a term which can be used both for maximization and minimization.

Local search is surprisingly effective because most combinatorial optimization problems have a very **rich internal structure** relating the configuration X and the f value. The analogy when the input domain is given by real numbers in $\mathbb{R}^n$ is that of a continuously differentiable function $f(x)$ optimized with gradient descent (a.k.a. steepest descent).

A **neighborhood** is suitable for local search if it reflects the problem structure. For example, if the solution is given by a permutation (in the Traveling Salesman Problem a permutation of the cities to be visited) an improper neighborhood choice would be to consider single-bit changes of a binary string describing the current solution, which would immediately cause *illegal configurations*, not corresponding to encodings of permutations. A better neighborhood can be given by all *transpositions* which exchange two elements and keep all others fixed. In general, a sanity check for a neighborhood controls if the f values in the neighborhood are correlated to the f value of the current point. If one starts at a good solution, solutions of similar quality can, on the average, be found more *in its neighborhood* than by sampling a completely unrelated random point. In addition, sampling a random point generally is much more expensive than sampling a neighbor, provided that the f value of the neighbors can be updated ("**incremental evaluation**") and it does not have to be recalculated from scratch, as we will see n the next Section.

24.3 Local search and big valleys

Local search stops at local minima but it can be the initial building block of more complex schemes, which will be presented in future chapters. For many optimization problems of interest, a closer approximation to the global optimum is required, and therefore more complex schemes are needed in order to continue the investigation into new parts of the search space, i.e., to *diversify* the search and encourage *exploration*. Here a second structural element comes to the rescue, related to the overall distribution of local minima and corresponding f values. In many relevant problems local minima tend to be *clustered*, furthermore good local minima tend to be closer to other good minima. **Promising local minima like to be in good company**. Let us define as **attraction basin** associated with a local optimum the set of points X which are mapped to the given local optimum by the local search trajectory. An hydraulic analogy, where the local search trajectory is now the trajectory of drops of water pulled by gravity, is that of *watersheds*, regions bounded peripherally by a divide and draining ultimately to a particular lake.

Now, if local search stops at a local minimum, **kicking** the system to a close attraction basin can be much more effective than restarting from a random configuration. If evaluations of f are incremental, completing a sequence of steps to move to a nearby basin can also be *much faster* than restarting with a complete evaluation followed by a possibly long trajectory descending to another local optimum.

This structural property is also called **Big Valley property** (Fig. 24.7). To help the intuition, one may think about a smooth f surface in a continuous environment, with basins of attraction which tend to have a nested, "fractal" structure. According to Mandelbrot, a fractal is generally "a rough or fragmented geometric shape that can be subdivided into parts, each of which is (at least approximately) a reduced-size copy of the whole," a property called self-similarity[1].

A second continuous analogy is that of a (periodic) function containing components at different wavelengths when analyzed with a Fourier transform. If you are not an expert in Fourier transforms, think about looking at a figure with defocusing lenses. At first the large scale details will be revealed, for example a figure of a distant person, then, by focusing, finer and finer details will be revealed: face arms and legs, then fingers, hair, etc. The same analogy holds for music diffused by loudspeakers of different quality, allowing higher and higher frequencies to be heard. At each scale the sound is not random noise and a pattern, a non-trivial structure is always present. This **multi-scale** structure, where smaller valleys are nested within larger ones, is the basic motivation for methods like Variable Neighborhood

[1] The term *fractal* derives from the Latin *fractus* meaning "broken" or "fractured."

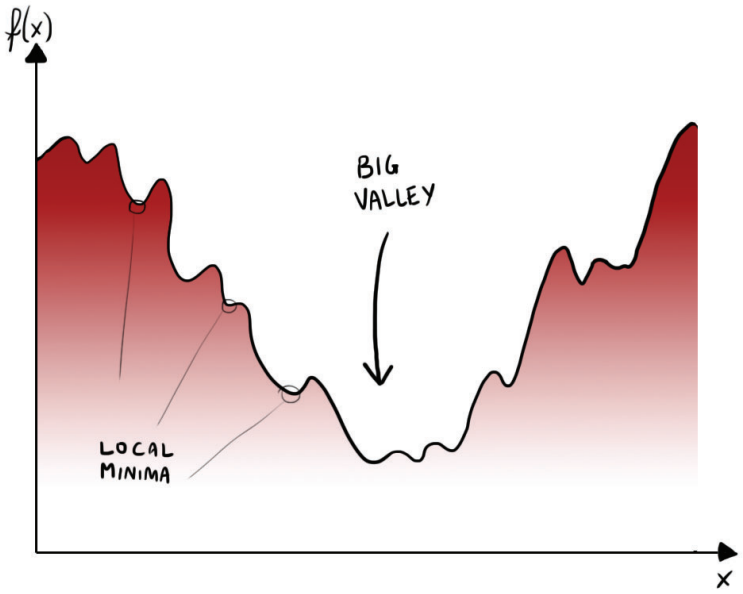


Figure 24.7: Structure in optimization problems: the "big valley" hypothesis.

Search (VNS) and Iterated Local Search (ILS), see for example [27] for a much more extended presentation and discussion of techniques, and [26] for a shorter presentation.

24.3.1 Local search and the TSP

Let's see how Local search comes to the rescue to improve tours in TSP. A random permutation can give us a first tour, or a greedy algorithm can be used to identify a first one with more intelligence. Local modifications of the tour can be obtained by removing some edges of the tour and reconnecting in a different way to obtain a legal tour again. Fig. 24.8 shows a *2-change* local move: two different edges are eliminated and a different legal tour is obtained by reconnecting the nodes. The *2-change neighborhood* of a tour is given by all tours which can be obtained by applying all possible $2 - changes$.

The size of the neighborhood is polynomial, of order $O(n^2)$. In addition, **calculating the change in tour length** after a 2-change is very fast: subtract the cost of the two eliminated edges, add the cost of the two new edges (**incremental evaluation**). The advantage with respect to calculating a new path cost becomes enormous for a large number of cities (4 operations w.r.t. n operations). This means that local search can analyze in a given CPU time a number of possible configurations along a search trajectory which is much larger that the number of unrelated configurations which could be analyzed in the same time.

Of course, 2-changes can be easily generalized to k-changes (obtained by removing k different edges and reconnecting in a different manner). However, more complex neighborhoods imply larger numbers of neighbors to explore, and a tradeoff between improvement and step complexity must be sought, possibly by means of an adaptive technique such as Variable Neighborhood Search (VNS, see Sec. 28.1).

Note that the neighborhood choice does not only imply a slower or faster convergence towards a (local) optimum:

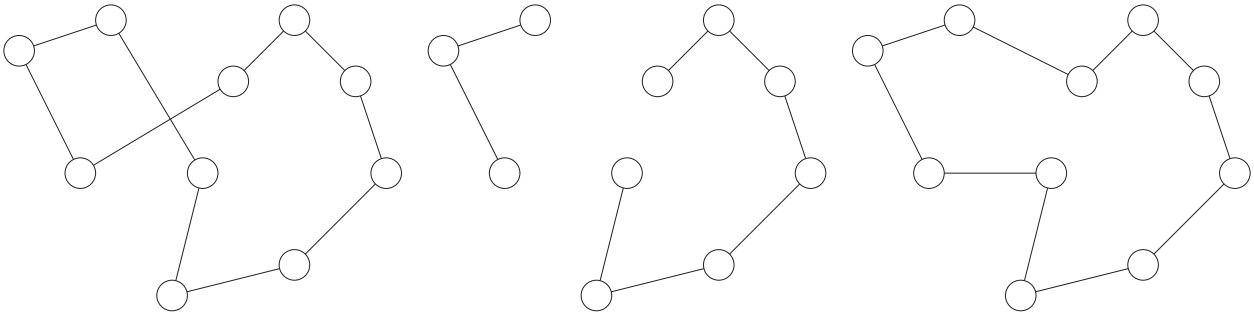


Figure 24.8: The Traveling Salesman Problem: a 2-change local move. Left to right: two edges are removed, leaving the tour partitioned, and the two segments are joined in the opposite way.

configurations that are locally optimal for a given neighborhood may be improved if more perturbations (e.g., a larger neighborhood) are allowed. Particularly bad choices of local moves mean that even simple instances of a problem can get stuck at an unsatisfactory local optimum, as shown in Fig. 24.9 where a TSP instance with 20 cities in a loop is initialized with a random configuration and solved by repeatedly selecting two cities and swapping them in the visiting order.

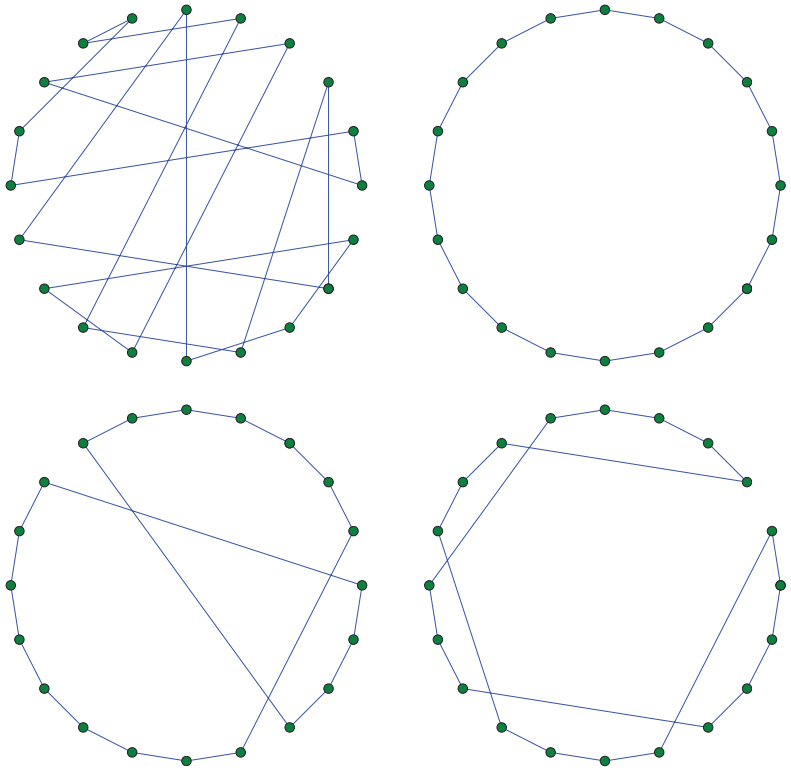


Figure 24.9: TSP configurations for a 20-city ring instance. Clockwise from top left: an initial random configuration, the optimal tour, two local minima. The local minima cannot be further improved by the chosen perturbation function, which consists in swapping the order of visit of two cities. A different local move (e.g., 2-swap) would improve the result.

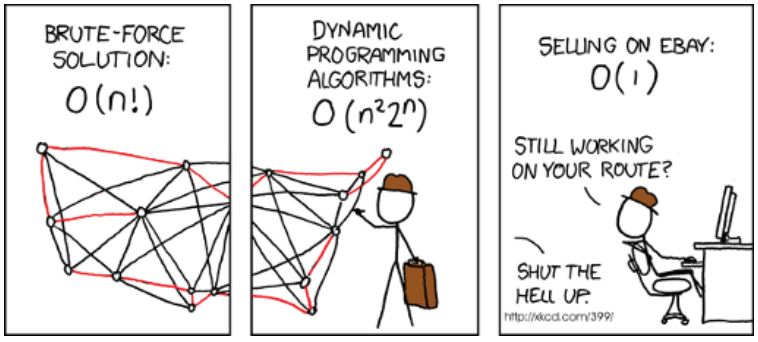


Figure 24.10: Do modern salesmen need TSP at all?

Gist

Greedy and local search are simple, human and effective ways to identify improving solutions for discrete optimization problems. They generate a sequence of changes, each change being *local*, i.e., affecting only a limited portion of the current solution. The **greedy principle** is related to short-sighted changes. The changes look appealing but maybe jeopardize reaching better solutions later on.

In a **greedy construction**, complete and admissible solutions are built is steps, by starting from incomplete ones and by fixing one element of the solution at a time. The single steps are not undone in the future steps.

In **perturbative local search**, one works with a complete solution and searches for a small (local) change leading to an improvement. The motivations for the relative success of local search is related to the rich structure of many problems a.k.a. **big valley** hypothesis. Local minima can often be reached by starting from nearby good tentative solutions. Local search tends to be fast because the **incremental evaluation** of neighbors of the current solution can be much faster that evaluating a new solution from scratch.

Greedy constructions and local search can be combined: one can build an initial complete solution with greedy construction and then improve it with local search.

Local search stops at **locally-optimal points**, when no improving neighbor exists. Additional **diversification** means are needed to escape from local attractors and avoid the search trajectory being *entrapped*, with little possibility of escaping from the attraction basin.

Local search is based on what is perhaps the oldest optimization method – trial and error. The idea is simple and natural and it is surprising to see how successful local search has been on a variety of difficult problems.

Chapter 25

Stochastic global optimization

While discrete variables have been considered in Chapter 24, in this chapter we search for **global optima of function of continuous (real) variables**. Having real variables means that "brute force" techniques like exhaustive enumeration of all possible solutions are impossible. In fact, the possible solutions are infinite! One may consider a range for each variable and *discretize* (for example $x \in [3, 22]$, with possible values $3.0, 3.1, 3.2, 3.3, ..., 21.9, 22.0$), but this dirty trick can lead to solutions of inferior quality. If the proper formulation is as a continuous problem, one should respect it. We will see in this Chapter that **throwing random points onto the input space** can be a robust brute-force method for the continuous optimization case.

Imagine to optimize (e.g., minimize) an objective function f through a so called **black-box interface**: the algorithm can query the value $f(\boldsymbol{x})$ for a sample point $\boldsymbol{x}$, but it cannot "look inside" f to see how it works: it cannot obtain gradient information, and it cannot make any assumptions on its analytic form (e.g., linear, quadratic, logarithmic, etc.). A black-box interface is optimal from the point of view of **separation of concerns**: to be as generally applicable as possible, optimization routines do not need to know anything about the application domain. With separation of

concerns, an optimization expert can improve profits for a financial institution or improve survivability of patients cured for cancer without any knowledge of economics or medicine.

In these cases an optimization scheme has to make do with just function evaluations. Of course, it can still decide *where* to place sample points, and it can use the information obtained to build internal models of the function and tune its own meta-parameters. A large amount of *stochasticity* in the generation of sample points usually helps to improve robustness and avoid that some false initial assumptions lead the optimization to deliver low-quality local optima. *Simplicity and ease of implementation* of the schemes are valuable: in many cases sophisticated schemes improve performance on some specialized tasks but can produce inferior results on different problems.

The simplicity, separation of concerns and general-purpose character of **Stochastic Global Optimization (SGO)** leads to a rapidly growing number of applications in engineering, computational chemistry, finance and medicine. An up-to-date book presenting this topic is [390]: this chapter is a brief overview highlighting some important points and theoretical findings.

SGO is a suite of methods ranging from the simple *global random search* to methods based on *probabilistic assumptions* about the objective function. They are not a panacea and the "**curse of dimensionality**" is unavoidable when the number of variables increases, and it leads to an exponential increase in the computational complexity. This is caused by the fact that neighbours are becoming exponentially isolated from each other as the dimension increases (in a ball of radius r in a large-dimensional space, most points fall in a narrow crust near the surface of the ball).

Many algorithms have been proposed heuristically, in some cases based on sexy analogies with natural processes, like **evolutionary algorithms** and **simulated annealing**. Heuristic global optimization algorithms are very popular in applications, often without a sound theoretical basis. After some basic notions and definitions are given is Sec. 25.1, we start from Pure Random Search (PRS) in Sec. 25.3, develop our intuition with some statistics in Sec. 25.4, discuss probabilistic and statistical models in Secs. 25.5 and 25.6.

25.1 Stochastic Global Optimization Basics

Let $f(\boldsymbol{x})$ be the (real-valued) function to optimize on the feasible region A, usually some region of a multi-dimensional real space $A \subseteq \mathbb{R}^n$.

Global optimization problem:

$$\begin{array}{rl} \text{Given} & f : A \to \mathbb{R} \\ \text{find} & \boldsymbol{x}^* \in A \\ \text{such that} & f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}) \text{ for every } \boldsymbol{x} \in A. \end{array}$$

A point $\boldsymbol{x}^*$ satisfying the above condition is called a **global optimum**, by definition it is the best possible solution to the problem.

Imagine that the optimization scheme generates a sequence of input values $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$, let the **record value** (the best-so-far value) at iteration n be $\hat{y}_n = \min_{i=1,\ldots,n} f(\boldsymbol{x}_i)$, and let $\hat{\boldsymbol{x}}_n$ be the sequence of record points (inputs) corresponding to the record values. The sequence $\hat{y}_1, \hat{y}_2, \ldots$ never increases, and only decreases (i.e., improves) at iterations in which a new record value is found.

The objective of global optimization is to have the sequence of record points $\hat{\boldsymbol{x}}_n$ approach the minimum $\boldsymbol{x}^*$ as n increases.

As one can expect, the development of efficient global optimization methods is more difficult than that of the local optimization methods, the diversity of multi-modal functions is simply too large. Do not expect universal strategies for global optimization and be prepared to live with a wide variety of alternative approaches. In addition, be prepared to **combine robust global schemes with fast local search schemes**.

Multistart local search is one of these possibilities, consisting of firing more runs of local search starting from a set of interesting points. In particular,it can be worth making several iterations of a local descent immediately after obtaining a new record point. In addition, every global optimization procedure should finish with a local descent from one or several of the best points $\hat{\boldsymbol{x}}_i$.

The "global" schemes should narrow down the area of search for the minimizer $\boldsymbol{x}^*$ (hopefully the area should be in the **region of attraction** of $\boldsymbol{x}^*$), leaving the problem of finding the exact location of $\boldsymbol{x}^*$ to a local optimization technique.

As we have already seen in the local search scenario in Chapter. 24, the region of attraction can be different for different methods. In particular, some techniques are able to jump over high-frequency components of the objective function and to reach the global optimum even if started far, provided that the low-frequency structure gives consistent information about locating the minimum value.

This situation is illustrated in Fig. 25.1. The objective function (solid line) has a large number of local minima; however, a local search procedure might as well be oblivious to its roughness and, due to fairly large steps (the grey arrows), only be sensitive to the overall trend. In this case, the procedure correctly identifies the region in which the true global minimum lies (left-hand side of Fig. 25.1). The dashed line is a "model" of the function, built by considering the sample points, that captures its large-scale behavior without bothering about fine-grained details. On the other hand, as shown in the right-hand side, a particularly needle-like minimum might escape this procedure and lie outside the most promising region identified by the model.

Most if not all wide-purpose stochastic global optimization schemes are "once scorned, now respectable" heuristic methods [386] without formal proofs of efficiency. Most theoretical results are related only to asymptotic convergence. In the absence of strong assumptions about an optimization problem, the only converging algorithms are those generating **everywhere dense sequences of observation points**. If A is compact and f continuous in the neighborhood of a local minimizer, an algorithm converges ($y_{on} \to m$ as $n \to \infty$) if and only if the algorithm generates a sequence of points x_i which is everywhere dense in A. Much stronger assumptions like "limited variation" or Lipschitz continuity (see below) are needed to ensure convergence also for sparser sequences of sample points.

Randomness can appear in many ways in global optimization: because of random errors in the evaluation of f (a frequent case in the real world), because of the randomized generation of sample points, or because of probabilistic assumptions about the objective function. We do not insist on deterministic methods in this chapter, they are used rarely only with specific constraints on the function f, in particular limited-variability.

Deterministic global optimization schemes aim at reducing the **worst-case** difference w.r.t. the optimal value. Proofs are complex and conditions of validity very fragile. If optimization has to be repeated for many different functions with similar characteristics, one can take a more realistic direction by assuming **statistical models of the functions** to be optimized and aiming at some **average optimality** criteria (for example demanding that the *average* error converges rapidly to zero).

Global random search algorithms are very popular because they tend to be simple, insensitive to the structure of the objective function and of the feasibility region, easily modifiable to **guarantee theoretical convergence**. In spite of these positive facts, convergence if often more loved by theoreticians than by practitioners because it can be painfully slow, and practical efficiency can depend a lot on tuning algorithm parameters to the instances to be solved, so that automated self-tuning methods can be of value, as well as a sound **experimental approach** in the design and test of competitive algorithms.

25.2 A digression on Lipschitz continuity

Elegant deterministic global optimization schemes can be obtained with strong assumptions about the function to be optimized. One of the "natural" assumptions for many phenomena in the physical world is "**limited-variability**," a.k.a. **Lipschitz continuity**. A function is Lipschitz continuous if its change in value is bounded by the corresponding change in the evaluation point (multiplied by a constant factor); i.e., there is a constant $K \geq 0$ such that, for all $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ in the function's domain,

$$|f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)| \leq K\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|.$$

For each point evaluated x, a **cone** $f(\boldsymbol{x}) - K\|\boldsymbol{x} - \boldsymbol{x}'\|$ defines an area on the plot in which the function cannot enter (because it cannot change too fast), and therefore an underestimate.

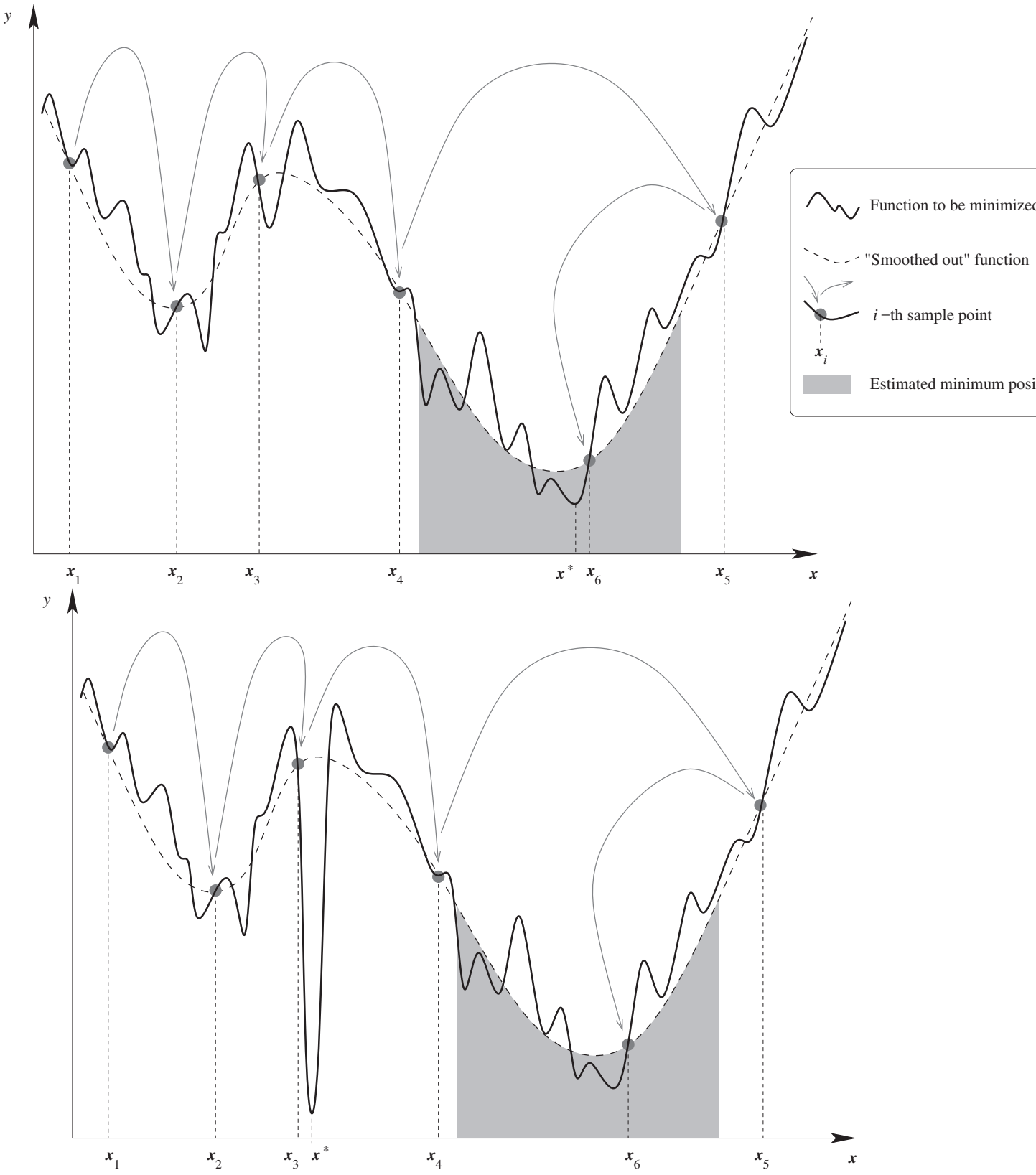


Figure 25.1: Top: function f (solid line) and a low-frequency, smoothed-out version (dashed line) leading to the region in which the global minimum lies. Bottom: a lesser well-behaved function where the global minimum lies at an unexpected position.

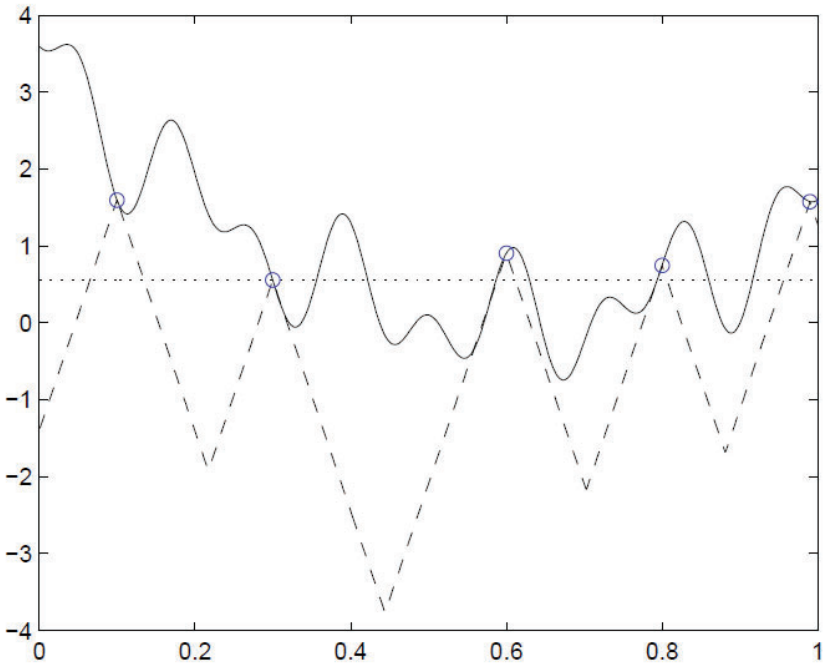


Figure 25.2: Pijavski-Shubert optimization scheme. The values of the objective function at t^k marked with dots determine the saw-tooth underestimate H^K (image derived from [390]).

Fig. 25.2 shows a function, same sampled points, and the current underestimate. It makes sense to place the next observation (sample) at the point of *minimum of the underestimate*, as in a well-known algorithm by Pijavski-Shubert.

Branch and Bound can be applied in a way similar to that for discrete problems: the admissible region A can be partitioned in subareas. If the underestimate over an area is too bad (so that the current record cannot be beaten), the area is cut and not considered for sampling. Advanced partitioning (and sampling) methods are presented in [327, 328]. As you can imagine, the situation for which a non-trivial Lipschitz constant K is known (and therefore underestimates are very tight) are very rare, but one can aim at estimating it in an online and local manner while the global optimization scheme is working [352], in a way similar to the RSO methods for discrete optimization considered in Chapter 27.

The idea of 'branching and bounding' can be extended to incorporate stochastic techniques. Branching of the feasible region A into a tree of subsets $A_i (i = 1, ..., l)$ is done in a similar manner. In global random search algorithms, the test points $x_j \in A_i$ are random; therefore, statistical methods can be used for estimating the attainable minimum on a specific region: $m_i = \inf_{x \in A_i} f(x)$ and for testing hypotheses about the values m_i, see Section 25.4.

25.3 Pure random search (PRS)

PRS is based on the **repeated generation of random points to be evaluated**, typically with the same probability distribution on the admissible region A, like a *uniform distribution*. It sounds too easy to work, but PRS is a solid building block not to be scorned, against which more sophisticated algorithms should be benchmarked. It is often found as a component of other more complex global optimization schemes, and its asymptotic properties, such as convergence rates, are easy to study.

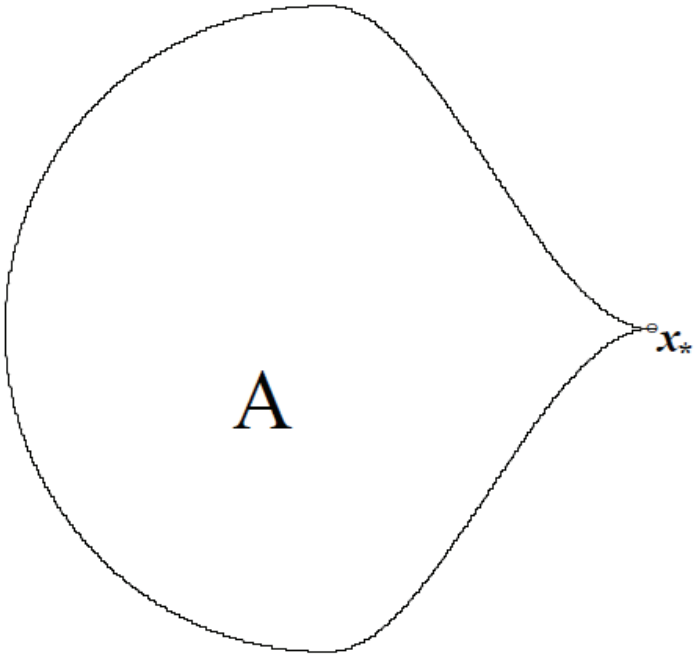


Figure 25.3: A pathological situation for PRS: the minimizer x^* is at a cusp of A so that the portion of a small ball centered around it which is feasible goes rapidly to zero.

In the following, we assume that the feasible region A and function f satisfy assumptions to avoid pathological cases, assumptions like smooth boundaries of A, small balls centered on minimizers having an approximately constant proportion of points in A, see Fig. 25.3 and [390] for the complete list.

Let's consider a general global random search in which point $\boldsymbol{x}_j$ in the sequence is generated from a probability distribution $P_j(\boldsymbol{x})$, which (for generality's sake) is allowed to be different at each iteration and to depend on the previous function evaluations. Because reaching the *exact* minimizer has probability zero (we are in a continuous setting), it is better to allow for some slack. Let's consider balls of a certain radius around points: $B(\boldsymbol{x}, \epsilon) = \{\boldsymbol{x}' \in A : \|\boldsymbol{x}' - \boldsymbol{x}\| \leq \epsilon\}$.

A classical form of the **convergence theorem** (derived from the "zero-one law" in classical probability theory, and rediscovered many times by different researchers) is the following one.

Let f have a finite number of minimizers, let $\boldsymbol{x}^*$ be such a minimizer, m be the minimum value, and f be continuous in the vicinity of $\boldsymbol{x}^*$. Also assume that

$$\sum_{j=1}^{\infty} \inf p_j\big(B(\boldsymbol{x}^*, \epsilon)\big) = \infty$$

for any $\epsilon > 0$, and the infimum is taken over all possible previous histories (sequence of evaluated points and function values). Then, for any $\delta > 0$ the sequence of points $\boldsymbol{x}_j$ with distribution P_j falls infinitely often into the set $W_\delta = \{\boldsymbol{x} \in A : f(\boldsymbol{x}) - m \leq \delta\}$, therefore δ-close to the optimal value.

Imagine placing arbitrarily small balls around each point (including the global optima). If, for all possible histories of stochastic sample generation, the probability distributions $p_j(\boldsymbol{x})$ guarantee that each ball is hit with probability which sum up to infinity, ***an arbitrary δ-close approximation of the optimum will be found infinitely often, with probability one***. In particular, a uniform probability distribution $P_j = P_{\text{uniform}}$ is an immediate way to satisfy the above requirements.

Before you jump on your chair to celebrate, let's remember that, more than this kind of convergence, you are probably more interested in the **rate of convergence**, i.e., in how many iterations you will have to wait in practice before arriving sufficiently close to the minimum, which is the topic of the following discussion.

Because a uniform probability distribution "does not learn" from the previous history of the search, to speedup the convergence in practice, a popular choice combines a "smart" probability density $Q_j(\boldsymbol{x})$, which takes into account what can be learnt from the previously evaluated points, plus a fraction of uniform probability density $P(\boldsymbol{x})$, as:

$$P_j(\boldsymbol{x}) = \alpha_j P(\boldsymbol{x}) + (1 - \alpha_j) Q_j(\boldsymbol{x})$$

so that the above convergence theorem still holds. In general, the distribution $Q_j(\boldsymbol{x})$ should egnerate more points in the most interesting areas. The different methods differe by the way in which they measure the interest level. For example, sampling from Q_j can consist of running a local search descent from the current record value, which is for sure an interesting point.

To take home: **convergence is not difficult to prove**, just mix some uniform probability with your favourite smart ("learning") sampling strategy. Note: this may satisfy your colleagues in maths, but not necessarily your colleagues in applied areas looking for *fast* optimization methods, producing high-quality solutions in a finite CPU time, not for infinite time!

25.3.1 Rate of Convergence of Pure Random Search

"Abandon all hope, you who enter here" was written at the entrance of Dante's Inferno.

In a similar manner, to start your global optimization effort without being fooled by marketing hype, a useful exercise is to derive the **rate of convergence** of PRS. It is a simple application of basic rules about deriving probabilities of repeated unlucky events.

Imagine that the different samples $\boldsymbol{x}_j$ are **independent and identically distributed (i.i.d)** with distribution $p(\boldsymbol{x})$, and let our objective be to hit the "target" set $B(\boldsymbol{x}^*, \epsilon) = \{\boldsymbol{x} \in A : \|\boldsymbol{x} - \boldsymbol{x}^*\| \leq \epsilon\}$ with one or more of the points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. A "success" event means that some $\boldsymbol{x}_j$ hits B, "failure" is the alternative event.

A single sampling event has success probability $p(B)$, which we assume to be strictly positive for all values of ϵ. It looks like throwing a dice with many faces, and calculating the probability that we get a particular face in a sequence of trials. PRS generates a sequence of **independent Bernoulli trials**. We fail in the sequence only if we fail in all trials.

In our case:

$$\Pr\{\boldsymbol{x}_j \notin B\} = 1 - p(B), \quad \text{for all } j.$$

Because of the independent generation of sample points, probabilities are multiplied:

$$\Pr\{\boldsymbol{x}_1 \notin B, \ldots, \boldsymbol{x}_n \notin B\} = \big(1 - p(B)\big)^n.$$

Because $p(B)$ is positive, this probability tends to zero when $n \to \infty$, and the probability that at least one $\boldsymbol{x}_j$ lies in B tends to one.

The average number of PRS iterations required for a first hit of our ball (and therefore solving the problem) is:

$$E(\text{first hitting time}) = \frac{1}{p(B)}.$$

If the distribution is uniform we can continue our exercise to reflect on concrete rates of convergence. In this case, for a d-dimensional problem:

$$p(B) = \frac{\text{vol}(B)}{\text{vol}(A)} = \frac{\pi^{d/2}\epsilon^d}{\Gamma(d/2+1)\text{vol}(A)}$$

in which $\Gamma(\cdot)$ is the Gamma function (an extension of the factorial function) and we used the formula for the volume of a d-dimensional Euclidean ball of radius ϵ.

In the interest of brevity one can consider approximations. If one wants to hit the ϵ-ball B with probability at least $1 - \gamma$, one needs to perform at least the following number of PRS iterations:

$$N^* \approx -\ln\gamma \cdot \frac{\Gamma(d/2+1)}{\pi^{d/2}\epsilon^d} \cdot \text{vol}(A).$$

The dependence on γ is not crucial: what is more alarming is the exponential increase w.r.t. the dimension d. This should not be a surprise. We are searching for a ball with radius equal to ϵ in a simple d-dimensional box of volume 1. If we shoot one random point we hit the target with probability proportional to ϵ in one dimension, ϵ^d in d dimensions, vanishing exponentially to zero when the dimension increases. Fig. 25.4 shows the simpler cases of $d = 1, 2, 3$ and a radius $\epsilon = 0.05$; them measure $p(B)$ goes from $2 \cdot 0.05 = 0.1$ in the 1-dimensional case to $4 \cdot \pi \cdot 0.05^3/3 \approx 0.0005$ in three dimensions.

If you are into combinatorics rather than geometry, you can also see the optimal point $\boldsymbol{x}^*$ as an array of d entries (its coordinates). For a randomly generated array $\boldsymbol{x}_j$ to be close to $\boldsymbol{x}^*$, each of its entries x_{jk} must be close to the corresponding entry x_k^* of the optimal point. Imagine how unlikely it is to generate $d = 100$ independent numbers, and finding that each of them is not farther than ϵ from a target value! Clearly, hitting an ϵ-ball is even harder than that; asymptotically, however, hitting a ball is not so different from hitting a cube: when d is fixed and $\epsilon \to 0$, the number of iterations for success increases approximately as:

$$N^* = O\left(\frac{1}{\epsilon^d}\right).$$

"Abandon all hope, you who enter here". If the number of dimensions is large, there is no magic algorithm to rapidly approximate the global optimum for a generic function in less than exponential number of iterations. If there is hope, it is related to **functions with special forms, so that regularities can be learnt** from an initial sampling, albeit in approximated form, and used to identify shortcuts leading rapidly to close approximations of the optimal solution.

The question is: what is the chance that we will encounter this kind of highly-structured objective functions in real applications? Luckily for us, the chance is not negligible. Think about three-dimensional protein folding. Identifying

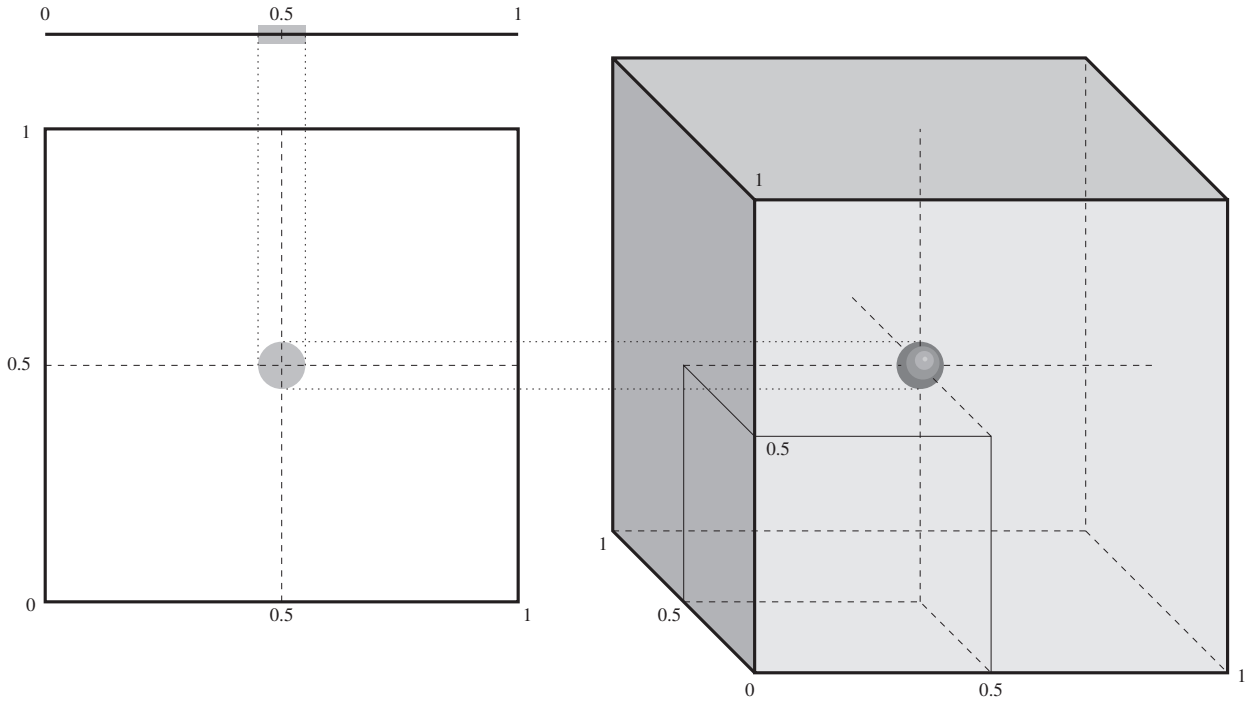


Figure 25.4: The curse of dimensionality: balls of the same radius in spaces of increasing dimensionality become harder and harder to hit due to their growingly negligible size w.r.t. the total volume.

the optimal structure (minimizing the potential energy) seems daunting but mother Nature does it rapidly with very noisy and humid hardware in our cells.

The bottom line is that convergence of some global optimization algorithms may please a theoretician but may not mean much in practice.

25.4 Statistical inference in global random search

If you are an expert in statistics, **order statistics** can be used to derive probabilities of extreme events of interest. In order statistics one starts from a probability distribution and derives distributions for the minimum (or maximum or k-th) **record value** in a sample of a certain size. In addition to optimization, results in this area, also under the name of **extreme value theory**, are applied to **estimating the risk of extreme, rare events**, like earthquakes, floods, beating records in athletic disciplines. We summarize some concepts pointing to useful references if you are interested in further investigations.

The objective is to answer questions of this kind: what is the probability to get a new record value in the next n iterations? What is the average waiting time? What is the probability that we already found a locally optimal point (or an approximation thereof)? Interesting asymptotic distributions arise when the number of iterations becomes very large. Some results are, at the beginning, counter-intuitive. For example, on average one has to make infinitely many iterations of PRS to get an improvement over the current best value. Of course, this result refers to an unbounded search domain and no prior knowledge about the function structure, so be patient! Some potentially useful results are related to statistical inference about the optimal value, which can be used for a statistically sound termination criterion. For example, **maximum likelihood estimations** of m can be derived, as well as **confidence intervals**. All techniques in statistics must be used with extreme competence and care. In certain cases one demands a large number of sample

points in the vicinity of the global optimizer in order to derive estimates, bounds, etc. But reaching the region of attraction of the global minimizer can be extremely difficult so that statistical inference will be made about some other local minimum!

A couple of interesting applications of statistical inference in global optimization are **branch and probability bound** and **random multistart**.

Branch and probability bound generalizes branch and bound to continuous variables when no error-proof bound is available. It consists of several iterations with the following steps:

1. split (branch) the set of admissible values into subsets, obtaining a tree organization;

2. decide about the potential value of the individual subsets for further investigation;

3. select the most interesting subsets for additional splits.

Hyper-rectangles are a frequent and easy choice for subdividing the input space. Statistical techniques to judge about the potential value of a subset Z are based on the (statistical) rejection of the hypothesis that the global minimum m cannot be reached in Z. Statistics can be obtained by sampling Z, or by running short local search streams starting in Z. In practice, *branch and probability bound* methods can be used for low-dimensional problems (up to about 10 dimensions). For much larger dimensions, the number of tree nodes explodes and the efficiency of the statistical procedures degrades.

Random multistart is another simple method consisting of repeating local search from random initial points, and running each search until a local minimum is met, or a close approximation thereof. An interesting question to answer is: given that multistart already found a certain set of locally-optimal points, some of them multiple times, what is the probability that all local optima have been found (including therefore the global optimum)? Let our continuous function f have a finite but unknown number L of local optimizers $\boldsymbol{x}^{*(1)}, \ldots, \boldsymbol{x}^{*(L)}$, and let $\theta_i = p(A_i^*)$ be the probability of "hitting" the attraction basin A_i^* of the local minimizer $\boldsymbol{x}^{*(i)}$. The size and form of the attraction basins depends on the specific local search scheme (which may for example filter out very small sub-basins).

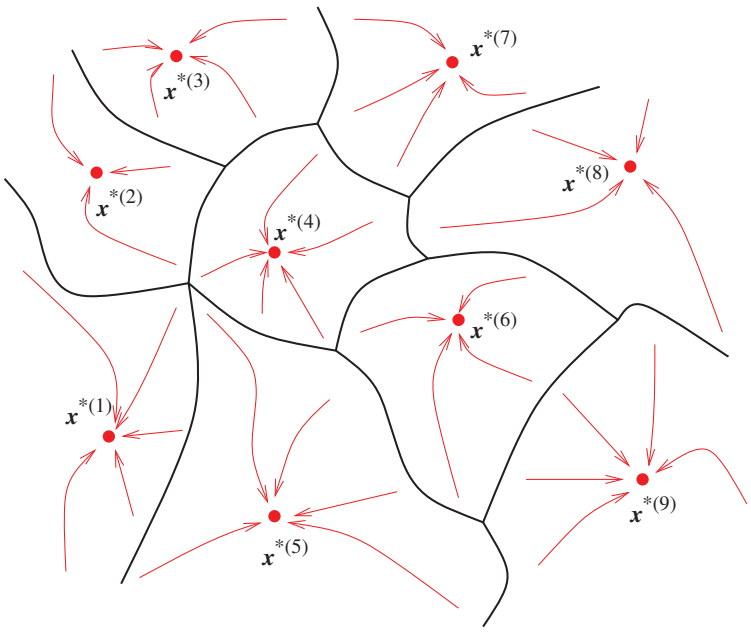


Figure 25.5: Attraction basins around locally optimal points.

By definition, $\theta_i > 0$ and $\sum_i \theta_i = 1$ (every local search will eventually end up on a local minimizer). Let n_i be the number of generated initial points belonging to the i-th basin, out of a sample of n points. The random vector $(n_1, \ldots, n_L)$ follows the **multinomial distribution**:

$$\Pr(n_1, ..., n_L) = \frac{n!}{n_1!...n_L!}\theta_1^{n_1}...\theta_L^{n_L}.$$

Statistical inference can be used to estimate the number of trials n^* with a probabilistic guarantee that all local minimizers have been found.

If the number of local minimizers L is known, at least as an upper bound L, we can obtain the following approximation for the number of starts of local search which guarantee to find all local minima with probability at least γ:

$$n^* \approx L \ln L + L \ln(-\ln \gamma)$$

If the number of local minimizers is not known, a Bayesian approach with an *a priori* distribution for the number of local minima can be used (but the practical issue becomes that of identifying a proper *a priori* probability, a kind of magic art in the absence of information about the possible function to be minimized).

25.5 Markov processes and Simulated Annealing

This section presents ideas which can be seen as stochastic modifications of Local Search, and which apply for both continuous and discrete optimization problems.

Local search stops at a locally optimal point. Now, for problems with a rich internal structure encountered in many applications (remember the "big valley" hypothesis), *searching in the vicinity of good local minima* may lead to the discovery of even better solutions. In this section the neighborhood structure is *fixed*, but the move generation and acceptance are stochastic and one also permits a "controlled worsening" of solution values to escape from the local attractor.

Now, if one is sitting on a local minimum and extends local search by *accepting worsening moves* (moves leading to worse f values) the trajectory moves to a neighbor of a local minimum. But the danger is that, after raising the solution value at the new point, the local minimum will be chosen again at the next iteration, leading to an endless cycle of "trying to escape and falling back immediately to the starting point." This situation surely happens in the deterministic case if the local minimum is *strict* (all neighbors have worse f values) and if more than one intermediate step is needed before points with f values better than that of the local minimum become accessible. Better points can become accessible when they can be reached from the current solution point by a local search trajectory. The situation becomes more chaotic in stochastic versions, but still one may be stuck jumping around in an attraction basin around a local optimizers for very long times.

The **Simulated Annealing (SA)** method has been investigated to avoid deterministic cycles and to allow for worsening moves, while still biasing the exploration so that low f values are visited more frequently than large values. The terms and the initial application comes from annealing in metallurgy, a process involving heating and controlled cooling of a metal to increase the size of its crystals and reduce their defects. The heat causes the atoms to be shaken from their initial positions, a local minimum of the internal energy, and wander randomly through states of higher energy; the slow cooling gives them more chances of finding states with lower internal energy than the initial one, corresponding to a stronger metal.

We summarize the technique and hint at mathematical results.

25.6 Simulated Annealing and Asymptotics

In the general model of Pure Random Search the probability distribution $p_j(\boldsymbol{x})$ for generating the j-th sample can depend on the previously extracted points (and on the corresponding objective function values). A radical simplification,

but possible improvement w.r.t. using a uniform distribution P, is to have P_j just depend on the latest generated point and f value. The sequence of sample points becomes a **Markov chain**. Markov is synonymous with **memory-less**: the entire previous history of the search process (points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{j-2}$ and corresponding f values) is forgotten to keep only the latest point. As you can imagine, Markovian algorithms are often practically inefficient because of their poor use of information and learning-while-searching possibilities. Nonetheless, they have enjoyed a huge popularity, partly caused by intriguing analogies with physical processes, partly because many mathematicians could demonstrate asymptotic theorems, rather useless to get practical guidelines, but helpful to obtain an aura of respectability.

The Simulated Annealing method [234] is based on the theory of Markov processes. The trajectory is built in a randomized manner: the successor of the current point is chosen stochastically, with a probability that depends only on the current point and not on the previous history.

$$\begin{aligned} \boldsymbol{x}' &\leftarrow \text{NEIGHBOR}(\ N(\boldsymbol{x}_j)\ ) \\ \boldsymbol{x}_{j+1} &\leftarrow \begin{cases} \boldsymbol{x}' & \text{if } f(\boldsymbol{x}') \leq f(\boldsymbol{x}_j) \\ \boldsymbol{x}' & \text{if } f(\boldsymbol{x}') > f(\boldsymbol{x}_j), \text{ with probability } p = e^{-\frac{f(\boldsymbol{x}) - f(\boldsymbol{x}_j)}{T}} \\ \boldsymbol{x}_j & \text{otherwise.} \end{cases} \end{aligned} \tag{25.1}$$

The neighbor of the current point can be obtained in discrete problems by applying a limited set of local changes to the current solution. In continuous problem it can be obtained by sampling with a probability distribution centered on the current point, e.g., with a Gaussian distribution.

SA introduces a *temperature* parameter T which determines the probability that worsening moves are accepted: a larger T implies that more worsening moves tend to be accepted, and therefore a larger diversification occurs. The rule in equation (25.1) is called *exponential acceptance rule*. If T goes to infinity, then the probability that a move is accepted becomes 1, whether it improves the result or not, and one obtains a **random walk**. *Vice versa*, if T goes to zero, only improving moves are accepted as in the standard local search. Being a Markov process, SA is characterized by a memory-less property: if one starts the process and waits long enough, the memory about the initial configuration is lost, the probability of finding a given configuration at a given state will be stationary and only dependent on the value of f. If T goes to zero **the probability will peak only at the globally optimal configurations**. This basic result raised high hopes of solving optimization problems through a simple and general-purpose method, starting from seminal work in physics [268] and in optimization [298, 87, 234, 2].

Unfortunately, after some decades it became clear that SA is not a *panacea*. Furthermore, most mathematical results about **asymptotic convergence** (converge of the method when the number of iterations goes to infinity) are quite **irrelevant for optimization**. First, one does not care whether the *final* configuration at convergence is optimal or not, but that an optimal solution (or a good approximation thereof) is encountered —and memorized— during the search. Second, asymptotic convergence usually requires a patience which is excessive considering the limited length of our lives. Actually, repeated local search [126], and even pure random search [88] have better asymptotic results for some problems.

A practitioner has to place asymptotic results in the background and develop heuristics where high importance is attributed to **learning from a task in an online manner during the search**. In the following we briefly consider some asymptotic convergence results of SA.

25.6.1 Asymptotic convergence results

Let $(\mathcal{X}, f)$ be an instance of a combinatorial optimization problem, $\mathcal{X}$ being the search space and f being the objective function. Let $\mathcal{X}^*$ be the set of optimal solutions. One starts from an initial configuration $X^{(0)}$ and repeatedly applies equation (25.1) to generate a trajectory $X^{(t)}$. Under appropriate conditions, the probability of finding one of the optimal solutions tends to one when the number of iterations goes to infinity:

$$\lim_{k \to \infty} \Pr(X^{(k)} \in \mathcal{X}^*) = 1. \tag{25.2}$$

Let $\mathcal{O}$ denote the set of possible outcomes (states) of a sampling process, let $X^{(k)}$ be the stochastic variable denoting the outcome of the k-th trial, then the elements of the *transition probability matrix* P, given the probability that the configuration is at a specific state j given that it was at state i before the last step, are defined as:

$$p_{ij}(k) = \Pr(X^{(k)} = j | X^{(k-1)} = i). \tag{25.3}$$

A *stationary distribution* of a finite time-*homogeneous* (meaning that transitions do not depend on time) Markov chain is defined as the stochastic vector $\boldsymbol{q}$ whose components are given by

$$q_i = \lim_{k\to\infty} \Pr(X^{(k)} = i | X^{(0)} = j), \text{ for all } j \in \mathcal{O} \tag{25.4}$$

If a stationary distribution exists, one has $\lim_{k\to\infty} \Pr(X^{(k)} = i) = q_i$. Furthermore $\boldsymbol{q}^T = \boldsymbol{q}^T P$, the distribution is not modified by a single Markov step.

If a finite Markov chain is homogeneous, *irreducible* (for every i, j, there is a positive probability of reaching i from j in a finite number of steps) and *aperiodic* (the greatest common divisor $\gcd(\mathcal{D}_i) = 1$, where $\mathcal{D}_i$ is the set of all integers $n > 0$ with $(P^n)_{ii} > 0$), there exist a unique stationary distribution, determined by the equation:

$$\sum_{j\in\mathcal{O}} q_j p_{ji} = q_i \tag{25.5}$$

Unfortunately the rate of convergence of SA is very slow, being based similar arguments as for the convergence of pure random search.

Homogeneous model

In the homogeneous model one considers a sequence of infinitely long homogeneous Markov chains, where each chain is for a fixed value of the temperature T.

Under appropriate conditions [1] (the generation probability must ensure that one can move from an arbitrary initial solution to a second arbitrary solution in a finite number of steps) the Markov chain associated to SA has a stationary distribution $q(T)$ whose components are given by:

$$q_i(T) = \frac{e^{-f(i)/T}}{\sum_{j\in\mathcal{X}} e^{-f(j)/T}} \tag{25.6}$$

and

$$\lim_{T\to 0} q_i(T) = q_i^* = \frac{1}{|\mathcal{X}^*|} I_{\mathcal{X}^*}(i) \tag{25.7}$$

where $I_{\mathcal{X}^*}$ is the characteristic function of the set $\mathcal{X}^*$, equal to one if the argument belongs to the set, zero otherwise.

It follows that:

$$\lim_{T\to 0} \lim_{k\to\infty} \Pr_T(X^{(k)} \in \mathcal{X}^*) = 1 \tag{25.8}$$

The algorithm asymptotically finds an optimal solution with probability one, "converges with probability one."

Inhomogeneous model

In practice one cannot wait for a stationary distribution to be reached. The temperature must be lowered before converging. At each iteration k one has therefore a different temperature T_k, obtaining a non-increasing sequence of values T_k such that $\lim_{k\to\infty} T_k = 0$.

If the temperature decreases in a sufficiently slow way:

$$T_k \geq \frac{A}{\log(k + k_0)} \tag{25.9}$$

for $A > 0$ and $k_0 > 2$, then the Markov chain converges in distribution to q^* or, in other words

$$\lim_{k \to \infty} \Pr(X^{(k)} \in \mathcal{X}^*) = 1 \tag{25.10}$$

The theoretical value of A depends on the depth of the deepest local, non-global optimum, a value which is not easy to calculate for a generic instance.

The above cited asymptotic convergence results of SA in both the homogeneous and inhomogeneous model are unfortunately *irrelevant for the application of SA to optimization*. In any finite-time approximation one must resort to approximations of the asymptotic convergence. The "speed of convergence" to the stationary distribution is determined by the second largest eigenvalue of the transition probability matrix $P(T)$ (not easy to calculate!). The number of transitions is at least *quadratic* in the total number of possible configurations in the search space [1]. For the inhomogeneous case, it can happen (e.g., Traveling Salesman Problem) that the *complete enumeration of all solutions* would take less time than approximating an optimal solution arbitrarily closely by SA [1].

In addition, repeated local search [126], and even random search [88] has better asymptotic results. According to [1] "approximating the asymptotic behavior of SA arbitrarily closely requires a number of transitions that for most problems is typically larger than the size of the solution space. Thus, the SA algorithm is clearly unsuited for solving combinatorial optimization problems to optimality." Of course, SA can be used in practice with fast *cooling schedules*, i.e., ways to progressively reduce the temperature during the search, but then the asymptotic results are not directly applicable. The optimal finite-length annealing schedules obtained on specific simple problems do not always correspond to those expected from the limiting theorems [351].

More details about cooling schedules can be found in [273, 196, 169]. Extensive experimental results of SA for graph partitioning, coloring and number partitioning are presented in [219, 220]. A comparison of SA and Reactive Search Optimizaiton (RSO) is presented in [38, 39].

The authors share with [390] an overall skepticism about using **Markov Chain Montecarlo** (MCMC) methods for efficiently optimizing functions, even in small dimensions.

25.7 The Inertial Shaker algorithm

A suggestion derived from our experience with optimization is: **Be lazy! Always try simple methods first, add complication only if motivated by a measurable improvement.**

The simpler **Inertial Shaker (IS)** technique, outlined in Fig. 25.6 can be a practical choice to go beyond Pure Random Search, while allowing on-the-job learning to rapidly adapt the probability of generating the next sample.

In IS the generation probability is uniform over a **search box** identified by vectors parallel to the coordinate axes (therefore the search box is defined by a single vector $\boldsymbol{b}$. In addition, a *trend direction* is identified by averaging a number of previous displacements [36]: the *find_trend* function used at line 7 simply returns a weighted average of the m_{disp} previous displacements:

$$\boldsymbol{\delta}_t = \textit{amplification} \cdot \frac{\sum_{u=1}^{T} \boldsymbol{\delta}_{t-u} e^{-\frac{u}{(\textit{history\_depth})^2}}}{\sum_{u=1}^{T} e^{-\frac{u}{(\textit{history\_depth})^2}}},$$

where *amplification* and *history_depth* are defined in the algorithm, while m_{disp} is chosen in order to cut off negligible exponential weights and to keep the past history reasonably small.

f	(input)	Function to minimize
$\boldsymbol{x}$	(input)	Initial and current point
$\boldsymbol{b}$	(input)	Box defining search region $\mathcal{R}$ around $\boldsymbol{x}$
$\boldsymbol{\delta}$	(parameter)	Current displacement
amplification	(parameter)	Amplification factor for future displacements
history_depth	(parameter)	Weight decay factor for past displacement average

```
1.  function InertialShaker (f, x, b)
2.      t ← 0
3.      repeat
4.          success ← double_shot_on_all_components ( δ)
5.          if success = true
6.              x ← x + δ
7.              find_trend ( δ)
8.              if f( x + δ) < f( x)
9.                  x ← x + δ;
10.                 increase amplification and history_depth
11.             else
12.                 decrease amplification and history_depth
13.     until convergence criterion is satisfied
14.     return x;
```

Figure 25.6: The Inertial Shaker algorithm, from [36].

Fig. 25.9 shows how the double-shot strategy is applied to all components of the search position $\boldsymbol{x}$. A displacement is applied at every component as long as it improves the result. If no improvement is possible, then the function returns `false`, and the search box is accordingly shrunk.

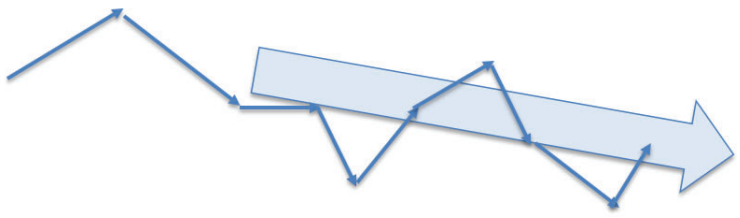


Figure 25.7: An illustration of the trend direction in the Inertial Shaker.

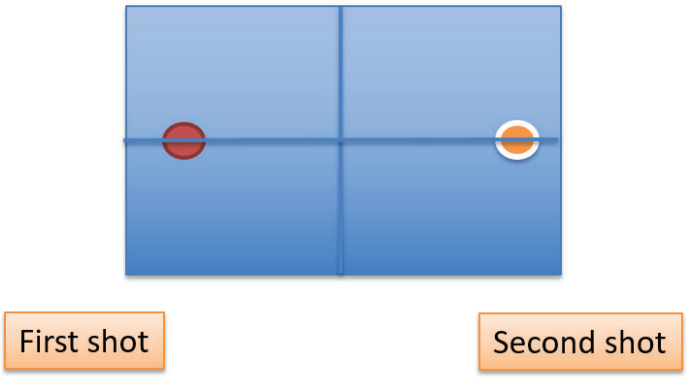


Figure 25.8: An illustration of the "double shot" in the Inertial Shaker.

f	Function to minimize
$\boldsymbol{x}$	Current position
$\boldsymbol{b}$	Vector defining current search box
$\boldsymbol{\delta}$	Displacement

```
1.  function double_shot_on_all_components (f, x, b, δ)
2.      success ← false
3.      x̂ ← x
4.      for i ∈ {1, ..., n}
5.          E ← f( x̂)
6.          r ← random in [−b_i, b_i]
7.          x̂_i ← x̂_i + r
8.          if f(x̂) > E
9.              x̂_i ← x̂_i − 2r
10.             if f(x̂) > E
11.                 b_i ← ρ_comp b_i
12.                 x̂_i ← x̂_i + r
13.             else
14.                 b_i ← ρ_exp b_i
15.                 success ← true
16.         else
17.             b_i ← ρ_exp b_i
18.             success ← true
19.     if success = true
20.         δ ← x̂ - x
21.     return success
```

Figure 25.9: The double-shot strategy from [36]: apply a random displacement within the search box to all coordinates, keep the improving steps; return `false` if no improvement is found.

Gist

Stochastic Global Optimization is simple and robust technique relying on the **random generation of sample points** in the search space. It is a bit like in the piñata or *pentolaccia* game in which a blindfolded kid tries to break a container filled with treats. If one is patient, sooner or later a random point will fall in the vicinity of a global optimum. **Asymptotic convergence** can be demonstrated but it is irrelevant in practice

The **curse of dimensionality** is unavoidable: when the dimension is large there are just too many places to hide the global optimum and the number of sample grows exponentially. But there is still hope if the objective function has a very rich structure with regularities (e.g., a "big valley" structure), a frequent case in practical applications

Simulated Annealing goes beyond local optima by allowing for controlled worsening of the current solutions, and it generates Markov chains (memory-less).

The more effective methods in SGO use some form of learning from the past generated samples along the search directory. **Statistical inference and "learning" techniques** can be used to modify probabilities of generating new samples, so that they are more concentrated on interesting regions.

The **Inertial Shaker** is a pragmatic example: samples are generated from a search box. Both the search box and a the trend direction are adapted (learnt) during the run.

Chapter 26

Derivative-Based Optimization

In this world - I am gonna walk
Until my feet - refuse to take me any longer
Yes I' m gonna walk - and walk some more.
(Macy Gray and Zucchero Fornaciari)

Most if not all problems can be cast as finding the optimal value for a suitable *objective function*, subject to constraints. If you are buying a house, you will have a bounded budget and objectives like number of rooms, neighborhood, view, vicinity to workplace, schools, etc. If you are searching for a partner you will have objectives like intelligence, beauty, companionship, etc. If you are running a company you will aim at maximizing profit, given constraints regarding your resources of people and equipment... You may notice that defining the proper function to be optimized is not trivial (think about the preference function for your preferred partner) but, once this crucial preliminary work is finished, one is left with the problem of **minimizing or maximizing a function which maps independent variables to an output value**. Maximizing means identifying the input values causing the maximum output value. If constraints are given, the input needs to be **feasible**, i.e., it needs to satisfy all constraints.

Methods to optimize functions are **the source of power** for most problem-solving and decision making activities, either explicitly or implicitly, a sound motivation for understanding the basic ideas and tools. While one does not need to know the underlying math before using the technology, mastering the basis facilitates faster and more effective choices. We consider the following related problems.

- **Nonlinear equations**, i.e. solving a set of nonlinear equations (individual functions f_i are collected in vector F):

  $$\begin{aligned} &\text{Given } F : \mathbb{R}^n \longrightarrow \mathbb{R}^n \\ &\text{find } x^* \in \mathbb{R}^n \text{ such that } F(x^*) = 0 \in \mathbb{R}^n. \end{aligned}$$

  The solution x^*, if it exists, minimizes $\sum_{i=1}^{n} (f_i(x))^2$. This is obvious because the sum of squares is not negative, and equal to zero if and only if all individual functions evaluate to zero.

- **Unconstrained minimization**:

  $$\begin{aligned} &\text{Given } f : \mathbb{R}^n \longrightarrow \mathbb{R} \\ &\text{find } x^* \in \mathbb{R}^n \text{ such that } f(x^*) \leq f(x) \text{ for every } x \in \mathbb{R}^n. \end{aligned}$$

  A point x^* satisfying the above condition is called a *global optimum*, by definition it is the best possible solution to the problem: no other solution is better.

In this chapter we collect some basic and traditional methods to optimize **smooth functions of continuous variables** (of real numbers), with some demonstrations about their convergence properties.

In mathematics, a function is smooth if a **derivative** exists. You may recollect the mathematical definition of derivative $f'(a)$ of function f at point a, as the limit for a step h going to zero of the difference in function value divided by the step:

$$f'(a) \;=\; \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}. \tag{26.1}$$

What does this abstract definition have in common with learning, making approximated models, optimizing? A lot.

The practical "meaning" of the derivative is that, if x values are very close to a starting point of interest a (so that the displacement $h = x - a$ is small) one can use it to obtain a good **local approximation** as:

$$f(a+h) \;\approx\; f(a) + f'(a)\, h \tag{26.2}$$

The approximation is linear in the displacement h and it gets better and better when h becomes very small. The original functions can be approximated, in a local area, with its **tangent line**, which is linear and therefore easier to handle (with linear algebra!).

Counter-examples where derivatives do not exist are discontinuous functions (with sudden jumps) which cannot be approximated with a straight line at the jump position, or functions with sharp corners, which again do not have a single well-defined tangent line. A ski on a gentle slope is a nice image for a derivative of a smooth function. Skiing works because tangent lines are a good approximation to the slope. A ski on a spiky rock gets broken because there is no local smooth distribution of forces along a tangent line.

Like skis are tools for descending slopes, first and second derivatives are **useful tools to build local models** to improve a tentative configuration x. The improvements are typically obtained in an iterative manner, from an initial point x_1, to an improving point x_2 (with a different local model), to an improving x_3 (with a different local model)...

If the model is linear, one gets only indications about possible descent directions. In one dimension, after knowing the derivative at point x_1, one knows whether to do a small step to the right or to the left, depending on the slope. For sure, the step has to be *sufficiently small*. If the derivative is not zero, the model given by the tangent line does not lead to a well-defined minimization problem, the y value is unbounded and goes to minus infinity. A *quadratic* model using also the second derivatives can lead to a well-defined minimization problem if the parabola is U-shaped (opening to the top). In this case, one can to define the next point x_2 as the minimum of the parabolic model. A quadratic model built by using the fist and second derivatives of the function in Fig. 26.3 is shown in Fig. 26.2.

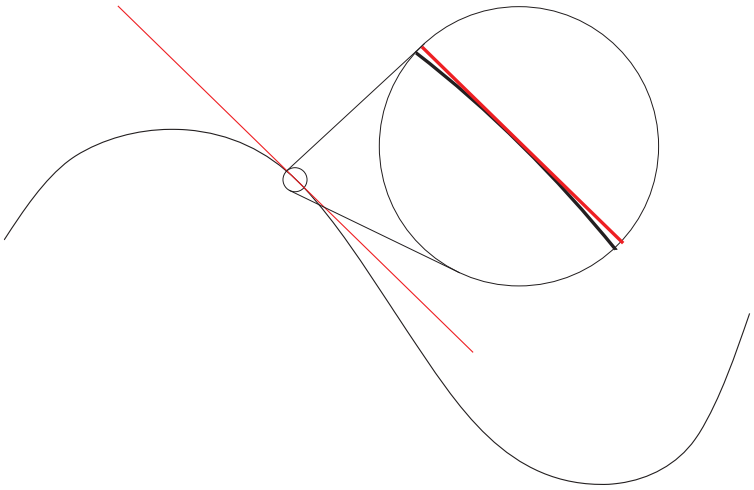


Figure 26.1: The graph of a function, drawn in black, and a tangent line to that function, drawn in red. The circular inset shows that the tangent line is a good local model of the function when one stays close to the point of interest.

26.1 Optimization and machine learning

To increase our motivation before jumping into the more terse mathematical results, let's note that there is a strong connection between machine learning and optimization.

When going in the direction of using **optimization for learning**, for sure *optimization* has to be used to select, among a class of models, one that is most consistent with the data provided for learning, one that better explains the observed data. An example is the usual "sum of squared differences" used in fitting curves and in supervised learning. Of course, the final scope of learning is *generalization*, but this only means that the function to be minimized will contain more pieces, to take the *model complexity* into account, so that simpler models will be favored over more complex ones.

When going in the contrary direction of using **learning for optimization**, some forms of *learning* are also used in efficient optimization algorithms. Preliminary examples of "learning" methods, although the inventors did not use that term, can be found in standard techniques for continuous optimization, where local models (obtained by using local information about function and derivatives) are constructed, whose validity can be limited to regions around the current point (*model-trust-regions* methods). Both the models and the *trust-regions* are typically adapted during a sequence of steps homing at a local minimizer.

While these techniques are traditionally associated with continuous optimization, the same principle of having a **local model learned during optimization** (or parameters tuned to the instance and local properties) can be useful in the different context of discrete (combinatorial) optimization, see for example the local search (Chapter 24) and Reactive Search Optimization (RSO) techniques [26] (Chapter 27).

A *leitmotiv* of many methods is to build the final solution by **a sequence of steps which modify a tentative solution**. At each step a **local model of the function** being optimized is built and used for a local move, modifying the tentative solution by small changes. The methods are therefore *short-sighted* and there is no guarantee of convergence to the global optimum. Nonetheless, in practice the presence of local minima (where local searchers will be stuck) did not prevent gradient-descent, local search and related techniques from becoming probably the simplest and most successful problem-solving machines.

Let us now see how the principle of having a *flexible local model (with parameters) learned during optimization of a given instance* acts in the case of continuous functions. The purpose of the following sections is to taste some of the most basic and successful paradigms of continuous optimization, with emphasis on intuition more than on mathematical details, that can be found in [115].

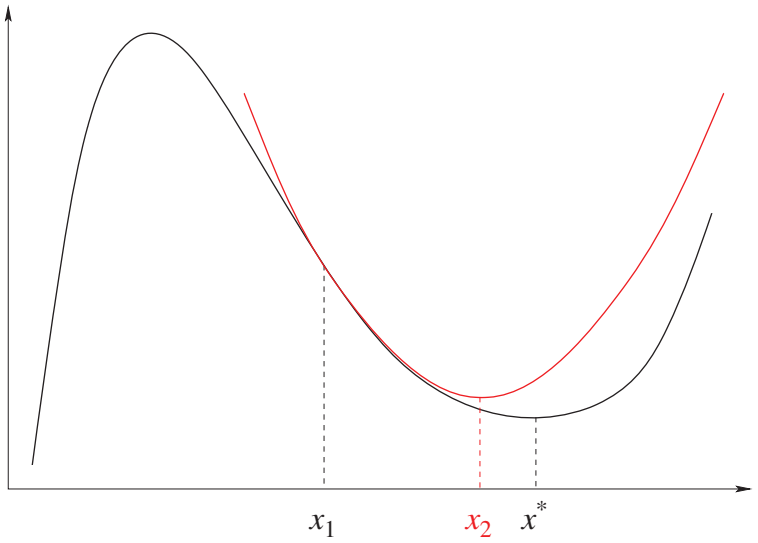


Figure 26.2: The graph of a function, drawn in black, and a local quadratic model at point x_1. The minimizer of the parabola x_2 is close to the (local) minimizer x^* of the function.

A discrimination has to do with the **availability of derivatives** of the function to be minimized. In most real-world cases derivatives are not available, actually in many cases the relationship between inputs and outputs can be *discontinuous*, or some inputs can be *discrete* (for example integer values). Try asking a businessman for the derivative of profit as a function of significant business choices, we doubt that you will get an answer!

If you are lucky and you are dealing with a function $f(x)$ of real numbers, which is continuous and differentiable, standard methods can be used. In particular, we summarize methods for one-dimensional optimization (Section 26.2), then review techniques for solving models (quadratic positive definite forms) in more dimensions (Section 26.3) and methods that use the model-solving techniques for the optimization of nonlinear functions of many variables (Section 26.4).

If your function does not have derivatives, you may consider the methods based on function evaluations only, like those described in Chapter 25.

26.2 Derivative-based techniques in one dimension

Intuition is easier in one dimension and let's therefore start with some classical results for functions of one variable. The historic, and still fundamental way to find a point where a differentiable function $f(x)$ is equal to zero, a.k.a. a **root**, is to start with a point *sufficiently close* to the target and iterate the two following steps:

1. find a **local solvable model**,
2. solve the local model.

The local model around the current point x_c can be derived from **Taylor series approximation** by stopping at the quadratic term:

$$f(x) = f(x_c) + f'(x_c)(x - x_c) + \frac{f''(x_c)(x - x_c)^2}{2!} + \dots,$$

or from Newton's theorem:

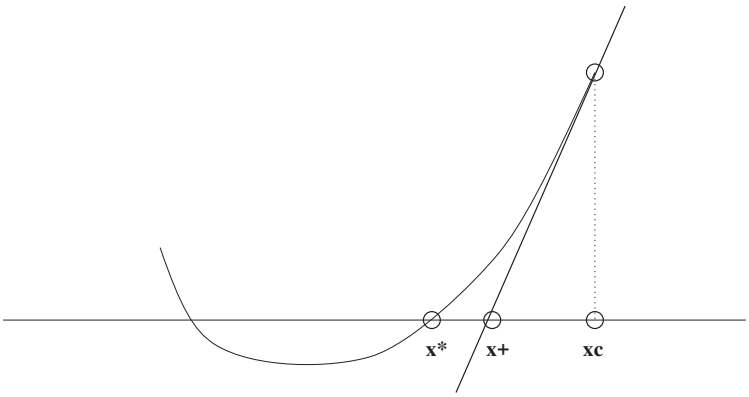


Figure 26.3: Local model for Newton's method.

$$f(x) = f(x_c) + \int_{x_c}^{x} f'(z)\ \mathrm{d}z \approx f(x_c) + f'(x_c)(x - x_c).$$

A **local model** (actually an **affine model**) around the current estimate x_c is therefore

$$M_c(x) = f(x_c) + f'(x_c)(x - x_c),$$

and by finding the root of the model one gets a prescription for the next value x_+ of the current estimate (the local step is from x_c to x_+), as illustrated in Fig. 26.3:

$$x_+ = x_c - \frac{f(x_c)}{f'(x_c)}.$$

If the function is linear, convergence occurs in one step. If the function is nonlinear, let us study the *local convergence* properties of Newton's method: we demonstrate that, if one starts with a point x_c sufficiently close to the root, one will eventually converge to it. The proof proceeds by *bounding the lack of linearity* of the model, and by demonstrating that the **distance to the target root is contracted at every step**.

The lack of linearity, or the error by using the model is:

$$f(x) - M_c(x) = \int_{x_c}^{x} [f'(z) - f'(x_c)]\ \mathrm{d}z.$$

We need now to bound the variation of a function proportionally to the difference in its inputs.

Definition 1 (Lipschitz continuity) *A function g is Lipschitz continuous with constant γ in a set X ($g \in Lip_\gamma(X)$) if for every $x, y \in X$:*

$$|g(x) - g(y)| \leq \gamma |x - y|.$$

Lemma 1 *Let $f' \in Lip_\gamma(D)$ for an open interval D. Then for any $x, y \in X$:*

$$|f(y) - f(x) - f'(x)(y - x)| \leq \gamma \frac{(x - y)^2}{2}.$$

Proof.

$$|f(y) - f(x) - f'(x)(y - x)| = \int_0^1 [f'(x + t(y - x)) - f'(x)](y - x)\ \mathrm{d}t,$$

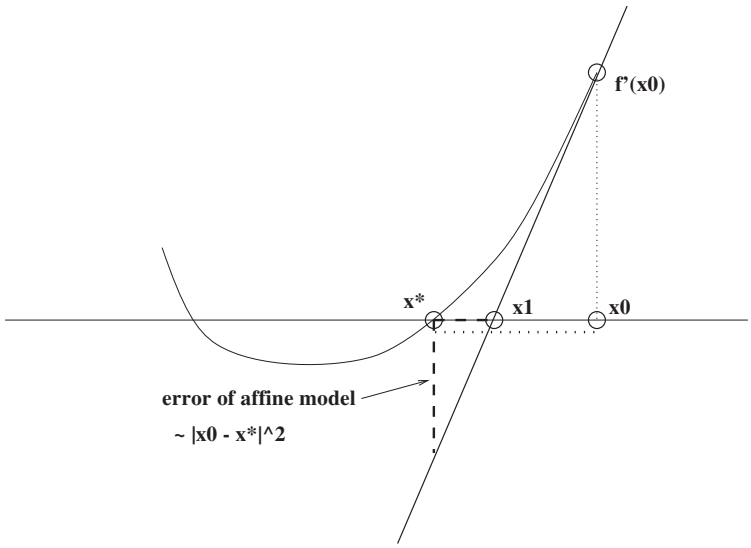


Figure 26.4: Convergence is guaranteed if the starting point x_0 is close to x^*.

and by using the triangle inequality and Lipschitz continuity:

$$\leq |y-x| \int_0^1 \gamma |t(y-x)| \, \mathrm{d}t = \gamma |y-x|^2/2.$$

We are now ready to demonstrate the **convergence theorem of Newton's method in one dimension**. Fig. 26.4 can help to follow the demonstration.

Theorem 1 *Let $f : D \longrightarrow \mathbb{R}$ for open interval D, $f' \in Lip_\gamma(D)$ (Lipschitz), $|f'(x)| \geq \rho$ (derivative bounded away from zero) in D.*

If $f(x) = 0$ has a solution $x^* \in D$, then the solution can be found by Newton method if the starting point x_0 is sufficiently close:
there is $\eta > 0$ such that if $|x_0 - x^*| < \eta$, the sequence:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

exists and converges to x^*. In addition:

$$|x_{k+1} - x^*| \leq \frac{\gamma}{2\rho} |x_k - x^*|^2.$$

Proof. Find a starting *ball* such that:

$$|x_{k+1} - x^*| \leq \tau |x_k - x^*| \text{ for a } \tau \in (0,1),$$

this would also imply that the point *remains* in the ball. Let's see.

$$x_1 - x^* = x_0 - x^* - \frac{f(x_0)}{f'(x_0)} = x_0 - x^* - \frac{f(x_0) - f(x^*)}{f'(x_0)},$$

$$= \frac{1}{f'(x_0)}[f(x^*) - f(x_0) - f'(x_0)(x^* - x_0)] = \frac{1}{f'(x_0)}[f(x^*) - M_0(x^*)],$$

where we identify the *error of the affine model* based on x_0 at x^*, bounded by $\frac{\gamma}{2}|x_0 - x^*|^2$. Therefore

$$|x_1 - x^*| \leq \frac{\gamma}{2|f'(x_0)|}|x_0 - x^*|^2 \leq \frac{\gamma}{2\rho}|x_0 - x^*|^2.$$

The distance is contracted if $\frac{\gamma}{2\rho}|x_0 - x^*| < 1$, or

$$|x_0 - x^*| \leq \frac{2\rho}{\gamma}\tau,$$

and contraction is obtained if one starts from the ball of radius:

$$\eta = \tau\frac{2\rho}{\gamma} \text{ (possibly reduced to fit the interval } D).$$

The above theorems guarantee **convergence in a fast (quadratic) manner, provided that one starts already sufficiently close to the target root**. This can be the case if a good approximation of the solution is already obtained, but, in general, one starts far away and does not have any guarantee that the starting point will be such that the steps will eventually lead to the solution.

The issue is **global convergence**. In practice, in the absence of strong guarantees, many techniques are **hybrid**, using Newton method when it works, otherwise falling back to a slower but safe global method, like the **bisection** method, as illustrated in Fig. 26.5.

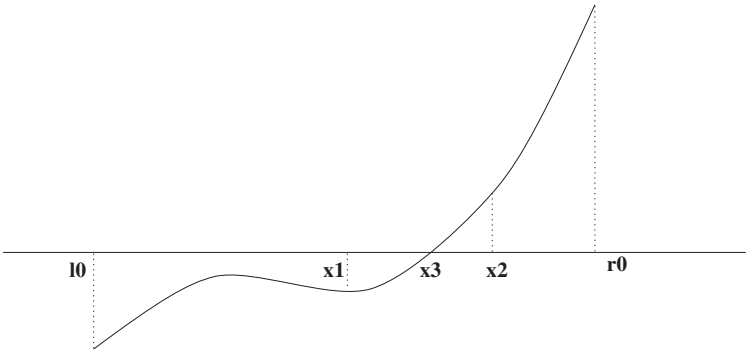


Figure 26.5: The bisection method. The initial interval is divided into two equal parts. One of the two sub-intervals is chosen depending on a test at the middle point. The subdivision is repeated for the chosen interval...

In the **bisection method for root finding**, one looks for a root of a continuous function by subdividing an initial interval (from l_0 to r_0) into two equal parts, observing the f value at the middle point x_1 and then continuing the search by considering only the left or the right sub-interval (while of course maintaining the invariant that the picked sub-interval contains the root). The function is continuous (smoothness implies continuity) and therefore it cannot have abrupt jumps. If the value is negative at l_0 and positive at r_0, there must be an internal point with value zero. If the value at x_1 is negative, at least one root must be in the right sub-interval. If the value at x_1 is positive, at least one root must be in the left sub-interval. One picks the appropriate sub-interval and iterates.

The bisection method is simple and effective, and it converges in a logarithmic number of steps. In fact, each step divides the interval into two equal parts, so that the length of the interval is divided by 2^s after s steps. It is unfortunate that this simple method is not easily extended to more than one dimension.

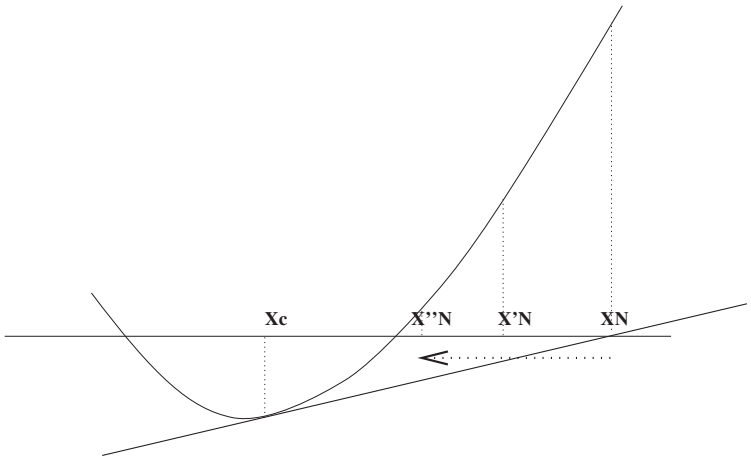


Figure 26.6: Backtracking: Newton step gives the direction.

1. **function** `hybrid_quasi_newton` ($f : \mathbb{R} \to \mathbb{R}$, x_0)
2. **while not** finished
3. Make local model of f around x_k, find x_N that solves the model;
4. **if** x_{k+1} is acceptable **then** move
5. **else** pick x_{k+1} by using a safe global strategy.

Figure 26.7: The hybrid quasi-Newton algorithm.

Fig. 26.6 illustrates the idea of **backtracking**: if Newton's step leads too far, beyond the position of the root, one reverts the direction coming back closer to the root position. One moves from Newton point x_N towards the starting point x_c until one finds x_+ with $|f(x_+)| < |f(x_c)|$.

A generic scheme for **hybrid methods** is to combine global convergence and fast local convergence, as illustrated in Fig. 26.7. One should try Newton step first but always insisting that the iteration decreases some measure of the closeness to a solution.

26.2.1 Derivatives can be approximated by the secant

If derivatives are not available or they are too costly to calculate, one can approximate them with the *secant* passing through two points (with a finite-difference approximation). The **secant method** consists of using the previous iterate x_- as follows:

$$a_c = \frac{f(x_c) - f(x_-)}{x_c - x_-}.$$

A convergence theorem is valid although the convergence rate is now slower (linear).

Theorem 2 *Let $f : D \longrightarrow \mathbb{R}$ for open interval D, $f' \in Lip_\gamma(D)$ (Lipschitz), $|f'(x)| \geq \rho$ (derivative bounded away from zero) in D.*

If $f(x) = 0$ has a solution $x^* \in D$, then there exist positive constants η, η' such that if $0 < |h_k| \leq \eta'$ and if $|x_0 - x^*| < \eta$, then the sequence

$$x_{k+1} = x_k - \frac{f(x_k)}{a_k}\ ,\ a_k = \frac{f(x_k + h_k) - f(x_k)}{h_k}$$

converges $q - linearly$ *to* x^*.

The lesson is that methods based on derivatives can often be used as starting points to develop methods without derivatives. The approximations will sacrifice some efficiency but convergence can still be obtained.

26.2.2 One-dimensional minimization

Up to now we discussed finding a root, a point where the value of f is equal to zero. To minimize a differentiable function one starts from this necessary condition [1]: the minimum must be at a point with $f'(x^*) = 0$. It all amounts to finding a *root* of the derivative function and we now know how to solve it! We can use the Hybrid Newton's method, plus the requirement that $f(x_k)$ decreases. After substituting the original function f with the first derivative f' one obtains

$$x_+ = x_c - \frac{f'(x_c)}{f''(x_c)}.$$

Note that the affine model of f' implies a **quadratic model** of f around x_c:

$$m_c(x) = f(x_c) + f'(x_c)(x - x_c) + \frac{1}{2}f''(x_c)(x - x_c)^2.$$

The iteration will converge locally and Q-quadratically to x^* of $f(x)$ if $f'''(x^*) \neq 0$ and f'' satisfies the Lipschitz condition near x^*. If it is necessary, one backtracks until $f(x_+) < f(x_c)$.

26.3 Solving models in more dimensions (positive definite quadratic forms)

Before using local quadratic models for optimization, let's increase our motivation by confirming that these local models can actually be *solved*. Let's now consider more than one variable. Solving the local quadratic model amounts to solving a quadratic form. An plot of a quadratic positive-definite form is shown in Fig. 26.8.

Newton's method now requires that the **gradient** of the model be equal to zero. Given a step s the quadratic model is

$$Q(s) = \sum_{i=1}^{n} g_i s_i + \sum_{i=1}^{n}\sum_{j=1}^{n} H_{ij} s_i s_j \equiv g^T s + \frac{1}{2} s^T H s.$$

After deriving the gradient, one demands

$$\nabla Q(s) = 0 = g + Hs; \tag{26.3}$$

$$Hs^N = -g \quad \text{(Newton equation)} . \tag{26.4}$$

The solution of the linear system can be found in one step of cost $O(n^3)$ for the matrix inversion [2].

Because of the finite-precision computation carried out by computers one has to deal with issues of **numerical stability**: with some techniques the errors accumulate in a dangerous way, and one may end up with a numerical solution which is wildly different from the exact mathematical solution (reachable only if real numbers could be represented with infinite precision in computers).

Ill-conditioning is a term used to measure how the solution is sensitive to changes in the data (because of finite precision computation). Fig. 26.9 shows an example in two dimensions (two similar equations corresponding to almost parallel lines in the plane).

[1] Sufficient additional condition is $f''(x^*) > 0$, e.g., use Taylor series with remainder:

$$f(x) - f(x^*) = f'(x^*)(x - x^*) + \frac{1}{2}f''(\bar{x})(x - x^*)^2.$$

[2] Actually matrix inversion can be done with $O(n^{\log_2 7})$ or even better asymptotic requirements with more refined techniques, which are nonetheless not often used in practice because of complexity and numerical computing issues.

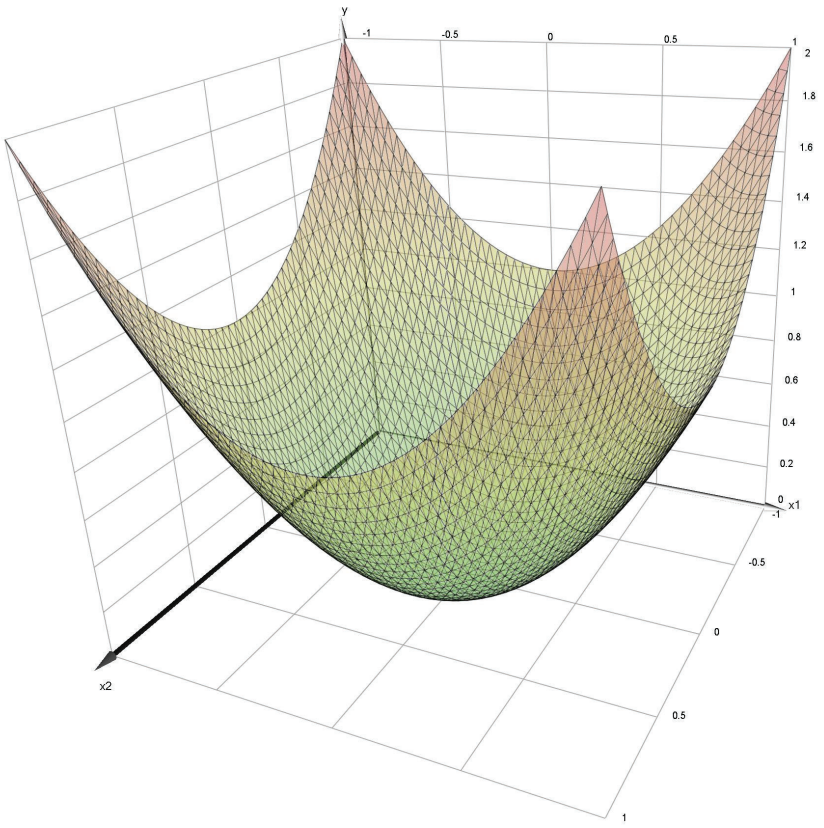


Figure 26.8: Quadratic positive definite f of two variables.

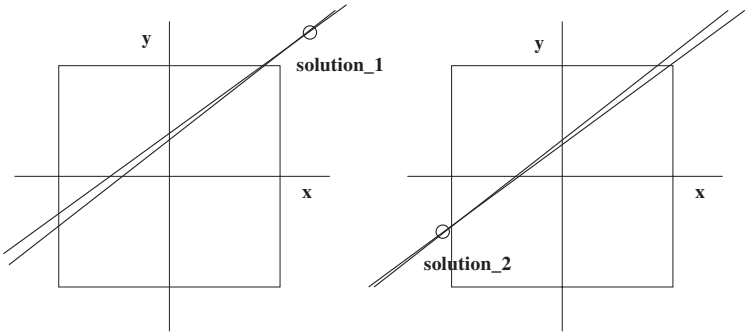


Figure 26.9: Ill-conditioning: solution is very sensitive to changes in the data. In this case two linear equations are very similar and a small change in the line direction is sufficient to shift the solution by a large amount.

In detail, one introduces the **condition number** $\kappa(H)$ of a matrix H defined as $\|H\|\,\|H^{-1}\|$, where $\|*\|$ is the matrix operator norm induced by the vector norm: $\|H\| = \max_x(\|Hx\|/\|x\|)$. The conditioning number is the ratio of the maximum to the minimum stretch induced by H and measures the **sensitivity of the solution of a linear system to finite-precision arithmetic**. If a linear system $Hx = b$ is perturbed in the following way with an error proportional to ϵ:

$$(H + \epsilon F)s(\epsilon) = g + \epsilon f, \tag{26.5}$$

the relative error in the solution can be bounded as:

$$\frac{\|s(\epsilon) - s\|}{\|s\|} \leq \kappa(H)\,(\frac{\|\epsilon F\|}{\|H\|} + \frac{\|\epsilon f\|}{\|g\|}) + O(\epsilon^2).$$

For the case of symmetric and positive definite matrices, an extremely stable triangular decomposition can be found with **Cholesky factorization**. Writing H (symmetric positive definite) as

$$H = LDL^T,$$

with L unit lower-triangular, D diagonal with strictly positive elements (LDL^T factorization).

Because the diagonal is strictly positive:

$$H = LD^{1/2}D^{1/2}L^T = \bar{L}\bar{L}^T = R^T R,$$

where R is a general upper triangular, the Cholesky factor can be considered the **"square root" of the matrix** H, a generalization of the usual square root for the case of matrices.

R can be computed directly from the element-by-element equality:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} r_{11} & & & \\ r_{21} & r_{22} & & \\ \vdots & \vdots & \ddots & \\ r_{n1} & r_{n2} & \dots & r_{nn} \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ & r_{22} & \dots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{pmatrix}.$$

Let's equate element (1,1):

$$a_{11} = r_{11}^2, \;\; r_{11} = \sqrt{a_{11}},$$

continue with first row:

$$a_{12} = r_{11}r_{12}, \;\; a_{13} = r_{11}r_{13}, \dots.$$

After the entire first row is known, start with second row, element:

$$a_{22} = r_{12}^2 + r_{22}^2.$$

The above process needs about $\frac{1}{6}n^3$ multiplications and additions and n square roots (which are avoided if LDL^T is used). No dramatic growth in the elements of R occurs because the following holds:

$$a_{kk} = r_{1k}^2 + r_{2k}^2 + \dots + r_{kk}^2.$$

Now the original equation becomes

$$R^T Rs = g, \tag{26.6}$$

and it can be solved by back-substitution (repeatedly solving for one variable and substituting into the remaining equations). Peel off factors in this way:

$$R^T s_1 = -g \quad \text{use forward substitution;} \tag{26.7}$$

$$Rs = s_1 \quad \text{use backward substitution.} \tag{26.8}$$

The cost for solving the equation is $O(n^2)$ and therefore the dominant cost is in the factorization.

Figure 26.10: Two gradient-descent experts on the mountains surrounding Trento, Italy.

26.3.1 Gradient or steepest descent

In many cases, finding the minimum of the quadratic model by matrix inversion is not the most efficient and robust manner if the linear system becomes very large, a frequent case in machine learning. Furthermore, in many cases the H matrix of second partial derivatives is not available or it is too costly to calculate.

In all these cases **gradient descent** is a possible simple strategy to gradually improve a starting solution aiming at a locally optimal configuration.

If the gradient is different from zero, and one moves along the negative gradient:

$$x_+ = x_c - \epsilon \nabla f,$$

considering the Taylor expansion of equation 4.4, there is a sufficiently small ϵ so that the function decreases $f(x_+) < f(x_c)$. Although naive and requiring some care to choose a small ϵ value, the above technique is used in many different applications (see for example the popular error back-propagation method for training neural networks in Chapter 10).

Steepest descent has very natural and intuitive interpretations. A drop of water on a surface moves according to the local gradient scouting for local minima, at least approximately. Skiers, like those in Fig. 26.10, know very well the meaning of steepest descent, and the fact that skis must be positioned perpendicularly to the gradient to stop. Discrete analogies of steepest descent have been encountered in Chapter 24 in the form of *local search*. The idea is that a search process decides about a local step by sampling the function values at neighboring configurations and then deciding. **No form of global vision is available to guide the search, only local information**.

In addition to being a descent direction, it is well known that the negative gradient $-g$ is the direction of *fastest* descent. A method which looks promising is to execute a one-dimensional minimization while moving along the gradient direction:

$$\min_t Q(x_c - gt).$$

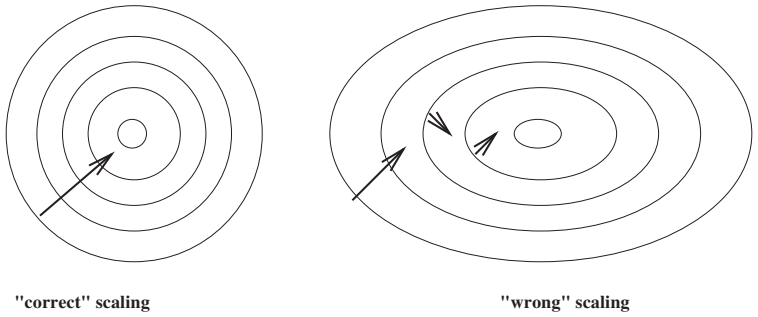


Figure 26.11: The gradient is not always an appropriate direction when searching for a minimizer.

Unfortunately, the intuition is wrong: in many cases spending a lot of effort in minimizing along the gradient direction is not the best way to solve a minimization problem.

The problem is caused by the fact that, when the matrix is *ill-conditioned*, the gradient direction does not point towards the optimal value but tends more and more to point in a perpendicular direction! Ill-conditioning in two dimensions can be visualized by thinking about contour lines becoming more and more *stretched* along a specific direction, see Fig. 26.11. When one follows the gradient, the resulting trajectory *zigzags* and the time to reach the minimum increases.

It can be shown that, when steepest descent is used to minimize a quadratic function $Q(s) = g^T s + \frac{1}{2} s^T H s$ (H symmetric and positive definite) the convergence can become very slow. In detail, by using the condition number κ, when κ increases, the difference between the current value and the best value is multiplied at each iteration by a number which tends to 1:

$$\begin{aligned} |Q(s_{k+1}) - Q(s_*)| &\approx \left(\frac{\eta_{max} - \eta_{min}}{\eta_{max} + \eta_{min}}\right)^2 |Q(s_k) - Q(s_*)| \\ &\approx \left(\frac{\kappa - 1}{\kappa + 1}\right)^2 |Q(s_k) - Q(s_*)|. \end{aligned}$$

If you permit us a far-fetched analogy, the above case may have some implication for life: being greedy and aiming at minimizing as much as possible along an appealing local direction can make one miss more "global" opportunities.

26.3.2 Conjugate gradient

The concept of **non-interfering directions** motivates the conjugate gradient method (CG) for minimization. Two directions are *mutually conjugate* with respect to the matrix H if

$$p_i^T H p_j = 0 \quad \text{when} \quad i \neq j. \tag{26.9}$$

After minimizing in direction p_i, the gradient at the minimizer will be perpendicular to p_i. If a second minimization is in direction p_{i+1}, the change of the gradient along this direction is $g_{i+1} - g_i = \alpha H p_{i+1}$ (for some constant α). The matrix H is indeed the Hessian, the matrix containing the second derivatives, and in the quadratic case the model coincides with the original function. Now, if equation (26.9) is valid, this change is *perpendicular* to the previous direction ($p_i^T (g_{i+1} - g_i) = 0$), therefore the *gradient at the new point remains perpendicular* to p_i and the previous minimization is not spoiled. For a quadratic function the conjugate gradient method is guaranteed to converge to the minimizer in at most $(n + 1)$ function and gradient evaluations (at least for infinite-precision calculations). For a general function the steps must be iterated until a suitable approximation to the minimizer is obtained.

Let us introduce the vector $y_k = g_{k+1} - g_k$. The first search direction p_1 is given by the negative gradient $-g_1$. Then the sequence x_k of approximations to the minimizer is defined by:

$$x_{k+1} = x_k + \alpha_k p_k, \tag{26.10}$$

$$p_{k+1} = -g_{k+1} + \beta_k p_k, \tag{26.11}$$

where g_k is the gradient, α_k is chosen to minimize E along the search direction p_k and β_k is given by:

$$\beta_k = \frac{y_k^T g_{k+1}}{g_k^T g_k} \qquad \text{(Polak-Ribiere choice)}, \tag{26.12}$$

or by:

$$\beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k} \qquad \text{(Fletcher-Reeves choice)}. \tag{26.13}$$

The different choices coincide for a quadratic function [331]. A major difficulty with the above forms is that, for a general function, the obtained directions are *not* necessarily descent directions and numerical instability can result.

The use of a *momentum* term to avoid oscillations in *back-propagation* [311] can be considered as an approximated form of conjugate-gradient.

26.4 Nonlinear optimization in more dimensions

Let's now consider the convergence properties of Newton's method in more dimensions. The method consists of solving the quadratic model:

$$m_c(x_c + p) = f(x_c) + \nabla f(x_c)^T p + \frac{1}{2} p^T \nabla^2 f(x_c) p,$$

and iterating, as shown in Fig. 26.12.

If the initial point is *close* to the minimizer x^* and $\nabla^2 f(x^*)$ is positive definite, the method converges Q-quadratically to x^*.

The possible problems arise if:

- the Hessian is not positive definite: there are directions of negative curvature $p^T H p < 0$, which means that the quadratic local model can assume arbitrarily large negative values when the step length along p increases to infinity;
- the Hessian is singular or ill-conditioned, leading to the impossibility or numerical difficulty of inverting the matrix.

The above problems lead to what are called **Modified Newton's methods**, which change the local model to obtain a sufficiently positive-definite and non-singular matrix. Furthermore they deal with global convergence, indefinite H, and iterative approximations to H. The method is to **combine a fast tactical local method with a robust strategic method** to assure global convergence.

26.4.1 Global convergence through line searches

Global convergence is obtained by adopting line searches along the identified direction: one tries Newton's method *first* and then possibly *backtracks*. Of course one needs to ensure that the direction is indeed a *descent direction*! Fortunately, if H (that is symmetric) is positive definite, Newton's direction *is* a descent direction:

$$\frac{df}{d\lambda}(x_c + \lambda s^N) = \nabla f(x_c)^T s^T = -\nabla f(x_c)^T H_c^{-1} \nabla f(x_c) < 0.$$

1. **function** `multi_dimensional_newton` ($f : \mathbb{R}^n \to \mathbb{R}$, $\boldsymbol{x}_0 \in \mathbb{R}^n$)
2. ⌈ *f is twice continuously differentiable*
3. | **while not** finished
4. | ⌈ solve $\nabla^2 f(\boldsymbol{x}_c) s^N = -\nabla f(\boldsymbol{x}_c)$;
5. | ⌊ $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + s^N$.

Figure 26.12: Newton method in more dimensions.

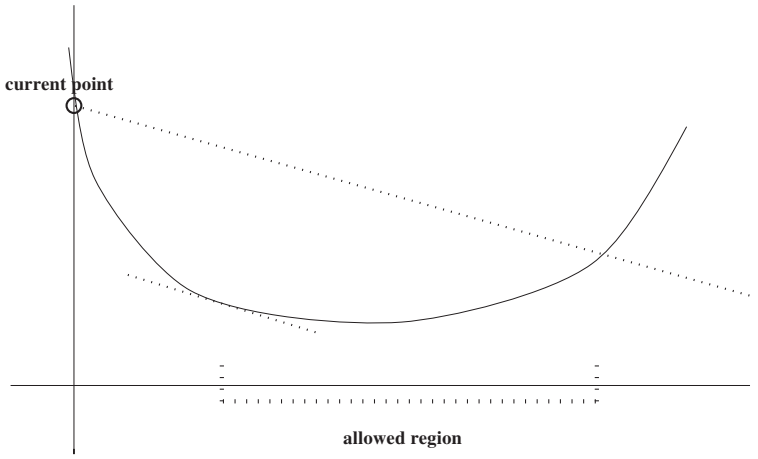


Figure 26.13: Armijo - Goldstein conditions.

If the Hessian has to be approximated, for sure one wants to maintain the *symmetry* and *positive definiteness*, so that the descent direction is guaranteed.

A way to ensure global convergence is to demand that the f value decreases by a sufficient amount with respect to the step length, that the step is long enough and that the search direction is kept away from being orthogonal to the gradient. A popular way to guarantee the above points is by Armijo and Goldstein conditions [157], also illustrated in Fig. 26.13:

1. $$f(x_c + \lambda_c p) \leq f(x_c) + \alpha \lambda_c \nabla f(x_c)^T p,$$
   where $\alpha \in (0, 1)$ and $\lambda_c > 0$;
2. $$\nabla f(x_c + \lambda_c p)^T p \geq \beta \nabla f(x_c)^T p,$$
   where $\beta \in (\alpha, 1)$.

If the Armijo-Goldstein conditions are satisfied at each iteration and if the error is bounded below, one has the following **global convergence** property:

$$\lim_{k \to \infty} \nabla f(x_c) = 0,$$

provided that each step is away from orthogonality to the gradient:

$$\lim_{k \to \infty} \nabla f(x_c) s_k / \|s_k\| \neq 0.$$

If the Armijo-Goldstein conditions are maintained, one can use **fast approximated one-dimensional searches** without losing global convergence [115].

26.4.2 Cure for indefinite Hessians

If the Hessian is indefinite one can use the **modified Cholesky** method. Let's consider the spectral decomposition:

$$H = U\Lambda U^T = \sum_{i=1}^{n} \eta_i u_i u_i^T,$$

where Λ is diagonal and Λ_{ii} is the eigenvalue η_i.

It is easy to see what will happen if η_i are negative (no minimum exists: values can grow to minus infinity) or close to zero (the inverse will have eigenvalues close to infinity).

If H is not positive definite or it is ill-conditioned one remedies in a very direct way by adding a simple diagonal matrix:

$$H' = \nabla f(x_c) + \mu_c I, \;\; \mu_c > 0$$

to correct the Hessian so that $\nabla^2 f(x_c) + \mu_c I$ is positive definite and well conditioned.

This leads to the **modified Cholesky factorization**: one finds the Cholesky factors of a different matrix $\bar{H}_c$, differing only by a diagonal matrix K with non-negative elements:

$$\bar{H}_c = LDL^T = H_c + K,$$

where all elements in D are positive and all elements of L are uniformly bounded

$$d_k > \delta, \;\; |l_{ij}|\sqrt{d_k} \leq \beta,$$

see [147] for an appropriate choice of β. The modified Cholesky factorization is used to correct Hessian [115, 20] so that $\nabla^2 f(x_c) + \mu_c I$ is positive definite and well conditioned.

This amounts to adding a positive definite quadratic form to our original model. The effect is that large steps tend to be penalized.

26.4.3 Relations with model-trust region methods

The previous techniques were based on finding a *search direction* and moving by an acceptable amount in that direction ("step-length-based methods").

Because the last modification consisted of adding to the local model a quadratic term:

$$m_{modified}(x_c + s) = m_c(x_c + s) + \mu_c s^T s,$$

one may suspect that minimizing the new model is equivalent to minimizing the original one with the constraint that the step s not be too large.

This can be executed by choosing first the maximum step length and then using the full (and not one-dimensional) quadratic model to determine the appropriate direction. In **model-trust region methods** the model is trusted only within a region, that is updated by using the experience accumulated during the search process.

Theorem 3 *Suppose that we are looking for the step s_c that solves:*

$$\min m_c(x_c + s) \;\; = \;\; f(x_c) + \nabla f(x_c)^T s + \frac{1}{2} s^T H_c s \tag{26.14}$$

$$\textit{subject to } \|s\| \leq \delta_c. \tag{26.15}$$

The above problem is solved by:

$$s(\mu) = -(H_c + \mu I)^{-1} \nabla f(x_c), \tag{26.16}$$

for the unique $\mu \geq 0$ such that the step has the maximum allowed length ($\|s(\mu)\| = \delta_c$), unless the step with $\mu = 0$ is inside the trusted region ($\|s(0)\| \leq \delta_c$), in which case $s(0)$, the Newton step, is the solution.

The diagonal modification of the Hessian is a **compromise between gradient descent and Newton's method**: when μ tends to zero the original Hessian is (almost) positive definite and the step tends to coincide with Newton's step; when μ has to be large the diagonal addition μI tends to dominate and the step tends to one proportional to the negative gradient:

$$s(\mu) = -(H_c + \mu I)^{-1} \nabla f(x_c) \approx -\frac{1}{\mu} \nabla f(x_c).$$

There is no need to decide from the beginning, the algorithm selects in an adaptive manner the move that is appropriate to the local configuration of the error surface.

26.4.4 Secant methods

Secant techniques are useful if the Hessian is not available or costly to calculate.

In one dimension the second derivative can be approximated with the slope of the secant through the values of the first derivatives at two near points:

$$\frac{d^2 f(x)}{dx^2}(x_2 - x_1) \approx \left(\frac{df(x_2)}{dx} - \frac{df(x_1)}{dx} \right). \tag{26.17}$$

In more dimensions one equation is not sufficient. Let the current and next point be x_c and x_+, respectively, and let's define $s_c = x_+ - x_c$ and $y_c = \nabla f(x_+) - \nabla f(x_c)$ (difference of gradients). The analogous "secant equation" is

$$H_+ s_c = y_c. \tag{26.18}$$

The above equation does not determine a unique H_+ but leaves the freedom to choose from a $(n^2 - n)$ dimensional affine subspace $Q(s_c, y_c)$ of matrices obeying equation (26.18).

A possibility to cure this issue is to use the previous "history." In other words, equation (26.18) will not be used to *determine* but to *update* a previously available approximation.

In particular (Broyden method), one can use a *least change* principle, finding the matrix in $Q(s_c, y_c)$ ("quotient") that is closest to the previously available matrix. This is obtained by *projecting* the matrix onto $Q(s_c, y_c)$, in the Frobenius norm (matrix as a long vector).

The resulting Broyden's update is

$$(H_+)_1 = H_c + \frac{(y_c - H_c s_c) s_c^T}{s_c^T s_c}. \tag{26.19}$$

Unfortunately, Broyden's update does not guarantee a *symmetric* matrix (remember that we want *descent* directions).

Projecting Broyden's matrix onto the subspace of *symmetric* matrices is not enough: the obtained matrix may be out of $Q(s_c, y_c)$.

Fortunately, if the two above projections are repeated, the obtained sequence $(H_+)_t$ converges to a matrix that is both symmetric and in $Q(s_c, y_c)$. This is the *symmetric* secant update of Powell:

$$H_+ = H_c + \frac{(y_c - H_c s_c) s_c^T + s_c (y_c - H_c s_c)^T}{s_c^T s_c} - \frac{< y_c - H_c s_c, s_c > s_c s_c^T}{(s_c^T s_c)^2}. \tag{26.20}$$

We are closer to a satisfactory update, but we insist on a *positive definite* approximation of the Hessian. The matrix H_+ is symmetric and positive definite if and only if $H_+ = J_+ J_+^T$, for some non-singular J_+. A proper update can be obtained by using Broyden's method to derive a suitable J_+.

The resulting update is historically known as the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update [68] and is given by:

$$H_+ = H_c + \frac{y_c y_c^T}{y_c^T s_c} - \frac{H_c s_c s_c^T H_c}{s_c^T H_c S_c}. \tag{26.21}$$

The positive-definite secant update converges $q - superlinearly$ [68].

It is possible to take the initial matrix H_0 as the identity matrix, so that the first step is along the negative gradient.

ϵ	Learning rate
$\bar{\epsilon}$	Average learning rate
$\boldsymbol{w}_{\text{curr}}$	Weights
$\boldsymbol{d}$	Search direction

```
1.  procedure oss_minimize
2.  ⌈ begin_or_restart
3.  | ϵ ← 10^−5
4.  | ϵ̄ ← 10^−5
5.  | w_curr ← random initial weights
6.  | iterations ← 1
7.  | while convergence criterion is not satisfied
8.  | ⌈ if iterations is multiple of N
9.  | |    begin_or_restart
10. | | iterations ← iterations + 1
11. | | d ← find_search_direction                    See Eq. (26.22)
12. | | if fast_line_search( d) = false
13. | ⌊ ⌊    begin_or_restart
14. procedure begin_or_restart
15. ⌈ find the current energy value
16. | ϵ ← ϵ̄
17. | d ← −g
18. ⌊ fast_line_search( d)
```

Figure 26.14: The one-step secant algorithm (Part I), from [19].

26.4.5 Closing the gap: second-order methods with linear complexity

Computing the exact Hessian requires order $O(n^2)$ operations and order $O(n^2)$ memory to store the Hessian components, in addition the solution of the equation to find the step (or search direction) in Newton's method (see Fig. 26.12) requires $O(n^3)$ operations, at least when using traditional linear algebra routines. Fortunately, some second-order information can be calculated by starting from the last gradients, and therefore reducing the computation and memory requirements to find the search direction to $O(n)$. The term "secant methods" used in [115] is reminiscent of the fact that derivatives are approximated by the secant through two function values.

Historically the one-step-secant method *OSS* is a variation of what is called *one-step (memory-less) Broyden-Fletcher-Goldfarb-Shanno* method, see [331]. The **OSS method** has been used for multilayer perceptrons in [19] and [31]. The main procedures are illustrated in Fig. 26.14–26.15.

Note that BFGS (see [376]) stores the whole approximated Hessian, while the *one-step* method requires **only vectors** computed from gradients. In fact, the new search direction p_+ is obtained as:

$$p_+ = -g_c + A_c s_c + B_c y_c, \tag{26.22}$$

where the two scalars A_c and B_c are the following combination of scalar products of the previously defined vectors s_c, g_c and y_c (last step, gradient and difference of gradients):

$$A_c = -\left(1 + \frac{y_c^T y_c}{s_c^T y_c}\right)\frac{s_c^T g_c}{s_c^T y_c} + \frac{y_c^T g_c}{s_c^T y_c} \;\; ; \;\; B_c = \frac{s_c^T g_c}{s_c^T y_c}.$$

The search direction is the negative gradient at the beginning of learning and it is restarted to $-g_c$ every N steps (N being the number of weights in the network).

The fast one-dimensional minimization along the direction p_c is crucial to obtain an efficient algorithm. This part of the algorithm (derived from [115]) is described in Fig. 26.15. The one-dimensional search is based on the backtracking strategy. The last successful learning rate λ is increased ($\lambda \leftarrow \lambda \times 1.1$) and the first tentative step is executed. To use the same notation as that of Fig. 26.14–26.15 let us denote with E ("energy") the function to be optimized. If the new value E is not below the "upper-limiting" curve, then a new tentative step is tried by using successive quadratic interpolations until the requirement is met. Note that the learning rate is decreased by L_{decr} after each unsuccessful trial. Quadratic interpolation is not wasting computation. In fact, after the first trial one has exactly the information that is needed to fit a parabola: the value of E_0 and E_0' at the initial point and the value of E_λ at the trial point. The parabola $P(x)$ is

$$P(x) = E_0 + E_0' x + \left[\frac{E_\lambda - E_0 - \lambda E_0'}{\lambda^2}\right] x^2, \tag{26.23}$$

and the minimizer λ_{min} is

$$\lambda_{min} = \frac{-E_0'}{2\left[\frac{E_\lambda - E_0 - \lambda E_0'}{\lambda^2}\right]} \leq \frac{1}{2(1 - G_{\text{decr}})}\lambda. \tag{26.24}$$

If the "gradient-multiplier" G_{decr} in Fig. 26.14 is 0.5, the λ_{min} that minimizes the parabola is less than λ.

26.5 Constrained optimization: penalties and Lagrange multipliers

Imagine that you are the owner of a factory and you want to maximize production. With unconstrained optimization your workers may complain if asked to work day and night without pauses, lunches and vacation. A reasonable constraint can be that each worker has to work for exactly 48 hours per week. Truck drivers have safety requirements on the number of continuous driving hours before a break is required, otherwise they risk falling asleep while driving (e.g., drive for no more that 6 hours).

A general **constrained minimization problem** is:

$$\begin{array}{rll} \min & f(\mathbf{x}) & \\ \text{subject to} & g_i(\mathbf{x}) = c_i \quad \text{for } i = 1, \ldots, n & \text{Equality constraints} \\ & h_j(\mathbf{x}) \geqq d_j \quad \text{for } j = 1, \ldots, m & \text{Inequality constraints} \end{array} \tag{26.25}$$

where $g_i(\mathbf{x}) = c_i$ for $i = 1, \ldots, n$ and $h_j(\mathbf{x}) \geq d_j$ for $j = 1, \ldots, m$ are constraints that are required to be satisfied. **Hard constraints** need to be satisfied for a solution to be feasible. Even for smooth objective functions, hard constraints complicate matters by introducing unsurmountable walls in the input parameter space.

Some ways to take care of constraints are problem-specific, e.g. they are easily handled in Linear and Quadratic Programming tasks (Chapter 33). But there are two simple general-purpose ways to address constraints. Both ways **transform an optimization problem with constraints into an unconstrained one** and are widely used.

The first method considers that **hard constraints are very rare in practice**. 48 working hours can be surpassed in some cases provided that overtime is paid more than the standard rate. In engineering, all physical measures are in any case subject to stochastic errors. If a truck on a freeway cannot weigh more than 10 tons, a policeman will hopefully not confiscate it if the maximum weight is surpasses by one gram. **Soft constraints** can be violated but there is a **penalty** to be paid, henceforth the name of **penalty method**. The harder the constraint, the higher the penalty.

For each equality constraint a quadratic penalty can be added for violation, leading to a **penalized objective function**:

$$\min f(\mathbf{x}) + \sum_i \gamma_i (g_i(\mathbf{x}) - c_i)^2 \tag{26.26}$$

If the constraint is not satisfied, a penalty proportional to γ_i times the (quadratic) violation is paid. Of course, different penalties are possible, e.g., absolute values, logarithmic, or different formulas for inequality constraints.

One may be tempted to set γ_i to a huge positive value to bring the constraint very close to perfect satisfaction. In fact, even a small violation of the constraint will cause an explosion of the penalty in the objective function. Unfortunately, very large constraints imply that very steep walls (although not perpendicular) will be created in the penalized objective, creating numerical problems and difficulties in the practical minimization. In practice, the proper γ_i values are a result of a tradeoff between minimizing f and accepting some violation. One can start with tentative values, see the results, discuss acceptability of constraint violation, repeat with different γ_i values until satisfied. Multiple-objective optimization (Chapter 40) gives equal standing to the initial objective and to the constraint violations, leading to a more systematic management of the solution.

If the function is smooth and has partial derivatives, a second possibility in mathematical optimization is the method of **Lagrange multipliers**. The problem is transformed into an unconstrained one by adding each constraint multiplied by a parameter λ_i (a Lagrange multiplier).

$$\min f(\mathbf{x}) + \sum_i \lambda_i g_i(\mathbf{x}) \tag{26.27}$$

Minimizing the transformed function yields a *necessary* condition for optimality. Additional checks are therefore necessary (for example, the identified point can be a saddle point and not a global minimum), but in many cases, in the presence of a single global optimum, the method of **Lagrange multipliers** will deliver the correct solution.

Let's develop some intuition with a graphical analysis.

Consider a two-dimensional problem:

$$\begin{array}{ll} \text{maximize} & f(x,y) \\ \text{subject to} & g(x,y) = c. \end{array}$$

We can visualize contours of f given by

$$f(x,y) = d$$

for various values of d and the contour of g given by $g(x,y) = c$, as shown in Fig. 26.17.

Suppose we walk along the contour line with $g = c$. In general the contour lines of f and g may be distinct, so the contour line for $g = c$ will intersect with or cross the contour lines of f. This is equivalent to saying that while moving along the contour line for $g = c$ the value of f can vary. Only when the contour line for $g = c$ meets contour lines of f tangentially, do we neither increase nor decrease the value of f — that is, when the contour lines touch but do not cross.

The contour lines of f and g touch when the **tangent vectors of the contour lines are parallel**. Since the gradient of a function is perpendicular to the contour lines, this is the same as saying that the gradients of f and g are parallel. Thus we want points (x,y) where $g(x,y) = c$ and

$$\nabla f(x,y) = \lambda \nabla g(x,y).$$

The above analysis is valid for more than two input dimensions. If the two gradients are *not* parallel at the minimizer, a local direction $\Delta\mathbf{x}$ can be found which is perpendicular to the constraint gradient, and therefore keeps the constraint satisfied in the linear approximation of Taylor series, but not perpendicular to the gradient of the original objective function. A small step along $\Delta\mathbf{x}$ (or minus $\Delta\mathbf{x}$) will therefore lead to smaller f values, contradicting the assumption of minimality.

The Lagrange multiplier λ specifies how one gradient needs to be multiplied to obtain the other one!

Because of linearity of the gradient, requiring $\nabla f(x,y) - \lambda \nabla g(x,y) = 0$ is the same as requiring $\nabla \left[f(x,y) - \lambda g(x,y) \right] = 0$. But this is the necessary condition for a stationary point of function:

$$f(x,y) - \lambda g(x,y)$$

the original function minus the constraint multiplied by the Lagrange multiplier.

The above can be generalized for more constraints and for inequality constraints.

A practical application of Lagrange multipliers is in economics. Let's remember that the gradient is used to find a first-order difference when $\mathbf{x}$ is changed, and that the two gradients are parallel and related by the λ^* multiplier. A Lagrange multiplier can be interpreted as the "marginal" change in the optimal value of the objective function (profit) due to **a change (relaxation) of a given constraint**. In such a context λ^* is the marginal cost of the constraint, and is referred to as the **shadow price**.

The shadow price can provide decision-makers with insights. For instance if a constraint limits the amount of labor available to 40 hours per week, the shadow price tells how much you should be willing to pay for an additional hour of labor. If pay more than the shadow price, the increase in in the total production value (the objective function) will be less that your labor cost.

Applications of Lagrange multipliers in ML are for example in the LASSO technique in Section 9.3, of in SVM in Section 12.1.1.

$\boldsymbol{d}$	Search direction
$\boldsymbol{g}$	Function gradient
$\boldsymbol{w}$	Weights
d_l	Projection of $\boldsymbol{d}$ along the gradient
E	Current energy
E_saved	Best energy
ok	Improving step found
trials	Number of iterations
MAXTRIALS	Maximum allowed number of iterations
L_decr	Step decrease at each iteration

```
1.  procedure fast_line_search ( d)
2.      d_l ← g · d
3.      if d_l > 0
4.          d ← - g; d_l ← g · d
5.      E_saved ← E
6.      ε ← L_incr ε; ok ← false; trials = 0
7.      repeat
8.          trials ← trials + 1
9.          w ← w_curr + ε d
10.         E ← E( w)
11.         if E < E_saved + G_decr d_l ε
12.             ok ← true
13.         else
14.             ε_quad ← parabola_minimizer(E_saved, d_l,f)        See Eq. (26.24)
15.             w ← w_curr + ε_quad d
16.             E ← E( w)
17.             if E < E_saved + G_decr d_l ε_quad
18.                 ok ← true; ε ← ε_quad
19.             else
20.                 ε ← L_decr ε
21.     until ok = true or trials > MAXTRIALS
22.     if ok = true
23.         p ← ε d
24.         w_curr ← w
25.         g ← ∇_w E(w)
26.         ε̄ ← 0.9 ε̄ + 0.1 ε
27.     return ok
```

Figure 26.15: The one-step secant algorithm (Part II), from [19]: fast one-dimensional search along the chosen direction $\boldsymbol{d}$.

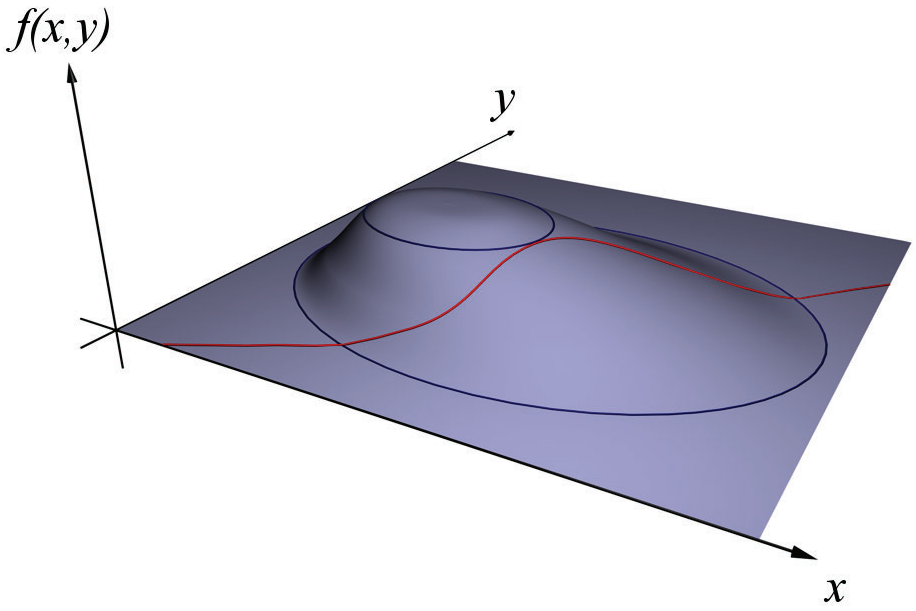


Figure 26.16: Find x and y to maximize $f(x, y)$ subject to a constraint (shown in red) $g(x, y) = c$.

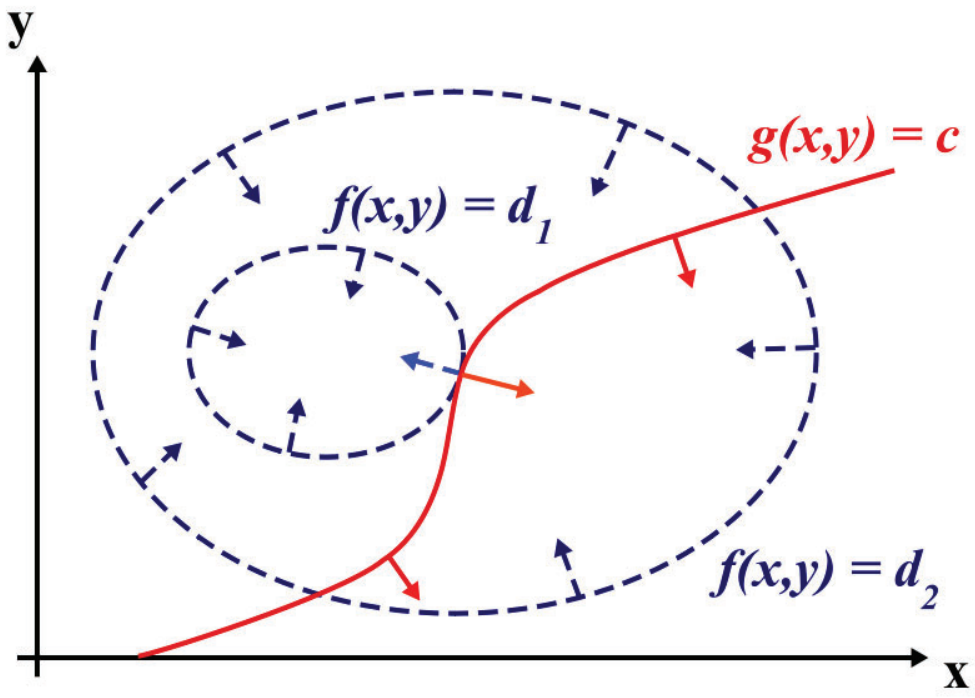


Figure 26.17: Lagrange multipliers.

Gist

Optimization of functions (models) with real numbers as input parameters is an old area, starting approximately during the second world war, and now reaching high levels of sophistication. The purpose is to design **automated techniques to identify inputs leading to maximum (or minimum) output values**.

In spite of the mathematical sophistication, the basis of most optimization techniques is very understandable, even if you do not remember anything from your courses in calculus (the mathematical study of change). A **drop of water** fallen from the sky reaches the sea without any conscious mathematical finesse.

The basic steps are as follows. Start from an initial value for the input parameters. Apply small local changes to the various inputs and test their effects (are they leading to higher or smaller output values?). Based on the test results decide whether to accept the local change or not. Repeat until there is progress, leading to better and better output values.

If one can calculate *derivatives*, one has a simple way to predict the effect of small local changes. In fact, you can consider the **derivative as a local predictor of change**. If the step is sufficiently small, the approximation "*change equals derivative times step*" tends to be very good. If derivatives are not available, one can test small changes directly (like in RAS) and keep **locally adapted models** to reduce wasted function evaluations. Local adaptation occurs by **learning from the previous steps of the search**.

Understanding the principles, even without math theorems, is sufficient to use optimization software with competence, and to avoid most pitfalls. After all, you do not need calculus and mathematical analysis to ski without falling down and with a reasonable guarantee to reach your gondola ski lift.

Part IV

Learning for intelligent optimization

Chapter 27

Reactive Search Optimization (RSO): Online Learning Methods

This then is the first duty of an educator:
to stir up life but leave it free to develop
(Maria Montessori)

After considering basic Local Search and memory-less (Markovian) search, this chapter presents the more advanced schemes for using machine learning to improve optimization,

In many cases a single relevant instance has to be solved, so that online learning schemes for optimization (**Reactive Search Optimization, RSO**) are of particular interest.

Even if the initial optimization problem is black-box, the more points are generated in input space and evaluated, the more knowledge is accumulated, in implicit form. Data about the past history of the search can be exploited to generate internal explicit models and improve the efficiency and effectiveness of the future optimization effort. In a way, RSO tends towards **truly intelligent problem-solving machines, which learn and self-improve the more they work**, in a way similar to humans, or similar to reactive biological systems. Think about the lifelong learning of a violinist, from the first mechanical and "symbolic" rule-based movements, to the real mastery of a Paganini.

The above figure shows an example in the history of bicycle design. Do not expect historical fidelity here, this book is about the LION way and not about bike technology. The first model is a starting solution with a single wheel, it works but it is not optimal yet. The second model is a randomized attempt to add some pieces to the original design, the situation is worse. One could revert back to the initial model and start other changes. But let's note that, if one *insists* and proceeds with a second addition, one may end up with the third model, clearly superior from a usability and safety point of view. This story has a lesson: **local search by small perturbations is a tasty ingredient but additional spices are in certain cases needed** to obtain superior results.

Reactive Search Optimization (RSO) advocates the integration of online machine learning techniques into optimization heuristics. The word *reactive* hints at a ready response to events during the search through an internal feedback loop for **online self-tuning and dynamic adaptation**. In RSO the past history of the search and the knowledge accumulated while moving in the configuration space is used for self-adaptation in an automated manner: the algorithm maintains the internal flexibility needed to address different situations during the search, but the adaptation is automated, and executed *while* the algorithm runs on a single instance and reflects on its past experience. Machine learning is therefore an essential ingredient in the RSO soup, as illustrated in Fig. 27.1.

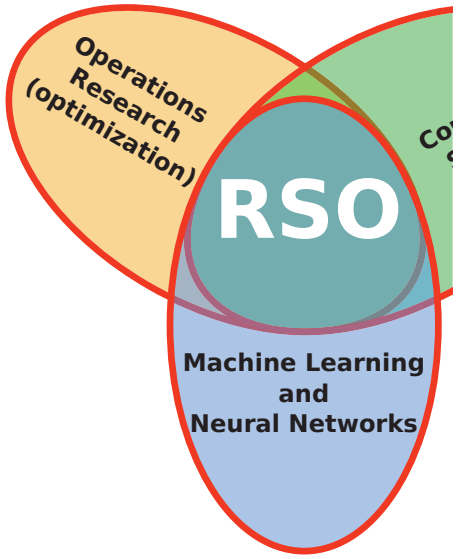


Figure 27.1: RSO is at the intersection of optimization, computer science (algorithms and data structures) and machine learning.

Reactive Search Optimization adopts ideas and methods from machine learning and statistics, in particular reinforcement learning, active or query learning, and neural networks. Let's note that the effort towards adaptive and self-tuning optimization schemes is deeply rooted also in the generation of adaptive probability density functions in stochastic global optimization (Chapter 25) or in the use of derivative-based local models in continuous optimization (Chapter 26). RSO deals with the systematic application of ML into optimization, also for solving a single instance!

The following sections are mostly dedicated to discrete optimization, but with some examples also for functions of real variables. After an introduction (Sec. 27.1), we present some notable ways of reacting from the search history to affect different parameters or critical choices of the search mechanism, in particular reacting on temporary prohibition

periods (Sec. 27.2), adapting the neighborhood in variable-neighborhood search (Chapter 28), iterating local search to escape from an attractor (Chapter 29), self-tuning the temperature in Simulated Annealing (Chapter 30), dynamically adapting fitness surfaces or the amount of stochasticity (Chapter 31).

27.1 RSO: Learning while searching

Let's cite the main motivations for passing from simple local search to Reactive Search Optimization, by summarizing the more extended presentations of the subject [25, 27, 26].

Many problem-solving methods are characterized by a certain number of **choices and free parameters**, whose appropriate setting and tuning is complex. In some cases the parameters are tuned through a feedback loop that includes **the user as a crucial learning component**: different options are developed and tested until acceptable results are obtained. The quality of results is not automatically transferred to different instances and the feedback loop can require a slow "trial and error" process when the algorithm has to be tuned for a specific application. In Machine Learning a rich variety of "design principles" is available that can be used in the area of parameter tuning and optimal choice for heuristics. The lack of human intervention does not imply higher unemployment rates for researchers. On the contrary, one is now loaded with a heavier task: **the algorithm developer must transfer his intelligent expertise into the algorithm** itself, a task that requires the exhaustive description of the tuning phase in the algorithm. The algorithm complexity will increase, but the price is worth paying if the two following objectives are reached.

- **Reproducibility of results** through complete and unambiguous documentation. The algorithm becomes self-contained and its quality can be judged independently from the designer or specific user. This requirement is particularly important for science, in which objective evaluations are crucial. The widespread usage of software archives further simplifies the test and simple re-use of heuristic algorithms.

- **Automation**. The time-consuming handmade tuning phase is now substituted by an automated process, as illustrated in Fig. 27.2. Let us note that only the final user will typically benefit from an automated tuning process. On the contrary, the algorithm designer faces a longer and harder development phase. There are no free meals: complexity does not disappear, it is only shifted from the decision maker to the method (and software) designer.

The metaphors for Reactive Search Optimization derive mostly from the individual human experience. Its motto is "**learning on the job**." As already mentioned, real-world problems have a rich structure. While many alternative solutions are tested in the exploration of a search space, patterns and regularities appear. The human brain quickly learns and drives future decisions based on previous observations. This is the main inspiration source for inserting online machine learning techniques into the optimization engine of RSO. Memetic algorithms share a similar focus on learning, although their concentration is on cultural evolution, describing how societies develop over time, more than on the capabilities of a single individual.

Nature and biology-inspired metaphors for optimization abound today. It is to some degree surprising that most of them derive from genetics and evolution, or from the emergence of collective behaviors from the interaction of simple living organisms which are mostly hard-wired with little or no learning capabilities. One almost wonders whether this is related to ideological prejudices in the spirit of Jean-Jacques Rousseau, who believed that man was good when in the state of nature but is corrupted by society, or in the spirit of the "evil man against nature" principle of commercial Hollywood B-movies. Metaphors can lead us astray from our main path: we are strong supporters of a pragmatic approach, **an algorithm is effective if it solves a problem in a competitive manner without requiring an expensive tailoring**, not because it corresponds to one's favorite elaborate, fanciful or sexy analogies. Furthermore, at least for a researcher, in most cases an algorithm is of scientific interest if there are ways to analyze its behavior and explain *why and when* it is effective. Seminal papers related to using memory in a strategic manner to guide heuristics to continue exploration beyond local minima are the ones by Fred Glover about tabu search, scatter search and path relinking, and related metaheuristics, see for example [150], [153] and the amusing [152]. Other inspiring papers that

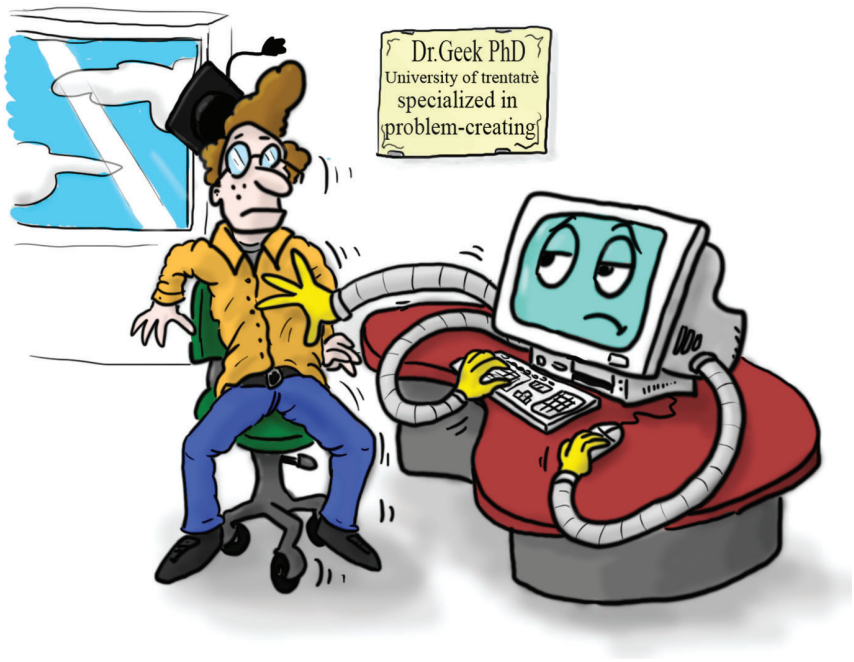


Figure 27.2: Algorithms with self-tuning capabilities like RSO make life simpler for the final user. Complex problem solving does not require technical expertise but is available to a much wider community of final users (adapted from [27]).

you may want to read to get a taste of related topics are [275] about memetic algorithms, [388] about evolving neural networks, [56] about meta-heuristics, and [200] about performance prediction and automated tuning.

27.2 RSO based on prohibitions

The idea of using **prohibitions to encourage creativity and diversification**, i.e., to encourage a decision maker, an engineer, or a designer, to consider radically new alternatives, is deeply rooted in the practice of research. Quoting from Konrad Lorenz, the Austrian Nobel-prize winner and creator of modern ethology, "*it is a good morning exercise for a research scientist to discard a pet hypothesis every day before breakfast. It keeps him young*." What a brilliant illustration of the fact that you want to *prohibit* the consideration of old solutions in order to be truly creative.

In the initial figure, one has to prohibit the consideration of monocycles and insist with local changes to eventually invent bicycles, although the process may include some intermediate inferior designs. Success is fueled by persistence and acceptance of repeated failure.

As mentioned above, local search generates a *trajectory* $X^{(t)}$ of points in the admissible search space. The successor of a point X is selected from a *neighborhood* $N(X)$ that associates to the current point X a subset of $\mathcal{X}$. A point X is *locally optimal with respect to N*, or a *local minimum* if: $f(X) \leq f(Y)$ for all $Y \in N(X)$. For the following discussion we consider the case in which $\mathcal{X}$ consists of binary strings with a finite length L: $\mathcal{X} = \{0,1\}^L$ and the neighborhood is obtained by applying the *elementary moves* $\mu_i (i = 1, ..., L)$ that change the i-th bit of the string $X = [x_1, ..., x_i, ..., x_L]$:

$$\mu_i([x_1, ..., x_i, ..., x_L]) = [x_1, ..., \bar{x}_i, ..., x_L]; \tag{27.1}$$

where $\bar{x}_i$ is the negation of the i-th bit: $\bar{x}_i \equiv (1 - x_i)$.

To avoid entrapment in local attraction basins, one can bias the search toward points with low f values but incorporate **reactive prohibition strategies** to discourage the repetitions of already-visited configurations. Local moves are executed even if f increases with respect to the value at the current point, to exit from local minima of f. But soon as a move is applied, **the inverse move is temporarily prohibited** (the name "tabu search" derives from this prohibition).

In detail, at a given iteration t, the set of moves $\mathcal{M}$ is partitioned into the set $\mathcal{T}^{(t)}$ of the *tabu* moves, and the set $\mathcal{A}^{(t)}$ of the admissible moves. Superscripts with parenthesis are used for quantities that depend on the iteration. At the beginning, the search starts from an initial configuration $X^{(0)}$, that is generated randomly, and all moves are admissible: $\mathcal{A}^{(0)} = \mathcal{M}$, $\mathcal{T}^{(0)} = \emptyset$. The search trajectory $X^{(t)}$ is then generated, by applying the best admissible move $\mu^{(t)}$ from the set $\mathcal{A}^{(t)}$:

$$X^{(t+1)} = \mu^{(t)}(X^{(t)}) \quad \text{where} \quad \mu^{(t)} = \arg\min_{\nu \in \mathcal{A}^{(t)}} f(\nu(X^{(t)})).$$

In isolation, the "modified greedy search" principle *can* generate cycles. For example, if the current point $X^{(t)}$ is a strict local minimum, the cost function at the next point must increase: $f(X^{(t+1)}) = f(\mu^{(t)}(X^{(t)})) > f(X^{(t)})$, and there is the possibility that the move at the next step will be its *inverse* ($\mu^{(t+1)} = {\mu^{(t)}}^{-1}$) so that the state after two steps will come back to the starting configuration

$$X^{(t+2)} = \mu^{(t+1)}(X^{(t+1)}) = {\mu^{(t)}}^{-1} \circ \mu^{(t)}(X^{(t)}) = X^{(t)}.$$

At this point, if the set of admissible moves is the same, the system will be trapped forever in a cycle of length 2. In this example, the cycle is avoided if the inverse move ${\mu^{(t)}}^{-1}$ is prohibited at time $t+1$. In general, the inverses of the **moves executed in the most recent part of the search are prohibited for a period** T. For binary strings a move coincides with its inverse: a move is prohibited if and only if its most recent use has been at time $\tau \geq (t - T^{(t)})$. The period is finite because the prohibited moves can be necessary to reach the optimum in a later phase of the search. In prohibition-based-RSO (tabu-RSO for short) the prohibition period $T^{(t)}$ is time-dependent.

The diversification effect of prohibiting moves on the search trajectory has been clarified by the **fundamental relationships between prohibition and diversification** demonstrated by Battiti in [42]. Let $H(X, Y)$ be the Hamming distance between two strings X and Y, defined as the number of corresponding bits that are different in the two strings. Now, if only allowed moves are executed, and T satisfies $T \leq (n-2)$, which guarantees that at least two moves are allowed at each iteration, one obtains the following.

- The Hamming distance H between a starting point and successive points along the trajectory is strictly increasing for $T+1$ steps:
$$H(X^{(t+\tau)}, X^{(t)}) = \tau \quad \text{for} \quad \tau \leq T+1.$$
- The minimum repetition interval R along the trajectory is $2(T+1)$:
$$X^{(t+R)} = X^{(t)} \;\Rightarrow\; R \geq 2(T+1).$$

The above relationships clearly show how **the prohibition is related to the amount of diversification**: the larger T, the larger is the distance H that the search trajectory must travel before it is allowed to come back to a previously visited point. But T cannot be too large, otherwise a shrinking number of moves will be allowed after an initial phase, leading to less freedom of movement.

The demonstration of the relationships is immediate as soon as one notices that, after a bit is changed, it is "frozen" for the next T iterations. To visualize this behavior, Fig. 27.3 shows the evolution of the configuration $X^{(t)}$, when the function to be optimized is given by $f(X) \equiv number(X)$, where $number(X)$ is obtained by considering X as the standard binary encoding of an integer number. The prohibition T is equal to three.

In a physical analogy, as soon as a bit is changed, an *ice cube* is placed on it so that it will not be changed during the future T iterations. When the period T elapses, the ice cube melts down and the bit can be changed again. The situation in which a bit is "frozen" and cannot be changed is shown with a shaded box in Fig. 27.3. In the example, the configuration starts with the all-zero string, a locally optimal point. At iteration 0, the best move changes the least

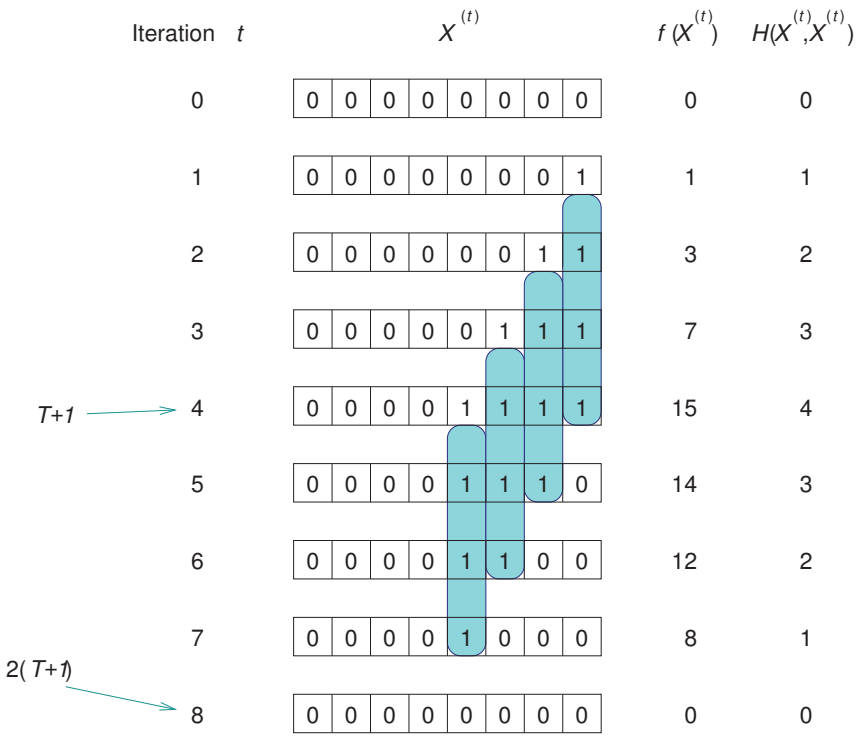


Figure 27.3: An example of the relationship between prohibition T, and diversification measured by the Hamming distance $H(X^{(t)}, X^{(0)})$. T = 3 in the example. Figure adapted from [42].

significant bit. At iteration 1 the least significant bit is frozen and the best *allowed* move changes the second bit. The maximum Hamming distance is reached at iteration $(T + 1)$, then the distance decreases and the initial configuration is repeated at iteration $2(T + 1)$. When a cycle like the one above is generated, the set of configurations visited during the initial part, up to $H = T + 1$, is *different* from the set visited when H decreases back toward zero. In other words, the trajectory looks like **a lasso around a local optimum**, and one does not waste CPU time to revisit previously visited configurations. In general, after visiting a locally optimal point, better values can be obtained by visiting other local optima. Of course, as soon as a local minimizer is found, all points in its *attraction basin* (i.e., all points that are mapped to the given minimizer by the local search dynamics) are not of interest. In fact, by definition, their f value is equal to or larger than the value at the local minimizer. The value of T should be chosen so that a new attraction basin leading to a new and possibly better local minimizer can be reached after reaching Hamming distance $T + 1$.

Because the minimal Hamming distance required (a sort of *attraction radius* for the given attraction basin) is not known, one should **determine T in a reactive way**, by learning the proper value *while* the search executes. The basic prohibition mechanism cannot guarantee the absence of cycles [37, 40]. In addition, the choice of a fixed T without *a priori* knowledge about the possible search trajectories that can be generated in a given $(\mathcal{X}, f)$ problem is difficult. If the search space is inhomogeneous, a size T that is appropriate in a region of $\mathcal{X}$ may be inappropriate in other regions. For example, T can be too small and insufficient to avoid cycles, or too large, so that only a small fraction of the movements are admissible and the search is inefficient.

Tabu-RSO uses a simple mechanism to change T during the search so that the value $T^{(t)}$ is appropriate to the local structure of the problem (therefore the term "reactive"). The underlying design principle is that of determining, for a given local configuration, **the minimal prohibition value which is sufficient to escape** from an attraction basin around a minimizer, as illustrated in Fig. 27.4. The basic principle is that T is equal to one at the beginning

(only immediately returning to a just left configuration is prohibited), T increases if the trajectory is *trapped* in an attraction basin around a local optimum (repetitions of previously visited configuration can signal this situation), and T decreases if unexplored search regions are visited, leading to different local optima. If the problem is so simple that a single local optimum is present, and therefore it coincides with the global optimum, the power of tabu-RSO is not needed, although not dangerous. Tabu-RSO will simply discover the optimal solution, save it, and then search for an (impossible) improvement. It has to be noted that most real-world problems are infested with many locally optimal points, so that tabu-RSO is crucial to transform a local search building block into an effective and efficient solver.

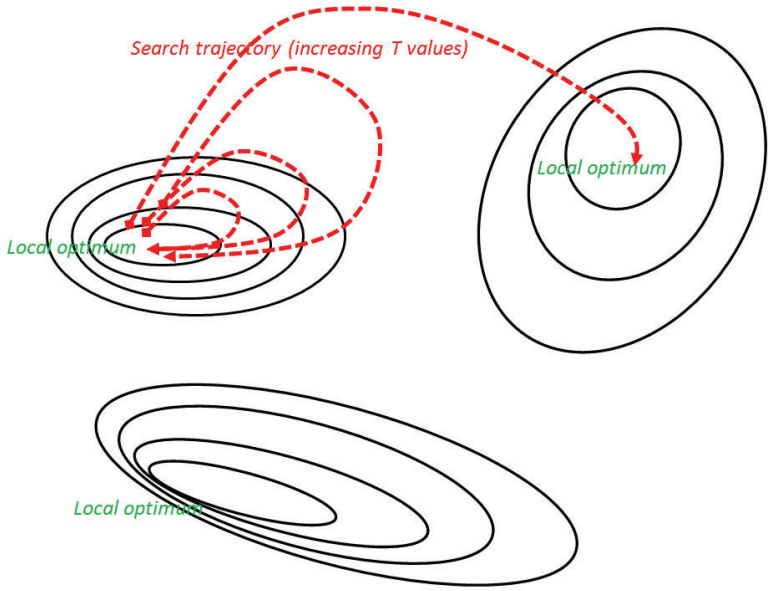


Figure 27.4: RSO with prohibitions in action. Three locally optimal points are shown together with contour lines of the function to be optimized. When starting from a locally optimal point, RSO executes loops which reach bigger and bigger distances from the attractor, until another attraction basin is encountered (if present).

The overhead (additional CPU time and memory) introduced by the reactive mechanisms is of small number of CPU cycles and bytes, approximately constant for each step in the search process. By using *hashing* functions[1] to store and retrieve the relevant data, the additional memory required can be reduced to some bytes per iteration, while the time is reduced to that needed to calculate a memory address from the current configuration and to execute a small number of comparisons and updates of variables [37]. RSO with prohibitions has been used for problems ranging from combinatorial optimization to the minimization of continuous functions and to sub-symbolic machine learning tasks, a partial list is contained in [26].

[1] Hashing is a nice trick — although maybe not well known outside of the computer science community — to create *dictionaries* (associations of data with keywords), so that retrieval is fast and approximately constant-time, on average. A *hash function* is a deterministic procedure that takes an arbitrary block of data (in our case the keyword) and returns a limited-size integer, the hash value, such that a change to the data will typically change the hash value. The obtained integer can be used as a memory *address* to store the block of data, so that lookup is immediate: get the hash value and go the memory address to read the data. Technical details related to having different keywords sharing by chance the same address can be resolved by *chaining*, i.e., associating a linked list of data items with the memory address.

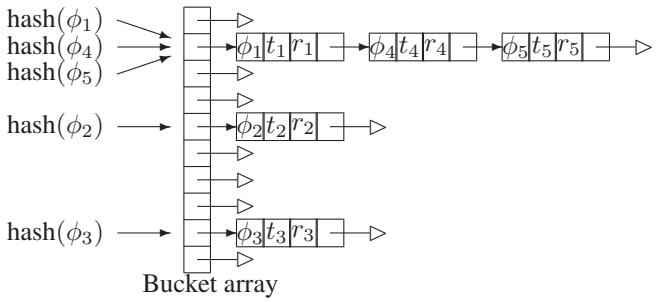


Figure 27.5: Open hashing scheme: items (configuration, or compressed hashed value, etc.) are stored in "buckets." The index of the bucket array is calculated from the configuration.

27.3 Fast data structures for using the search history

The storage and access of the past events is executed through the well-known *hashing* or radix-tree techniques in a CPU time that is approximately constant with respect to the number of iterations. Therefore the overhead caused by the use of the history is negligible for tasks requiring a non-trivial number of operations to evaluate the cost function in the neighborhood.

An example of a memory configuration for the hashing scheme is shown in Fig. 27.5. From the current configuration phi one obtains an index into a "bucket array." The items (configuration or hashed value or derived quantity, last time of visit, total number of repetitions) are then stored in linked lists starting from the indexed array entry. Both storage and retrieval require an approximately constant amount of time if: i) the number of stored items is not much larger than the size of the bucket array, and ii) the hashing function scatters the items with a uniform probability over the different array indices. More precisely, given a hash table with m slots that stores n elements, a load factor $\alpha = n/m$ is defined. If collisions are resolved by chaining, searches take $O(1 + \alpha)$ time, on average.

27.3.1 Persistent dynamic sets

Persistent dynamic sets are proposed to support memory–usage operations in history-sensitive heuristics in [24, 22].

Ordinary data structures are *ephemeral* [119], meaning that when a change is executed the previous version is destroyed. Now, in many contexts like computational geometry, editing, implementation of very high level programming languages, and, last but not least, the context of history-based heuristics, multiple versions of a data structure must be maintained and accessed. In particular, in heuristics one is interested in *partially persistent* structures, where all versions can be accessed but only the newest version (the *live* nodes) can be modified. A review of *ad hoc* techniques for obtaining persistent data structures is given in [119] that is dedicated to a systematic study of persistence, continuing the previous work of [285].

Hashing combined with persistent red-black trees

The basic observation is that, because Tabu Search is based on local search, configuration $X^{(t+1)}$ differs from configuration $X^{(t)}$ only because of the addition or subtraction of a single index (a single bit is changed in the string). It is therefore reasonable to expect that more efficient techniques can be devised for storing *a trajectory of chained configurations* than for storing arbitrary states. The expectation is indeed true, although the techniques are not for beginners. You are warned, proceed only if not scared by advanced data structures.

Let us define the operations INSERT(i) and DELETE(i) for inserting and deleting a given index i from the set. As cited above, configuration X can be considered as a set of indices in $[1, L]$ with a possible realization as a balanced red-black tree, see [46, 166] for two seminal papers about red-black trees. The binary string can be immediately obtained from the tree by visiting it in symmetric order, in time $O(L)$. INSERT(i) and DELETE(i) require $O(\log L)$ time, while at most a single node of the tree is allocated or deallocated at each iteration. Re-balancing the tree after insertion or deletion can be done in $O(1)$ rotations and $O(\log L)$ color changes [357]. In addition, the amortized number of color changes per update is $O(1)$, see for example [259].

Now, the REM method [148, 149] is closely reminiscent of a method studied in [285] to obtain partial persistence, in which the entire update sequence is stored and the desired version is rebuilt from scratch each time an access is performed, while a systematic study of techniques with better space-time complexities is present in [312, 119]. Let us now summarize from [312] how a partially persistent red-black tree can be realized. An example of the realizations that we consider is presented in Fig. 27.6.

The trivial way is that of keeping in memory all copies of the ephemeral tree (see the top part of Fig. 27.6), each copy requiring $O(L)$ space. A smarter realization is based on *path copying*, independently proposed by many researchers, see [312] for references. Only the path from the root to the nodes where changes are made is copied: a set of search trees is created, one per update, having different roots but sharing common subtrees. The time and space complexities for INSERT(i) and DELETE(i) are now of $O(\log L)$.

The method that we will use is a space-efficient scheme requiring only linear space proposed in [312]. The approach avoids copying the entire access path each time an update occurs. To this end, each node contains an additional "extra" pointer (beyond the usual left and right ones) with a time stamp. When attempting to add a pointer to a node, if the extra pointer is available, it is used and the time of the usage is registered. If the extra pointer is already used, the node is copied, setting the initial left and right pointers of the copy to their latest values. In addition, a pointer to the copy is stored in the last parent of the copied node. If the parent has already used the extra pointer, the parent, too, is copied. Thus copying proliferates through successive ancestors until the root is copied or a node with a free extra pointer is encountered. Searching the data structure at a given time t in the past is easy: after starting from the appropriate root, if the extra pointer is used the pointer to follow from a node is determined by examining the time stamp of the extra pointer and following it if and only if the time stamp is not larger than t. Otherwise, if the extra pointer is not used, the normal left-right pointers are considered. Note that the pointer direction (left or right) does not have to be stored: given the search tree property it can be derived by comparing the indices of the children with that of the node. In addition, colors are needed only for the most recent (live) version of the tree. In Fig. 27.6 NULL pointers are not shown, colors are correct only for the live tree (the nodes reachable from the rightmost root), extra pointers are dashed and time-stamped.

The worst-case time complexity of INSERT(i) and DELETE(i) remains $O(\log L)$, but the important result derived in [312] is that the amortized space cost per update operation is $O(1)$. Let us recall that the total amortized space cost of a sequence of updates is an upper bound on the actual number of nodes created.

Let us now consider the context of history-based heuristics. Contrary to the popular usage of persistent dynamic sets to search past versions at a specified time t, one is interested in checking whether a configuration has already been encountered in the previous history of the search, at *any* iteration.

A convenient way of realizing a data structure supporting X-SEARCH(X) is to combine *hashing* and *partially persistent dynamic sets*, see Fig. 27.7. From a given configuration X an index into a "bucket array" is obtained through a hashing function, with a possible incremental evaluation in time $O(1)$. Collisions are resolved through chaining: starting from each bucket header there is a linked list containing a pointer to the appropriate root of the persistent red-black tree and satellite data needed by the search (time of configuration, number of repetitions).

As soon as configuration $X^{(t)}$ is generated by the search dynamics, the corresponding persistent red-black tree is updated through INSERT(i) or DELETE(i). Let us now describe X-SEARCH$(X^{(t)})$: the hashing value is computed from $X^{(t)}$ and the appropriate bucket searched. For each item in the linked list the pointer to the root of the past version of the tree is followed and the old set is compared with $X^{(t)}$. If the sets are equal, a pointer to the item on the linked list is returned. Otherwise, after the entire list has been scanned with no success, a NULL pointer is returned.

In the last case a new item is linked in the appropriate bucket with a pointer to the root of the live version of the

tree (X-INSERT(X, t)). Otherwise, the last visit time t is updated and the repetition counter is incremented.

After collecting the above cited complexity results, and assuming that the bucket array size is equal to the maximum number of iterations executed in the entire search, it is straightforward to conclude that each iteration of *reactive-TS* requires $O(L)$ average-case time and $O(1)$ amortized space for storing and retrieving the past configurations and for establishing prohibitions.

In fact, both the hash table and the persistent red-black tree require $O(1)$ space (amortized for the tree). The worst-case time complexity per iteration required to update the current $X^{(t)}$ is $O(\log L)$, the average-case time for searching and updating the hashing table is $O(1)$ (in detail, searches take time $O(1 + \alpha)$, α being the load factor, in our case upper bounded by 1). The time is therefore dominated by that required to compare the configuration $X^{(t)}$ with that obtained through X-SEARCH$(X^{(t)})$, i.e., $O(L)$ in the worst case. Because $\Omega(L)$ time is needed during the neighborhood evaluation to compute the f values, the above complexity is optimal for the considered application to history-based heuristics.

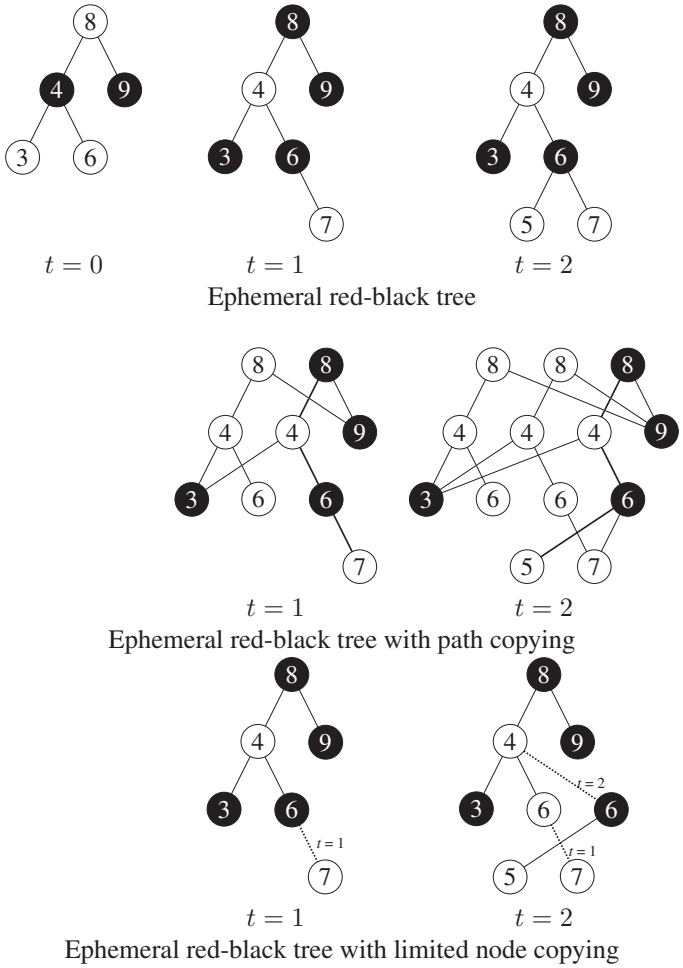


Figure 27.6: How to obtain a partially persistent red-black tree from an ephemeral one (top), containing indices 3,4,6,8,9 at t=0, with subsequent insertion of 7 and 5. Path copying (middle), with thick lines marking the copied part. Limited node copying (bottom) with dashed lines denoting the "extra" pointers with time stamp.

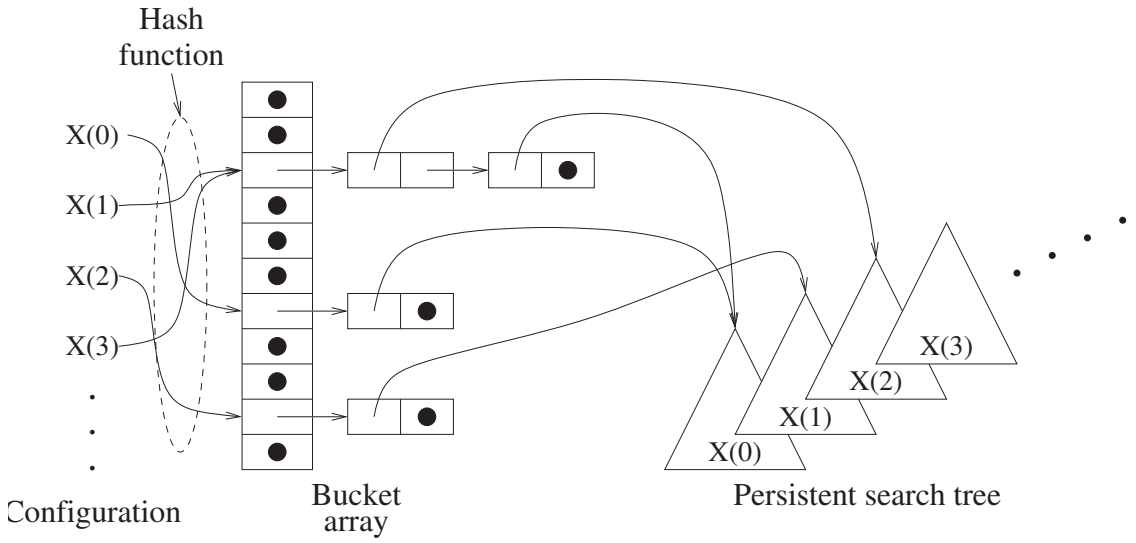


Figure 27.7: Open hashing scheme with persistent sets: a pointer to the appropriate root for configuration $X^{(t)}$ in the persistent search tree is stored in a linked list at a "bucket." Items on the list contain satellite data. The index of the bucket array is calculated from the configuration through a hashing function.

Gist

Reactive Search Optimization (RSO) uses learning and adaptation during the optimization process, so that the search technique can be fine-tuned to the instance being solved and to the local characteristics around the current tentative solution. RSO deals with designing **an intelligent module overseeing the basic local search process**, balancing diversification and intensification, optimizing components of the optimization process itself (meta-optimization or meta-heuristics).

Reactive Search Optimization **adapts in an online manner meta-parameters of heuristics**. In particular prohibition mechanisms ("stay away from this area") can be used to encourage diversification and discovery of improving solutions. It is well known that real creativity and innovation requires staying away from current solutions ("lateral thinking", "out-of-the-box thinking" are popular terms in the management literature). In a similar manner, prohibition mechanisms added to simple local-search schemes can be a very direct manner to continue the search beyond local optimality.

Reactive Search Optimization schemes require **saving and retrieving past configurations**, operations which can be very fast with appropriate data structures.

Let's note that the term reactive as "**readily responsive to a stimulus**" used in our context is not in contrast with proactive as "acting in anticipation of future problems, needs, or changes." In fact, in order to obtain a reactive algorithm, the designer needs to be proactive by appropriately planning modules into the algorithm, to endow it with the capability of autonomous response. In other words, **Reactive Search Optimization algorithms need proactive algorithm designers**.

Chapter 28

Adapting neighborhoods and selection

Speak through me, O Muse,
of that man of many devices
who wandered much
once he'd sacked the sacred citadel of Troy.
(Odyssey by Homer, as translated by Stein,
Charles)

It looks like there is little online learning to be considered for a simple technique like local search. Nonetheless, we already encountered a first possibility, that of prohibiting some local moves in Chapter 27. This chapter considers a more structured organization of the possible local moves. Various possible moves are collected into a set of different neighborhoods, which are chosen in a strategic manner depending on the current state of the search.

As a mythological analogy consider how Odysseus (or Ulysses in Roman myths) escaped from the Sirens, beautiful yet dangerous creatures, who lured nearby sailors with their enchanting music and voices to shipwreck on the rocky coast of their island. He had all of his sailors plug their ears with beeswax and tie him to the mast. He ordered his men to leave him tied tightly to the mast, no matter how much he would beg. What a nice analogy of the effort and different set of moves required to escape from the attraction of a local minimum.

Let's recollect the building block of local search considered in Section 24.2 In the function IMPROVING-NEIGHBOR one has to decide about a *neighborhood* (a set of local moves to be applied) and about a *way to pick one of the neighbors* to be the next point along the search trajectory. Selecting a **neighborhood structure appropriate to a given problem** is the most critical issue in LS. Let's concentrate on *online* learning strategies which can be applied *while* local search runs on a specific instance. They can be applied in two contexts: selection of the neighbor or selection of the neighborhood.

Let's start from the first context where a neighborhood is chosen before the run is started, and only the selection of an improving neighbor is dynamic. The average progress in the optimization per unit of computational effort (the average "speed of descent" Δf_{best} per second) will depend on two factors: the average *improvement per move* and the average *CPU time per move*. There is a trade-off: the longer to evaluate the neighborhood, the better the chance of identifying a move with a large improvement, but the shorter the total number of moves which one can execute in the CPU time allotted. The optimal setting depends on the problem, the specific instance, and the local configuration of the f landscape.

The immediate brute-force approach consists of considering **all neighbors**, by applying all possible basic moves, evaluating the corresponding f values and moving to the neighbor with the best value, breaking ties randomly if they occur. The best possible neighbor is chosen at each step. To underline this fact, the term "**best-improvement** local search" is used.

A second possibility consists of evaluating **a sample of the possible moves**, a subset of neighbors. In this cases IMPROVING-NEIGHBOR can return the first candidate with a better f value. This option is called **first-move**. If no such candidate exists the trajectory is at a local optimum. A randomized examination order can be used to avoid spurious effects. FIRSTMOVE is clearly adaptive: the exact number of neighbors evaluated before deciding the next move depends not only on the instance but on the particular local properties in the configuration space around the current point. One expects to evaluate a small number of candidates the early phase of the search, whereas identifying an improving move will become more and more difficult during the later phases, close to local optimality. The analogy is that of learning a new language: the progress is fast at the beginning but it gets slower and slower after reaching an advanced level. To summarize, FIRSTMOVE works according to "keep evaluating until either an improving neighbor is found or all neighbors have been examined".

28.1 Variable Neighborhood Search: Learning the neighborhood

There are cases when the definition of a *fixed* neighborhood for a problem is not an optimal choice because **adapting the neighborhood to the local configuration** around the current point is beneficial.

A possible way of tuning the neighborhood considers **a set of neighborhoods**, defined *a priori* at the beginning of the search, and then aims at using the most appropriate one during the search, as illustrated in Fig. 28.1. This is the seminal idea of the Variable Neighborhood Search (VNS) technique, see [171].

Let the set of neighborhoods be $\{N_1, N_2, ..., N_{kmax}\}$. A proper VNS strategy has to deal with the following issues:

1. Which neighborhoods to use and how many of them. Larger neighborhoods may include smaller ones or be disjoint.

2. How to schedule the different neighborhoods during the search (order of consideration, transitions between different neighborhoods)

3. Which neighborhood evaluation strategy to use (first move, best move, sampling, etc.)

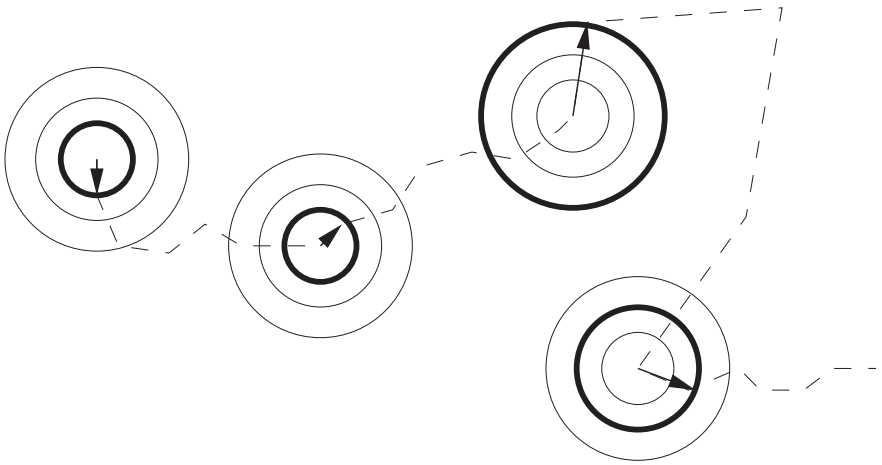


Figure 28.1: Variable neighborhood search: the used neighborhood ("circle" around the current configuration) varies along the search trajectory.

The first issue can be decided based on detailed problem knowledge, preliminary experimentation, or simply availability of off-the-shelf software routines for the efficient evaluation of a set of neighborhoods.

The second issue leads to a range of possible techniques. A simple implementation can just **cycle randomly among the different neighborhoods** during subsequent iterations: no online learning is present but possibly more robustness for solving instances with very different characteristics or for solving an instance where different portions of the search space have widely different characteristics.

Let's note that **local optimality depends on the neighborhood**: as soon as a local minimum is reached for a specific N_k, improving moves can in principle be found in other neighborhoods N_j with $j \neq k$. A possibility to use online learning is based on the principle "**intensification first, minimal diversification only if needed**" which we often encounter in heuristics [37]. One orders the neighborhoods according to their *diameter*, or to the *strength* of the perturbation executed. For example, if the search space is given by binary strings, one may consider as N_1 all changes of a single bit, N_2 all changes of two bits, etc. If local search makes progress one sticks to the default neighborhood N_1. As soon as a local minimum with respect to N_1 is encountered one tries to go to greater Hamming distances from the current point aiming at discovering a nearby attraction basin, possibly leading to a better local optimum.

Fig. 28.2 illustrates the reactive strategy: point a corresponds to the local minimum, point b is the best point in neighborhood N_1, and point c the best point in N_2. The value of point c is still worse, but the point is in a different attraction basin so that a better point e could now be reached by the default local search. The best point d in N_3 is already improving on a.

From the example one already identifies two possible strategies. In both cases one uses N_1 until a local minimum of N_1 is encountered. When this happens one considers N_2, N_3, ... In the first strategy one stops when an improving neighbor is identified (point d in the figure). In the second strategy one stops when one encounters a neighbor in a different attraction basin with an improving local minimum (point c in the figure). How does one know that c is in a different basin? One can perform a local search run from it and of look at which point the local search converges.

For both strategies, one reverts back to the default neighborhood N_1 as soon as the *diversification* phase considering neighborhoods of increasing diameter is successful. Note a strong similarity with the design principle of Reactive Tabu Search, see Sec. 27.2, where diversification through prohibitions is activated when there is evidence of entrapment in an attraction basin and gradually reduced when there is evidence that a new basin has been discovered.

Many VNS schemes using the set of different neighborhoods in an organized way are possible [173]. Variable Neighborhood Descent (VND), see Fig. 28.3, uses the default neighborhood first, and the ones with a higher number

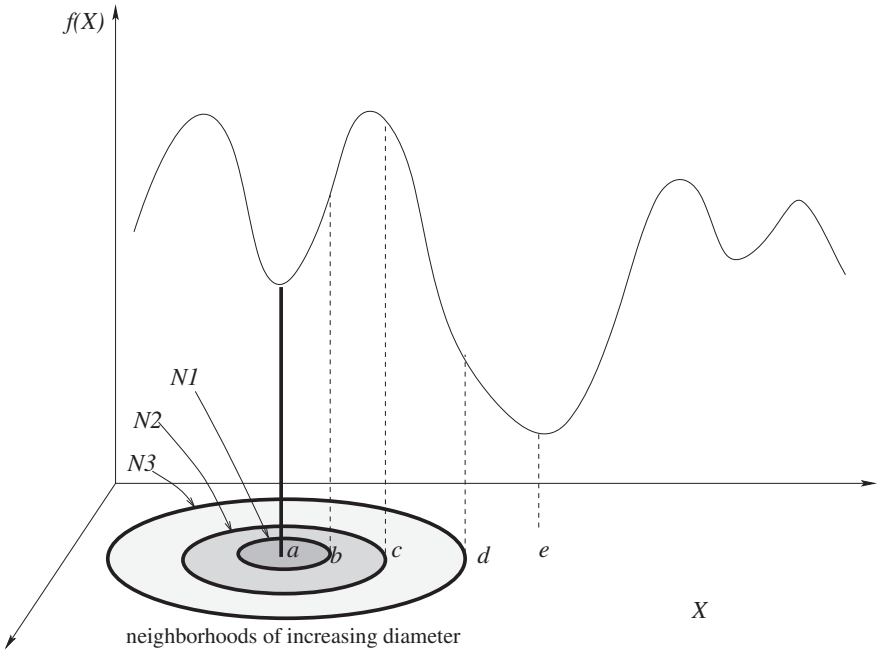


Figure 28.2: Variable neighborhoods of different diameters. Neighboring points are on the circumferences at different distances. The figure is intended to help the intuition: the actual neighbors considered in the text are discrete.

only if the default neighborhood fails (i.e., the current point is a local minimum for N_1), and only until an improving move is identified, after which it reverts back to N_1. When VND is coupled with an ordering of the neighborhoods according to the *strength* of the perturbation, one realizes the principle "**use the minimum strength perturbation leading to an improved solution**."

REDUCED-VNS is a stochastic version where only one random neighbor is generated before deciding about moving or not. Line 5 of Fig. 28.3 is substituted with:

$$X' \leftarrow \text{RANDOMEXTRACT}(N_k(X))$$

SKEWED-VNS modifies the move acceptance criterion by **accepting also worsening moves if they lead the search trajectory sufficiently far** from the current point ("I am not improving but at least I keep moving without worsening too much during the diversification"), see Fig. 28.4. This version requires a suitable distance function $\rho(X, X')$ between two solutions to get controlled diversification (e.g., $\rho(X, X')$ can be the Hamming distance for binary strings), and it requires a *skewness* parameter α to regulate the trade-off between movement distance and willingness to accept worse values. By looking at Fig. 28.2, one is willing to accept the worse solution c because it is sufficiently far to possibly lead to a different attraction basin. Of course, determining an appropriate metric and skewness parameter is not a trivial task in general.

Other versions of VNS employ a stochastic move acceptance criterion, in the spirit of Simulated Annealing as implemented in the large-step Markov-chain version [264, 262], where "kicks" of appropriate strength are used to exit from local minima, see also Chapter 29 about Iterated Local Search.

An explicitly reactive-VNS is considered in [64] for the VRPTW problem (vehicle routing problem with time windows), where a construction heuristic is combined with VND using first-improvement local search. Furthermore, the objective function used by the local search operators is modified to consider the waiting time to escape from a local minimum. A preliminary investigation about a self-adaptive neighborhood ordering for VND is presented in [192].

```
1. function VARIABLENEIGHBORHOODDESCENT (N_1, ..., N_{k_max})
2.     repeat until no improvement or max CPU time elapsed
3.         k ← 1                                          default neighborhood
4.         while k ≤ k_max:
5.             X' ← BESTNEIGHBOR (N_k(X))                 neighborhood exploration
6.             if f(X') < f(X)
7.                 X ← X' ; k ← 1                         success: back to default neighborhood
8.             else
9.                 k ← k + 1                              try with the following neighborhood
```

Figure 28.3: The VND routine. Neighborhoods with higher numbers are considered only if the default neighborhood fails and only until an improving move is identified. X is the current point.

```
1. function SKEWEDVARIABLENEIGHBORHOODSEARCH (N_1, ..., N_{k_max})
2.     repeat until no improvement or max CPU time elapsed
3.         k ← 1                                          default neighborhood
4.         while k ≤ k_max
5.             X' ← RANDOMEXTRACT (N_k(X))
6.             X'' ← LOCALSEARCH(X')                      shake local search to reach local minimum
7.             if f(X'') < f(X) + αρ(X, X'')
8.                 X ← X'' ; k ← 1                        success: back to default neighborhood
9.             else
10.                k ← k + 1                              try with the following neighborhood
```

Figure 28.4: The SKEWED-VNS routine. Worsening moves are accepted provided that the change leads the trajectory sufficiently far from the starting point. X is the current point. $\rho(X, X'')$ measures the solution distance.

Ratings to the different neighborhoods are derived according to their observed benefits in the past and used periodically to order the various neighborhoods.

To conclude this section, let's note some similarities between VNS and the adaptation of the search region in stochastic search technique for continuous optimization. In both cases the neighborhood is adapted to the local position in the search space. In addition to many specific algorithmic differences, let's note that the set of neighborhoods is discrete in VNS while it consists of a portion of $\mathbb{R}^n$ for continuous optimization. Neighborhood adaptation in the continuous case, see for example the Reactive Affine Shaker algorithm [36] in Sec. 32.1, is considered to speed-up convergence to a local minimizer, not to jump to nearby valleys.

Gist

Local search heuristics build upon the definition of a suitable neighborhood (set of basic moves to apply to tentative solutions). Unfortunately, it is difficult to know *a priori* which neighborhood will produce the best results. In addition, configurations which are locally optimal for one neighborhood can be further improved by a different neighborhood.

Variable Neighborhood Search (VNS) organizes the neighborhoods as sets of different strengths (amount of perturbation), it uses the simplest neighborhood at the beginning, until a local minimum is reached. At this point, neighborhoods leading to bigger perturbations are tried. Like the flexible Ulysses, the man of many devices, rapidly adapts to new contexts to escape Polyphemus and the Sirens, **VNS adapts the neighborhood in a reactive fashion to the characteristics of a specific problem instance and to the local situation along the search trajectory** to escape locally optimal points.

VNS is a king of greedy strategy about the use of neighborhoods (therefore a kind of meta-greedy-strategy).

Chapter 29

Iterated local search

Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time.
(Thomas A. Edison)

If a **local search "building block"** is available, for example as a concrete software toolkit, how can it be used by some upper layer coordination mechanism to get better results? An answer is given by *repeating* calls to the local search routine each time starting from a properly chosen configuration. If the starting configuration is random, one starts from scratch and discards knowledge about the previous searches. This trivial form actually is called simply **repeated local search**. It has no memory and just keeps trying, like the mythological Sisyphus, punished for his deceitfulness and forced to roll an immense boulder up a hill, only to watch it roll back down, repeating this action for eternity.

Learning implies that the previous history, for example the memory about the previously found local minima, is mined to produce better and better starting points. The implicit assumption is again that of a clustered distribution of local minima: determining good local minima is easier when starting from a local minimum with a low f value than when starting from a random point. It is also faster because trajectory lengths from a local minimum to a nearby

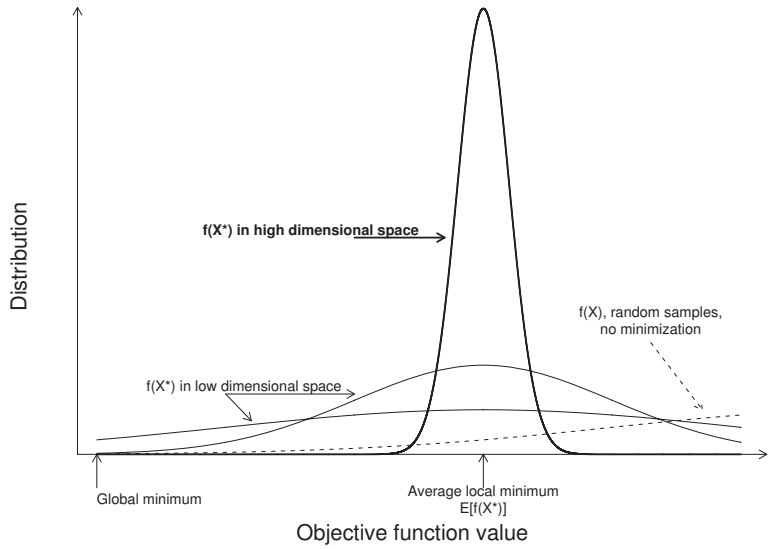


Figure 29.1: Probability of low f values is larger for local minima in $\mathcal{X}^*$ than for a point randomly extracted from $\mathcal{X}$. Large-dimensional problems tend to have a very spiky distribution for $\mathcal{X}^*$ values.

one tend to be shorter. Furthermore an incremental evaluation of f can often be used instead of re-computation from scratch if one starts from a new point. *Updating* f values after a move can be much faster than computing them from scratch. As usual, the only caveat is to avoid confinement in a given attraction basin, so that the "kick" to transform a local minimizer into the starting point for the next run has to be appropriately strong, but not too strong to avoid reverting to memory-less random restarts (if the kick is stochastic). **Iterated Local Search** is based on building a sequence of locally optimal solutions by: (i) perturbing the current local minimum; (ii) applying local search after starting from the modified solution.

As it happens with many simple — but sometimes very effective — ideas, the same principle has been rediscovered multiple times, for example in [45]. One may also argue that iterated local search shares many design principles with variable neighborhood search. A similar intuition is present in the iterated Lin-Kernighan algorithm of [221], where a local minimum is modified by a 4-change (a "big kick" eliminating four edges and reconnecting the path) and used as a starting solution for the next run of the Lin-Kernighan heuristic. In the stochastic local search literature based on Simulated Annealing, the work about large-step Markov chain of [264, 262, 263, 370] contains very interesting results coupled with a clear description of the principles.

Our description follows mainly [254]. LOCALSEARCH is seen by ILS as a black box. It takes as input an initial configuration X and ends up at a local minimum X^*. Globally, LOCALSEARCH maps from the search space $\mathcal{X}$ to the reduced set $\mathcal{X}^*$ of local minima. Obviously, the values of the objective function f at local minima are better than the values at the starting points, unless one is so lucky to start already at a local minimum. If one searches for low-cost solutions, sampling from $\mathcal{X}^*$ is therefore more effective than sampling from $\mathcal{X}$, this is in fact the basic feature of local search, see Fig. 29.1.

One may be tempted to sample in $\mathcal{X}^*$ by repeating runs of local search after starting from different random initial points. Unfortunately, general statistical arguments [316] related to the "law of large numbers" indicate that when the size of the instance increases, the *probability distribution of the cost values* f *tends to become extremely peaked about*

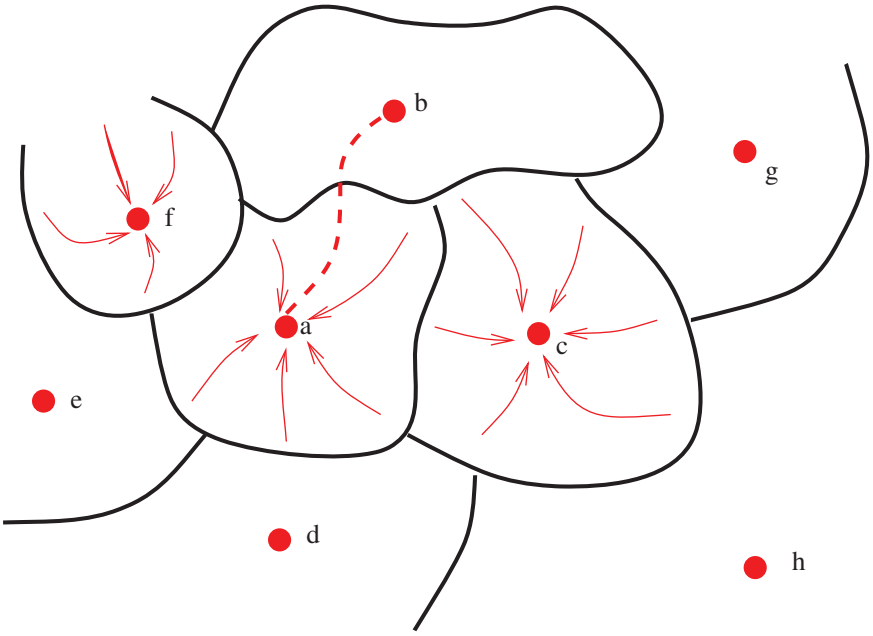


Figure 29.2: Neighborhood among attraction basins induced a neighborhood definition on local minima in $\mathcal{X}^*$.

the mean value $E[f(x^*)]$, mean value which can be offset from the best value f_{best} by a fixed percent excess. If we repeat a random extraction from $\mathcal{X}^*$ we are going to get very similar values with a large probability.

Relief comes from the rich structure of many optimization problem, which tends to cluster good local minima together. Instead of a random restart it is better to search in the neighborhood of a current good local optimum. What one needs is a **hierarchy of nested local searches**: starting from a proper neighborhood structure on $\mathcal{X}^*$ (proper as usual means that it makes the internal structure of the problem "visible" during a walk among neighbors). Hierarchy means that one uses local search to identify local minima, and then defines a local search *in the space of local minima*. One could continue, but in practice one limits the hierarchy to two levels. The sampling among $\mathcal{X}^*$ will therefore be *biased* and, if properly designed, can lead to the discovery of f values significantly lower than those expected by a random extraction in $\mathcal{X}^*$.

A neighborhood for the space of local minima $\mathcal{X}^*$, which is of theoretical interest, is obtained from the structure of the *attraction basins* around a local optimum. An attraction basin contains all points which are mapped to the given optimum by local search. The local optimum is an *attractor* of the dynamical system obtained by applying the local search moves. By definition, two local minima are neighbors if and only if their attraction basins are neighbors, i.e., they share part of the boundary. For example, in Fig. 29.2, local minima b, c, d, e, f are neighbors of local minimum a. Points g, h are not neighbors of a.

A weaker notion of closeness (neighborhood) which permits a fast stochastic search in $\mathcal{X}^*$ and which does not require an exhaustive determination of the attraction basins geography — a daunting task indeed — is based on creating a *randomized path* in $\mathcal{X}$ leading from a local optimum to one of the neighboring local optima, see the path from a to b in the figure.

A final design issue is how to build the path connecting two neighboring local minima. An heuristic solution is the following one, see Fig. 29.3 and Fig. 29.4: generate a sufficiently strong perturbation leading to a new point and then apply local search until convergence at a local minimum. The perturbation strength and the acceptance decision can change in a *reactive* manner, depending on the history of search. For example, the perturbation can become *stronger* if the recent history indicates that the search is trapped in the neighborhood of a local minimum, it can become *lighter* if

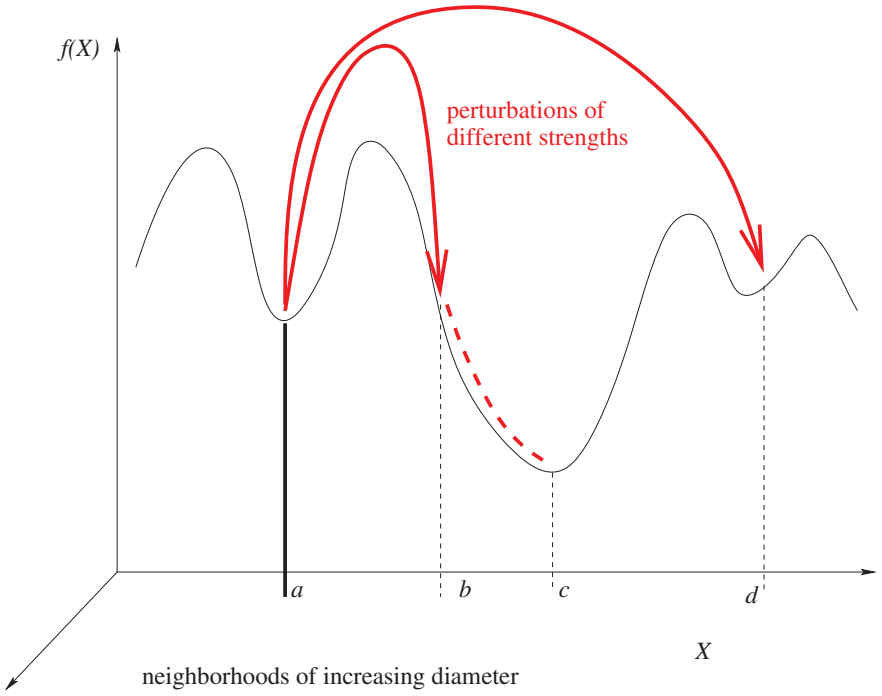


Figure 29.3: ILS: a perturbation leads from a to b, then local search to c. If perturbation is too strong one may end up at d therefore missing the closer local minima.

1. **function** ITERATEDLOCALSEARCH ()
2. $X^0 \leftarrow$ INITIALSOLUTION()
3. $X^* \leftarrow$ LOCALSEARCH (X^0)
4. **repeat**
5. $\quad X' \leftarrow$ PERTURB (X^*, history)
6. $\quad X^{*'} \leftarrow$ LOCALSEARCH (X')
7. $\quad X^* \leftarrow$ ACCEPTANCEDECISION (X^*, $X^{*'}$, history)
8. **until** (no improvement or termination condition)

Figure 29.4: Iterated Local Search. The perturbation strength and the acceptance decision depend on the history of the search process.

new record values are generated after the most recent kicks (an evidence that new local optima are being visited after the kicks).

One has to adapt the appropriate *strength* of the perturbation to avoid cycling and keep exploring. If the perturbation is too small, one risks that the solution returns back to the starting local optimum. As a result, if the perturbation and local search are deterministic, an endless cycle would be produced.

Learning based on the previous search history is of paramount importance to avoid cycles and similar traps. The principle of "intensification first, minimal diversification only if needed" can be applied, together with stochastic elements to increase robustness and discourage cycling. As we have seen for VNS, minimal perturbations maintain the trajectory in the starting attraction basin, while excessive ones bring the method closer to a random sampling, therefore loosing the boost from the problem structure properties. A possible solution consists of perturbing by a short random walk of a length which is *adapted* by statistically monitoring the progress in the search.

It is already clear that the design principles underlying many superficially different techniques are in reality strongly related. We already mentioned the issue related to designing a proper perturbation, or "kick," or selecting the appropriate neighborhood, to lead a solution away from a local optimum, as well as the issue of using online reactive learning schemes to increase the adaptation and robustness.

Gist

Local search mechanisms are simple to build and run but stop at the first locally optimal point encountered along the search trajectory.

Repeated local search consists of repeating local search after starting from different (possibly randomized) initial solutions. The best solutions found in all repetitions in then returned. Repeated local search is oblivious (memory-less).

In most cases better results cab be obtained by the smarter **iterated local search**, which can be seen as **local search among locally optimal points**. Kicks (perturbations) are applied to push a local optimum sufficiently away that the trajectory will not fall back to the starting point, but not too far away to make it similar to random search (and therefore not exploiting the rich "Big-Valley" problem structure).

In a soccer analogy, a player kicks the ball and passes it to a different player to reach the goal, but if the kick is too strong the referee will signal with a flag that the ball wnet off the field of play, leaving to wasted time in the action.

Chapter 30

Online learning in Simulated Annealing

"The poets did well to conjoin music and medicine, in Apollo, because the office of medicine is but to tune the curious harp of man's body and reduce it to harmony."
(Francis Bacon, The Advancement Of Learning)

A notable feature of simulated annealing is its asymptotic convergence, a notable drawback is its *asymptotic* convergence. For a practical application of SA, if the local configuration is close to a local minimizer and the temperature is already very small in comparison to the upward jump which has to be executed to escape from the attractor, although the system will *eventually* escape, an enormous number of iterations can be spent around the attractor. Given a finite patience time, all future iterations can be spent while "circling like a fly around a light-bulb" (the light-bulb being a local minimum). Animals with superior cognitive abilities get burnt once, learn, and avoid doing it again!

The memoryless property (current move depending only on the current state, not on the previous history) makes SA look like a dumb animal indeed. It is intuitive that a better performance can be obtained by using memory, by self-analyzing the evolution of the search, and by activating more direct *escape* strategies. One needs a better time-management than the "let's go to infinite time" principle. In the following sections we summarize the main memory-based approaches developed in the years to make SA a competitive strategy.

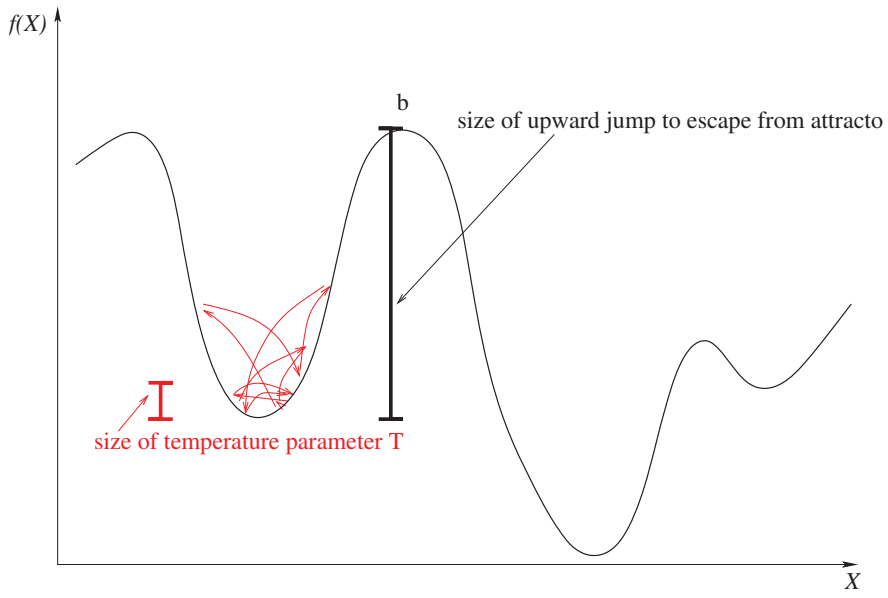


Figure 30.1: Simulated Annealing: if the temperature is very low w.r.t. the jump size SA risks a practical entrapment close to a local minimizer.

30.1 Combinatorial optimization problems

Even if a vanilla version of a cooling schedule for SA is adopted (starting temperature T_{start}, geometric cooling schedule $T_{t+1} = \alpha\, T_t$, with $\alpha < 1$, final temperature T_{end}), a sensible choice has to be made for the three involved parameters T_{start}, α, and T_{end}. If the scale of the temperature is wrong, extremely poor results are to be expected. The work [380] suggests to estimate the distribution of f values, note that f is usually called "energy" when exploiting the physical analogies of SA. The standard deviation of the energy distribution defines the maximum-temperature scale, while the minimum change in energy defines the minimum-temperature scale. These temperature scales tell us where to begin and end an annealing schedule.

The analogy with physics is pursued in [247], where concepts related to *phase transitions* and *specific heat* are used. A **phase transition** is related to solving a sub-part of a problem. Before reaching the state after the transition, big reconfigurations take place and this is signaled by *wide variations* of the f values. In thermodynamics and statistical mechanics, the specific heat describes how the average function value (energy value) changes with temperature. A phase transition occurs when the specific heat is maximal, a quantity estimated by the ratio between the estimated variance of the objective function and the temperature: ${\sigma_f}^2/T$. After a phase transition corresponding to the big reconfiguration, finer details in the solution have to be fixed, and this requires a slower decrease of the temperature. It is a very bad idea to stop SA immediately after a phase transition, when new record values keep being generated. Concretely, one defines two temperature-reduction parameters α and β, monitors the evolution of f along the trajectory and, after the phase transition takes place at a given T_{msp}, one switches from a faster temperature decrease given by α to the slower one given by β. The value T_{msp} is the temperature corresponding to the maximum specific heat, when the scaled variance reaches its maximal value.

A monotonic decrease of the temperature has some weaknesses: for fixed values of T_{start} and α in the vanilla version one will reach an iteration so that the temperature will be so low that *practically* no tentative move will be accepted with a non-negligible probability (given the finite users' patience). The best value reached so far, f_{best}, will remain stuck in a helpless manner even if the search is continued for very long CPU times, see also Fig. 30.1. In other words, given a set of parameters T_{start} and α, the useful span of CPU time is practically limited. After the initial

period the temperature will be so low that the system *freezes* and, with large probability, no tentative moves will be accepted anymore within the finite span of the run. In many cases one would like to obtain an **anytime algorithm**, so that longer allocated CPU times are related to possibly better and better values until the user decides to stop. Anytime algorithms — by definition — return the best answer possible even if they are not allowed to run to completion, and may improve on the answer if they are allowed to run longer.

Let's note that, in many cases, the stopping criterion should be decided *a posteriori*, for example after estimating that additional time has little probability to improve significantly on the result.

To avoid this problem is related to a monotonic temperature decrease, one considers **non-monotonic cooling schedules**, see [97, 283, 3]. A very simple proposal [97] suggests to reset the temperature once and for all at a constant temperature high enough to escape local minima but also low enough to visit them, for example, at the temperature T_{found} when the best heuristic solution is found in a preliminary SA simulation.

The basic design principle for a non-monotonic schedule is related to: i) exploiting an attraction basin rapidly by decreasing the temperature so that the system can settle down close to the local minimizer, ii) *increasing the temperature* to diversify the solution and visit other attraction basins, iii) decreasing again after reaching a different basin. As usual, the temperature increase in this kind of non-monotonic cooling schedule has to be rapid enough to avoid falling back to the current local minimizer, but not too rapid to avoid a random-walk situation (where all random moves are accepted) which would not capitalize on the local structure of the problem ("good local minima close to other good local minima"). The implementation details have to do with **determining an entrapment** situation, for example from the fact that no tentative move is accepted after a sequence t_{max} of tentative changes, and determining the detailed temperature decrease-increase evolution as a function of events occurring during the search. Possibilities to increase the temperature include resetting the temperature to $T_{\text{reset}} = T_{\text{found}}$, the temperature value when the current best solution was found [283]. If the reset is successful, one may progressively reduce the reset temperature: $T_{\text{reset}} \leftarrow T_{\text{reset}}/2$. Alternatively [3] geometric **re-heating** phases can be used, which multiply T by a heating factor γ larger than one at each iteration during reheat. Enhanced versions involve a learning process to choose a proper value of the heating factor depending on the system state. In particular, γ is close to one at the beginning, while it increases if, after a fixed number of escape trials, the system is still trapped in the local minimum. More details and additional bibliography can be found in the cited papers.

Let's note that similar "strategic oscillations" have been proposed in tabu search, in particular in the reactive tabu search [37] see Sec. 27.2, and in variable neighborhood search, see Sec. 28.1.

Experimental evidence shows that the *a priori* determination of SA parameters and acceptance function does not lead to efficient implementations [278]. Adaptations may be done "*by the algorithm itself using some learning mechanism* or by the user using his own learning mechanism." The authors appropriately note that the optimal choices of algorithm parameters depend not only on the problem but also on the particular instance and that a proof of convergence to a globally optimum is not a selling point for a specific heuristic: in fact a simple random sampling, or even exhaustive enumeration (if the set of configurations is finite) will eventually find the optimal solution, although they are not the best algorithms to suggest. A simple adaptive technique suggested in [278] is the SEQUENCEHEURISTIC: a perturbation leading to a worsening solution is accepted if and only if a fixed number of trials could not find an improving perturbation. This method can be seen as deriving evidence of "entrapment" in a local minimum and reactively activating an escape mechanism. In this way the temperature parameter is eliminated. The positive performance of the SEQUENCEHEURISTIC in the area of design automation suggests that the success of SA is "due largely to its acceptance of bad perturbations to escape from local minima rather than to some mystical connection between combinatorial problems and the annealing of metals" [278].

30.2 SA for global optimization of continuous functions

The application of SA to continuous optimization (optimization of functions defined on real variables in $\mathbb{R}$) is pioneered by [101]. The basic method is to generate a new point with a random step along a direction $\boldsymbol{e}_h$, to evaluate the function and to accept the move with the probability given in equation (25.1). One cycles over the different directions $\boldsymbol{e}_h$ during successive steps of the algorithm. A first critical choice has to do with the range of the random step along

the chosen direction. A fixed choice obviously may be very inefficient: this opens a first possibility for *learning* from the local f surface. In particular a new trial point $\boldsymbol{x}'$ is obtained from the current point $\boldsymbol{x}$ as:

$$\boldsymbol{x}' = \boldsymbol{x} + \text{Rand}(-1, 1)v_h\boldsymbol{e}_h$$

where Rand$(-1, 1)$ returns a random number uniformly distributed between -1 and 1, $\boldsymbol{e}_h$ is the unit-length vector along direction h, and v_h is the step-range parameter, one for each dimension h, collected into the vector $\boldsymbol{v}$. The exponential acceptance rule is used to decide whether to update the current point with the new point $\boldsymbol{x}'$. The v_h value is adapted during the search with the aim of maintaining the number of *accepted* moves at about one-half of the total number of tried moves. Although the implementation is already reactive and based on memory, the authors encourage more work so that a "good monitoring of the minimization process" can deliver precious feedback about some crucial internal parameters of the algorithm.

In Adaptive Simulated Annealing (ASA), also known as very fast simulated re-annealing [205], the parameters that control the temperature cooling schedule and the random step selection are automatically adjusted according to algorithm progress. If the state is represented as a point in a box and the moves as an oval cloud around it, the temperature and the step size are adjusted so that all of the search space is sampled at a coarse resolution in the early stages, while the state is directed to promising areas in the later stages [205].

Gist

The use of Simulated Annealing for optimization has been motivated with **asymptotic convergence results**. When the number of iterations goes to infinity and the temperature is decreased slowly, the probability of observing a global optimum goes to one. Unfortunately, life is too short for asymptotic results, even the life of the universe is too short in some cases.

The problem is that the temperature parameter, when compared with the step in function value which must be executed to escape from a local attraction basin, can be so low that all future steps can be spent in the neighborhood of a single local optimum, in spite of the theoretical asymptotic convergence.

One has to **abandon vanilla "Markov-process" SA** in favor of more pragmatic versions which estimate progress, determine a possible entrapment, and activate direct ways of escaping, e.g., by raising the temperature (**re-heating**). The conditions for the validity of most math theorems are not valid anymore, theoretical analysis becomes very complex if not impossible, but better local optima can be determined in most practical cases.

Traditional SA can be compared to **a fly randomly circling around a light bulb**, getting burnt again and again, with no memory and no learning possibility. The light bulb is an analogy for a local optimum entrapping a search directory. When touching something hot, a kid will get burnt once, maybe twice, but then online learning takes place. Which process sounds more sensible?

Chapter 31

Dynamic landscapes and noise levels

'Morsel' is a perfect word. Forming those six letters on the lips and tongue prompts an instantaneous physiological reaction. The mouth waters. The lips purse.
(Shawn Amos)

This chapter considers reactive modification of the objective function in order to support appropriate diversification of the search process. Contrary to the prohibition-based techniques of Sec. 27.2 the focus is not that of pushing the search trajectory away from a local optimum though explicit and direct prohibitions but on **modifying the objective function so that previous promising areas in the solution space appear less favorable**, and the search trajectory will be gently pushed to visit new portions of the search space. To help the intuition, see also Fig. 31.1, one may think about pushing up the search landscape at a discovered local minimum, so that the search trajectory will flow into neighboring attraction basins. For a physical analogy, think about you sitting in a tent while it is raining outside. A

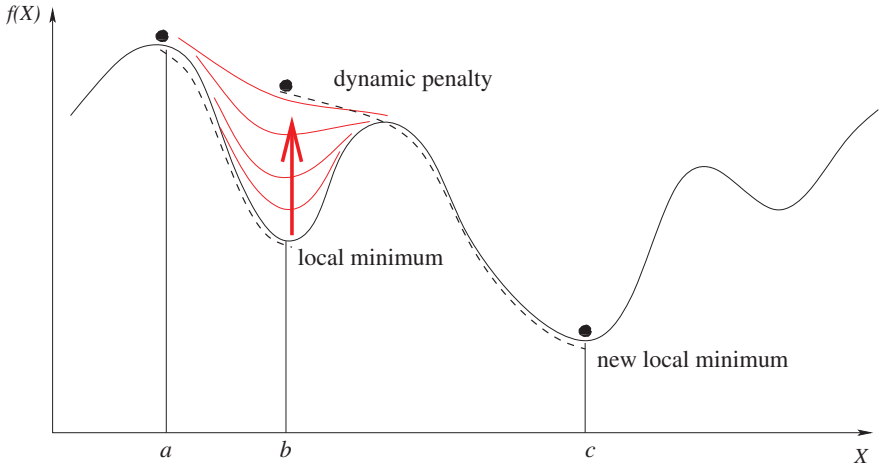


Figure 31.1: Transformation of the objective function to gently push the solution out of a given local minimum.

```
GSAT-WITH-WALK
1       for i ← 1 to MAX-TRIES
2           X ← random truth assignment
3           for j ← 1 to MAX-FLIPS
4               if Rand(0,1)< p then
5                   var ← any variable occurring in some unsatisfied clause
6               else
7                   var ← any variable with largest Δf
8               FLIP(var)
```

Figure 31.2: The "GSAT-with-walk" algorithm. Rand$(0, 1)$ generates random numbers in the range $[0, 1]$

way to eliminate dangerous pockets of water stuck on flat convex portions is to gently push the tent fabric from below until gravity will lead water down.

As with many algorithmic principles, it is difficult to pinpoint a seminal paper in this area. The literature about stochastic local search for the Satisfiability (SAT) problem is of particular interest. Different variations of local search with randomness techniques have been proposed for Satisfiability and Maximum Satisfiability (MAX-SAT) starting from the late eighties, for some examples see [164], [324], and the updated review of [208]. These techniques were in part motivated by previous applications of "min-conflicts" heuristics in the area of Artificial Intelligence, see for example [161] and [271].

Before arriving at the objective function modifications, let's summarize the influential algorithm GSAT [324]. It consists of multiple runs of LS$^+$ local search, each one consisting of a number of iterations that is typically proportional to the problem dimension n. Let f be the number of satisfied clauses. At each iteration of LS$^+$, a bit which maximizes Δf is chosen and flipped, even if Δf is negative, i.e., after flipping the bit the number of satisfied clauses decreases.

The algorithm is briefly summarized in Fig. 35.16. A certain number of tries (MAX-TRIES) is executed, where each try consists of a number of iterations (MAX-FLIPS). At each iteration a variable is chosen by two possible criteria and then flipped by the function FLIP, i.e., x_i becomes equal to $(1 - x_i)$. One criterion, active with *noise* probability p, selects a variable occurring in some unsatisfied clause with uniform probability over such variables, the other one is the standard method based on the function f given by the number of satisfied clauses. For a generic move μ applied at iteration t, the quantity $\Delta_\mu f$ (or Δf for short) is defined as $f(\mu\ X^{(t)}) - f(X^{(t)})$. The straightforward book-keeping part of the algorithm is not shown. In particular, the best assignment found during all trials is saved and

reported at the end of the run. In addition, the run is terminated immediately if an assignment is found that satisfies all clauses. Different noise strategies to escape from attraction basins are added to GSAT in [322, 323]. In particular, the GSAT-with-walk algorithm.

The breakout method suggested in [274] for the constraint satisfaction problem measures the cost as the sum of the weights associated to the violated constraints (to the nogoods). Each weight is one at the beginning, at a local minimum the weight of each nogood is increased by one until one escapes from the given local minimum (a breakout occurs).

Clause–weighting has been proposed in [321] in order to increase the effectiveness of GSAT for problems characterized by strong asymmetries. A positive weight is associated to each clause to determine how often the clause should be counted when determining which variable to flip. The weights are dynamically modified and the qualitative effect is that of "filling in" local optima while the search proceeds. Clause–weighting and the breakout technique can be considered as "reactive" techniques where a repulsion from a given local optimum is generated in order to induce an escape from a given attraction basin. The local adaptation is clear: weights are increased until the original local minimum disappears, and therefore the current weights depend on the local characteristic of a specific local minimum point.

In detail, a weight w_i is associated to each clause, and the guiding evaluation function becomes not a simple count of the satisfied clauses but a sum of the corresponding weights. New parameters are introduced and therefore new possibilities for tuning the parameters based on feedback from preliminary search results. The algorithm in [325] suggests a different way to use weights to encourage more priority on satisfying the "most difficult" clauses. One aims at **learning how difficult a clause is to satisfy**. These hard clauses are identified as the ones which remain unsatisfied after a try of local search descent followed by plateau search. Their weight is increased so that future runs will give them more priority when picking a move. If weights are only increased, after some time their size becomes large and their relative magnitude will reflect the overall statistics of the SAT instance, more than the local characteristics of the portion of the search space where the current configuration lies. To combat this problem, two techniques are proposed in [130], either *reducing* the clause weight when a clause is satisfied, or by a weight decay scheme (each weight is reduced by a factor ϕ before updating it). Depending on the size of the increments and decrements, one achieves "continuously weakening incentives not to flip a variable" instead of the strict prohibitions of Tabu Search. The second scheme takes the *recency of moves* into account, the implementation is through a weight decay scheme updating so that each weight is reduced before a possible increment by δ if the clause is not satisfied:

$$w_i \leftarrow \ \phi \, w_i + \delta$$

where one introduces a decay rate ϕ and a "learning rate" δ. A faster decay (lower ϕ value) will limit the temporal extension of the context and imply a faster forgetting of old information. The effectiveness of the weight decay scheme is interpreted by the authors as "learning the best way to conduct local search by discovering the hardest clauses relative to recent assignments." Some collateral damage (*warping* effects) can be caused clause-weighting dynamic local search on the fitness surface [360]. The fitness surface is changed in a *global* way after encountering a local minimum. Points which are very far from the local minimum, but which share some of the unsatisfied clauses, will also see their values changed. This does not correspond to the naive "push-up" picture where only the area close to a specific local minimum is raised, and the effects on the overall search dynamics are far from simple to understand. Be aware of dangerous analogies!

A more recent proposal of a dynamic local search (DLS) for SAT is in [361]. The authors start from the Exponentiated Sub-Gradient (ESG) algorithm [317], which alternates search phases and weight updates, and develop a scheme with low time complexity of its search steps: Scaling and Probabilistic Smoothing (SAPS). Weights of satisfied clauses are multiplied by α_{sat}, while weights of unsatisfied clauses are multiplied by α_{unsat}, then all weights are smoothed towards their mean $\bar{w}$: $w \leftarrow w \ \rho + (1 - \rho) \ \bar{w}$. A *reactive version* of SAPS (RSAPS) is then introduced that adaptively tunes one of the algorithm's important parameters.

31.1 Guided local search

While we concentrated on the SAT problem above, a similar approach has been proposed with the term of Guided Local Search (GLS) [371, 373] for other applications. GLS aims at enabling intelligent search schemes that exploit problem- and search-related information to guide a local search algorithm in a search space. Penalties depending on solution features are introduced and dynamically manipulated to distribute the search effort over the regions of a search space.

Let us stop for a moment with an historical digression to show how many superficially distinct concepts are in fact deeply related. Inspiration for GLS comes from a previously proposed neural net algorithm (GENET) [375] and from tabu search [148], simulated annealing [234], and tunneling [252]. The use of "neural networks" for optimization consists of setting up *a dynamical system whose attractors correspond to good solutions of the optimization problem*. Once the dynamical system paradigm is in the front stage, it is natural to use it not only to search for but also to escape from local minima. According to the authors [372], GENET's mechanism for escaping resembles *reinforcement learning* [18]: patterns in a local minimum are stored in the constraint weights and are discouraged to appear thereafter. GENET's learning scheme can be viewed as a method to *transform the objective function so that a local minimum gains an artificially higher value*. Consequently, local search will be able to leave the local minimum state and search other parts of the space. In tunneling algorithms [252] the modified objective function is called the tunneling function. This function allows local search to explore states which have higher costs around or further away from the local minimum, while aiming at nearby states with lower costs. In the framework of continuous optimization similar ideas have been rediscovered multiple times. Rejection-based stochastic procedures are presented in [252, 15, 282]. Citing from a seminal paper [252], one combines "a minimization phase having the purpose of lowering the current function value until a local minimizer is found and a tunneling phase that has the purpose of finding a point ... such that when employed as starting point for the next minimization phase, the new stationary point will have a function value no greater than the previous minimum found." The "strict" prohibitions of tabu search become "softer" penalties in GLS, which are determined by *reaction to feedback from the local optimization heuristic under guidance* [373].

A complete GLS scheme [373] defines appropriate solution features f_i, for example the presence of an edge in a TSP path, and combines three ingredients:

feature penalties p_i to diversify the search away from already-visited local minima (the *reactive* part)

feature costs c_i to account for the *a priori* promise of solution features (for example the edge cost in TSP)

a neighborhood activation scheme depending on the current state.

The *augmented cost function* $h(X)$ is defined as:

$$h(X) = f(X) + \lambda \sum_i p_i \; I_i(X) \tag{31.1}$$

where $I_i(X)$ is an indicator function returning 1 if feature i is present in solution X, 0 otherwise. The augmented cost function is used by local search instead of the original function.

Penalties are zero at the beginning: there is no need to escape from local minima until they are encountered! Local minima are then the "learning opportunities" of GLS: when a local minimum of h is encountered the augmented cost function is modified by updating the penalties p_i. One considers all features f_i present in the local minimum solution X' and increments by one the penalties which maximize:

$$I_i(X') \; \frac{c_i}{1 + p_i} \tag{31.2}$$

The above mechanism kills more birds with one stone. First a higher cost c_i, and therefore an inferior *a priori* desirability for feature f_i in the solution, implies a higher tendency to be penalized. Second, the penalty p_i, which is also a counter of how many times a feature has been penalized, appears at the denominator, and therefore discourages

penalizing features which have been penalized many times in the past. If costs are comparable, the net effect is that penalties tend to alternate between different features present in local minima.

GLS is usually combined with "fast local search" FLS. FLS includes implementation details which speedup each step but do not impact the dynamics and do not change the search trajectory, for example an incremental evaluation of the h function, but it also includes qualitative changes in the form of *sub-neighborhoods*. The entire neighborhood is broken down into a number of small sub-neighborhoods. Only active sub-neighborhoods are searched. Initially all of them are active, then, if no improving move is found in a sub-neighborhood, it becomes inactive. Depending on the move performed, a number of sub-neighborhoods are activated where one expects improving moves to occur as a result of the move just performed. For example, after a feature is penalized, the sub-neighborhood containing a move eliminating the feature from the solution is activated. The mechanism is equivalent to prohibiting examination of the inactive moves, in a tabu search spirit. As an example, in TSP one has a *sub-neighborhoods* per city, containing all moves exchanging edges where at least one of the edges terminates at the given city. After a move is performed, all *sub-neighborhoods* corresponding to cities at the ends of the edges involved in the move are activated, to favor a chain of moves involving more cities.

While the details of *sub-neighborhoods* definition and update are problem-dependent, the lesson learned is that much faster implementations can be obtained by avoiding a brute-force evaluation of the neighborhood, the motto is "evaluate only a subset of neighbors where you expect improving moves." In addition to a faster evaluation per search step one obtains a possible additional diversification effect related to the implicit prohibition mechanism. This technique to speedup the evaluation of the neighborhoods is similar to the "don't look bits" method in [50]. One flag bit is associated to every node, and if its value is 1 the node is not considered as a starting point to find an improving move. Initially all bits are zero, then if an improving move could not be found starting at node i the corresponding bit is set. The bit is cleared as soon as an improving move is found that inserts an edge incident to node i.

The parameter λ in equation (31.1) controls the importance of penalties w.r.t the original cost function: a large λ implies a large diversification away from previously visited local minima. A reactive determination of the parameter λ is suggested in [373].

While the motivations of GLS are clear, the interaction between the different ingredients causes a somewhat complicated dynamics. Let's note that different units of measure for the cost in equation (31.1) can impact the dynamics, something which is not particularly desirable: if the cost of edge in TSP is measured in kilometers the dynamics is not the same as if the cost is measured in millimeters. Furthermore, the definition of costs c_i for a general problem is not obvious and the consideration of the "costs" c_i in the penalties in a way duplicates the explicit consideration of the real problem costs in the original function f. In general, when penalties are added and modified, a desired effect (minimal required diversification) is obtained *indirectly* by modifying the objective function and therefore by possibly causing *unexpected effects*, like new spurious local minima, or shadowing of promising yet-unvisited solutions. For example, an unexplored local minimum of f may not remain a local minimum of h and therefore it may be skipped by modifying the trajectory.

A penalty formulation for TSP including memory-based trap-avoidance strategies is proposed in [374]. One of the strategies avoids visiting points that are close to points visited before, a generalization of the STRICT-TS strategy [23]. A recent algorithm with an *adaptive clause weight redistribution* is presented in [206], it adopts resolution-based preprocessing and reactive adaptation of the total amount of weight to the degree of stagnation of the search. Stagnation is identified after a long sequence of flips without improvement, long periods of stagnation will produce "oscillating phases of weight increase and reduction."

31.2 Adapting noise levels

Up to know we have seen how to modify the objective function or the LS rules in a dynamic manner depending on the search history and current state in order to deviate the search trajectory away from local optimizer. Another possibility to reach similar (stochastic) deviations of the trajectory is by adding a controlled amount of randomized movements. *Clinamen* is the name the philosopher Lucretius gave to the spontaneous microscopic swerving of atoms from a vertical path as they fall, considered in discussions of possible explanation for free will. A kind of algorithmic *clinamen* can

be used to influence the diversification of an SLS technique. A related usage of "noise" is also considered in Sec 25.5 about Simulated Annealing, in which upward moves are accepted with a probability depending on a temperature parameter.

An opportunity for self-adaptation considering different amounts of randomness is given by **adaptive noise** mechanism for WalkSAT. In WalkSAT [322], one repeatedly flips a variable in an unsatisfied clause. If there is at least one variable which can be flipped without breaking already satisfied clauses, one of them is flipped. Otherwise, a noise parameter p determines if a random variable is flipped, or if a greedy step is executed, with probability $(1-p)$, favoring minimal damage to the already satisfied clauses.

In [265] it appears that appropriate noise settings achieve a good balance between the greedy "steepest descent" component and the exploration of other search areas away from already considered attractors. The work in [265] considers this generalized notion of a noise parameter and suggests tuning the proper noise value for a specific instance by testing different settings through a preliminary series of short runs. Furthermore, the suggested statistics to monitor, which is closely related to the algorithm performance, is the *ratio* between the average final values obtained at the end of the short runs and the variance of the f values over the runs. Quite consistently, the best noise setting corresponds to the one leading to the lowest empirical ratio increased by about 10%. faster tuning can be obtained if the examination of a predefined series of noise values is substituted with a faster adaptive search which considers a smaller number of possible values, see [294] which uses Brent's method [67]. An adaptive noise scheme is also proposed in [188], where the noise setting p is dynamically adjusted based on search progress. Higher noise levels are determined in a reactive manner if and only if there is evidence of search stagnation. In detail, if too many steps elapse since the last improvement, the noise value is increased, while it is gradually decreased if evidence of stagnation disappears. A different approach based on optimizing the noise setting on a given instance prior to the actual search process (with a fixed noise setting) is considered in [294].

Gist

Imagine dynamic pricing for street parking: the higher the price, the more people will avoid parking at city centers. The "fitness landscape" for the appeal of parking spaces changes and drivers modify their trajectories to stay in the periphery.

In a similar manner, when local minima are discovered and saved in memory, and therefore they become uninteresting for additional explorations, **artificially raising the value of the objective function** can be a way to encourage the search process to stay away from the known local minima. Unfortunately, depending on how dynamic penalties are placed, some new promising solutions cab be hidden, so that the method has to be used with great care.

Adapting the level of randomness during the search is a second mechanism to tradeoff intensification and exploration. By turning the "noise knob" one transforms a greedy perturbative search more and more into a random walk. One aims at a compromise which is appropriate for the local characteristics along the optimization trajectory of a specific instance.

Chapter 32

Adaptive Random Search

Men nearly always follow the tracks made by others and proceed in their affairs by imitation, even though they cannot entirely keep to the tracks of others or emulate the prowess of their models. So a prudent man should always follow in the footsteps of great men and imitate those who have been outstanding. If his own prowess fails to compare with theirs, at least it has an air of greatness about it. He should behave like those archers who, if they are skilful, when the target seems too distant, know the capabilities of their bow and aim a good deal higher than their objective, not in order to shoot so high but so that ***by aiming high they can reach the target****."*

(Niccolo Machiavelli)

In many real-world situations partial derivatives cannot be used because the function is not differentiable or because computing derivatives is too expensive. This motivates the study of optimization techniques based only on the knowledge of function values, like variants of the **adaptive random search** algorithm based on the theoretical framework of [340].

The general scheme starts by choosing an initial point in the configuration space and an initial *search region* surrounding it and proceeds by iterating the following steps.

1. A new candidate point is generated by **sampling the search region** according to a given probability measure.

2. The **search region is adapted** according to the value of the function at the new point. It is compressed if the new function value is greater than the current one (unsuccessful sample) or expanded otherwise (successful sample).

3. If the sample is successful, the new point becomes the current point, and the **search region is moved** so that the current point is at its center for the next iteration.

For effective implementations, simple search regions around the current point suffice, for example regions defined by *boxes* (with edges given by a set of linearly independent vectors) with uniform probability distributions inside the box. In this case generating a random displacement is simple: the basis vectors are multiplied by random numbers in the real range (-1.0, 1.0) and added: $\boldsymbol{\delta} = \sum_j \text{Rand} \times \boldsymbol{b}_j$.

The requirement that the box edges are parallel to the coordinate axes can be relaxed so that the frames can be compressed or expanded along arbitrary directions by using affine transformations, as explained in the following section.

32.1 RAS: adaptation of the sampling region

A simple but surprisingly effective self-adaptive and derivative-free method is the **Reactive Affine Shaker (RAS)** algorithm [69], based on [36]. RAS adapts a search region by an affine transformation. An affine transformation (from the Latin, *affinis*, "connected with") between two vector spaces consists of a linear transformation followed by a translation:

$$x \mapsto Ax + b.$$

Geometrically, an affine transformation in Euclidean space preserves:

- (i) the collinearity relation between points; i.e., the points which lie on a line continue to be collinear after the transformation,

- (ii) ratios of distances along a line; i.e., for distinct collinear points $p_1$1,$p_2$2,$p_3$3, the ratio $|p_2 - p_1|/|p_3 - p_2|$ is preserved. In general, an affine transformation is composed of linear transformations (rotation, scaling or shear) and a translation (or "shift").

In RAS, the region is translated when a successful sample is found, elongated along arbitrary success directions, and compressed along the unsuccessful ones. The modification takes into account the *local knowledge* derived from trial points generated with a uniform probability in the search region. The aim is to scout for local minima in the attraction basin where the initial point falls, by adapting the step size and direction to maintain heuristically **the largest possible movement per function evaluation**. The design is complemented by the analysis of some strategic choices, like the double-shot strategy and the initialization [69]. Let's now comment on the name (Reactive Affine Shaker). The solver's movements try to minimize the number of jumps towards the minimum region, and this is achieved by constantly changing the movement direction and size. Search region and therefore step adjustments are implemented by a feedback loop guided by the evolution of the search itself, therefore implementing a "reactive" self-tuning mechanism. The generation of samples is *tuned to the local properties* of the f surface, in the spirit of the Reactive Search Optimization principles explained in Chapter 24. The constant change in step size and direction creates a "shaky" trajectory, with abrupt leaps and turns.

The pseudo-code of the RAS algorithm is shown in Fig. 32.1. At every iteration, a displacement $\boldsymbol{\Delta}$ is generated so that the point $\boldsymbol{x} + \Delta$ is uniformly distributed in the local search region $\mathcal{R}$ (line 4). To this end, the basis vectors are multiplied by different random numbers in the real range $[-1, 1]$ and added:

$$\boldsymbol{\Delta} = \sum_j \text{Rand}(-1, 1)\boldsymbol{b}_j.$$

f	(input)	Function to minimize
$\boldsymbol{x}$	(input)	Initial point
$\boldsymbol{b}_1, \ldots, \boldsymbol{b}_d$	(input)	Vectors defining search region $\mathcal{R}$ around $\boldsymbol{x}$
ρ	(input)	Box expansion factor
t	(internal)	Iteration counter
$\mathbf{P}$	(internal)	Transformation matrix
$\boldsymbol{x}$, $\boldsymbol{\Delta}$	(internal)	Current position, current displacement

```
1.  function ReactiveAffineShaker (f, x, ( b_j), ρ)
2.      t ← 0;
3.      repeat
4.          Δ ← Σ_j Rand(−1, 1) b_j;
5.          if f( x + Δ) < f( x)
6.              x ← x + Δ;
7.              P ← I + (ρ − 1) ΔΔ^T / ‖ Δ ‖^2;
8.          else if f( x - Δ) < f( x)
9.              x ← x - Δ;
10.             P ← I + (ρ − 1) ΔΔ^T / ‖ Δ ‖^2;
11.         else
12.             P ← I + (ρ^−1 − 1) ΔΔ^T / ‖ Δ ‖^2;
13.         ∀j b_j ← P b_j;
14.         t ← t+1
15.     until convergence criterion;
16.     return x;
```

Figure 32.1: The Reactive Affine Shaker pseudo-code.

Rand$(-1, 1)$ represents a call of the random-number generator. If one of the two points $\boldsymbol{x} + \Delta$ or $\boldsymbol{x} - \Delta$ improves the function value, then it is chosen as the next point. Let us call $\boldsymbol{x}'$ the improving point. In order to enlarge the box along the promising direction, the box vectors $\boldsymbol{b}_i$ are modified as follows.
The direction of improvement is $\boldsymbol{\Delta}$. Let us call $\boldsymbol{\Delta}'$ the corresponding vector normalized to unit length:

$$\boldsymbol{\Delta}' = \frac{\boldsymbol{\Delta}}{\|\boldsymbol{\Delta}\|}.$$

Then the projection of vector $\boldsymbol{b}_i$ along the direction of $\boldsymbol{\Delta}$ is

$$\boldsymbol{b}_i|_{\boldsymbol{\Delta}} = \boldsymbol{\Delta}'(\boldsymbol{\Delta}' \cdot \boldsymbol{b}_i) = \boldsymbol{\Delta}'\boldsymbol{\Delta}'^T\boldsymbol{b}_i.$$

To obtain the desired effect, this component is enlarged by a coefficient $\rho > 1$, so the expression for the new vector $\boldsymbol{b}'_i$ is

$$\begin{aligned} \boldsymbol{b}'_i &= \boldsymbol{b}_i + (\rho - 1)\boldsymbol{b}_i|_{\boldsymbol{\Delta}} \\ &= \boldsymbol{b}_i + (\rho - 1)\boldsymbol{\Delta}'\boldsymbol{\Delta}'^T\boldsymbol{b}_i \\ &= \boldsymbol{b}_i + (\rho - 1)\frac{\boldsymbol{\Delta}\boldsymbol{\Delta}^T}{\|\boldsymbol{\Delta}\|^2}\boldsymbol{b}_i \\ &= P\boldsymbol{b}_i, \end{aligned} \tag{32.1}$$

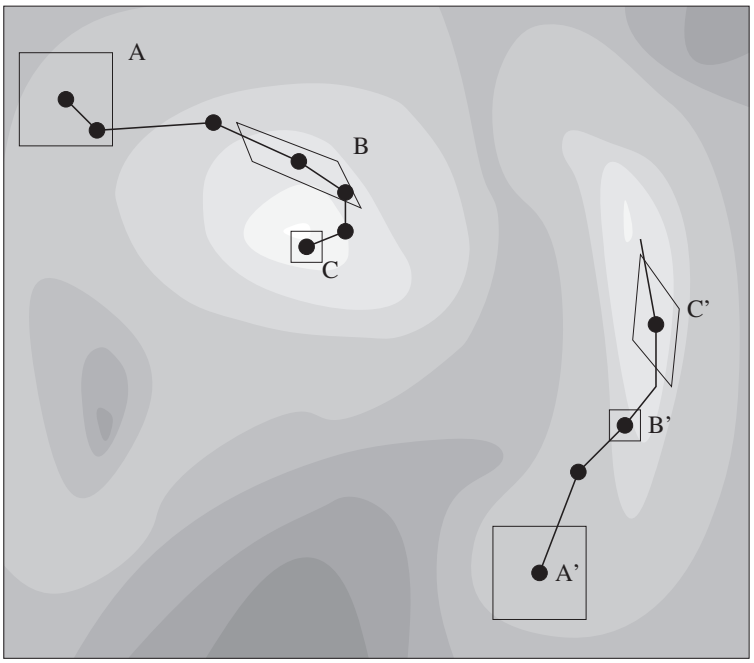


Figure 32.2: Reactive Affine Shaker geometry: two search trajectories leading to two different local minima, adapted from [69].

where

$$P = \mathbf{I} + (\rho - 1)\frac{\mathbf{\Delta}\mathbf{\Delta}^T}{\|\mathbf{\Delta}\|^2}. \tag{32.2}$$

The fact of testing the function improvement on both $\boldsymbol{x} + \mathbf{\Delta}$ and $\boldsymbol{x} - \mathbf{\Delta}$ is called *double-shot strategy*: if the first sample $\boldsymbol{x} + \mathbf{\Delta}$ is not successful, the specular point $\boldsymbol{x} - \mathbf{\Delta}$ is considered. This choice drastically reduces the probability of generating two consecutive unsuccessful samples. For a mental image, consider fitting a *plane* around the current point: if a step increases f, the opposite step decreases it. Going from mental images to math, if one considers differentiable functions and small displacements, the directional derivative along the displacement is proportional to the scalar product between displacement and gradient $\mathbf{\Delta} \cdot \nabla f$. If the first is positive, a change of sign will trivially cause a negative value, and therefore a decrease in f for a sufficiently small step size. The empirical validity for general functions, not necessarily differentiable, is caused by the correlations and structure contained in most of the functions corresponding to real-world problems.

If the double-shot strategy fails, then the affine transformation (32.1) is applied by replacing the expansion factor ρ with its inverse ρ^{-1} (line 12 of Fig. 32.1), causing a compression of the search area.

An illustration of the geometry of the Reactive Affine Shaker algorithm is shown in Fig. 32.2, where the function to be minimized is represented by a contour plot showing *isolines* at fixed values of f, and two trajectories (ABC and A'B'C') are plotted. The search regions are shown for some points along the search trajectory. The design criteria of RAS are given by an *aggressive search for local minima*: the search speed is increased when steps are successful (points A and A'), reduced only if no better point is found after the double shot. When a point is close to a local minimum, the repeated reduction of the search frame produces a very fast convergence of the search (point C). Note that another cause of reduction for the search region can be a narrow descent path (a "canyon", such as in point B'), where only a small subset of all possible directions improves the function value. However, once an improvement is found, the search region grows in the promising direction, causing a faster movement along that direction.

f	(input)	Function to minimize
ρ,	(input)	Box expansion factor
$L_1, \ldots, L_d, U_1, \ldots, U_d$	(input)	Search range
$L'_1, \ldots, L'_d, U'_1, \ldots, U'_d$	(input)	Initialization range
$\boldsymbol{b}_1, \ldots, \boldsymbol{b}_d$	(internal)	Vectors defining search region $\mathcal{R}$ around $\boldsymbol{x}$
$\boldsymbol{x}$, $\boldsymbol{x}'$	(internal)	Current position, final position of run

1. **function** RepeatedReaciveAffineShaker (f, ρ, (L'_j), (U'_j), (L_j), (U_j))
2. $\quad \forall j\ \boldsymbol{b}_j \leftarrow \dfrac{U_j - L_j}{4} \cdot \boldsymbol{e}_j$;
3. $\quad$ **pardo**
4. $\quad\quad \boldsymbol{x} \leftarrow$ random point $\in [L'_1, U'_1] \times \cdots \times [L'_d, U'_d]$;
5. $\quad\quad \boldsymbol{x}$' $\leftarrow$ ReactiveAffineShaker(f, $\boldsymbol{x}$, ($\boldsymbol{b}_j$), ρ);
6. $\quad$ **return** best position found;

Figure 32.3: The RAS algorithm, from [69].

32.2 Repetitions for robustness and diversification

In general, an effective estimation of the number of steps required for identifying a global minimum is clearly impossible. Even when a local minimum is found, it is generally impossible to determine whether it is global or not, in particular if the knowledge about the function derives only from evaluations of $f(x)$ at selected points, which is a frequent case in dealing with real-world applications.

Because RAS does not include mechanisms to escape from local minima, it should be stopped as soon as the trajectory is sufficiently close to a local minimizer. For instance, a single RAS run can be terminated if the search region becomes smaller than a threshold value. In fact, the box tends to reduce its volume in proximity of a local minimum because of repeated failures in improving the function value.

RAS searches for local minimizers and is stopped as soon as one is found. A simple way to continue the search is to **restart from a different initial random point**. This approach is equivalent to a "population" of RAS searchers, in which each member of the population is *independent*, completely unaware of what other members are doing (Fig. 32.3). More complex ways of coordinating a team of searchers are considered in the *C-LION* framework in Chapter 38.

Gist

Adaptive random search adapts a current **search region around the current tentative solutions**. New points are sampled from the search region. Depending on finding better or worse solutions, the search region is then adapted. In particular Reactive Affine Shaker (RAS) stretches or squeezes the search region along the direction of the latest successful or unsuccessful step, respectively.

This chapter concludes our presentation of RSO. To summarize, Reactive Search Optimization can **adapt in an online manner** at least the following choices or meta-parameters: prohibition periods ("stay away from this area"), neighborhood ("if no improving move, get another neighborhood"), iterations – not simply repetitions – of local search ("kick the system to a new attractor and call LS"), dynamic modifications of the objective function ("push up so that a local minimum disappears") amount of randomness in SA and other stochastic schemes ("noise level up to jump out of an attractor").

Part V

Special optimization problems and advanced topics

Chapter 33

Linear and Quadratic Programming

I think many people's deviant behavior starts with dreams because dreams are so non-linear... as if there's an assumption that everything has to be linear or has to be plotted.
(Robyn Hitchcock)

In the previous chapters we consider general-purpose methods for solving optimization problems, both discrete and continuous. But before considering the above techniques, it is worth checking if your problem belongs to **a few**

categories which can be solved exactly in acceptable (polynomial) time. These cases are not so frequent but some of them are incredibly useful for relevant applications.

Full-fledged real world applications with realistic constraints can *rarely* be solved to optimality (in the worst case) in acceptable CPU time. In theoretical computer science one abstracts from particular hardware and operating systems and studies how the worst-case CPU time grows when the input size grows. One does not care about constants but about rate-of-growth. A CPU time which increases as a polynomial in the input size n (e.g., like n^2 or n^5) is usually considered affordable. For sure, it is better than an exponential increase like 2^n which makes solving large-size problems impossible.

On the other hand, **a demonstration of global optimality is not required by most applications**, and it your problem cannot be solved in polynomial time in the worst case (for the worst possible input configuration) it does not mean that your problem cannot be addressed in practice. First, the input configurations causing very large CPU times can be very rare. Second, if a business obtains a 10% increase in profit by some optimization technique, the proof is in the pudding and it may be of academic interest to demonstrate that the optimal solution would have been an increase of 10.5%.

In any case, before starting your solution effort, you should always check that your problem is not in the list of those solvable in acceptable CPU time, with dedicated algorithms (updated lists of problems can be found in the web). Even if your case does not correspond exactly to one of these notable problems, in some cases you can consider radical simplifications leading to solvable cases, insight and deeper knowledge.

While we cannot cover many specialized cases in this introductory book, one interesting case is **Linear Programming (LP)**, the solution of problems with a **linear objective function and linear constraints**. By the way, "programming" has nothing to do with the modern meaning related to software, but with the organized solution with help of a tabular representation (a *tableau*). **Linear Optimization** can be a modern name, but less used. Let's consider the **diet problem**, originally motivated the 1930s by the Army's desire to minimize the cost of feeding GIs in the field while still providing a healthy diet. Its goal is to select a set of food quantities that will satisfy a set of daily nutritional requirement at minimum cost. The constraints regulate the minimum and maximum number of calories and the amount of vitamins, minerals, fats, sodium, etc. in the diet. LP is considered in Sec. 33.1.

Integer Linear Programming (ILP) has the same objective function and constraints, with the additional requirements that input values are integers, which makes the problem much more difficult to solve (Sec. 33.2). Imagine a diet problem in which foods are not "continuous-valued" like flour, but can be bought in boxes like canned soup in a supermarket.

Because the number of interesting problems is simply to large to consider in a single book, we concentrate on some relevant algorithm design principles, with at least a concrete example for each case: Linear and Quadratic Programming in this Chapter, Branch and bound (Sec. 34.1), Dynamic Programming (Sec. 34.2), Perturbative and Constructive Greedy in the introductory sections (Sec. 24.1).

33.1 Linear Programming (LP)

The diet problem can be easily stated as follows:

- Minimize the cost of food eaten during one day
- Subject to the requirements that the diet satisfy a person's nutritional requirements and that not too much of any one food be eaten.

The decision variables are the quantities for the different foods in an optimal diet. After knowing the relevant constants like the *cost per unit of weight* and the *content per unit of weight* of nutrients, vitamins etc., for the different foods, it is straightforward to demonstrate that both the objective function and the constraints are **linear functions of the decision variables**. In fact, the cost is a sum over all foods of "quantity times unitary cost", and the constraints will bound above or below sums over all foods of "quantity times unitary content". E.g., a sum will require calories to be in a certain range, another sum will require vitamin A to be in a second range, etc. In addition to the historic

diet problem, the number of applications of LP is wide and growing, including scheduling flight crews, drilling for oil based on geological surveys, solving many graph-related and combinatorial problems[102].

Let's now develop some **geometrical intuition** about minimizing a linear function subject to linear constraints. We already encountered linear functions and constraints (of the form $\boldsymbol{w} \cdot \boldsymbol{x}$) in the Chapters 4 and 5 about linear models . If x is one-dimensional, a linear function is a straight line, constraints are inequalities like $a \leq x$ or $x \leq b$. If there are values of x satisfying the inequalities, and the slope of the line is different from zero, it should be evident that an optimal solution will be one of the boundary values (a or b). Proof by contradiction: if optimal value x is an interior point, we can move it right or left (depending on the slope) to get a lower f value, a contradiction. Be careful: constraints like $a < x$ are *not* acceptable (imagine $f(a)$ is the minimum value). In fact, given a point close to a we could always find a smaller x value with a smaller f value: one needs **compact sets**, closed —that is, containing all its limit points— and bounded.

In higher dimensions constraints become planes and then hyper-planes, linear functions become inclined planes and gently varying functions (boring, without any curvature, with straight and equally-separated contour lines). For a concrete image, imagine a two-dimensional example, an inclined billiard table with four different legs. Minimization means leaving a billiard ball to reach the lowest possible position. Guess what, the billiard ball will stop at a corner. If the two lowest legs are equal, the ball may stop at another position along the lowest edge, but we can easily move it to one corner without spending energy.

Let's us now move from intuition to math. Linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints. LP can be expressed in **canonical** form as

$$\begin{aligned} \text{maximize} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \qquad (33.1) \\ \text{and} \quad & \boldsymbol{x} \geq 0 \end{aligned}$$

(33.2)

where $\boldsymbol{x}$ represents the vector of n variables (to be determined), $\boldsymbol{c}$ and $\boldsymbol{b}$ are vectors of coefficients, A is an $m \times n$ matrix of coefficients, one row for each of the m constraints. If the form is not canonical, simple transformations can lead to it [102].

There are two standard ways to reason about LP: **the geometrical and the algebraic view**. Let's consider the geometric way first.

The inequalities $Ax \leq b$ and $x \geq 0$ are the constraints which specify a geometrical figure known as **convex polytope** over which the objective function is to be optimized.

Each constraint, the scalar product between a row of matrix and the vector of variables: $A_i \cdot \mathbf{x} \leq b_i$, defines a half-space of points satisfying it. Because *all* constraints have to be satisfied, the *intersection* of the half-spaces, when bounded and non-empty, defines the **convex polytope**. Interestingly, the geometrical features can be defined also as the convex hull of a set of vertices of the polytope.

Convexity plays a big role. A set is **convex** if, given any two points A, B in that set, the line AB joining them lies entirely within that set. The demonstration that the LP polytope is convex is simple: a half-space is convex, and the intersection of convex spaces is convex (the line AB belongs to all half-spaces and therefore to the intersection).

The polytope boundary consists of *facets* of dimension $d - 1$ (where one or more constraints are satisfied with equality), *vertices* (faces of dimension zero, i.e., points) and *edges*, faces of dimension one, i.e., line segments.

The LP polytope contains an **infinite** number of points. A brute-force algorithm of the kind "consider all feasible points and output one with maximum value of the objective function" is impossible.

Luckily, an optimal solution can always be found at a vertex. For a demonstration, convexity and linearity are the keys. First, because the feasible region is convex and the objective function is linear, **a local optimum is a global optimum**. The demonstration is easy, imagine the above does not hold, there will be a point $\mathbf{x}_{loc}$ which is locally optimal but with a global optimum point $\mathbf{x}_{opt} \neq \mathbf{x}_{loc}$ of higher objective value. If one considers the line segment *connecting* the two points, because of convexity belonging to the feasible region, the objective along this segment

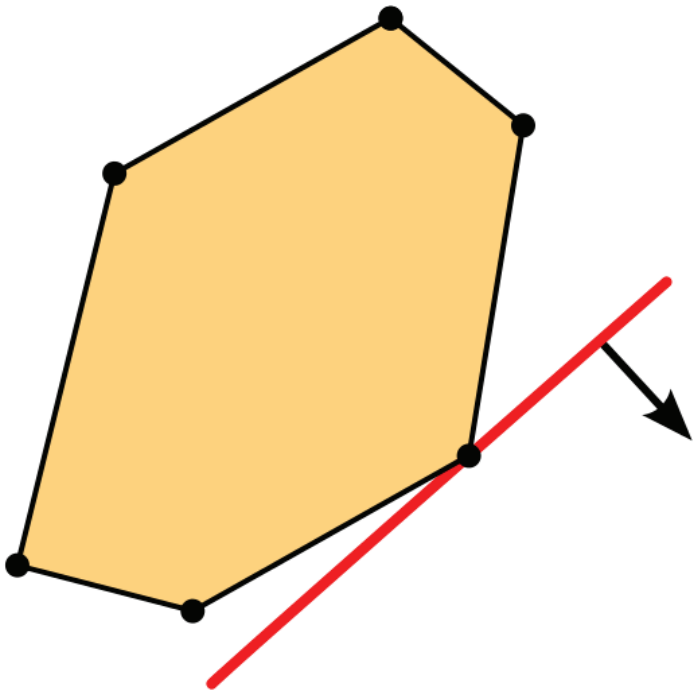


Figure 33.1: A a simple linear program with two variables and six inequalities. The set of feasible solutions is depicted in yellow and forms a polygon, a 2-dimensional polytope. The linear cost function is represented by the red line and the arrow: The red line is a level set of the cost function, and the arrow indicates the direction in which we are optimizing. Contour lines are also shown.

will increase. In particular, it will increase also for points along the segment in all local balls surrounding the local optimum, a contradiction with the fact that is is locally optimal.

In addition, a **global optimum can always be found among the vertices**. The proof is again by contradiction. Assume that a local optimum x_{opt} is in the interior of the LP polytope. If the gradient of the linear objective function is non-zero($\mathbf{c} \neq 0$), one can more away from the local optimum along the line $\mathbf{x}_{opt} + t\mathbf{c}$ $(t \geq 0)$ and obtain higher objective values, until a point on the boundary is met. If the point on the boundary is not a vertex, one considers the vectors spanning the facet or edge. Either the gradient is perpendicular (in which case moving along the faces will not change the objective), or its projection defines a direction to follow to get even higher objective values. In all cases one can safely move (getting higher or equal objective values) until a vertex is reached.

This is an important result: instead of considering an infinite number of points one can consider **only the vertices**. Actually, not all of them have to be checked. By the above conclusions that local optimality is sufficient one derives the **simplex algorithm** for LP, possibly among the ten most used algorithms in the world. The term simplex has to do with its operation on simplicial cones, the corners (i.e., the *neighborhoods of the vertices*) of the polytope.

The high-level description of the simplex algorithm is as follows.

- It starts at some vertex. If the problem is solvable (the feasible region is not empty), an initial vertex can be determined by applying the simplex algorithm to a modified and easily solvable version of the original program.
- A sequence of **local search** (LS) steps are executed, in which the possible neighbors (in local search terminology) of the current vertex are the vertices that can be reached by moving along an edge. LS always moves to an improving local vertex (one with higher objective value). Different versions have to do with rules for picking an

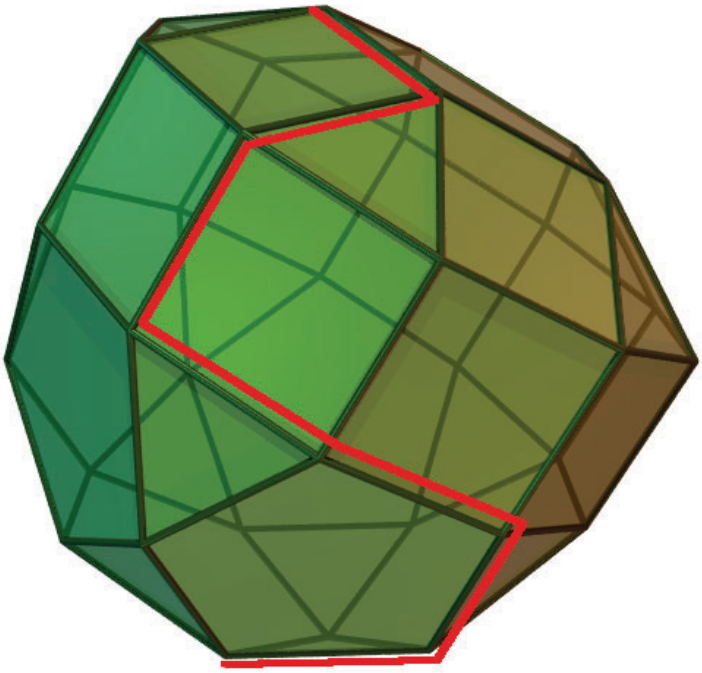


Figure 33.2: The simplex algorithm for solving LP problems.

improving vertex. In the LP terminology, a **pivot** is related to this passage to a neighboring vertex, and **pivoting rules** are LS rules for selecting an improving neighbor.

- Simplex terminates when it reaches a **local maximum**, a vertex from which all neighboring vertices have a smaller objective value. Because of convexity of feasible region and linearity of objective function, this is actually a global optimum.

The simplex algorithm works by constructing a feasible solution at a vertex of the polytope and then **walking along a path on the edges of the polytope** to vertices with non-decreasing values of the objective function until an optimum is reached.

A couple of questions are related to convergence (in a finite number of steps) and computational complexity (number of operations as a function of the dimension of the problem). For the first issue, the simple version above has to be cured o avoid possible cycles (endless repetitions of a sequence of vertices) in rare but possible situations. This "stalling" is possible only in the *degenerate* case of neighboring vertices with equal objective values. A randomized rule for picking a vertex in case of ties is a simple way to cure cycling. For the second issues, unfortunately he worst-case complexity of simplex method as formulated by Dantzig is exponential time. An exponential number of iterations is rarely encountered for many practical problems, but there is no guarantee.

The existence of solutions algorithms for LP with guaranteed polynomial complexity has been an open problem in computer science for many years. The LP problem was first shown to be solvable in polynomial time by Leonid Khachiyan in 1979, but a larger breakthrough in the field came in 1984 when Narendra Karmarkar introduced a new **interior-point method**. Nonetheless, the simplex algorithms with smart pivoting rules is still the *de facto* standard for most applications of LP.

33.1.1 An algebraic view of linear programming

Although the geometric view is intuitive, the algebraic view is useful to derive a workable algorithm. We refer to [287, 102] for detailed demonstrations and give only a short summary here. A first observation is that the boundaries of the polytope corresponds to some inequalities in the constraints becoming equality (the constraint is *tight* or *active*). If one sits at a vertex, a small change in some directions will make the point exit the feasibility region.

Now, dealing with equalities only is simpler than dealing with inequalities. Luckily, one can transform an LP problem into the **slack form**:

$$\begin{array}{cc} \text{maximize} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & A\mathbf{x} = \mathbf{b} \\ \text{and} & \mathbf{x} \geq \mathbf{0} \end{array} \tag{33.3}$$

by a simple trick of adding additional **slack variables**. For each constraint:

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \tag{33.4}$$

one introduces a new variable s and rewrites the above inequality as the two contraints:

$$\sum_{j=1}^{n} a_{ij}x_j + s = b_i \tag{33.5}$$

$$s \geq 0 \tag{33.6}$$

A constraint is satisfied if and only if there is a non-negative *slack* or difference between the left- and right-hand of equation (33.4).

If A is the $m \times n$ matrix, with $m < n$, the algebraic definition of a corner is obtained as follows. There are m linearly independent columns A_j of A, which make up a basis $\mathcal{B} = \{\mathcal{A}_{|\infty}, ..., \mathcal{A}_{|\Updownarrow}\}$ of the linear space spanned by all columns (A is of (full) rank, and "row rank equals column rank", therefore rank is the minimum of the two dimensions). We can collect the columns of the basis $\mathcal{B}$ as an $m \times m$ nonsingular (invertible) matrix $B = [A_{j_i}]$. Being a basis, all vectors in R^m can be derived by linear combination, in particular the vector $\mathbf{b}$, the other columns are not involved (their coefficient in the linear combination is zero).

A **basic solution** corresponding to the basis $\mathcal{B}$ is a vector $\mathbf{x} \in R^n$ with elements different from zero only for indices corresponding to the basis columns. In detail:

$$\begin{array}{rcll} x_j & = & 0 & \text{for} A_j \notin \mathcal{B} \\ x_{j_k} & = & \text{the } k\text{-th component of } B^{-1}\mathbf{b}, & k = 1, ..., m \end{array}$$

The connection between geometry and algebra is that **basic feasible solutions correspond to vertices** of the polytope.

When LP is formulated in the slack form, the simplex algorithm works with the system of equalities by rewriting it in an equivalent form. After a number of iterations, the system is rewritten so that the solution is immediate to obtain. In a way, **the simplex algorithm can be considered as a kind of "Gaussian elimination for inequalities"**.

At each step, the linear program is reformulated so that the current basic solution has a greater objective value. The action of moving along the edge of the polytope from a vertex to a neighboring one corresponds to changing the basis B. At each step one chooses a nonbasic variable (initially set to zero - not in the basis) which appears with a positive coefficient in the objective (if one increases the variable, the objective increases). The variable is raised away from zero until a constraint becomes tight (some basis variable becomes zero). At this point, one can rewrite the slack form, exchanging the roles of the basic and non-basic variables (a column exits the base B and a new column enters). After rewriting, if all multiplicative constants in the objective function are negative, we are done. By increasing non-basic variables we can only worsen the solution and therefore the vertex (basic feasible solution) is optimal. **Pivoting** corresponds of course to having a new non-basic variable enter the basis (entering variable) and a basic variable exit (leaving variable).

33.2 Integer Linear Programming

An Integer Linear Programming problem (ILP) is an LP problem with the additional constraints that the variables **x** must take on **integral values**. Imagine the diet problem, in which foods can only be bought in boxes of 1,2,3 ... kilograms.

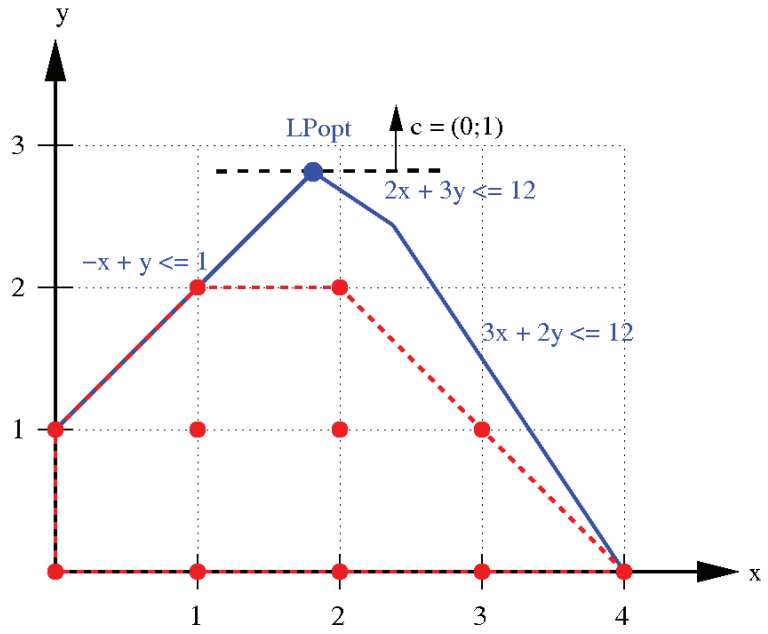


Figure 33.3: Integer Linear Program and relaxation.

Now, it can be demonstrated that ILP is NP-hard, therefore belonging to a hard class of problems for which there is unlikely to be a polynomial-time algorithm in the worst case. What looks like a "small" change in the definition (from real values to integers) completely changes the theoretical hardness of the problem. Actually, the problem is so hard that just determining whether an ILP problem has a feasible solution is NP-hard.

Intuitively, the feasible region is not a nice polytope anymore but a set of dots corresponding to integer (feasible) coordinates. No way to move continuously from vertex to vertex!

A heuristic way to proceed is to simply remove the constraint that **x** is integral, and solve the corresponding LP (called the **LP relaxation of the ILP**). The optimal LP value is for sure an *upper bound* (when one relaxes to real values, in particular one considers also integral values, and therefore has *more* possibilities to improve a solution). One can then round the entries of the solution to the LP relaxation to the nearest integers. Of course, a solution obtained by rounding may not be optimal, it may not even be feasible (it may violate some constraint). ILP problems can be solved with heuritic local search techniques (but of course abandoning hopes of guaranteed optimality in polynomial times).

33.3 Quadratic Programming (QP)

In addition to Linear Programming, a notable special case is that of **Quadratic Programming (QP)**, the problem of optimizing a *quadratic* function of several variables subject to linear constraints on these variables. An example has been encountered in dealing with Support Vector Machines (Chapter 12). Quadratic Programming is defined as:

$$\text{minimize} \frac{1}{2}\mathbf{x}^{\mathrm{T}} Q\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} \quad \text{subject to} \quad A\mathbf{x} \leq \mathbf{b}.$$

For positive-definite Q, the ellipsoid method solves the problem in polynomial time. But if Q is indefinite, the problem becomes NP-hard. The mental image for positive-definite Q is that of Fig. 26.8. QP is considered to be solvable in practice for dimensions ranging up to thousands of variables (depending on the problem characteristics).

An excellent introduction of some classic algorithms for discrete (combinatorial) are [287] and [102], an updated "Bible of algorithms". More than on a list of special cases it is more interesting to concentrate on **high-level algorithm design principles** for optimization problems (with some concrete examples), in addition to the case of Local Search considered in Section 24.2.

Gist

Linear Programming, the solution of optimization problems with linear objective functions and linear constrains, creates mental images of gentle slopes and straight walls, and of balls ending up in low corners. LP is among the most widely used problems.

The analysis of LP and the development of the simplex algorithm is an elegant mixture of continuous (infinite) but convex feasible areas, and local search among the discrete set of vertices. The two different views (geometric and algebraic) are useful to develop insight about the problem structure and its solution.

LP can be solved in polynomial time, although the most used algorithm (the simplex) has no worst-case guarantee of always converging in a polynomial number of steps.

Remember that small changes in the definition can make huge changes in the difficulty of solving the problem: if input values are constrained to be integers the problem (ILP) becomes one of the hardest to be solved to optimality. But useful bounds and heuristic solutions can be obtained by relaxation: the problem is solved with continuous variables (LP) and the results is then rounded to the nearest integers.

Quadratic Programming can be used to solve efficiently problems with quadratic objective functions and linear constraints.

If you encounter an LP or QP in your business, enjoy, there is a wide variety of off-the-shelf software which can be immediately used for its solution.

Chapter 34

Branch and bound, dynamic programming

Big fish eats little fish.

The number of different optimization problems is so large that one can easily feel lost in a maze of details. Luckily, in many cases some **common underlying algorithm design principles** can be used for their solution. We therefore concentrate on the most relevant algorithmic patterns in addition to greedy and local search: Branch and Bound (Sec. 34.1) and Dynamic Programming (Sec. 34.2), with at least a concrete example for each algorithmic principle.

Branch and bound is a little smarter than the exhaustive enumeration of all possible solutions. When **branching** one considers all possible ways of fixing the value of a variable in the solution (leading to a subtree - a branch - in the visual representation of the process). In the construction of a complete solution from a partial one, a **bound** is associated to the current partial solution. One knows that, in whatever manner a solution is completed, one cannot obtain more than the value given by the bound. If this value is surpassed by the current "record" solution, the completion of the tentative solution is aborted and some iterations are spared.

Dynamic Programming is based on decomposing a problem, finding solutions to smaller instances, and re-using them to build solutions to larger and large instances, in a bottom-up manner. Because smaller instances are found many times, it makes sense to store their solutions in an organized memory structure. In an analogy, the big problem contains little problems in its bellies, which can be eaten to build the complete solutions.

34.1 Branch and bound

If one needs to find a best configuration out of a finite set, a brute-force algorithm is **exhaustive search**: generate all possible configuration and evaluate the corresponding objective-function values, deliver the best one.

If a configuration corresponds to picking specific values for a set of variables, the entire set of configurations can be organized as a **solution tree**: the root is the starting situation (no decision yet taken - no value given to the variables)), the first level corresponds to different (but finite) ways of assigning a value to the first variable, the second-level children correspond to the different values for the second variable, etc. **Complete solutions** correspond to the leaves of the tree, while internal nodes are **partial solutions**. Let's note that the tree is not unique, variables can be ordered in different ways and not necessarily balances (some variables can be assigned values conditionally on the values of other ones). Exhaustive search then is implemented by generating the entire tree and examining its leaves.

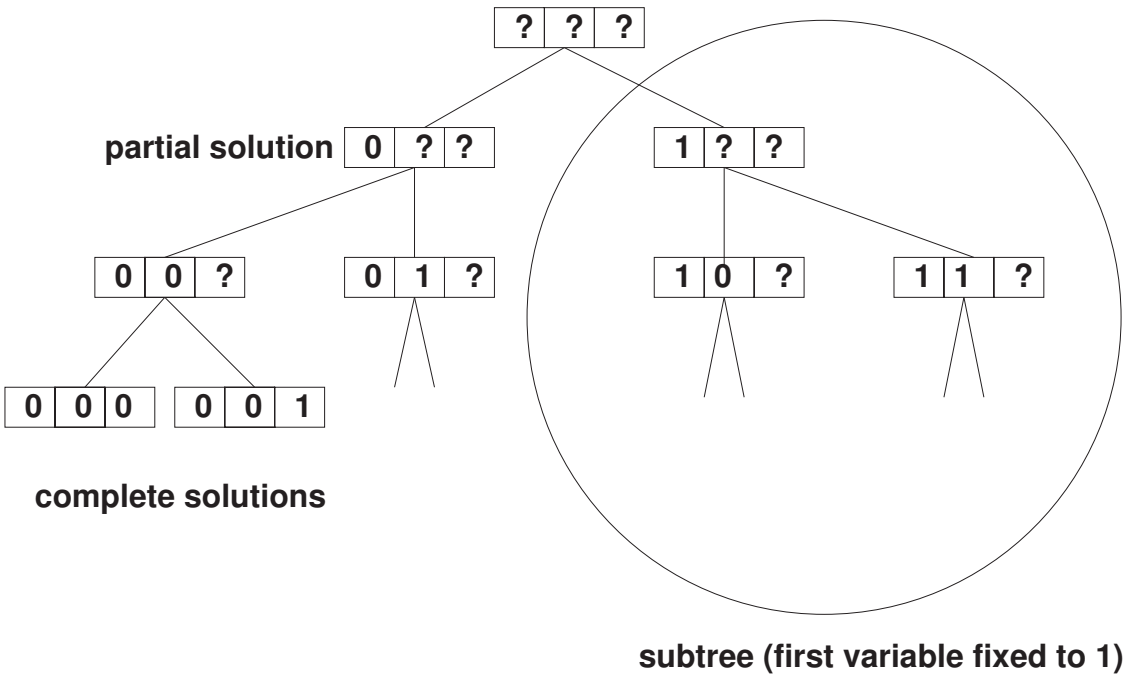


Figure 34.1: A solution tree for a binary sting with three variables.

As a concrete example, if the solution consists of a binary string, after a specific subset of bits is fixed (to 1 or 0) in a **partial solution**, one is left with the possibility to fix in all possible ways the remaining variables ("free variables"). When then the first free variable is fixed to zero one gets the left subtree, when it is fixed to one one gets the right subtree (Fig 34.1).

As you imagine, exhaustive search is too simple to be a practical solution for most cases: the difficulty lies in the enormous number of leaves, which can grow exponentially (e.g., if C values are possible for n variables, the total number of possibilities is C^n).

A trembling flame of hope in the tunnel of exhaustive search, which is some cases can lead to a notable saving of CPU time, is called **Branch and Bound (BB or B&B)**. The core idea is that some parts of the trees can be "**pruned**" (do not need to be generated) because one can demonstrate that **some partial solutions have no hope of becoming optimal** when completed. **B&B** was used al lot in the early stages of Artificial Intelligence. A generalization of branch and bound also subsumes the A*, B* and alpha-beta search algorithms from artificial intelligence [279]. In most cases, although a lot of CPU time can be spared, the total remains exponential and the size of instances which can be solved

to optimality increases only by a small quantity. Nonetheless, B&B can be used also with early stopping and therefore becomes a useful heuristic when an optimal solution cannot be guaranteed in the allotted CPU time.

B&B is an algorithm design paradigm, mostly for discrete and combinatorial optimization problems. It consists of a systematic enumeration of candidate solutions by means of state space search. Before we considered nodes as decision points (fixing the value of a variable). An alternative and complementary view is to associates nodes with **sets of solutions**, the complete slutions which can be obtained by fixing the remaining free variables in all possible ways. The full set of candidate solutions is at the root. The algorithm explores branches of this tree, which represent subsets of the solution set (all leaves of the subtree originating at a node). **Branching** means assigning a value to a variable in a partial solution and finding, in a recursive manner, the best value in the corresponding subtree. Before enumerating the candidate solutions of a branch, the branch is checked against an **optimistic bound** on the optimal solution, and is discarded if it cannot produce a better solution than the current record value (the best value found so far by the algorithm). We talk about "optimistic" bound to avoid the usual confusion between upper or lower bound, depending on the max or min direction of optimization. The advantage w.r.t. exhaustive search depends on the existence, quality and speed of computation of the *optimistic bound* used for pruning. If the bound is tight, a big fraction of the search space can be cut.

To summarize (and after deciding for minimization), branch-and-bound aims at minimizing the value of a real-valued function $f(x)$, in which $x \in S$, the set of admissible, or candidate solutions, also called **search space**. A B&B algorithm operates according to two principles:

1. It recursively splits the search space into smaller spaces (**branching**), then minimizing $f(x)$ on these smaller spaces.

2. It keeps track of an **optimistic bound** on the minimum value which can be reached by completing a given partial solution in all possible ways. If the bound is larger than the current record value, the current subtree is pruned, and control is returned to the first encountered higher level in the tree which has alternative ways to fix variables not yet explored. This operation is called **backtracking**.

If one eliminates backtracking one obtains a single construction. The construction is greedy if the most promising choice is executed at each step. Greedy is myopic and cannot undo early choices. With backtracking, early choices will be undone in a systematic manner, until all possibilities are eventually tried. As one can imagine, programming languages with **recursive function calls** permit very elegant software implementations of branch-and-bound methods.

In visiting the solution tree starting from the root, like in all graph visits, one can proceed depth-first or breadth-first. The **depth-first** variant (going down the tree to produce an initial complete solution) quickly produces complete solutions, and therefore record values. Intelligent implementations aim at producing a first high-quality solution early in the search, so that its record value will help in pruning many future subtrees.

Fig. 34.3 shows a toy example related to maximizing the number of queens which can be placed on a chessboard so that no queen can attack another queen. For each positioned queen, no other queen can be placed in the same row, in the same column and in the two diagonals centered on the queen (Fig. 34.3). The first level of the tree is given by placing the first queen in all possible squares in the first row. The second level is obtained by placing the second queen in all possible *admissible* squares in the second row. The inadmissible squares do not lead to a subtree to explore, and are immediately dismissed. Each subtree is deepened until no additional queen can be placed. When this happens, the system backtracks to the first higher level in the tree with alternatives which have not been tried yet. A simple optimistic bound on the *new* queens that can be placed is given by the number of free squares remaining after eliminating all rows, columns and diagonals in which a queen is already placed.

Notice that Branch and Bound (much like Brute Force enumeration of all possible solutions) creates a search tree that explodes exponentially as the problem size increases. Therefore it will hit the same scalability wall, only a little bit later. In spite of its popularity in the initial study of Artificial Intelligence, Branch And Bound is unusable for most real-world problem due to its CPU time requirements. In any case, careful implementations are in some cases crucial to pass from solving problems with a few variables (say up to 5 or 10) to solving bigger problems, say with up to 20 or 30 variables. In some cases this permits to go from toy problems to cases which are closer to more complex real-world

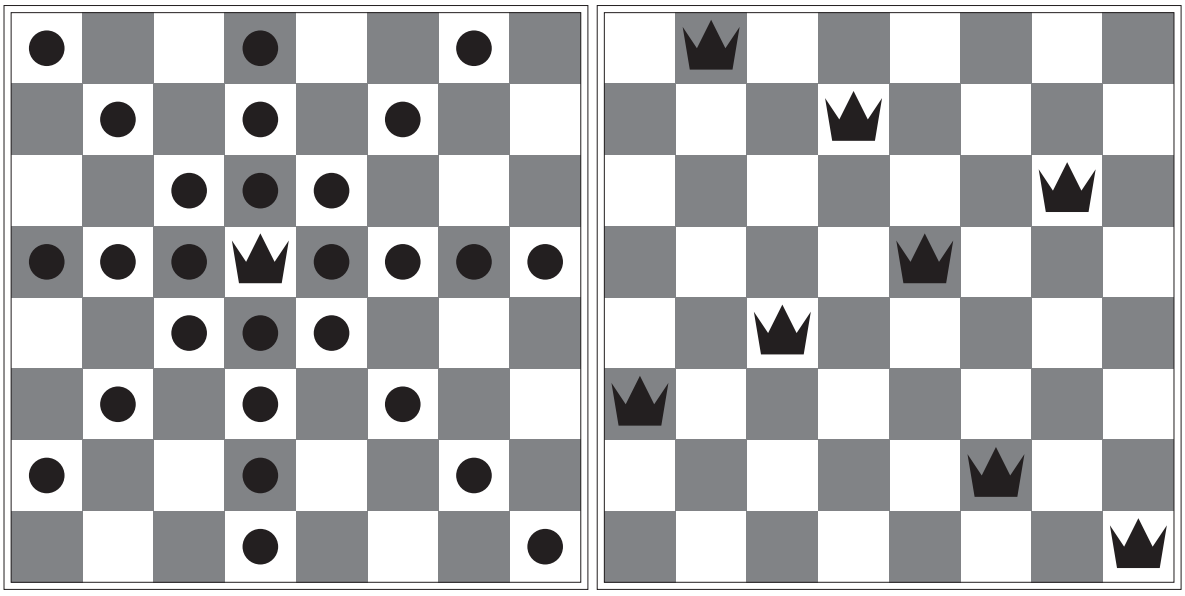


Figure 34.2: In chess, a queen is allowed to "take" any other queen on the board by moving along row, column or diagonals (left). Eight queens (right), is this a solution so that no queen can eat another one?

situations. On the other hand, when used as an heuristic (with early termination and no proof of optimality) B&B can be competitive if one employs smart bounds and rapid creation of good record values.

34.2 Dynamic programming

In some cases, the optimal solutions has a **structure with interesting nesting characteristics**, which can help in building it efficiently and in demonstrating its optimality. In Dynamic Programming, "*big fish eat little fish*", big problems have **shared optimal subproblems** in their bellies.

Dynamic programming (DP) shares with the general divide-and-conquer method the principle of solving problems by combining solutions to subproblems, with the specific flavor that the subproblems are not independent, i.e., **problems share subproblems**. DP solves each subproblem *only once* and then saves its result in a table, therefore avoiding useless re-computation. As it was the case for LP, "programming" has nothing to do with software but with the use of a tabular solution method.

DP algorithms, originally studied by R. Bellman in 1955, are developed according to four steps:

1. Study and characterize the structure of an optimal solution (how it *contains solutions to smaller instances*).
2. *Define the value* of an optimal solution *in a recursive manner* from the values of solutions of subproblems.
3. *Compute the value* of the optimal solution in a bottom-up manner (by starting from the smallest subproblems).
4. *Construct* an optimal solution from information computed while finding the value.

The above may sound abstract but dynamic programming is actually fundamental in making cellular phones work (Viterbi's algorithm), in bio-informatics (finding longest common subsequences in DNA), scheduling to maximize profit, in calculating shortest paths, and in hundreds of concrete applications including menial ones like pretty-printing and LaTEx formatting. Some for of DP has been used also for making this book look nice!

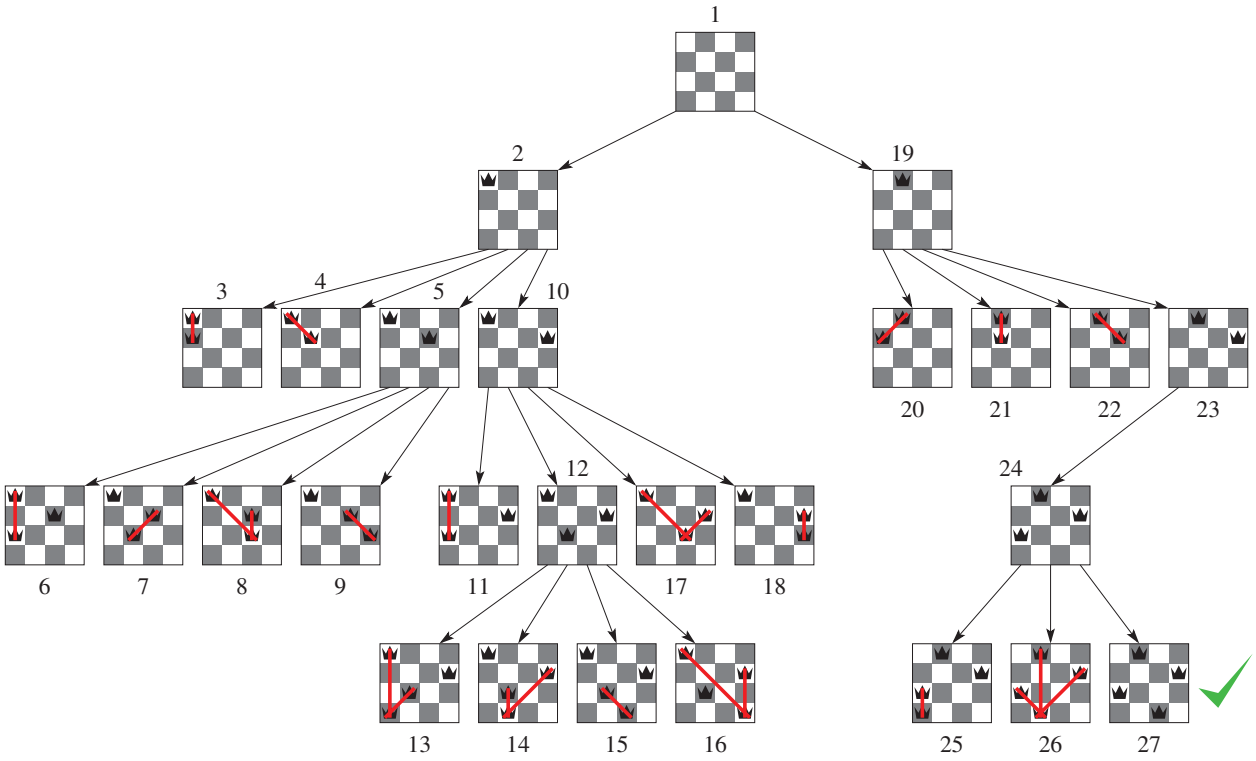


Figure 34.3: Branch and bound in action to maximize the number of queens placed on a 4×4 chessboard. Queens are positioned one row after another one. Positions are shown up to the first legal solution.

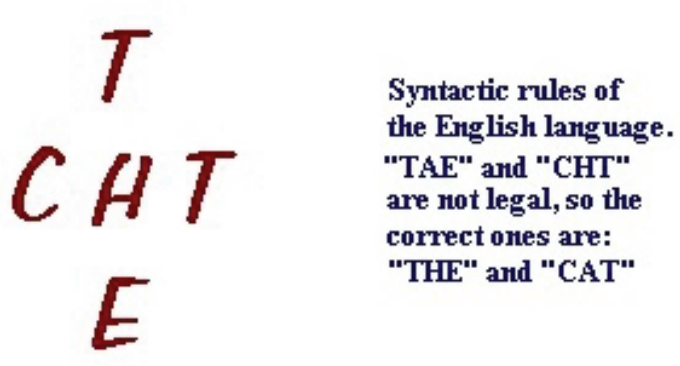


Figure 34.4: Viterbi's dynamic programming algorithm applied to text recognition.

We focus on **Viterbi's algorithm** here.

To help the intuition, let's consider a simplified problem of text recognition first (Fig. 34.4, 34.5). Recognition of characters from a text source can be done Optical Character Recognition (OCR) machines. The recognition of text can work by trying to recognize *individual* characters but the error rate can be large if the image is corrupted by noise or if the same letter can be written in many different ways. Imagine interpreting a hand-written prescription of medicines by doctors. Pharmacists can do it because they have a large amount of additional background information.

A model going beyond individual characters considers the fact that written text is a *sequence* generated from an underlying correct word or phrase. The probability of recognizing a character at a certain position depends on the underlying sequence, and therefore on the context given by additional nearby characters. In Fig. 34.4 ("THE-CAT" example) the recognition of the central ambiguous character as "H" or "A" is for sure influenced by the presence of a previous character "T" or "C". In the same manner, our brain recognizes spoken words and phrases even in very noisy and difficult situations, by considering its sequential nature and a lot of additional contextual knowledge.

Sequences abound in the real-world, as well as simple models to calculate overall probabilities of complete sequences. As noted before when discussing **maximum a posteriori (MAP) probability**, the principle of **searching for the most probable sequence given the observed signals and given a model of sequence generation** is a sound heuristic principle for identifying a "hidden" explanation of the observation. The "hidden" explanation of a recording is the correct phrase which originated the specific utterance.

Very useful and used models of sequences generated by an underlying hidden process are called **hidden Markov model (HMM)**. The intention is to model a system with a **hidden state** (not directly measurable) which produces one among a set of output signals (or symbols), with a probability for each symbol. In addition, the hidden state changes probabilistically in discrete time (1,2,3...), with a certain transition probability. One measures the output signals and aims at identifying **the most probable sequence of hidden states** producing the observed sequence of signals. Events are independent so that probabilities are multiplied. We can view the probability of a *path* (a specific sequence if hidden states in time) as the probability that a "random walk" beginning at time one with a given probability of being in the different initial states will follow the given path.

If you want, you can introduce a fictitious state 0 with a probability π_i of transiting to state i at time 1.

Mathematically the **hidden Markov model** is characterized by a state space S, initial probabilities π_i of being in state i, transition probabilities $a_{i,j}$ from state i to state j, and emission probabilities $b_{i,o}$ that state i emits a certain observation o.

Say we observe outputs $y_1, \ldots, y_T$.

Let $V_{t,k}$ is the probability of the most probable state sequence ending at time t with k as its final (hidden) state. The best way to reason about the **shared optimal substructure** of the optimal solution is to consider its structure *in time*. Assume that $(s_1, ..., s_{t-1}, k)$ is the optimal sequence (the most probable one) ending at state k at iteration t, and consider the most probable sequences stopping one iteration before, at $t-1$. Let $V_{t-1,x}$ be the probability of the most probable path reaching configuration x at the previous time. Two other events must happen to reach k, a transition from x to k and the emission of the measured signal y_t. Because we are searching for the most probable sequence, we must maximize the probability over all possible previous points x.

Therefore, the most likely state sequence $x_1, \ldots, x_T$ that produces the observations is given by the recurrence relations:

$$\begin{aligned} V_{1,k} &= \mathrm{P}\big(y_1 \mid k\big) \cdot \pi_k \\ V_{t,k} &= \max_{x \in S} \left(\mathrm{P}\big(y_t \mid k\big) \cdot a_{x,k} \cdot V_{t-1,x} \right) \end{aligned}$$

After determining the optimal value, the Viterbi path can be retrieved by saving **back pointers** that remember which state x was the winning one in the second equation. Let $\mathrm{Ptr}(k,t)$ be the function that returns the value of x used to compute $V_{t,k}$ if $t > 1$, or k if $t = 1$. Then:

$$\begin{aligned} x_T &= \arg\max_{x \in S}(V_{T,x}) \\ x_{t-1} &= \mathrm{Ptr}(x_t, t) \end{aligned}$$

The complexity of this algorithm is $O(T \times |S|^2)$ while a complexity of a brute-force scheme considering all possible paths is $O(|S|^T)$. The reduction in CPU time is enormous for long sequences.

To illustrate how the Viterbi algorithm works, consider the short example in Fig. 34.5. One assumes a 4-letter alphabet, $A = \{T, A, C, O\}$, and the observed string given by the OCR machine is $Z = -CAT-$, where "$-$" denotes blank or space. The features vectors $z1, z2$, and $z3$ are obtained when the feature extractor looks at C,A,T. The available and relevant information that Viterbi's algorithm traverses to make a decision on the word, is expressed in

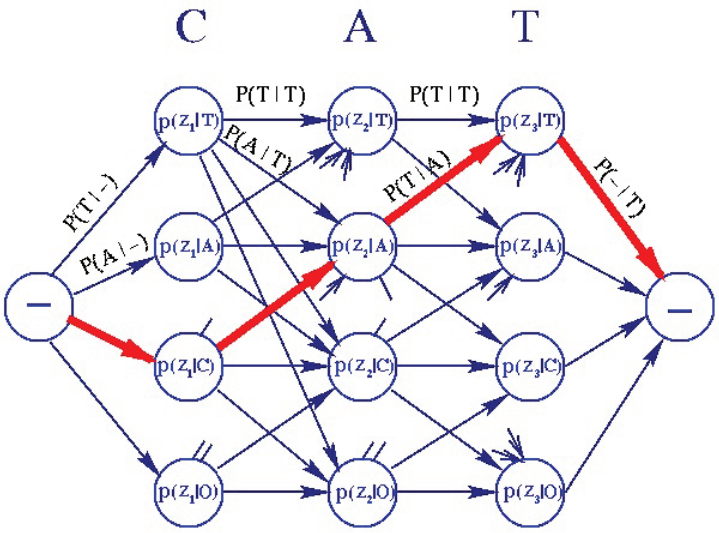


Figure 34.5: Viterbi's dynamic programming algorithm applied to text recognition.

terms of a directed graph or **trellis** as in Fig. 34.5. All nodes (except the blank nodes "-") and edges have probabilities associated with them. The edge probabilities (Markov transition probabilities between letters) remain fixed no matter what sequence of letters is presented to the machine. They represent *static information*. The node probabilities (likelihood of the feature vectors obtained from the characters), on the other hand, are a function on the actual characters presented to the machine. They represent *dynamic information*. We see from the figure that any path from the start node to the end node (the "–" nodes) represents a sequence of letters but not necessarily a valid word. Consider the bold path, which represent the letter sequence CAT. The aim is to find the letter sequence which maximizes the product of the probabilities of its corresponding path.

The pseudo-code of the algorithm is shown in Fig.34.6. Two support matrices are used to save values of the optimal subproblems (T_1) and to save the previous state in the temporal path, so that the optimal solution can be reconstructed at the end.

Andrew Viterbi proposed its algorithm in 1967 as a decoding algorithm for "convolutional codes" over noisy digital communication links. Being an application of dynamic programming, it has, however, a history of multiple invention.

A Dynamic Programming algorithm called the Bellman-Ford Algorithm is used to for the **single-source shortest path problem**, while the Floyd-Warshall algorithms can be used to find **shortest paths between all pairs** of vertices. The starting observation is that a shortest path between two vertices contains other shortest paths within it (otherwise one could substitute sub-paths obtaining shortest overall paths).

Similar algorithms, with generalizations, are now in wide use for maximization problems involving probabilities like statistical parsing, to find the most likely assignment of all or some subset of latent variables in some graphical models, e.g., Bayesian networks, Hidden Markov Models (HMM), and Markov random fields. The latent variables need in general to be connected in a way somewhat similar to an HMM, with a limited number of connections between variables and some type of linear structure among them.

O	Set of observations emitted by states
S	Set of states
$\boldsymbol{\pi}$	Initial probabilities for states
$\boldsymbol{y}$	Observed outputs
A	Matrix of transition probabilities between states
B	Matrix of emission probabilities

```
1.  function Viterbi (O, S, π, y, A, B)
2.      for each state s_i
3.          T_1[i, 1] ← π_i · B_{i,y_1}
4.          T_2[i, 1] ← 0
5.      for i ← 2, 3, ..., T
6.          for each state s_j
7.              T_1[j, i] ← max_k(T_1[k, i − 1] · A_kj · B_{j,y_i})
8.              T_2[j, i] ← arg max_k(T_1[k, i − 1] · A_kj · B_{j,y_i})
9.      z_T ← arg max_k(T_1[k, T])
10.     x_T ← s_{z_T}
11.     for i ← T, T-1, ..., 2
12.         z_{i−1} ← T_2[z_i, i]
13.         x_{i−1} ← s_{z_{i−1}}
14.     return x
```

Figure 34.6: Viterbi algorithm pseudo-code.

Gist

When solving problems with a set of discrete variables, **brute-force** exhaustive generation of all possible solutions has a clear scalability issue: the CPU time explodes in an exponential manner when the number of variables grows. One can imagine producing a tree, in which an additional variable is set to all possible values at a node, generating the corresponding sub-trees (**branching**). Branching alone amounts to brute-force enumeration of candidate solutions and testing them all. To improve the performance, Branch and Bound keeps track of an **optimistic bound** on the solution which can be obtained by completing a partial solution. This bound is compared with the current record value at each iteration to "prune" the search space, eliminating partial solutions that are doomed (one is sure that their completions will not contain any optimal solution). When a node in the tree is doomed, the search **backtracks** and continues from an upper-layer node which contains new subtrees to explore.

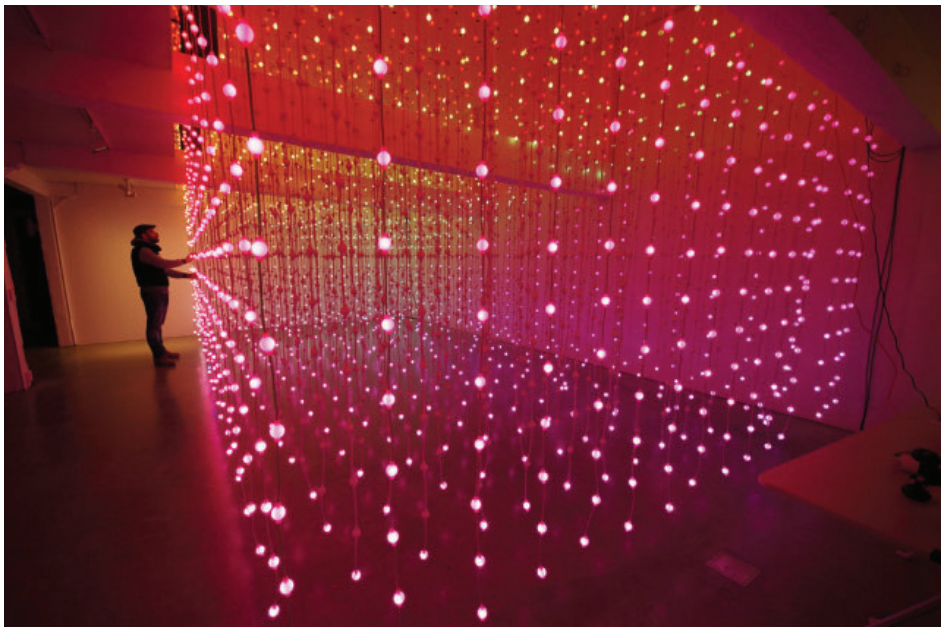
The popularity of B&B in the initial part of Artificial Intelligence is somewhat surprising: given only toy problems can be solved because of its exponential running times. For sure, the human species would not have survived in the forest with B&B when confronted by very fast predators.

The core idea of **dynamic programming** is to avoid repeated work by remembering partial results. It works when one can demonstrate that an **optimal solution contains optimal solutions to smaller sub-problems**. The smaller subproblems can be solved once and saved in memory to build solutions to many larger and larger problems. In some happy cases this process leads to an enormous reduction of CPU times (and better scalability) w.r.t. the brute-force solution. Cellular phones would be impossible without Viterbi's algorithm.

Chapter 35

Satisfiability

I can't get no satisfaction
'Cause I try and I try and I try and I try
(The Rolling Stones)



It is difficult to think about a problem which is more relevant for the applications and more interesting from an algorithmic point of view than **satisfiability of Boolean formulas**. It is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE so that the formula evaluates to TRUE.

Many instances of SAT that occur in practice, for example in **symbolic artificial intelligence, circuit design and automatic theorem proving**. While the roots of formalised logic go back to Aristotle, the dream of automating the derivation of all mathematical truth using axioms and inference rules of formal logic had a huge expansion in the twentieth century. Applications in integrated circuit design and verification are of particular commercial interest, given the huge losses caused by mistakes in the design of logic circuits. Solving *SAT* is a prototypical form of the more general **onstraint programming (CP)** method, searching for a state of the world in which a large number of

constraints are satisfied at the same time. The constraints are expressed as Boolean formulas of binary variables (with two values usually called "true" and "false").

In spite of their far-reaching mathematical interest, as an additional bonus, SAT and MAX-SAT are surprisingly easy to define and visualize, and therefore helpful to make abstract procedures very concrete in our minds. This chapter is somewhat more detailed than the other ones, and it requires therefore more effort and concentration, because most of the algorithmic building blocks for solving optimization problems can be encountered for *SAT* in a simple and terse version, with clear advantages for learning. In addition of the already presented topics, *SAT* gives the opportunity to discuss **approximation algorithms** (with guaranteed performance ratio) and **randomized algorithms** in which random numbers are used in the solution.

To start with a toy concrete example, let's consider the organization of a meeting. Consider the following constraints: John can only meet either on Monday, Wednesday or Thursday, Catherine cannot meet on Wednesday, Anne cannot meet on Friday, Peter cannot meet neither on Tuesday nor on Thursday. Question: Is the meeting possible and when can it take place?

The constraints can be encoded into the following Boolean formula:

$$(Mon \bigvee Wed \bigvee Thu) \bigwedge (\neg Wed) \bigwedge (\neg Fri) \bigwedge (\neg Tue \bigwedge \neg Thu)$$

The formula can be satisfied by setting Monday to TRUE, and the other days to FALSE, and therefore the meeting can take place on Monday. Sure, this can also be easily solved by hand, but imagine defining meetings in a large university subject to many constraints regarding professors, classrooms, etc. Even approximated solutions become very hard!

In the following sections, by following mostly [35], we summarize the main methods, considering both exact (complete) and approximated approaches.

35.1 Satisfiability and maximum satisfiability: definitions

In the Maximum Satisfiability (*MAX–SAT*) problem one is given a Boolean formula in conjunctive normal form, i.e., as a conjunction of clauses, each clause being a disjunction. The task is to find an assignment of truth values to the variables that satisfies the maximum number of clauses.

SAT is the decision version of the problem, i.e., to assess if *all* clauses can be satisfied or not. A *MAX–SAT* solution solves *SAT* if the maximum number of clauses satisfied is equal to the total number of clauses.

In our work, n is the number of variables and m the number of clauses, so that a formula has the following form:

$$\bigwedge_{1 \leq i \leq m} (\bigvee_{1 \leq k \leq |C_i|} l_{ik})$$

where $|C_i|$ is the number of literals in clause C_i and l_{ik} is a literal, i.e., a propositional variable u_j or its negation $\overline{u_j}$, for $1 \leq j \leq n$. The set of clauses in the formula is denoted by **C**. If one associates a *weight* w_i to each clause C_i one obtains the weighted *MAX–SAT* problem, denoted as *MAX W–SAT*: one is to determine the assignment of truth values to the n variables that maximizes the sum of the weights of the satisfied clauses. Of course, *MAX–SAT* is contained in *MAX W–SAT* (all weights are equal to one). In the literature one often considers problems with different numbers k of literals per clause, defined as *MAX–k–SAT* , or *MAX W–k–SAT* in the weighted case. In some papers *MAX–k–SAT* instances contain *up to* k literals per clause, while in other papers they contain *exactly* k literals per clause. We consider the second option unless otherwise stated.

MAX–SAT is of considerable interest not only from the theoretical side but also from the practical one. On one hand, the decision version *SAT* was the first example of an $\mathcal{NP}$–complete problem [100], moreover *MAX–SAT* plays an important role in the characterization of different approximation classes [13]. On the other hand, many issues in mathematical logic and symbolic **artificial intelligence** can be expressed in the form of satisfiability or some of its variants, like **constraint satisfaction**. Some relevant applications are consistency in expert system knowledge bases [281], integrity constraints in databases [11, 142], approaches to inductive inference [187, 227], asynchronous circuit synthesis [209, 302].

We summarize the basic approaches for the exact or approximated solution of the *MAX W–SAT* and *MAX–SAT* problem, to give a panoramic view of the extreme diversity of methods.

The presentation of algorithms for the *SAT* is limited to a quick overview.

35.1.1 Notation and graphical representation

MAX–SAT is easy to define and excellent to visualize, and therefore to remember.

A clause will be represented either as $C = \overline{u} \vee v \vee z$ or as a set of literals, as in $C = \{\overline{u}vz\}$.

For the following discussion, it can be useful to help the intuition with a graphical representation of a formula in conjunctive normal form, as depicted in Fig. 35.1. In the figure, one has a case of *MAX 3–SAT* : all clauses have three literals and the formula is:

$$(u_1 \vee \overline{u_3} \vee u_5) \wedge (\overline{u_2} \vee \overline{u_4} \vee \overline{u_5}) \wedge (\overline{u_1} \vee u_3 \vee \overline{u_4})$$

Truth values to variables are assigned by placing a black triangle to the left if the variable is **true**, to the right if it is **false**. Each literal is depicted with a small circle, placed to the left if the corresponding variable is **true**, to the right in the other case. If a literal is *matched* by the current assignment (e.g., if the literal asks for a **true** value and the variable is set to **true**, or if is asks for **false** and the variable is **false**), it is shown with a gray shade. The *coverage* of a clause is the number of literals in the clause that are matched by the current assignment, and it is illustrated by placing a black square in the appropriate position of an array with indices ranging from 0 to the number of literals in each clause $|C|$.

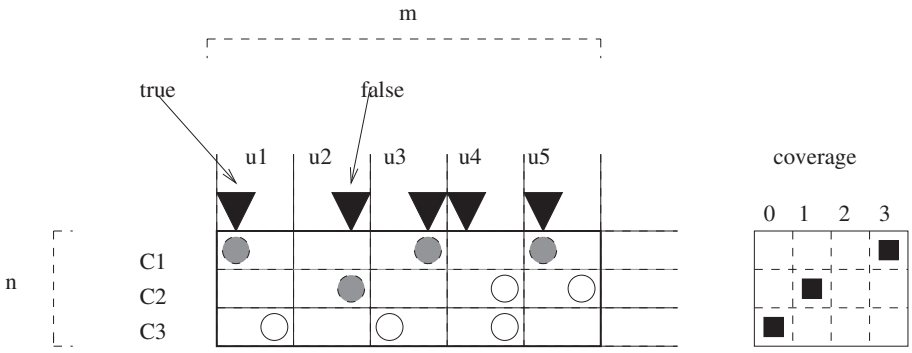


Figure 35.1: A formula in conjunctive normal form (CNF).

35.2 Resolution and Linear Programming

35.2.1 Resolution and backtracking for *SAT*

A simple approach to solve *SAT* consists of the smart generation and test of all possible truth assignment, adapting the Branch and Bound method described in Section 34.1 to the particular structure os *SAT* .

In the adaptation, a basic tool is that of **resolution**, given by the **recursive replacement of a formula by one or more formulae, the solution of which implies the solution of the original formula**.

In *resolution* a variable is selected and a new clause, called the *resolvent* is added to the original formula. The process is repeated to exhaustion or until an empty clause is generated. The original formula is not satisfiable if and only if an empty clause is generated [307].

One aims at demonstrating that the problem *cannot* be satisfied, if one fails, the problem is satisfiable.

Let us now consider some details: A clause R is the *resolvent* of clauses C_1 and C_2 iff there is a literal $l \in C_1$ with $\bar{l} \in C_2$ such that $R = (C_1 \setminus \{l\}) \cup (C_2 \setminus \{\bar{l}\})$ and $u(l)$, the variable associated to the literal, is the only variable appearing both positively and negatively.

For the two clauses $C_1 = (l \vee a_1 \vee ... \vee a_A)$ and $C_2 = (\bar{l} \vee b_1 \vee ... \vee b_B)$ the resolvent is therefore the clause $R = (a_1 \vee ... \vee a_A \vee b_1 \vee ... \vee b_B)$. The resolvent is a logical consequence of the logical *and* of the two clauses. Therefore, if the resolvent is added to the original set of clauses, the **set of solutions does not change**. It is immediate to check that, if both C_1 and C_2 are satisfied, i.e., have at least one matched literal, the resolvent must also be satisfied. In fact, if it is not, in the original clauses there are no matched literals apart from either $\bar{l}$ or l, but this implies that both clauses cannot be satisfied (see also Fig. 35.2 for a graphical illustration).

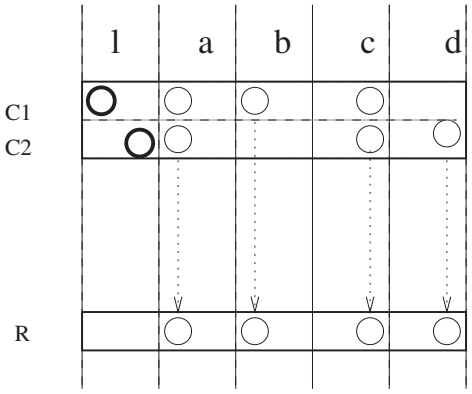


Figure 35.2: How to construct a resolvent, an example with variables l, a, b, c, d.

Davis and Putnam [111] started in 1960 the investigation of useful strategies for handling resolution. In addition to applying transformations that preserve the set of solutions they eliminate one variable at a time in a chosen order by using all possible resolvents on that variable. During resolution the lengths and the number of added clauses can easily increase and become extremely large.

DPLL(**C** : set of clauses)
Input: Boolean CNF formula $\mathbf{C} = \{C_1, C_2, \ldots, C_m\}$
Output: Yes or No (decision about satisfiability)
1 **if C** is empty **then return** Yes
2 **if C** contains an empty clause **then return** No
3 **if** there is a pure literal l in **C** **then return** DPLL($\mathbf{C}(l)$)
4 **if** there is a unit clause $\{l\} \in$ **C then return** DPLL($\mathbf{C}(l)$)
5 Select a variable u in **C**
6 **if** DPLL($\mathbf{C}(u)$) = Yes **then return** Yes
7 **else return** DPLL($\mathbf{C}(\overline{u})$)

Figure 35.3: The DPLL algorithm by Davis, Logemann and Loveland in recursive form. The recursive calls are executed on the problems derived after setting the truth value of the selected variable.

Davis, Logemann and Loveland [110] avoid the memory explosion of the original DP algorithm by replacing the resolution rule with the *splitting rule* (Davis, Putnam, Logemann and Loveland, or DPLL algorithm for short). In splitting, a variable u in a formula is selected. Now, if there exist a satisfying truth assignment for the original formula then either u is **true** or $\overline{u}$ is **true** in the assignment. In the first case the formula obtained by eliminating all clauses containing u and by deleting all occurrences of $\overline{u}$ must be satisfied, see Fig 35.4. This derived formula is called $\mathbf{C}(u)$ in Fig. 35.3. In the second case, the formula obtained by eliminating all clauses containing $\overline{u}$ and all occurrences of u must be satisfied. *Vice versa*, if both derived formulae cannot be satisfied, neither can the original problem.

A *tree* is therefore generated. At the root one has the original problem and no variables are assigned values. At each node of the tree one generates two children by *selecting* one of the yet unassigned variables in the problem

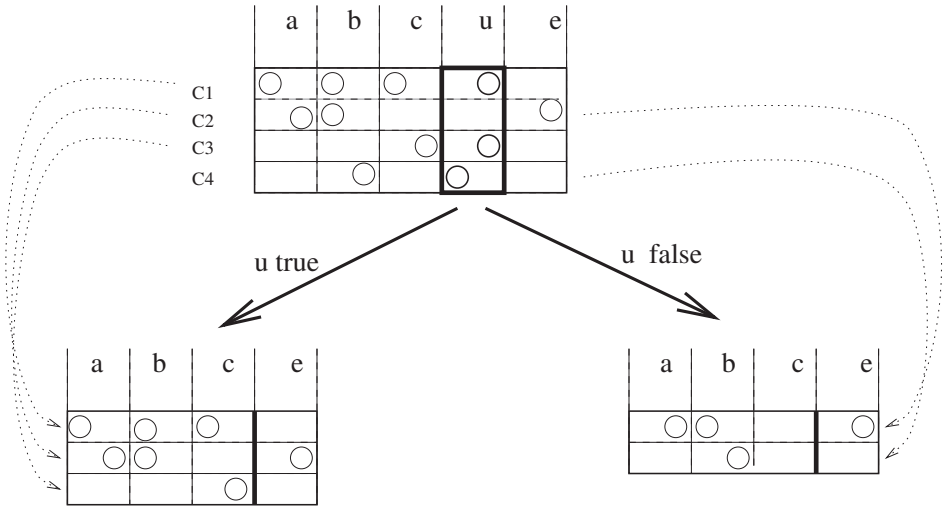


Figure 35.4: Example of splitting on a variable u.

corresponding to the node and by generating the two problems derived by setting the variable to **true** or **false**. A trivial upper bound on the number of nodes in the tree is proportional to the number of possible assignments, i.e., $O(2^n)$. In fact, sophisticated techniques are available to reduce the number of nodes, that nonetheless remains exponential in the worst case.

The techniques include:

- avoiding the examination of a subtree when the fate of the current problem is decided (problems with an empty clause have no solutions, problems with no clauses have a solution). If the current problem cannot be solved, or if it is solved but one wants all possible solutions, one *backtracks* to the first unexplored branch of the tree. Note that, when splitting is combined with a depth-first search of the tree (as in the DPLL algorithm) one avoids the memory explosion because only one subproblem is active at a given time.
- *selecting* the next variable for the splitting based on appropriate criteria. For example, one can prefer variables that appear in clauses of length one (*unit clause rule*), or select a *pure literal* (such that it occurs only positive, or only negative), or select a literal occurring in the *smallest clause*.

Interesting reviews are [163], and [246]. A parallel implementation in given in [57].

35.2.2 Integer programming approaches

The *MAX W–SAT* problem has a natural integer linear programming formulation (*ILP*), see Section 34.1. Let $y_j = 1$ if Boolean variable u_j is **true**, $y_j = 0$ if it is **false**, and let the Boolean variable $z_i = 1$ if clause C_i is satisfied, $z_i = 0$ otherwise. The integer linear program is:

$$\max \sum_{i=1}^{m} w_i \; z_i$$

subject to the following constraints:

$$\begin{aligned} \sum_{j \in U_i^+} y_j + \sum_{j \in U_i^-} (1 - y_j) \quad &\geq \quad z_i \; , \; i = 1, \cdots, m \\ y_j &\in \{0, 1\} \; , \; j = 1, \cdots, n \\ z_i &\in \{0, 1\} \; , \; i = 1, \cdots, m \end{aligned}$$

where U_i^+ and U_i^- denote the set of indices of variables that appear unnegated and negated in clause C_i, respectively.

Because the sum of the z_i w_i is maximized and because each z_i appears as the right-hand side of one constraint only, z_i will be equal to one if and only if clause C_i is satisfied.

If one neglects the objective function and sets all z_i variables to 1, one obtains an integer programming feasibility problem associated to the *SAT* problem [55].

The integer linear programming formulation of *MAX–SAT* suggests that this problem could be solved by a standard **branch-and-bound** method. A tree is generated, see also the DPLL method, where the root corresponds to the initial instance and two children are obtained by *branching*, i.e., by selecting one free variable and setting it **true** (left child) and **false** (right child). An *upper bound* on the number of satisfied clauses can be obtained by using a **linear programming relaxation**: the constraints $y_j \in \{0,1\}$ and $z_i \in \{0,1\}$ are replaced by $y_j \in [0,1]$ and $z_i \in [0,1]$. One obtains a Linear Programming (*LP*) problem that can be solved in polynomial time and, because the set of admissible solutions is enlarged with respect to the original problem, one obtains an upper bound.

Unfortunately this is not likely to work well in practice [172] because the solution $y_j = 1/2, j = 1, \cdots, n$, $z_i = 1, i = 1, \cdots, m$ is feasible for the *LP* relaxation unless there exist some constraint containing only one variable. The bounds so obtained would be very poor.

Better bounds can be obtained by using Chvátal cuts. In [187] it is shown that the resolvents in the propositional calculus correspond to certain cutting planes in the integer programming model of inference problems.

A general cutting plane algorithm for *ILP* , see for example [287], works as follows. One solves the *LP relaxation* of the problem: if the solution is integer the algorithm terminates, otherwise one adds linear constraints to the *ILP* that do not exclude integer feasible points. The constraints are added one at a time, until the solution to the *LP* relaxation is integer.

LP relaxations of integer linear programming formulations of *MAX–SAT* have been used to obtained upper bounds in [170, 387, 155]. A linear programming and rounding approach for *MAX 2–SAT* is presented in [85].

35.3 Continuous approaches

The *ILP* feasibility problem obtained from *SAT* as described in the previous section is solved with an **interior point algorithm** in [227, 228], which applies a function minimization method based on *continuous* mathematics to the inherently discrete *SAT* problem.

In [228] the application is to a problem of **inductive inference**, in which one aims at identifying a hidden Boolean function using outputs obtained by applying a limited number of random inputs to the hidden function. The task is formulated as a *SAT* problem, which is in turn formulated as an integer linear program:

$$A^T y \leq c \ , \ y \in \{-1,1\}^n \tag{35.1}$$

where A^T is an $m \times n$ real matrix and c a real m vector.

The interior point algorithm is based on finding a local minimum in the box $-1 \leq y_j \leq 1$ of the *potential function*:

$$\phi(y) = \log \left\{ \frac{n - y^T y}{\prod_{k=1}^m (c_k - {a_k}^T y)^{1/m}} \right\} \tag{35.2}$$

by an iterative method. The denominator of the argument of the log is the geometric mean of the *slacks* (a_k is the k-th column of matrix A). It is shown that, if the integer linear program has a solution, y^* is a global minimum of this potential function if and only if y^* solves the integer program. The next iterate y^{k+1} (interior point solution, i.e., such that $A^T y < c$) is obtained by moving in a descent direction Δy from the current iterate y^k such that $\phi(y^{k+1}) = \phi(y^k + \alpha \Delta y) < \phi(y^k)$. Each iteration in [228] is based on the *trust region approach* of continuous optimization where the Riemannian metric used for defining the search region is dynamically modified.

In some techniques the *MAX–SAT* (or *SAT*) problem is transformed into an **unconstrained optimization problem** on the real space R^n and solved by using existing global optimization techniques.

Some examples of this approach include the UNISAT models [207] and the *neural network* approaches [222, 76]. In general, these techniques do not have performance guarantees because they assure only the local convergence to a locally optimal point, not necessarily the global optimum. The local convergence properties of some optimization algorithms are considered in [162]. To obtain these results, one assumes that the initial solution is "sufficiently close" to the optimal solution.

35.4 Approximation algorithms

MAX–SAT is a playground for algorithms with **guaranteed quality of approximation**. The basic principle is to guarantee that the delivered result will be within a certain percentage of the optimal solution, which is of interest for many applications. Let's remember that the real world is complex and noisy so that small differences in the solutions can become irrelevant when compared with experimental noise and approximations related to defining the problem. For a detailed treatment of complexity classes we refer to [35], we focus here on some notable approximated algorithms.

The two first approximate algorithms for *MAX W–SAT*were proposed by Johnson [218] and use **greedy construction** strategies. The original paper [218] demonstrated for both of them a performance ratio $1/2$. Actually the second one reaches a performance ratio $2/3$ [83].

The first algorithm chooses, at each step, the literal that occurs in the maximum number of clauses. If the literal is positive, the corresponding variable is set to **true**; if the literal is negative, the corresponding variable is set to **false**. The clauses satisfied by the literal are deleted from the formula and the algorithm stops when the formula is satisfied or all variables have been assigned values. More formally, this procedure is developed in algorithm GREEDYJOHNSON1 of Fig. 35.5.

GREEDYJOHNSON1
Input: Boolean CNF formula $\mathbf{C} = \{C_1, C_2, \ldots, C_m\}$;
Output: Truth assignment U;
△ The satisfied clauses will be incrementally inserted in the set **S**;
△ U is the truth assignment;
△ for every literal l, $u(l)$ is the corresponding variable;
S $\leftarrow \emptyset$; LEFT $\leftarrow$ **C**; $V \leftarrow \{u \mid u$ variable in **C**$\}$;
repeat
  Find l, with $u(l) \in V$, that is in max. no. of clauses in LEFT
  Solve ties arbitrarily
  Let $\{C_{l_1}, \ldots, C_{l_k}\}$ be the clauses in which l occurs
  S $\leftarrow$ **S** $\cup \{C_{l_1}, \ldots, C_{l_k}\}$
  LEFT $\leftarrow$ LEFT $\setminus \{C_{l_1}, \ldots, C_{l_k}\}$
  if l is positive **then** $u(l) \leftarrow$ **true else** $u(l) \leftarrow$ **false**
  $V \leftarrow V \setminus \{u(l)\}$
until no literal l with $u(l) \in V$ is contained in any clause of LEFT
if $V \neq \emptyset$ **then forall** $u \in V$ **do** $u \leftarrow$ **true**
return U

Figure 35.5: The GREEDYJOHNSON1 algorithm, a $k/(k+1)$-approximate algorithm.

Theorem 35.4.1 *Algorithm* GREEDYJOHNSON1 *is a polynomial time 1/2-approximate algorithm for MAX–SAT .*

Proof. One can prove that, given a formula with m clauses, algorithm GREEDYJOHNSON1 always satisfies at least $m/2$ clauses, by induction on the number of variables. Because no optimal solution can be larger than m, the theorem

follows. The result is trivially true in the case of one variable. Let us assume that it is true in the case of $i-1$ variables $(i > 1)$ and let us consider the case in which one has i variables. Let u be the last variable to which a truth value has been assigned. We can suppose that u appears positive in k_1 clauses, negative in k_2 clauses and does not appear in $m - k_1 - k_2$ clauses. Without loss of generality suppose that $k_1 \geq k_2$. Then, by inductive hypothesis, algorithm GREEDYJOHNSON1 allows us to choose suitable values for the remaining $i-1$ variables in such a way to satisfy at least $(m - k_1 - k_2)/2$ clauses; if according to the algorithm we now choose u = **true** we satisfy

$$\frac{m - k_1 - k_2}{2} + k_1 \geq \frac{m}{2}$$

clauses.
□

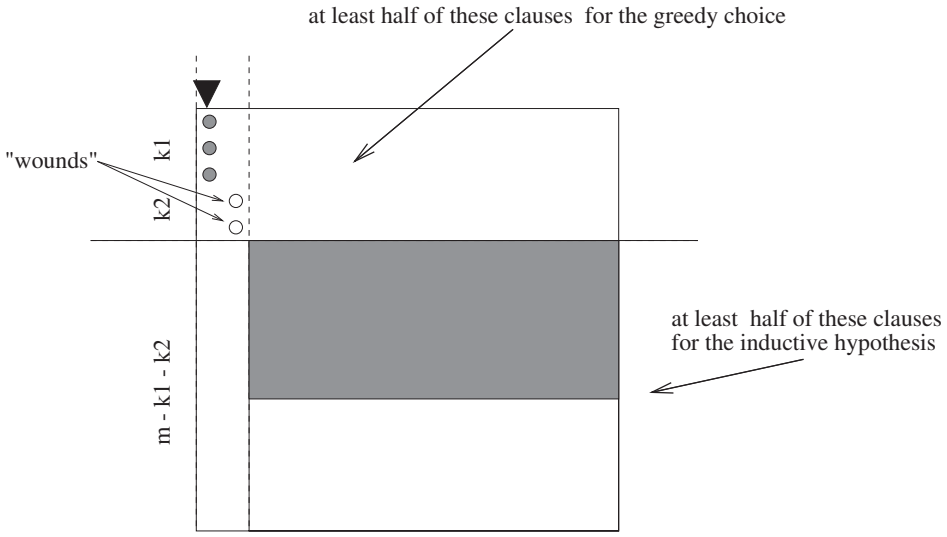


Figure 35.6: Illustration of the GREEDYJOHNSON1 algorithm.

Let us note that one does not use the fact that the chosen literal occurs in the *maximum* number of clauses for the above proof. What is required is that, given an unset variable that appears in at least an unsatisfied clause, the variable is set to **true** or **false** in a way that maximizes the number of newly satisfied clauses.

This result can be made more specific by considering the number of variables in a clause.

Theorem 35.4.2 *Let k be the minimum number of variables occurring in any clause of the formula. For any integer $k \geq 1$, algorithm* GREEDYJOHNSON1 *achieves a feasible solution y of an instance x such that*

$$\frac{m(x,y)}{m^*(x)} \geq 1 - \frac{1}{k+1}.$$

Proof. Because of the greediness, when literal l is picked in line 3 of Fig. 35.5, the number of newly satisfied clauses is at least as large as the number of new *wounds*, defined as the number of occurrences of literal $\bar{l}$ in clauses of LEFT that will never be matched in the future steps, given the choice of l, see Fig. 35.6. When the algorithm halts, the only clauses remaining in LEFT are those that have a number of wounds equal to the number of their literals, and hence are *dead*. This means that, when the algorithm halts, there are at least $k|\text{LEFT}|$ wounds, and therefore $|S| \geq k|\text{LEFT}|$. Thus $m^* \leq m = |S| + |\text{LEFT}| \leq \frac{(k+1)}{k}|S|$. The bound follows.
□

Note that, according to the definition of performance ratio, algorithm GREEDYJOHNSON1 is $\frac{k}{k+1}$–approximate. In particular, for $k = 1$, the performance ratio is $1/2$, for $k = 2$ the performance ratio is $2/3$, for $k = 3$ the performance ratio is $3/4$ and so on. This means that the goodness of the algorithm improves for larger values of k. Therefore the worst case is given by $k = 1$, that is, when one has unit clauses (clauses with just one literal).

Johnson introduced a second algorithm (GREEDYJOHNSON2). This algorithm improves the performance ratio and obtains a bound $2/3$ [83]. Until very recently, only a performance ratio $1/2$ was demonstrated [218]. The original theorem in [218] is here presented, because of its simplicity and paradigmatic nature and because it gives a better performance as a function of k, the minimum number of literals in some clause. In the algorithm one associates a *mass* $w(C_i) = 2^{-|C_i|}$ to each clause. The term *mass* is used instead of the original term "weight" in order to avoid confusions with the clause *weight* in the *MAX W–SAT* problem. The mass will be proportional to the weight in the version of the algorithm for the *MAX W–SAT* problem ($w(C_i) = w_i\ 2^{-|C_i|}$). In [218] the analysis of the performance of algorithm GREEDYJOHNSON2 leads to the following:

GREEDYJOHNSON2
Input: Boolean CNF formula $\mathbf{C} = \{C_1, C_2, \ldots, C_m\}$;
Output: Truth assignment U;
△ The satisfied clauses will be incrementally inserted in the set **S**;
△ U is the truth assignment;
△ for every literal l, let $u(l)$ be the corresponding variable;
1 **S** $\leftarrow \emptyset$; LEFT $\leftarrow$ **C**; $V \leftarrow \{u \mid u$ variable in **C**$\}$;
2 Assign to each clause C_i a mass $w(C_i) = 2^{-|C_i|}$
3 **repeat**
4 Determine $u \in V$, appearing in at least a clause $\in$ LEFT
5 Let **CT** be the clauses $\in$ LEFT cont. u, **CF** those cont. $\overline{u}$
6 **if** $\sum_{C_i \in \mathbf{CT}} w(C_i) \geq \sum_{C_i \in \mathbf{CF}} w(C_i)$ **then**
7 $u(l) \leftarrow$ **true**
8 **S** $\leftarrow$ **S** $\cup$ **CT**
9 LEFT $\leftarrow$ LEFT $\setminus$ **CT**
10 **forall** $C_i \in$ **CF do** $w(C_i) \leftarrow 2 \cdot w(C_i)$
11 **else**
12 $u(l) \leftarrow$ **false**
13 **S** $\leftarrow$ **S** $\cup$ **CF**
14 LEFT $\leftarrow$ LEFT $\setminus$ **CF**
15 **forall** $C_i \in$ **CT do** $w(C_i) \leftarrow 2 \cdot w(C_i)$
16 **until** no literal l in any clause of LEFT is such that $u(l)$ is in V
17 **if** $V \neq \emptyset$ **then forall** $u \in V$ **do** $u \leftarrow$ **true**
18 **return** U

Figure 35.7: The GREEDYJOHNSON2 algorithm, a $(1 - 1/2^k)$-approximate algorithm.

Theorem 35.4.3 *Let k be the minimum number of clauses occurring in any clause of the formula. For any integer $k \geq 1$, algorithm* GREEDYJOHNSON2 *achieves a feasible solution y of an instance x such that*

$$\frac{m(x,y)}{m^*(x)} \geq 1 - \frac{1}{2^k}.$$

Proof. Initially, because each clause has at least k literals, the total mass of all the clauses in LEFT cannot exceed $m/2^k$. During each iteration, the total mass of the clauses in LEFT cannot increase. In fact, the mass removed from LEFT is at least as large as the mass added to those remaining clauses which receive new *wounds*, see lines 6–15 of

Fig. 35.7. Therefore, when the algorithm halts, the total mass still cannot exceed $m/2^k$. But each of the *dead* clauses in LEFT when the algorithm halts must have been wounded as many times as it had literals, hence must have had its mass doubled that many times, and so must have final mass equal to one. Therefore $|\text{LEFT}| \leq m/2^k$, and so $|S| \geq m(1 - 1/2^k)$ and the bound follows.
□

Again, for larger values of k, algorithm GREEDYJOHNSON2 obtains better performance ratios and, generally speaking, because $1 - \frac{1}{2^k} > 1 - \frac{1}{k+1}$ for any integer $k \geq 2$, algorithm GREEDYJOHNSON2 has a better performance than that of algorithm GREEDYJOHNSON1.

The performance ratio $2/3$ has been proved in a paper by Chen, Friesen, and Zheng [83]. Because they consider the *MAX W–SAT* problem, line 2 in Fig. 35.7 must be modified to take the weights w_i into account: the *mass* becomes $w(C_i) = w_i\ 2^{-|C_i|}$. The preceding bound $1/2$ depends on the fact that the only upper bound used in the above proofs was given by the total weight of the clauses; of course this upper bound can be far from the optimal value. The novelty of the approach of [83] is that the performance ratio can be derived by using the correct value of the optimal solution. In order to prove that algorithm GREEDYJOHNSON2 has this better performance ratio let us introduce a generalization of the algorithm. It is important to stress that this generalization is introduced to perform a more accurate analysis of the performance ratio and it is used in the following as a theoretical tool.

The difference between GREEDYJOHNSON2 and its generalization is rather subtle. The generalized algorithm, that we denote as GENJOHNSON2, considers an arbitrary Boolean array $b[1..n]$ of size n as additional input, and examines b to decide what to do if an equality is present in line 6 of Fig. 35.7. Let us assume that the variable one is considering is u_j. In line 6 of GREEDYJOHNSON2 in Fig. 35.7, when $\sum_{C_i \in \mathbf{CT}} w(C_i) = \sum_{C_i \in \mathbf{CF}} w(C_i)$, the **if** condition is true and u_j is set to **true**. Now, instead, when one obtains an equality one considers two different cases: if the variable $b[j]$ is **true** u_j is set to **true**; if the variable $b[j]$ is **false** u_j is set to **false**.

This generalized algorithm is then used in the proof with this Boolean array equal to the optimal assignment. Of course the optimal assignment cannot be derived in polynomial time but here we are not interested in running an algorithm but in performing a theoretical analysis.

We will prove that GENJOHNSON2 has a performance ratio $2/3$ and this fact will imply that also GREEDYJOHNSON2 has performance ratio $2/3$.

Let us give some definitions needed in the proof.

Definition 2
- *A literal is positive if it is a Boolean variable u_i for some i.*
- *A literal is negative if it the negation $\bar{u}_i$ of a Boolean variable for some i.*

Definition 3 *Assume that algorithm* GENJOHNSON2 *is applied to a formula* **C** *and consider a fixed moment in the execution.*

- *A literal l is active if it has not been assigned a truth value yet.*
- *A clause C_j is killed if all literals in C_j are assigned value* **false**.
- *A clause C_j is negative if it is neither satisfied nor killed, and all active literals in C_j are negative literals.*

Definition 4 *Let $0 \leq t \leq n$. Assume that in* GENJOHNSON2 *the t–th iteration has been completed (a truth assignment has been given to t variables). Then $\mathbf{S}^t$ denotes the set of satisfied clauses, $\mathbf{K}^t$ denotes the set of killed clauses, $\mathbf{N}_i^t$ denotes the set of negative clauses with exactly i active liberals.*

Without less of generality, one assumes that each clause in the formula has at most r literals. The proof of the performance ratio $2/3$ depends on the following Lemma.

Given a set of clauses **C**, let us define as $w(\mathbf{C})$ the sum of the weights of all clauses of **C**.

Lemma 2 *For any formula* **C** *of MAX W–SAT and for any Boolean array* $b[1..n]$*, when the algorithm* GENJOHNSON2 *is applied on* **C** *the following inequality holds at all iterations* $0 \leq t \leq n$*:*

$$w(\mathbf{S}^t) \geq 2w(\mathbf{K}^t) + \sum_{i=1}^{r} \frac{1}{2^{i-1}} w(\mathbf{N}_i^t) - A_0 \tag{35.3}$$

where $A_0 = \sum_{i=1}^{r} \frac{1}{2^{i-1}} w(\mathbf{N}_i^0)$

The proof of the Lemma proceeds by induction on t and can be found in [83].

Theorem 35.4.4 *The performance ratio of algorithm* GREEDYJOHNSON2 *is* $2/3$*.*

Proof. Let **C** be an instance of *MAX W–SAT* and let U_0 an optimal truth assignment for **C**. Now one considers another formula $\mathbf{C}'$ that is derived from **C** as follows. If $U_0(u_t) =$ **false** for a variable u_t then one negates u_t (u_t and $\bar{u_t}$ are interchanged) in $\mathbf{C}'$. No change on the weights is done. Therefore there exists a one–to–one correspondence between the set of clauses in **C** and the set of clauses in $\mathbf{C}'$; moreover the corresponding clauses have the same weight. In addition, the Boolean array $b[1..n]$ is constructed such that $b[j]$ = **false** if and only if $U_0(u_j) =$ **false**.

It is easy to see (for the details, see again [83]) that

- the weight of an optimal assignment to $\mathbf{C}'$ is equal to the weight of an optimal assignment to **C**.
- the truth assignment for **C** found by GREEDYJOHNSON2 and the truth assignment for $\mathbf{C}'$ found by GENJOHNSON2 have the same weight.

This means that, if we prove that GENJOHNSON2 has a performance ratio $2/3$ on the formula $\mathbf{C}'$, the theorem is shown.

Note that the truth assignment $U_0^{'}$ for $\mathbf{C}'$ that gives value **true** to all variables corresponds to the optimal truth assignment U_0 for **C**. Therefore $U_0^{'}$ is optimal for $\mathbf{C}'$.

When GENJOHNSON2 stops, that is, for $t = n$, $\mathbf{S}^n$ is the set satisfied by the algorithm and $\mathbf{K}^n$ is the set of clauses not satisfied. $\mathbf{N}_i^n$ is the empty set for any i.

Applying the inequality 35.3 of Lemma 2 to this case, one obtains:

$$w(\mathbf{S}^n) \geq 2w(\mathbf{K}^n) - A_0. \tag{35.4}$$

On the other hand, A_0 can be upperbounded in the following way:

$$A_0 = \sum_{i=1}^{r} \frac{1}{2^{i-1}} w(\mathbf{N}_i^0) \leq \sum_{i=1}^{r} w(\mathbf{N}_i^0) \leq 2 \sum_{i=1}^{r} w(\mathbf{N}_i^0). \tag{35.5}$$

From inequalities 35.4 and 35.5 one has:

$$\frac{3}{2} w(\mathbf{S}^n) \geq w(\mathbf{S}^n) + w(\mathbf{K}^n) - \sum_{i=1}^{r} w(\mathbf{N}_i^0) \tag{35.6}$$

Note that, on one hand, $w(\mathbf{S}^n)$ is the weight of the truth assignment found by GENJOHNSON2. On the other hand, $\mathbf{S}^n \cup \mathbf{K}^n$ is the whole set of clauses in $\mathbf{C}'$ and the optimal truth assignment $U_0^{'}$ for $\mathbf{C}'$ that gives value **true** to all variables satisfies all clauses in $\mathbf{C}'$ except those belonging to $\mathbf{N}_i^0$ for $i = 1, 2, \ldots, r$.

Therefore an optimal truth assignment for $\mathbf{C}'$ has weight exactly

$$w(\mathbf{S}^n) + w(\mathbf{K}^n) - \sum_{i=1}^{r} w(\mathbf{N}_i^0).$$

Then the inequality 35.6 says that the weight of the truth assignment found by GENJOHNSON2 is at least $2/3$ of the weight of an optimal assignment to $\mathbf{C}'$. In consequence, the weight of the assignment constructed by the original GREEDYJOHNSON2 algorithm for the instance $\mathbf{C}$ is at least $2/3$ of the weight of an optimal assignment to $\mathbf{C}$, thus proving the theorem.
□

35.4.1 Randomized algorithms for *MAX W–SAT*

A randomized $1/2$–approximate algorithm for *MAX W–SAT*

One of the most interesting approaches in the design of new algorithms is the use of *randomization*. During the computation, random bits are generated and used to influence the algorithm process.

In many cases randomization allows to obtain better (expected) performance or to to simplify the construction of the algorithm. Particularly in the field of approximation, randomized algorithms are widely used and, for many problems, the algorithm can be "derandomized" in polynomial time while preserving the approximation ratio. However, it is important to note that, often, the derandomization leads to algorithms which are very complicated in practice.

Let us now use this approach to present more efficient approximate algorithms for *MAX W–SAT*. More precisely, this section introduces two different randomized algorithms that achieve a performance ratio of $3/4$. Moreover, it is possible to derandomize these algorithms, that is, to obtain deterministic algorithms that preserve the same bound $3/4$ for every instance.

The derandomization is based on the *method of conditional probabilities* that has revealed its usefulness in numerous cases and is a general technique that often permits to obtain a deterministic algorithm from a randomized one while preserving the quality of approximation.

Let us first present the algorithm RANDOM, a simple randomized algorithm, that, while just achieving a performance ratio $1/2$, will be used in the following subsections as an ingredient to reach the performance ratio $3/4$.

RANDOM
Input: Set $\mathbf{C}$ of weighted clauses in conjunctive normal form
Output: Truth assignment U, $\mathbf{C}'$, $\sum_{C_j \in \mathbf{C}'} w_j$
1 Independently set each variable u_i to **true** with probability $1/2$
2 Compute $\mathbf{C}' = \{C_j \in \mathbf{C} : C_j \text{ is satisfied } \}$
3 Compute $\sum_{C_j \in \mathbf{C}'} w_j$

Figure 35.8: The RANDOM algorithm, a randomized $(1 - 1/2^k)$-approximate algorithm.

It is difficult to think about a simpler (randomized) algorithm! Because the algorithm is randomized, one is interested in the **expected performance** when the algorithm is run with different sequences of random bits (i.e., with different random assignments).

Lemma 3 *Given an instance of MAX W–SAT in which all clauses have at least k literals, the expected weight W of the solution found by algorithm* RANDOM *is such that*

$$W \geq (1 - \frac{1}{2^k}) \sum_{C_j \in \mathbf{C}} w_j.$$

Proof. The probability that any clause with k literals is not satisfied by the assignment found by the algorithm is 2^{-k} (all possible k matches must fail). Therefore the probability that a clause is satisfied is $1 - 2^{-k}$. Then

$$W = (1 - \frac{1}{2^k}) \sum_{C_j \in \mathbf{C}} w_j.$$

□

As an immediate consequence of Lemma 3, one obtains the following Corollary.

Corollary 1 *Algorithm* RANDOM *finds a solution for MAX W–SAT whose expected value is at least one half of the optimum value.*

The performance of algorithm RANDOM is the same, in a probabilistic setting, as that of algorithm GREEDYJOHNSON2.

Actually it is possible to show that, by applying the method of conditional probabilities to derandomize [35] algorithm RANDOM, one essentially obtains algorithm GREEDYJOHNSON2.

For $k = 1$, algorithm RANDOMachieves an expected performance ratio $1/2$. The performance of the algorithm improves if we increase the number of literals. In particular, for $k = 2$, that is for formulae which do not contain unit clauses, one obtains an expected value which is at least $3/4$ of the optimal value. Therefore if one could discard unit clauses, one would already have a $3/4$-approximate algorithm for *MAX W–SAT*, after applying the derandomization. This observation will reveal its usefulness in the following.

A randomized $3/4$-approximate algorithm for *MAX W–SAT*

This subsection presents an algorithm that considerably improves the performance of algorithm RANDOM, and obtains a performance ratio $3/4$.

First of all we consider a generalization of algorithm RANDOM. In the previous case the value of every variable was chosen randomly and uniformly, that is with probability $1/2$; now the value of variable u_i is chosen with probability p_i, obtaining algorithm GENRANDOM.

GENRANDOM
Input: Set **C** of weighted clauses in conjunctive normal form
Output: Truth assignment U, $\mathbf{C}'$, $\sum_{C_j \in \mathbf{C}'} w_j$
1 Independently set each variable u_i to **true** with probability p_i
2 Compute $\mathbf{C}' = \{C_j \in \mathbf{C} : C_j \text{ is satisfied } \}$
3 Compute $\sum_{C_j \in \mathbf{C}'} w_j$

Figure 35.9: The GENRANDOM algorithm.

The expected number of clauses satisfied by algorithm GENRANDOM can be immediately computed as a function of p_i.

Lemma 4 *The expected weight W of the set of clauses* **C** *is:*

$$W = \sum_{C_j \in C} w_j (1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i)$$

where $U_j^+ (U_j^-)$ denotes the set of indices of the variables appearing unnegated (negated) in the clause C_j.

Proof. It is an obvious generalization of the proof given in the particular case $p_i = 1/2$.
□

Now, if one manages to find suitable values p_i such that $W \geq 3/4\ m^*(\mathbf{C})$ for every formula **C**, one would obtain a $3/4$-approximate randomized algorithm.

To aim at this result, let us consider the representation of the instances of *MAX W–SAT* as instances of an integer linear programming problem (*ILP*) already presented in Section 2:

$$max \sum_{C_j \in \mathbf{C}} w_j z_j$$

subject to :

$$\sum_{i \in U_j^+} y_i + \sum_{i \in U_j^-} (1 - y_i) \geq z_j, \forall C_j \in \mathbf{C}$$

$$y_i \in \{0, 1\}, 1 \leq i \leq n$$

$$z_j \in \{0, 1\}, \forall C_j \in \mathbf{C}$$

Let $u_1, \ldots, u_n$ be the Boolean variables appearing in the formula. An instance of *MAX W–SAT* is equivalent to an instance of *ILP* if we choose the following conditions:

- $y_i = 1$ iff variable u_i is **true**;
- $y_i = 0$ iff variable u_i is **false**;
- $z_j = 1$ iff clause C_j is satisfied;
- $z_j = 0$ iff clause C_j is not satisfied.

The linear inequality states the fact that a clause can be satisfied ($z_j = 1$) only if at least one of its literals is matched.

One cannot compute the optimal value in polynomial time because *ILP* is $\mathcal{NP}$–complete. However let us consider the *LP* relaxation (by *relaxation* one means that the set of admissible solution increases with respect to that of the original problem) in which one relaxes the conditions $y_i, z_j \in \{0, 1\}$ with the new constraints $0 \leq y_i, z_j \leq 1$. It is known that *LP* can be solved in polynomial time finding a solution

$$(y^* = (y_1^*, \ldots, y_n^*), z^* = (z_1^*, \ldots, z_m^*))$$

with value $m^*_{LP}(x) \geq m^*_{ILP}(x)$, for every instance x, where $m^*_{LP}(x)$ and $m^*_{ILP}(x)$ denote the optimal value of the *LP* and *ILP* instances, respectively. The upper bound is obvious given that the set of admissible solutions is enlarged by the relaxation.

Let us consider algorithm GENAPPROX, see Fig. 35.10, that works as follows: first it solves the linear programming relaxation and so computes the optimal values (y^*, z^*); then, given a function g to be specified later, it computes, for each i, $i = 1, \ldots, n$, the probabilities $p_i = g(y^*{}_i)$. By Lemma 4 we know that a solution of weight:

$$W = \sum_{C_j \in \mathbf{C}} w_j (1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i)$$

must exist; by applying the method of conditional probabilities, such solution can be deterministically found.

If the function g can be computed in polynomial time then algorithm GENAPPROX runs in polynomial time. In fact the linear relaxation can be solved efficiently and the computation of the feasible solution can be computed in polynomial time with the method of conditional probabilities explained before.

The quality of approximation naturally depends on the choice of the function g. Let us suppose that this function finds suitable values such that:

$$(1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i) \geq \frac{3}{4} z_j^*.$$

GENAPPROX
Input: Set **C** of clauses in disjunctive normal form
Output: Set $\mathbf{C}'$ of clauses, $W = \sum_{C_j \in \mathbf{C}'} w_j$
1 Express the input **C** as an equivalent instance x of *ILP*
2 Find the optimum value y^*, z^* of x in the linear relaxation
3 Choose $p_i \leftarrow g(y^*{}_i)$, $i = 1, 2, \ldots, n$, for a suitable function g
4 $W \leftarrow \sum_{C_j \in \mathbf{C}} w_j (1 - \prod_{i \in X_j^+} (1 - p_i) \prod_{i \in X_j^-} p_i)$
5 Apply the method of conditional probabilities to find
6 a feasible solution $\mathbf{C}' = \{C_j \in \mathbf{C} : C_j \text{ is satisfied}\}$ of value W

Figure 35.10: The GENAPPROX algorithm, deterministic version.

If this inequality is satisfied, then the algorithm is a $3/4$-approximate algorithm for *MAX W–SAT*. In fact one has :

$$\begin{aligned} W &= \sum_{C_j \in \mathbf{C}} w_j (1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i) \geq \frac{3}{4} \sum_{C_j \in \mathbf{C}} w_j z_j^* = \\ &= \frac{3}{4} m_{\mathit{LP}}^*(x) \geq \frac{3}{4} m_{\mathit{ILP}}^*(x) \end{aligned}$$

More generally if one has :

$$(1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i) \geq \alpha z_j^*$$

one obtains a α-approximate algorithm.

A first interesting way of choosing the function g consists of applying the following technique, called *Randomized Rounding*, to get an integral solution from a linear programming relaxation. In order to get integer values one rounds the fractional values, that is each variable y_i is independently set to 1 (corresponding to the Boolean variable u_i being set to **true**) with probability y_i^*, for each $i = 1, 2, \ldots, n$. Hence the use of the randomized rounding technique is equivalent to choosing $p_i = g(y_i^*) = y_i^*, i = 1, 2, \ldots, n$.

Lemma 5 *Given the optimal values (y^*, z^*) to LP and given any clause C_j with k literals, one has*

$$(1 - \prod_{i \in U_j^+} (1 - y_i^*) \prod_{i \in U_j^-} y_i^*) \geq \alpha_k z_j^*$$

where

$$\alpha_k = 1 - \left(1 - \frac{1}{k}\right)^k.$$

Proof. Let us consider a clause C_j and, for the sake of simplicity, let us assume that every variable is unnegated. If a variable u_i would appear negated in C_j, one could substitute u_i by its negation $\overline{u}_i$ in every clause and also replace y_i by $1 - y_i$. So we can assume $C_j = u_1 \vee \cdots \vee u_k$ with the associated condition $y_1^* + \cdots + y_k^* \geq z_j^*$. The Lemma is proved by showing that:

$$1 - \prod_{i=1}^{k} (1 - y_i^*) \geq \alpha_k z_j^*.$$

In the proof we exploit the geometric inequality based on the properties of the arithmetic mean: given a finite set of nonnegative numbers $\{a_1, \ldots, a_k\}$,

$$\frac{a_1 + \cdots + a_k}{k} \geq \sqrt[k]{a_1 a_2 \cdots a_k}.$$

Now we apply the geometric inequality to the set $\{1-y_1^*,\ldots,1-y_k^*\}$. Because $\sum_{i=1}^k \frac{1-y_i^*}{k} = 1 - \frac{\sum_{i=1}^k y_i^*}{k}$, one has

$$1-\prod_{i=1}^{k}(1-y_i^*) \geq 1-(1-\frac{\sum_{i=1}^k y_i^*}{k})^k \geq 1-(1-\frac{z_j^*}{k})^k.$$

We note that the function $g(z_j^*) = 1-(1-\frac{z_j^*}{k})^k$ is concave in the interval $[0,1]$; hence it is sufficient to prove that $g(z_j^*) \geq \alpha_k z_j^*$ at the extremal points of the interval. Because one has

$$g(0) = 0 \text{ and } g(1) = \alpha_k$$

the Lemma is shown.
□

One can conclude that algorithm GENAPPROX with the choice $p_i = y_i^*$ reaches an approximation ratio equal to α_k. In particular for $k = 2$, the ratio is $3/4$. Note that, because α_k is decreasing with k, algorithm GENAPPROX is an α_k–approximation algorithm for formulae with *at most* k literals per clause.

Moreover, it is well known that $\lim_{k\to\infty}(1-\frac{1}{k})^k = \frac{1}{e}$; hence for arbitrary formulae one finds approximate solutions whose value is at least $1-\frac{1}{e}$ times the optimal value. Because $1-\frac{1}{e} = 0.632\ldots$, the randomized rounding obtains a better performance than RANDOM, but it looks as if one is far from achieving a $3/4$-approximation ratio.

Luckily, with a suitable merging of the above algorithm with RANDOM one obtains the desired performance ratio. Firstly let us recall that RANDOM is a $3/4$-approximation algorithm if all clauses have *at least* two literals. On the other hand, GENAPPROX is a $3/4$-approximation algorithm if we work with clauses with *at most* two literals. One algorithm is good for large clauses, the other for short ones. A simple combination consists of running both algorithm and choosing the best truth assignment obtained. Let us now consider the expected value obtained from the combination.

Theorem 35.4.5 *Let W_1 be the expected weight corresponding to $p_i = 1/2$ and let W_2 be the expected weight corresponding to $p_i = y_i^*$, $i = 1,2,\ldots,n$. Then one has :*

$$\max(W_1,W_2) \geq \frac{3}{4} m^*_{\boldsymbol{LP}}(x), \textit{ for any instance } x.$$

Proof. Because $\max(W_1,W_2) \geq \frac{W_1+W_2}{2}$, it is sufficient to show that $\frac{W_1+W_2}{2} \geq \frac{3}{4} m^*_{\boldsymbol{LP}}(x)$ for any x. Let us denote by $\mathbf{C}^k$ the set of clauses with exactly k literals. By Lemma 3, because $0 \leq z_j^* \leq 1$ one has

$$W_1 = \sum_{k\geq 1}\sum_{C_j\in\mathbf{C}^k} \gamma_k w_j \geq \sum_{k\geq 1}\sum_{C_j\in\mathbf{C}^k} \gamma_k w_j z_j^* \tag{35.7}$$

where $\gamma_k = (1-\frac{1}{2^k})$.

Moreover, by applying Lemma 5, one obtains:

$$W_2 \geq \sum_{k\geq 1}\sum_{C_j\in\mathbf{C}^k} \alpha_k w_j z_j^*. \tag{35.8}$$

Summing 35.7 and 35.8 one has :

$$\frac{W_1+W_2}{2} \geq \sum_{k\geq 1}\sum_{C_j\in\mathbf{C}^k} \frac{\gamma_k+\alpha_k}{2} w_j z_j^*.$$

We note that $\gamma_1+\alpha_1 = \gamma_2+\alpha_2 = 3/2$ and for $k \geq 3$ one has that $\gamma_k+\alpha_k \geq 7/8+1-\frac{1}{e} \geq 3/2$; Therefore:

$$\frac{W_1+W_2}{2} \geq \sum_{k\geq 1}\sum_{C_j\in\mathbf{C}^k} \frac{3}{4} w_j z_j^* = \frac{3}{4} m^*_{\boldsymbol{LP}}(x).$$

□

Note that it is not necessary to separately apply the two algorithms but it is sufficient to randomly choose one of the two algorithms with probability $1/2$, as it is done in algorithm 3/4–APPROXIMATE SAT.

3/4–APPROXIMATE SAT
Input: Set **C** of clauses in conjunctive normal form
Output: Set $\mathbf{C}'$ of clauses, $W = \sum_{C_j \in \mathbf{C}'} w_j$
1 Express the input **C** as an equivalent instance x of *ILP*
2 Find the optimum value (y^*, z^*) of x in the linear relaxation
3 With probability $1/2$ choose $p_i = 1/2$ or $p_i = y_i^*, i = 1, 2, \ldots, n$
4 $W \leftarrow \sum_{C_j \in \mathbf{C}} w_j (1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i)$
5 Apply the method of conditional probabilities to find a feasible
6 solution $\mathbf{C}' = \{C_j \in \mathbf{C} : C_j \text{ is satisfied}\}$ of value W

Figure 35.11: The 3/4–APPROXIMATE SAT algorithm: deterministic with performance ratio 3/4.

Corollary 2 *Algorithm* 3/4–APPROXIMATE SAT *is a* $3/4$*-approximation algorithm for MAX W–SAT.*

Proof. The proof derives from the above theorem and from the use of the method of conditional probabilities.
□

35.5 Local search for *SAT*

MAX–SAT is among the problems for which perturbative local search, described in Section 24.2, has been very effective: different variations of local search with randomness techniques have been proposed for *SAT* and *MAX–SAT* starting from the late eighties, see for example [164, 324], motivated by previous applications of "min-conflicts" heuristics in the area of Artificial Intelligence [270].

The general scheme is based on generating a starting point in the set of admissible solution and trying to improve it through the application of simple *basic* moves. If a move ("trial") is successful one accepts it, otherwise ("error") one keeps the current point. Of course, the successfulness of a local search technique depends on the neighborhood chosen and there are often trade-offs between the size of the neighborhood (and the related computational requirements to calculate it) and the quality of the obtained local optima.

In addition, as it will be demonstrated in Sec. 35.5.2, the use of a guiding function different from the original one can in some cases guarantee local optima of better quality.

Because this presentation is dedicated to the *MAX–SAT* problem, the search space that we consider is given by all possible truth assignments. Of course, a truth assignment can be represented by a binary string. For this presentation, let us consider the elementary changes to the current assignment obtained by changing a single truth value. The definitions are as follows.

Let $\mathcal{U}$ be the discrete search space: $\mathcal{U} = \{0, 1\}^n$, and let $f : \mathcal{U} \longrightarrow R$ (R are the real numbers) be the function to be maximized, i.e., in our case, the number of satisfied clauses. In addition, let $U^{(t)} \in \mathcal{U}$ be the current configuration along the *search trajectory* at iteration t, and $N(U^{(t)})$ the neighborhood of point $U^{(t)}$, obtained by applying a set of basic moves μ_i $(1 \leq i \leq n)$, where μ_i complements the i-th bit u_i of the string: $\mu_i \ (u_1, u_2, ..., u_i, ..., u_n) = (u_1, u_2, ..., 1 - u_i, ..., u_n)$. Clearly, these moves are idempotent ($\mu_i^{-1} = \mu_i$).

$$N(U^{(t)}) = \{U \in \mathcal{U} \text{ such that } U = \mu_i \, U^{(t)}, i = 1, ..., n\}$$

The version of *local search* (LS) that we consider starts from a random initial configuration $U^{(0)} \in \mathcal{U}$ and generates a search trajectory as follows:

$$V = \text{BEST-NEIGHBOR}(\ N(U^{(t)})\ ) \tag{35.9}$$

$$U^{(t+1)} = \begin{cases} V & \textbf{if} \quad f(V) > f(U^{(t)}) \\ U^{(t)} & \textbf{if} \quad f(V) \leq f(U^{(t)}) \end{cases} \tag{35.10}$$

where BEST-NEIGHBOR selects $V \in N(U^{(t)})$ with the best f value and ties are broken randomly. V in turn becomes the new current configuration if f improves. Other versions are satisfied with an improving (or non-worsening) neighbor, not necessarily the best one. Clearly, local search stops as soon as the first local optimum point is encountered, when no improving moves are available, see eqn. 35.10. Let us define as LS$^+$ a modification of LS where a specified number of iterations are executed and the candidate move obtained by BEST-NEIGHBOR is *always* accepted even if the f value remains equal or worsens.

35.5.1 Quality of local optima

Let m^* be the optimum value and k the minimum number of literals contained in the problem clauses.

For the following discussion it is useful to consider the different degree of *coverage* of the various clause for a given assignment. Precisely, let us define as COV_s the subset of clauses that have exactly s literals matched by the current assignment, and by $\text{COV}_s(l)$ the number of clauses in COV_s that contain literal l.

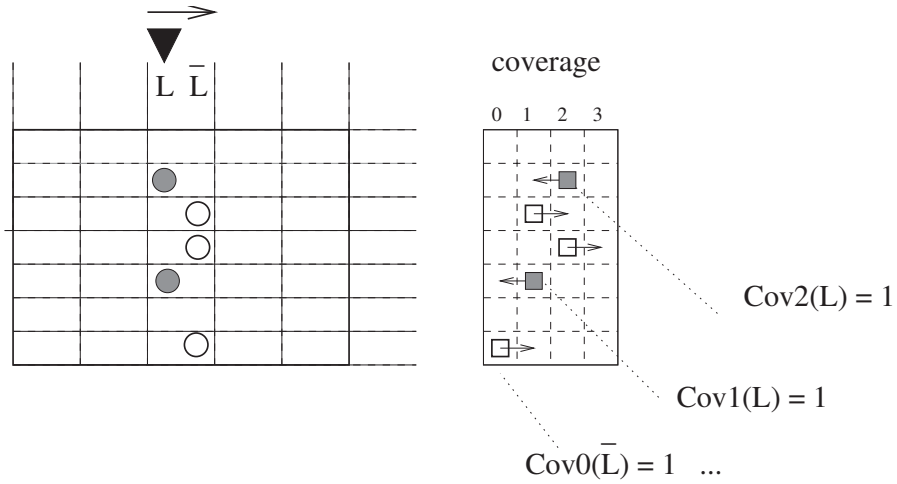


Figure 35.12: Literal L is changed from **true** to **false**.

One has the following theorem [172]:

Theorem 35.5.1 *Let m_{loc} be the number of satisfied clauses at a local optimum of any instance of MAX–SAT with at least k literals per clause. m_{loc} satisfies the following bound*

$$m_{loc} \geq \frac{k}{k+1} m$$

and the bound is sharp.

Proof. By definition, if the assignment U is a local optimum, one cannot flip the truth value of a variable (from **true** to **false** or *vice versa*) and obtain a net increase in the number of satisfied clauses f. Now, let $(\Delta f)_i$ by the increase in

f if variable u_i is flipped. By using the above introduced quantities one verifies that:

$$(\Delta f)_i = -\text{Cov}_1(u_i) + \text{Cov}_0(\overline{u_i}) \leq 0 \tag{35.11}$$

In fact, when u_i is flipped one looses the clauses that contain u_i as the single matched literal, i.e., $\text{Cov}_1(u_i)$ and gains the clauses that have no matched literal and that contain $\overline{u_i}$, i.e., $\text{Cov}_0(\overline{u_i})$.

After summing over all variables:

$$\sum_{i=1}^{n} \text{Cov}_0(\overline{u_i}) \leq \sum_{i=1}^{n} \text{Cov}_1(u_i) \tag{35.12}$$

$$k|\text{Cov}_0| \leq |\text{Cov}_1| \leq m_{loc} \tag{35.13}$$

where the equality $\sum_{i=1}^{n} \text{Cov}_0(\overline{u_i}) = k|\text{Cov}_0|$ and $\sum_{i=1}^{n} \text{Cov}_1(u_i) = |\text{Cov}_1|$ have been used. The equality are demonstrated by counting how many times a clause in Cov_0 (or Cov_1) is uncountered during the sum. For example, because all literals are unmatched for the clauses in Cov_0, each of them will be encountered k times during the sum.

The conclusion is immediate:

$$m = m_{loc} + |\text{Cov}_0| \leq (1 + \frac{1}{k})m_{loc} = \frac{k+1}{k} m_{loc} \tag{35.14}$$

□

The intuitive explanation is as follows: if there are too many clauses in Cov_0 , because each of them has k unmatched literals, there will be at least one variable whose flipping will satisfy so many of these clauses to lead to a net increase in the number of satisfied clauses.

There is therefore a very simple local search algorithm that reaches the same bound as the GREEDYJOHNSON1 algorithm. One starts from a truth assignments and keeps flipping variables that cause a net increase of satisfied clauses, until a local optimum is encountered. Of course, because one gains at least one clause at each step, there is an upper bound of m on the total number of steps executed before reaching the local optimum.

The following corollary is immediate:

Corollary 3 *If m_{loc} is the number of satisfied clauses at a local optimum, then:*

$$m_{loc} \geq \frac{k}{k+1} m^* \tag{35.15}$$

Besides *MAX–SAT* , many important optimization problems share the property that the ratio between the value of the local optimum and the optimal value is bounded by a constant. It is possible to define a class $\mathcal{GLO}$ composed of these problems. It is of interest to note that the closure of $\mathcal{GLO}$ coincides with $\mathcal{APX}$ [14].

35.5.2 Non-oblivious local optima

In the design of efficient approximation algorithms for *MAX–SAT* a recent approach of interest is based on the use of *non-oblivious functions* independently introduced in [8] and in [335].

Let us consider the classical local search algorithm LS for *MAX–SAT* , here redefined as *oblivious* local search (LS-OB). Clearly, the feasible solution found by LS-OB typically is only a *local* and not a *global* optimum.

Now, a different type of local search can be obtained by using a *different* objective function to direct the search, i.e., to select the best neighbor at each iteration. Local optima of the standard objective function f are not necessarily local optima of the different objective function. In this event, the second function causes an *escape* from a given local optimum. Interestingly enough, suitable *non-oblivious* functions f_{NOB} improve the performance of LS if one considers both the worst-case performance ratio and, as it has been shown in [34], the actual average results obtained on benchmark instances.

Let us mention a theoretical result for *MAX 2–SAT* . The d-neighborhood of a given truth assignment is defined as the set of all assignment where the values of at most d variables are changed. The theoretically-derived non–oblivious function for *MAX 2–SAT* is:

$$f_{NOB}(U) = \frac{3}{2}|\text{COV}_1| + 2|\text{COV}_2|$$

Theorems 7-8 of [335] state that:

Theorem 35.5.2 *The performance ratio for any oblivious local search algorithm with a d-neighborhood for MAX 2–SAT is* $2/3$ *for any* $d = o(n)$. *Non-oblivious local search with an 1-neighborhood achieves a performance ratio* $3/4$ *for MAX 2–SAT* .

Proof. While one is referred to the cited papers for the complete details, let us only demonstrate the second part of the theorem. The proof is a generalization of that for Theorem 35.5.1. Let the non-oblivious function be a weighted linear combination of the number of clauses with one and two matched literals:

$$f_{NOB} = a|\text{COV}_1| + b|\text{COV}_2|$$

Let $(\Delta f)_i$ by the increase in f if variable u_i is flipped. By using the definition of local optimum and the quantities introduced in Sec. 35.5.1 one has that $(\Delta f)_i \leq 0$ for each possible flip of a variable u_i. After expressing $(\Delta f)_i$ by using the above introduced quantities, one obtains:

$$-a|\text{COV}_1(u_i)| - (b-a)|\text{COV}_2(u_i)| + a|\text{COV}_0(\overline{u_i})| + (b-a)|\text{COV}_1(\overline{u_i})| \leq 0 \tag{35.16}$$

In fact, when u_i is flipped, all clauses that contain it decrease their coverage by one, while the clauses that contain $\overline{u_i}$ increase it by one, see also Fig. 35.12. As usual, let us assume that no clause contains both a literal and its negation.

After summing over all variables and collecting the sizes of the sets COV_i one obtains:

$$\sum_{i=1}^{n}(\Delta f)_i \leq 0 \tag{35.17}$$

$$\frac{b-a}{a}|\text{COV}_2| + \frac{2a-b}{2a}|\text{COV}_1| \geq |\text{COV}_0| \tag{35.18}$$

Now one can fix the relative size of the values a and b in order to get the best possible bound. This occurs when the coefficients of the terms $|\text{COV}_2|$ and $|\text{COV}_1|$ in equation 35.18 are equal, that is, for $b = \frac{4}{3}a$.

For these values one obtains the following bound:

$$|\text{COV}_2| + |\text{COV}_1| \geq 3\,|\text{COV}_0| \tag{35.19}$$

The number of satisfied clauses must be larger than three times the number of unsatisfied ones, which implies that $|\text{COV}_0| \leq \frac{1}{4}m$, or $m_{loc} \geq \frac{3}{4}m$.

□

Therefore LS-NOB, by using a function that weights in different ways the satisfied clauses according to the number of matched literals, improves considerably the performance ratio, even if the search is restricted to a much smaller neighborhood. In particular the "standard" neighborhood where all possible flips are tried is sufficient.

With a suitable generalization the above result can be extended: LS-NOB achieves a performance ratio $1 - \frac{1}{2^k}$ for *MAX–k–SAT* . The oblivious function for *MAX–k–SAT* is of the form:

$$f_{NOB}(U) = \sum_{i=1}^{k} c_i|\text{COV}_i|$$

and the above given performance ratio is obtained if the quantities $\Delta_i = c_{i+1} - c_i$ satisfy:

$$\Delta_i = \frac{1}{(k-i+1)\left(\begin{array}{c} k \\ i-1 \end{array} \right)} \left[\sum_{j=0}^{k-i} \left(\begin{array}{c} k \\ j \end{array} \right) \right]$$

Because the positive factors c_i that multiply $|\text{COV}_i|$ in the function f_{NOB} are strictly increasing with i, the approximations obtained through f_{NOB} tend to be characterized by a "redundant" satisfaction of many clauses. Better approximations, at the price of a limited number of additional iterations, can be obtained by a two-phase local search algorithm (NOB&OB): after a random start f_{NOB} guides the search until a local optimum is encountered [34]. As soon as this happens a second phase of LS is started where the move evaluation is based on f. A further reduction in the number of unsatisfied clauses can be obtained by a "plateau search" phase following NOB&OB: the search is continued for a certain number of iterations after the local optimum of OB is encountered, by using LS^+, with f as guiding function [34].

An example of non-oblivious search

Let us consider the following task with number of variables $n = 5$, and clauses $m = m^* = 4$, see also Fig. 35.13:

$$(\overline{u}_1 \vee \overline{u}_2 \vee u_3) \wedge (\overline{u}_1 \vee \overline{u}_2 \vee u_4) \wedge (\overline{u}_1 \vee \overline{u}_2 \vee u_5) \wedge (\overline{u}_3 \vee \overline{u}_4 \vee \overline{u}_5)$$

Let us assume that the assignment $U = (11111)$ is reached by OB local search. It is immediate to check that $U = (11111)$ is an oblivious local optimum with one unsatisfied clause (clause-4). While OB stops here, a possible sequence to reach the global optimum starting from U is the following: i) u_1 is set to **false**, ii) u_3 is set to **false**. Now, the first move does not change the number of satisfied clauses, but it changes the "amount of redundancy" (in clause-1 two literals are now satisfied, i.e., clause-1 enters COV_2) and the move *is* a possible choice for a selection based on the *non-oblivious* function. The *oblivious* plateau has been eliminated and the search can continue toward the globally optimal point $U = (01011)$.

35.5.3 Local search satisfies most *3–SAT* formulae

An intriguing result by Koutsoupias and Papadimitriou [239] shows that, for the vast majority of satisfiable *3–SAT* formulae, the local search heuristic that starts at a random truth assignments and repeatedly flips a variable that improves the number of satisfied clauses, almost always succeeds in discovering a satisfying truth assignment.

Let us consider all clauses that are satisfied by a given truth assignment $\hat{U}$ and let us pick each of them with probability $p = 1/2$ to build a *3–SAT* formula. The following theorem [239] is demonstrated:

Theorem 35.5.3 *Let $0 < \varepsilon < 1/2$. Then there exists c,*
$c \approx \left(1 - \sqrt{1-(1/2-\varepsilon)^2}\right)^2 /6$*, such that for all but a fraction of at most $n2^n e^{-cn^2/2}$ satisfiable 3–SAT formulae with n variables, the probability that local search succeeds in discovering a truth assignment in each independent trial from a random start is at least $1 - e^{-\varepsilon^2 n}$.*

Proof. Let us focus on the structure of the proof, without giving the technical details. One assumes that there is an assignment $\hat{U}$ that satisfies all clauses and shows that, if one starts from a *good* initial assignment, i.e., one that agrees with $\hat{U}$ in at least $(1/2 - \varepsilon)$ variables, the probability that the local search is ever *mislead* is small. By "*mislead*" one means that, when a variable is flipped, the Hamming distance between $U^{(t)}$ and $\hat{U}$ increases. The Hamming distance between two binary strings is given by the number of differing bits.

In detail, the quantity $1 - e^{-\varepsilon^2 n}$ in the theorem is the probability that the initial random truth assignment is *good* (use Chernoff bound). Then one demonstrates that, if the initial assignment is good, the probability that one does not reduce the Hamming distance between $U^{(t)}$ and $\hat{U}$ when an improving neighbor is chosen is at most $2e^{-cpn^2}$, the

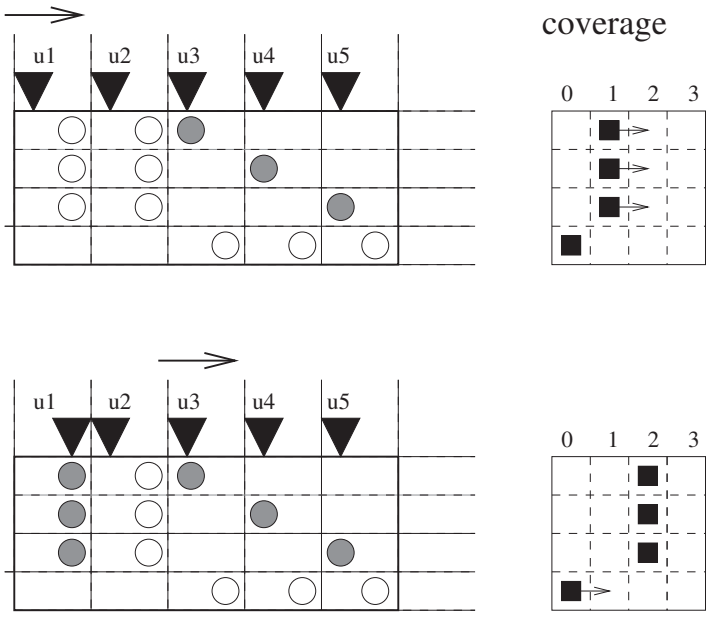


Figure 35.13: Non-oblivious search takes the different coverage into account.

probability being measured with respect to the random choice of the clauses to build the original formula ($p = 1/2$ for the above theorem).

Finally, the probability that local search starting from a good assignment will ever be misled by flipping a variable during the entire search trajectory is at most $n2^n e^{-cpn^2}$, since there are at most $n2^{n-1}$ such possible flippings – the number of edges of the n-hypercube.

□

The original formulation of the above theorem is for a *greedy* version of local search, using the function BEST-NEIGHBOR described in eqn. 35.9, but the authors note that greediness is not required for the theorem to hold, although it may be important in practice.

Let us finally note that the result, while of theoretical interest, is valid for formulae with many clauses (p must be such that the expected number of clauses is $\Omega(n^2)$), while the most difficult formulae have a number of clauses that is linear in n, see also Sec 35.8.1.

35.5.4 Randomized search for *2–SAT* (Markov processes)

A "natural" polynomial-time *randomized* search algorithm for *2–SAT* is presented in [288]. While it has long been known that *2–SAT* is a polynomially solvable problem, the algorithm is of interest because of its simplicity and is summarized here also because it motivated the GSAT-WITH-WALK algorithm of [325], see also Sec. 35.6.2.

In its "standard" form, local search is guided by the number of satisfied clauses and the basic criterion is that of accepting a neighbor only if more clauses are satisfied. The paper by Papadimitriou [288] changes the perspective by concentrating the attention to the *unsatisfied* clauses.

The algorithm for *2–SAT* , is extremely simple:

Let us note that worsening moves, leading to a lower number of satisfied clause, can be accepted during the search.

One can prove that:

Theorem 35.5.4 *The* MARKOVSEARCH *randomized algorithm for 2–SAT , if the instance is satisfiable, finds a satisfying assignment in $O(n^2)$ expected number of steps.*

```
MARKOVSEARCH
1    Start with any truth assignment
2    while there are unsatisfied clauses do
3        pick one of them and flip a random literal in it
```

Figure 35.14: The MARKOVSEARCH randomized algorithm for *2–SAT* .

Proof. The proof involves an aggregation of the states of the Markov chain so that the chain is mapped to the *gambler's ruin* chain. A sketch of the proof is derived from [276]. Given an instance with a satisfying assignment $\hat{U}$, and the current assignment $U^{(t)}$, the progress of the algorithm can be represented by a particle moving between the integers $\{0, 1, ..., n\}$ on the real line. The position of the particle indicates how many variables in $U^{(t)}$ agree with those of $\hat{U}$. At each iteration the particle's position can change only by one, from the current position i to $i + 1$ or $i - 1$ for $0 < i < n$. A particle at 0 can move only to 1, and the algorithm terminates when the particle reaches position n, although it may terminate at some other position with a satisfying assignment different from $\hat{U}$. The crucial fact is that, in an unsatisfied clause, at least one of the *two* literals has an incorrect value and therefore, with probability at least $1/2$, the number of correct variables increases by one when a randomized step is executed.

The random walk on the line is one of the most extensively studied stochastic processes. In particular, the above process is a version of the "gambler's ruin" chain with reflecting barrier (that is, the house cannot lose its last dollar). Average number of steps for the gambler to be ruined is $O(n^2)$.
□

35.6 Memory-less Local Search Heuristics

State-of-the-art heuristics for *MAX–SAT* are obtained by complementing local search with schemes that are capable of producing better approximations beyond the locally optimal points. In some cases, these schemes generate a sequence of points in the set of admissible solutions in a way that is fixed before the search starts. An example is given by *multiple runs* of local search starting from different random points. The algorithm does not take into account the *history* of the previous phase of the search when the next points are generated. The term *memory-less* denotes this lack of feedback from the search history.

In addition to the cited *multiple-run* local search, these techniques are based on Markov processes (Simulated Annealing), see Sec. 35.6.1, "plateau" search and "random noise" strategies, see Sec. 35.6.2, or combinations of randomized constructions and local search, see Sec. 35.6.3.

35.6.1 Simulated Annealing

Simulated Annealing has been described in Seciton 25.5. The use of a Markov process (Simulated Annealing or SA for short) to generate a stochastic search trajectory is illustrated in Fig. 35.15, adapted from [343].

For a certain number of tries, a random truth assignment is generated (line 2) and the $temperature$ parameter is set to MAX-TEMP. In the inner loop, new assignments are generated by probabilistically flipping each variable based on the improvement δ in the number of satisfied clauses that would occur after the flip. Of course, the improvement can be negative. The probability to flip is given by a logistic function that penalizes smaller or negative improvements (line 9). The inner loop controls the *annealing schedule*: when $iter$ increases the $temperature$ slowly decreases (line 5) until a minimum of MIN-TEMP is reached and the control exits the loop (line 6) Let us note that, when the $temperature$ is large, the moves are similar to those produced by a random walk, while, when the $temperature$ is low the acceptance criterion of the moves is that of local search and the algorithm resembles GSAT, that will be introduced in Sec. 35.6.2. Implementation details, the addition of a "random walk" modification inspired by [325], and experimental results are described in the cited paper.

```
SA
1   for tries ← 1 to MAX-TRIES
2       U ← random truth assignment ; iter ← 0
3       forever
4           if U satisfies all clause then return U
5           temperature ← MAX-TEMP × e^{−iter×decay_rate}
6           if temperature < MIN-TEMP then exit loop
7           for i ← 1 to n
8               δ ← increase of satisfied clauses if u_i is flipped
9               FLIP(u_i)with probability1/(1 + e^{−δ/temperature})
10          iter ← iter + 1
```

Figure 35.15: The Simulated Annealing algorithm for *SAT* .

35.6.2 GSAT with "random noise" strategies

SAT is of special concern to Artificial Intelligence because of its connection to *reasoning*. In particular, deductive reasoning is the complement of satisfiability: from a collection of base facts A one should deduce a sentence F if and only if $A \cup \overline{F}$ is *not* satisfiable, see also Sec. 35.2.2. The popular and effective algorithm GSAT was proposed in [324] as a *model-finding* procedure, i.e., to find an interpretation of the variables under which the formula comes out **true**. GSAT consists of multiple runs of LS$^+$, each run consisting of a number of iterations that is typically proportional to the problem dimension n. The experiments in [324] show that GSAT can be used to solve hard (see sec. 35.8.1) randomly generated problems that are an order of magnitude larger than those that can be solved by more traditional approaches like Davis-Putnam or resolution. Of course, GSAT is an incomplete procedure: it could fail to find an optimal assignment. An extensive empirical analysis of GSAT is presented in [145, 144].

Different "noise" strategies to escape from attraction basins are added to GSAT in [325, 323]. In particular, the GSAT-WITH-WALK algorithm has been tested in [323] on the Hansen-Jaumard benchmark of [172], where a better performance with respect to SAMD is demonstrated, although requiring much longer CPU times. See Sec. 35.7.1 for the definition of SAMD.

```
GSAT-WITH-WALK
1   for i ← 1 to MAX-TRIES
2       U ← random truth assignment
3       for j ← 1 to MAX-FLIPS
4           if RANDOMNUMBER < p then
5               u ← any variable occurring in some unsat. clause
6           else
7               u ← any variable with largestΔf
8           FLIP(u)
```

Figure 35.16: The GSAT-WITH-WALK algorithm. RANDOMNUMBER generates random numbers in the range $[0, 1]$.

The algorithm is briefly summarized in Fig. 35.16. A certain number of tries (MAX-TRIES) is executed, where each try consists of a number of iterations (MAX-FLIPS). At each iteration a variable is chosen by two possible criteria and then flipped by the function FLIP, i.e., U_i becomes equal to $(1 - U_i)$. One criterion, active with "noise" probability p, selects a variable occurring in some unsatisfied clause with uniform probability over these variables, the other one is the standard method based on the function f given by the number of satisfied clauses. The first criterion was motivated by [288], see also Sec. 35.5.4. For a generic move μ, the quantity $\Delta_\mu f$ (or Δf for short) is defined as $f(\mu\ U^{(t)}) - f(U^{(t)})$. The straightforward book-keeping part of the algorithm is not shown. In particular, the best assignment found during all trials is saved and reported at the end of the run. In addition, the run is terminated

immediately if an assignment is found that satisfies all clauses. The original GSAT algorithm can be obtained by setting $p = 0$ in the GSAT-WITH-WALK algorithm of Fig. 35.16.

35.6.3 Randomized Greedy and Local Search (GRASP)

A hybrid algorithm that combines a randomized greedy construction phase to generate initial candidate solutions, followed be a local improvement phase is the GRASP scheme proposed in [304] for the *SAT* and generalized for the *MAX W–SAT* problem in [305], a work that is briefly summarized in this section.

GRASP is an iterative process, with each iteration consisting of two phases, a construction phase and a local search phase.

During each construction, all possible choices are ordered in a candidate list with respect to a greedy function measuring the (myopic) benefit of selecting it. The algorithm is randomized because one picks in a random way one of the best candidates in the list, not necessarily the top candidate. In this way different solutions are obtained at the end of the construction phase.

Because these solutions are not guaranteed to be locally optimal with respect to simple neighborhoods, it is usually beneficial to apply a local search to attempt to improve each constructed solution.

```
GRASP(RCLSize, MaxIter, RandomSeed)
1       △ Input instance and initialize data structures
2       for i ← 1 to MaxIter
3           [ U ← CONSTRUCTGREEDYRAND(RCLSize, RandomSeed)
4           [ U ← LOCALSEARCH(U)
```

```
CONSTRUCTGREEDYRAND(RCLSize, RandomSeed)
1       for k ← 1 to n
2           [ MAKERCL(RCLSize)
3           [ s ← SELECTINDEX(RandomSeed)
4           [ ASSIGNVARIABLE(s)
5           [ ADAPTGREEDYFUNCTION(s)
```

Figure 35.17: The GRASP algorithm (above) and the randomized greedy construction (below).

A high-level description of the GRASP algorithm is presented in Fig. 35.17, a summarized version of the more detailed description in [305]. After reading the instance and initializing the data structures one repeats for $MaxIter$ iterations the construction of an assignment U and the application of local search starting from U to produce a possibly better assignment (lines 2–4). Of course, the best assignment found during all iterations is saved and reported at the end. In addition to $MaxIter$, the parameters are $RCLSize$, the size of the restricted candidate list of moves out of which a random selection is executed, and a random seed used by the random number generator. In detail, see function CONSTRUCTGREEDYRAND in Fig. 35.17, the restricted candidate list of assignments is created by MAKERCL, the index of the next variable to be assigned a truth value is chosen by SELECTINDEX, the truth value is assigned by ASSIGNVARIABLE and the greedy function that guides the construction is changed by ADAPTGREEDYFUNCTION to reflect the assignment just made.

The remaining details about the greedy function (designed to maximize the total weight of yet-unsatisfied clauses that become satisfied after a given assignment), the creation of the restricted candidate list, and local search (based on the *1-flip* neighborhood) are presented in [305], together with experimental results.

35.7 History-sensitive Heuristics

Different history-sensitive heuristics have been proposed to continue local search schemes beyond local optimality. These schemes aim at intensifying the search in promising regions and at diversifying the search into uncharted territories by using the information collected from the previous phase (the *history*) of the search. The *history* at iteration t is formally defined as the set of ordered couples (U, s) such that $0 \leq s \leq t$ and $U = U^{(s)}$.

Because of the internal feedback mechanism, some algorithm parameters can be modified and tuned in an *on-line* manner, to reflect the characteristics of the *task* to be solved and the *local* properties of the configuration space in the neighborhood of the current point. This tuning has to be contrasted with the *off-line* tuning of an algorithm, where some parameters or choices are determined for a given problem in a preliminary phase and they remain fixed when the algorithm runs on a specific instance.

35.7.1 Prohibition-based Search: TS and SAMD

Tabu Search (TS) is a *history-sensitive* heuristic proposed by F. Glover [148] and, independently, by Hansen and Jaumard, that used the term SAMD ("steepest ascent mildest descent") and applied it to the *MAX–SAT* problem in [172]. The main mechanism by which the history influences the search in TS is that, at a given iteration, some neighbors are *prohibited*, only a non-empty subset $N_A(U^{(t)}) \subset N(U^{(t)})$ of them is *allowed*. The general way of generating the search trajectory that we consider is given by:

$$N_A(U^{(t)}) = \text{ALLOW}(N(U^{(t)}), U^{(0)}, ..., U^{(t)}) \quad (35.20)$$

$$U^{(t+1)} = \text{BEST-NEIGHBOR}(\ N_A(U^{(t)})\ ) \quad (35.21)$$

The set-valued function ALLOW selects a non-empty subset of $N(U^{(t)})$ in a manner that depends on the entire previous history of the search $U^{(0)}, ..., U^{(t)}$. Let us note that worsening moves *can* be produced by eqn. 35.21, as it must be in order to exit local optima.

The introduction of algorithm SAMD is motivated in [172] by contrasting the technique with Simulated Annealing (SA) [234] for maximization. The directions of local changes are little explored by SA: for example, if the objective function increases, the change is always accepted however small it may be. On the contrary, it is desirable to exploit the information on the direction of *steepest* ascent and yet to retain the property of not being blocked at the first local optimum found. SAMD performs local changes in the direction of steepest ascent until a local optimum is encountered, then a local change along the direction of mildest descent takes place and the reverse move is *forbidden* for a given number of iterations to avoid cycling with a high probability. The details of the SAMD technique as well as additional specialized devices for detecting and breaking cycles are outlined in [172]. A computational comparison with SA and with Johnson's two algorithms is also presented. A specialized Tabu Search heuristic is used in [215] to speed up the search for a solution (if the problem is satisfiable) as part of a branch-and-bound algorithm for *SAT* , that adopts both a relaxation and a decomposition scheme by using polynomial instances, i.e., *2–SAT* and Horn *SAT* .

35.7.2 HSAT and "clause weighting"

In addition to the already cited SAMD [172] heuristic that uses the temporary prohibitions of recently executed moves, let us mention two variations of GSAT that make use of the previous history.

HSAT [145] introduces a tie-breaking rule into GSAT: if more moves produce the same (best) Δf, the preferred move is the one that has not been applied for the longest span. HSAT can be seen as a "soft" version of Tabu Search: while TS prohibits recently-applied moves, HSAT discourages recent moves if the same Δf can be obtained with moves that have been "inactive" for a longer time. It is remarkable to see how this innocent variation of GSAT can increase its performance on some *SAT* benchmark tasks [145].

Clause–weighting has been proposed in [321] in order to increase the effectiveness of GSAT for problems characterized by strong asymmetries. In this algorithm a positive weight is associated to each clause to determine how often the clause should be counted when determining which variable to flip. The weights are dynamically modified

during problem solving and the qualitative effect is that of "filling in" local optima while the search proceeds. Clause–weighting can be considered as a "reactive" technique where a repulsion from a given local optimum is generated in order to induce an escape from a given attraction basin.

35.7.3 The Hamming-Reactive Tabu Search (H-RTS) algorithm

The Reactive Search Optimization (RSO) principles of "learning while searching" have been introduced in Chapter 27.

An algorithm that combines the previously described techniques of local search (oblivious and non-oblivious), the use of prohibitions (see TS and SAMD), and a reactive scheme to determine the prohibition parameter is presented in [33]. The algorithm is called HAMMING-REACTIVE-TS algorithm, and its core is illustrated in Fig. 35.18.

```
HAMMING-REACTIVE-TS
1     repeat
2         t_r ← t
3         U ← random truth assignment
4         T ← ⌊T_f n⌋
5         repeat                                  { NOB local search }
6             U ← BEST-MOVE(LS, f_NOB)
7         until largest Δf_NOB = 0
8         repeat
9             repeat                              { local search }
10                U ← BEST-MOVE(LS, f_OB)
11            until largest Δf_OB = 0
12            U_I ← U
13            for 2(T + 1) iterations             { reactive tabu search }
14                U ← BEST-MOVE(TS, f_OB)
15            U_F ← U
16            T ← REACT(T_f, U_F, U_I)
17        until (t − t_r) > 10 n
18    until solution is acceptable or max. number of iterations reached
```

Figure 35.18: The *H-RTS* algorithm.

The initial truth assignment is generated in a random way, and non-oblivious local search (LS-NOB) is applied until the first local optimum of f_{NOB} is encountered. LS-NOB obtains local minima of better average quality than LS-OB, but then the guiding function becomes the standard oblivious one. This choice was motivated by the success of the NOB & OB combination [34] and by the poor diversification properties of NOB alone, see [33].

The search proceeds by repeating phases of local search followed by phases of TS (lines 8–17 in Fig. 35.18), until a suitable number of iterations are accumulated after starting from the random initial truth assignment (see line 17 in Fig. 35.18). A single elementary move is applied at each iteration. The variable t, initialized to zero, identifies the current iteration and increases after a local move is applied, while t_r identifies the iteration when the last random assignment was generated. Some trivial bookkeeping details (like the increase of t) are not shown in the figure.

During each combined phase, first the local optimum of f is reached, then $2(T + 1)$ moves of Tabu Search are executed. The design principle underlying this choice is that prohibitions are necessary for diversifying the search only after LS reaches a local optimum. The fractional prohibition T_f is changed during the run by the function REACT to obtain a proper balance of diversification and bias [33].

The random restart executed after $10\ n$ moves guarantees that the search trajectory is not confined in a localized portion of the search space.

Being an heuristic algorithm, there is not a natural termination criterion. In its practical application, the algorithm is therefore run until either the solution is acceptable, or a maximum number of moves (and therefore CPU time) has elapsed. What is demonstrated in the computational experiments in [33] is that, given a fixed number of iterations, HAMMING-REACTIVE-TS achieves much better average results with respect to competitive algorithms (GSAT and GSAT-WITH-WALK). Because, to a good approximation, the actual running time is proportional to the number of iterations, HAMMING-REACTIVE-TS should therefore be used to obtain better approximations in a given allotted number of iterations, or equivalent approximations in a much smaller number of iterations.

35.8 Models of hardness and threshold effects

Given the hardness of the problem and the relevancy for applications in different fields, the emphasis on the experimental analysis of algorithms for the *MAX–SAT* problem has been growing in recent years.

In some cases the experimental comparisons have been executed in the framework of "challenges," with support of electronic collection and distribution of software, problem generators and test instances. Practical and industrial *MAX–SAT* problems and benchmarks, with significant case studies are also presented in [120], see also the contained review [163].

In some cases it is of interest to model the hardness of instances with appropriate models. Let us describe some basic problem models that are considered both in theoretical and in experimental studies of *MAX–SAT* algorithms [163].

- **k-SAT model**, also called **fixed length clause model**. A randomly generated CNF formula consists of independently generated random clauses, where each clause contains exactly k literals. Each literal is chosen uniformly from $U = \{u_1, ..., u_n\}$ without replacement, and negated with probability p. The default value for p is $1/2$.
- **average k-SAT model**, also called **random clause model**. A randomly generated CNF formula consists of independently generated random clauses. Each literal has ha probability p of being part of a clause. In detail, each of the n variables occurs positively with probability $p(1-p)$, negatively with probability $p(1-p)$, both positively and negatively with probability p^2, and is absent with probability $(1-p)^2$.

Both models have many variations depending on whether the clauses are required to be different, whether a variable and its negation can be present in the same clause, etc.

Although superficially similar, the two models differ in the *difficulty* to solve the obtained formulae and in the mathematical analysis. In particular, when the initial formula comes from the average k-SAT model, a step that fixes the value of a variable produces a set of clauses from the same model, while if the same step is executed in the k-SAT model, the resulting clauses do not necessarily have the same length and therefore do not come from the k-SAT model.

Other *structured* problem models are derived from the mapping of instances of different problems, like *coloring*, n-queens, etc. [207]. The performance of algorithms on these more structured models tend to have little correlation with the performance tested on the above introduced random problems. Unfortunately, the theoretical analysis of these more structured problems is very hard. The situation worsens if one considers "real-world" practical applications, where one is typically confronted with a few instances and little can be derived about the "average" performance, both because the probability distribution is not known and because the number of instances tends to be very small.

A compromise can be reached by having parametrized generators that capture some of the relevant structure of the "real-world" problems of interest.

35.8.1 Hardness and threshold effects

Different algorithms demonstrate a different degree of effort, measured by number of elementary steps or CPU time, when solving different kinds of instances. For example, Mitchell et al. [272] found that some distributions used in past experiments are of little interest because the generated formulae are almost always very easy to satisfy. They also

reported that one can generate very hard instances of k-SAT, for $k \geq 3$. In addition, they report the following observed behavior for random fixed length *3–SAT* formulae: if r is the ratio r of clauses to variables ($r = m/n$), almost all formulae are satisfiable if $r < 4$, almost all formulae are unsatisfiable if $r > 4.5$. A rapid transition seems to appear for $r \approx 4.2$, the same point where the computational complexity for solving the generated instances is maximized, see [235, 99] for reviews of experimental results.

A series of theoretical analyses aim at approximating the unsatisfiability threshold of random formulae. Let us define the notation and summarize some results obtained.

Let **C** be a random k-SAT formula. The research problem that has been considered, see for example [236], is to compute the least real number κ such that, if r is larger than κ, then the probability of **C** being satisfiable converges to 0 as n tends to infinity. In this case one says that **C** is asymptotically almost certainly satisfiable. Experimentally, κ is a threshold value marking a "sudden" change from probabilistically certain satisfiability to probabilistically certain unsatisfiability. More precisely [5], given a sequence of events $\mathcal{E}_i$, one says that $\mathcal{E}_n$ occurs almost surely (a.s.) if $\lim_{n\to\infty} \mathbf{Pr}[\mathcal{E}_n] = 1$, where $\mathbf{Pr}[event]$ denotes the probability of an event. The behavior observed in experiments with random k-SAT leads to the following conjecture:
For every $k \geq 2$, there exist r_k such that for any $\varepsilon > 0$, random instances of k-SAT with $(r_k - \varepsilon)n$ clauses are a.s. satisfiable and random instances with $(r_k + \varepsilon)n$ clauses are a.s. unsatisfiable

For $k = 2$ (i.e., for the polynomially solvable *2–SAT*) the conjecture was proved [89, 156], in fact showing that $r_2 = 1$. For $k = 3$ much less progress has been made: neither the existence of r_3 nor its value has been determined.

In the fixed-length *3–SAT* model, the total number of all possible clauses is $8\binom{n}{3}$ and the probability that a random clause is satisfied by a truth assignment U is $7/8$.

Let $\mathcal{U}_n$ be the set of all truth assignments on n variables, and let $\mathcal{S}_n$ be the set of assignments that satisfy the random formula **C**. Therefore the cardinality $|\mathcal{S}_n|$ is a random variable. Given **C**, let $|\mathcal{S}_n(\mathbf{C})|$ be the number of assignments satisfying **C**.

The expected value of the number of satisfying truth assignments of a random formula, $\mathbf{E}[|\mathcal{S}_n|]$, is defined as:

$$\mathbf{E}[|\mathcal{S}_n|] = \sum_{\mathbf{C}} (\mathbf{Pr}[\mathbf{C}]\,|\mathcal{S}_n(\mathbf{C})|) \tag{35.22}$$

The probability that a random formula is satisfiable is:

$$\mathbf{Pr}[\text{the random formula is satisfiable}] = \sum_{\mathbf{C}} (\mathbf{Pr}[\mathbf{C}]\, I_{\mathbf{C}}) \tag{35.23}$$

where $I_{\mathbf{C}}$ is 1 if **C** is satisfiable, 0 otherwise.

From equations (35.22) and (35.23) the following Markov's inequality follows:

$$\mathbf{Pr}[\text{the random formula is satisfiable}] \leq \mathbf{E}[|\mathcal{S}_n|] \tag{35.24}$$

Let us now consider the "first moment" argument to obtain an upper bound for κ in the *3–SAT* model. First one observes that the expected number of truth assignments that satisfy **C** is $2^n(7/8)^{rn}$, then one lets this expected value converge to zero and uses the above Markov's inequality. From this one obtains

$$\kappa \leq \log_{8/7} 2 = 5.191$$

This result has been found independently by many people, including [129] and [90]. More refined studies are present for example in [77, 236, 226, 5]

Gist

SAT and *MAX–SAT* deal with assigning truth values to variables in Boolean formulas to make them true ("satisfied"). They are interesting and extremely relevant optimization problems, easy to define and visualize, for which most algorithmic ideas have been tried with success in creative combinations.

These include branch-and-bound, integer linear programming, continuous approaches, approximation algorithms, randomized algorithms, perturbative local search and greedy constructions, reactive search optimization, experimental studies of hardness.

A full grasp of the approaches for solving *SAT* and *MAX–SAT* is therefore beneficial to fully understand how to apply these principles for a very concrete simple problem, before considering more complicated versions like constraint programming (CP).

The satisfaction of studying and using *SAT* and *MAX–SAT* is never ending.

Chapter 36

Design of experiments, query learning, and surrogate model-based optimization

Scientists and engineers study the behavior of systems to find improving or optimal configurations. Consider for example the fuel consumption of a motor; it will depend on various parameters, including the motor geometry, the temperature of operation, the kind of fuel, etc. To minimize it, one needs a model of how the consumption depends on the design parameters.

In abstract terms, one deals with a system transforming a vector of inputs X into an output Y. Only in some very rare cases an explicit and exact analytic model is available. In other and more frequent cases, deriving the output Y requires running a simulator or even building prototypes. The two operations can be very costly in terms

of time or money. Minimizing the number of simulator runs or experiments is a therefore a critical issue. Building a surrogate (approximated) model based on the executed experiments can be a way, if the model evaluation is much faster than the real system. When an initial model is available, it can be used to make predictions of to identify optimal configurations by optimization. If calculating $Y(\mathbf{x})$ for different $\mathbf{x}$ values is very costly, all evaluations done are collected to build initial models, useful to guide the generation of the future input configurations to be evaluated. This is a paradigmatic case of "learning while optimizing", also known as **surrogate optimization**, or **optimization based on response surfaces** (Response Surface Methodology or RSM), and related to Kriging (Section 1.2), to locally-weighted regression (Section 9.2), and to global optimization schemes based on memory and statistical inference (Section 25.4).

In this chapter we focus on data-driven models for which **experimental data is not available at the beginning and needs to be acquired in a strategic and intelligent manner**. In medicine, one may want to understand if a medical treatment is successful or not. Experiments imply treating patients with different medicines in a carefully controlled context, to avoid *placebo* effects, or effects caused by different populations of patients. Some patients may have dangerous side-effects during the tests, some may even die. As one can imagine, these experiments are incredibly costly and slow, causing years of delay between the invention of new treatments and their commercialization. In manufacturing, one may want to fine-tune a casting process. Through casting a liquid material is poured into a mold, which contains a hollow cavity of the desired shape, and then allowed to solidify. It is the main method to produce most of our physical goods, but also very complex and fragile. Variations of temperature, pressure, speed of injection, composition of the metal, may lead to defective parts which need to be discarded. In manufacturing, simulating a specific setting or - even worse - running a production facility just to accumulate data can be very time-consuming and costly.

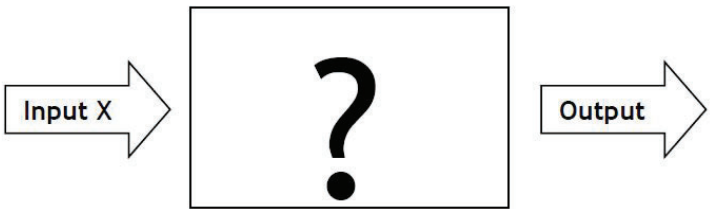


Figure 36.1: Design of Experiments (DOE): how to generate appropriate inputs to study and model an unknown system.

When one needs experimental data to study a system or to **generate training examples in machine learning**, the objectives are to limit costs, while getting informative data, sufficient to build precise models. The topic is not recent, being deeply related with the scientific method. In science and engineering it is known as **design of experiments**. In ML, with a different but related context, it is known as **active or query learning**.

36.1 Design of experiments (DOE)

Experimentation is a goal-directed activity. In spite of stories of apples falling on sleeping scientists leading to elegant theories of gravitation, most experiments need to be **designed** with precise objectives and with a careful allocation of physical or computational resources. Inspiration and creativity are crucial to define the initial goals but sterile without the 90% "perspiration" in the experimental work.

If the objective is to model a system and understand which inputs are relevant to predict the output, the questions (the input points $\mathbf{x}$) have to be produced in order to derive accurate models, and to identify the *factors of variation*. The concept of factors of variations is related to the concept of *informative features* in ML. Inputs that do not influence

the output in a significant manner can be eliminated from the model. **Design of experiments** refers to a systematic and principled manner to generate examples to characterize, predict, and then improve the behavior of any system or process.

All experimental activity is usually accompanied by **experimental noise** in the form of measurement errors, hidden and uncontrollable factors affecting the experiments, and by the **possibility to be fooled by chance** into deriving wrong conclusions.

Because physical measurements are subject to errors, one often assumes that the measured output $Y(X)$ is equal to a "true" response $Y_t(X)$ plus a random error term $\epsilon : Y(X) = Y_t(X) + \epsilon$, ϵ being a random variable chracterized by a given distribution (usually, a Gaussian with a given standard deviation σ).

We will first introduce some DOE methods popular with engineers and then (in Section 36.3) present the different but related concept of active and query learning.

In DOE, inputs are usually called "design variables" or "factors", output is called "response". Let's note that no simple receipt exists to apply DOE and response surfaces. All techniques depend on assumptions that can be easily violated in real systems, so that the focused attention by the experimenter and the critical use of more than one technique are required to avoid the most obvious mistakes. Let's just make a motivating example. Imagine that we want to measure the effect of a medication on cholesterol level and let's assume that the cholesterol levels depend both on taking the medication but also on age. If the original experimental data are not generated with a careful DOE, it can happen – by chance, or by having two different hospitals with patients of different age collecting the data - that the patients receiving the medication are much older, so that cholesterol levels are higher in spite of the medication. The effect of the medication can be canceled or reversed by the effect of age. To separate the medication effect, the sample of people should have their age randomized (no statistically significant age distribution should be present in the two groups).

In many engineering applications, a simulator can be used to generate example outputs Y corresponding to user-chosen inputs X. The situation can be different if examples require running a physical system, or observing a process in action. But if one is in the lucky case, designing an experiment means deciding the appropriate number of examples to generate and where they are generated in the input space. The overall DOE objective of getting a surrogate model is clear, but the devil is in the details, and one must consider the assumptions and how the model is going to be used. Selection of a proper DOE is a matter that requires expert advice, knowledge of the problem, and estimates of the time-budget allocated to the generation of sample output values, which usually is the most time-consuming part.

A simple constraint on the input can be given by specifying a design space, usually by setting separate upper and lower bounds on each x variable. Without loss of generality, if n is the input dimension, we can think about the hypercube $[-1, 1]^n$ or $[0, 1]^n$. The original range can then be recovered by appropriately scaling the individual variables. If one has reasons to believe that the form of the model is correct (for example, that the real physical process is described by a low-order polynomial), placing the examples near or on the boundaries of the design space makes sense because this choice will minimize the effect of the random error ϵ on the determination of the model parameters. Think about fitting a line with two points, the more separated they are in the x direction, the more stable will be the line with respect to random noise in the y position. On the contrary, if the model is a rough approximation, and one aims at using the model in the interior part of the design space, for example for finding an optimal configuration, a good fraction of the points should be placed also in the interior part of the design space. In other words, one could aim at minimizing the average model discrepancy (called generalization error in the language of machine learning) more than at determining individual coefficients with a high precision. Another critical parameter is the total number of examples. In general, the more examples the better (provided that they are well distributed), but the computational cost of running the simulator can be very large and one must settle for the minimal number which is sufficient to identify a workable model. A second possibility is to use multiple iterations, by first running an exploratory analysis in the entire design space, and then additional investigations in more concentrated areas of the design space.

As a rule of thumb, the number of examples has to be larger than the number of parameters in the model (i fpossible much larger). Otherwise the model will easily fit the examples but produce wildly-oscillating "nonsense" results in regions of the design space far from the chosen examples. As an overall advice, randomization will avoid the most serious faults inherent in advanced DOE methods based on specific assumptions. The popular design of experiment

techniques are summarized in the following sections.

36.1.1 Full factorial design

In spite of the sophisticated name, this is the most obvious (and naive) way to proceed. It is based on determining the number of levels (i.e. different values) for each variables and then generating sample points on a **regular grid**. For two levels the points are placed at the left and right boundary of the design space variable. For three levels, an additional point is added in the middle, and so on.

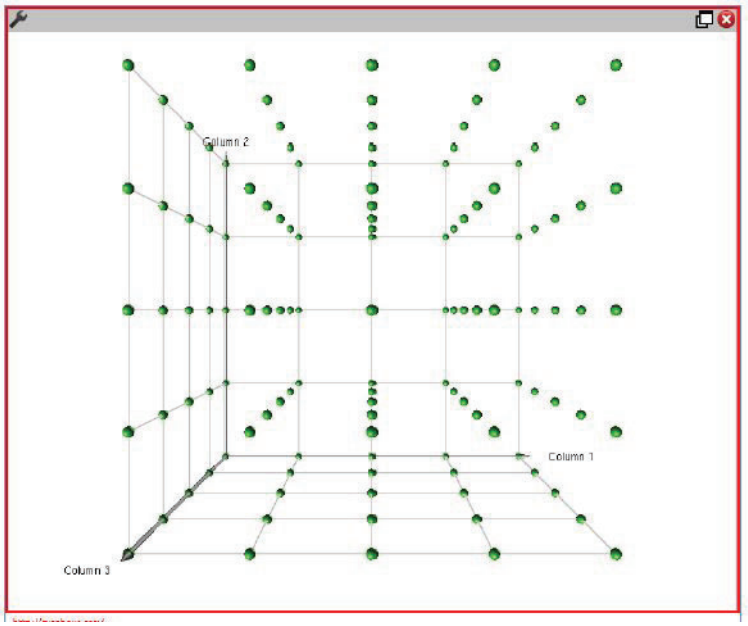


Figure 36.2: A full factorial design.

The figure shows a full-factorial DOE with 5 levels for each factor, in three dimensions. The advantage of the full factorial design is its simplicity; its disadvantage lies in its regular grid-like nature. The generated samples may miss some relevant effect between the sample points. The number of points generated, if the number of levels L is the same for all variables, is equal to L^n, a number which will explode very rapidly to make this method applicable only to problems with a small number of input variables. Full factorial designs are very expensive: the number of samples grows exponentially when the input dimension grows. It is not surprising that reduced factorial designs using only a subset of the full factorial design points are considered.

36.1.2 Randomized design: pseudo Montecarlo sampling

A full factorial design ie very expensive. In spite of its cost it can be fragile. If the interesting areas of parameters lie between the regularly-spaced points, they will never be identified. A more robust design, which can generate more points if more time is available, is based on a simple randomization. A randomized design will a uniform probability generates points in every area with a probability different from zero.

Pseudo-random number generators can be used to generate random numbers uniformly distributed in an interval, for example [0,1]. By repeated calls of a good-quality pseudo-random number generator, one can create random vectors uniformly distributed in the design space. $X = \{rand(), rand(), rand(), \dots\}$

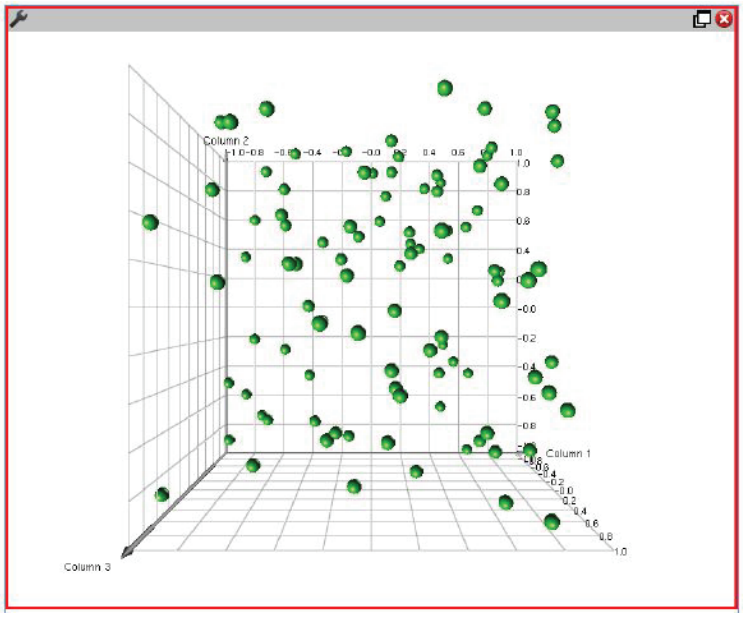


Figure 36.3: A randomized design.

Fig. 36.3 shows a pseudo-Monte Carlo DOE with 100 points in three dimensions. The distribution of points along the three different factors is shown in the parallel coordinates plot in Fig. 36.4 (the parallel coordinate plot is explained in Section 18.3). Two rather large gaps in the factor corresponding to the first coordinate are highlighted with an arrow. By chance, Some large areas can remain empty of sample points.

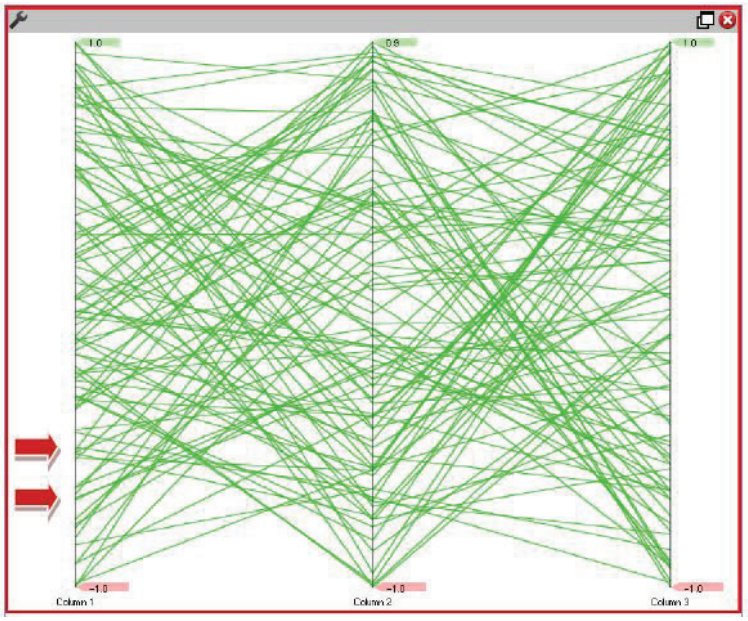


Figure 36.4: Parallel coordinates plot: a randomized design may leave some holes.

By suitable transformations, one can easily generate random samples for simple design spaces, like circles, circular regions, triangles, etc. In the absence of detailed and trusted information about the physical system under analysis, it is difficult to beat the simplicity and the robustness of the pseudo-random design. The more advanced techniques must be used with care and with more competence, and it is not guaranteed that they will add a lot of information with respect to what can be gained by simple randomization. Another advantage of a randomized design is that it works for any number of samples, and that it can be immediately repeated (by using a different random number seed and therefore a different sequence of random numbers) to check for the robustness or fragility of the first DOE. A weakness of the basic pseudo-random design is that, because of the random and independent nature of the samples, it will often leave large regions of the design space unexplored. The **stratified Monte Carlo sampling** was developed to cure the above problem, provided that one can afford the required number of samples. The idea is to divide each interval into subintervals (or "bins") of equal probability. Once the bins are defined, a sample position is then randomly selected within each bin. This trivially guarantees that at least a sample will be present in each interval. The drawback is that, if the minimum of two intervals is generated for each input dimension, the number of samples grows at best like 2^n.

36.1.3 Latin hypercube sampling

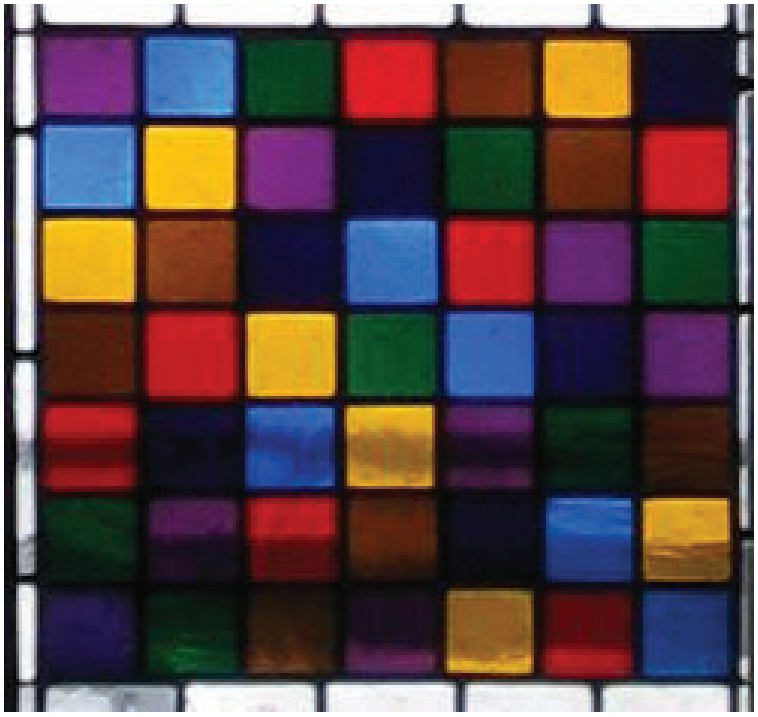


Figure 36.5: Stained glass window in the dining hall of Caius College, in Cambridge, commemorating Ronald Fisher and representing a Latin square.

Latin Hypercube sampling (LHS)[266, 204] is a modern and popular randomized DOE method than can work with any user-selected number of samples k. Under certain assumptions, LHS provides a more accurate estimate of the mean value of the Y function than the standard Monte Carlo sampling. Its motivation is to ensure that, for a given number of samples k, and k equally-spaced bins, for all one-dimensional projections of the samples there will be one and only one sample in each bin. A square grid containing sample positions is a Latin square if (and only if) there is only one sample in each row and each column. A Latin hypercube is the generalisation of this concept to an arbitrary number of dimensions, whereby each sample is the only one in each axis-aligned hyperplane containing it.

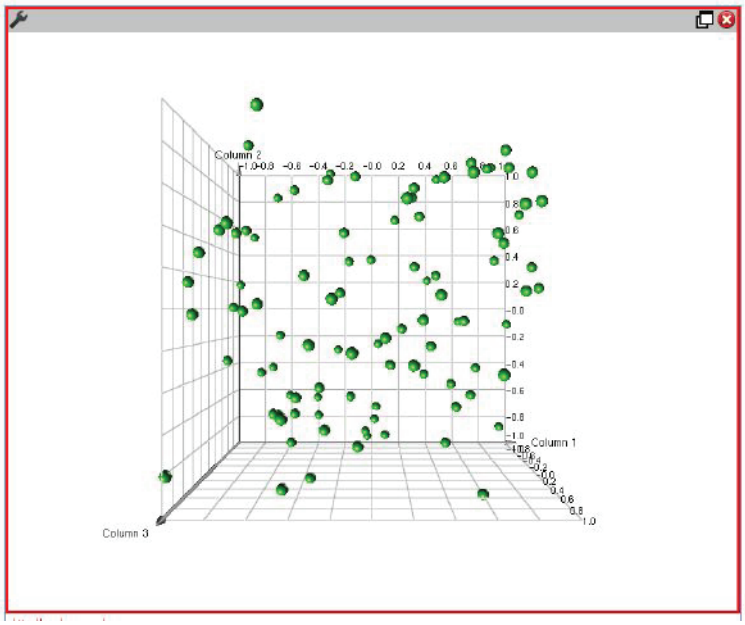


Figure 36.6: A Latin hypercube sampling.

Fig. 36.6 shows a LHS DOE with 100 samples, in three dimensions. The parallel coordinates visualization in Fig. 36.7 shows the uniform distribution of points along each factor: each one of the 100 bins is covered by one and only one point.

The method to generate LHS samples is the following. In one dimension x_1, partition the range into k bins and then generate one sample per bin with uniform probability. In two dimensions, x_1 and x_2, after generating the x_1 values, pick the bin in the x_2 direction according to a permutation of the x_1 bin index. The permutation will ensure that all vertical bins will be covered by one and only one sample. After generalizing, and picking a separate random permutation of the bins for each input variable, one obtains the complete LHS design. In detail, in the $[0,1]^n$ hypercube, if p is the number of samples and $x_j^{(i)}$ is the j-th component of the i-th sample, for $1 \leq j \leq n$ and $1 \leq i \leq k$, and $\pi_j^{(i)}$ is the permutation for variable j, permuting integers $0, 1, .., k-1$, evaluated at i, the coordinates are obtained as:

$$x_j^{(i)} = (x_j^{(i)} + U_j^{(i)})/k$$

where U is a uniform random value on $[0,1]$.

36.2 Surrogate model-based optimization

A **surrogate model** is an engineering method used when an outcome of interest cannot be easily directly measured, so a model of the outcome is used instead. Most engineering design problems require experiments and/or simulations to evaluate design objective and constraint functions as function of design variables. For example, in order to find the optimal airfoil shape for an aircraft wing, an engineer simulates the air flow around the wing for different shape variables (length, curvature, material, ..). For many real-world problems, however, a single simulation can take many minutes, hours, or even days to complete. As a result, routine tasks such as design optimization, design space exploration, sensitivity analysis and what-if analysis become impossible since they require thousands or even millions of simulation evaluations. To alleviate this burden on can use approximated models, known as **surrogate models, response surface models, meta-models or emulators**, that mimic the behavior of the system as closely as possible

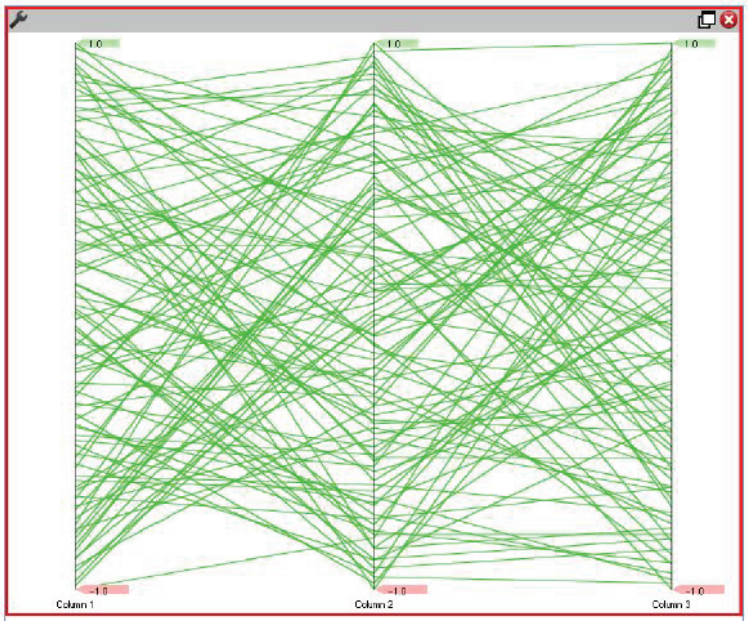


Figure 36.7: A Latin hypercube sampling: The parallel coordinates visualization shows the uniform distribution of points along each factor.

while being computationally cheaper to evaluate. Surrogate models are constructed by using a data-driven, bottom-up approach. The exact, inner working of the simulation code is not assumed to be known (or even understood), solely the input-output behavior is available (the system is considered as a **black box**). A surrogate model is constructed based on measuring the response of the simulator to a limited number of intelligently chosen data points obtained by a careful design of experiments. The surrogate model can be called "approximation model" "DACE model" (DACE stays for design and analysis of computer experiments), or "response surface approximation", leading to the term of **Response Surface Methodology (RSM)** in the optimization o fsystems. The main goal of response surface methodology is to create a predictive model of the relationship between the inputs and the outputs and then using the model to determine optimal operating settings for the system.

The surrogate models need to be determined ("trained") by considering a set of example pairs (X_e, Y_e), giving input and the corresponding output for a set of example values, used to fix the model parameters.

Let's consider a single input and output parameter: if the models of the relationship between input and output is given by a polynomial function with M parameters $a_0, a_1, \ldots, a_{(M-1)}$ then $Y(X) = a_0 + a_1 X + ... + a_{(M-1)} X^{(M-1)}$. The model parameters can be fixed by requiring that some measure of the average difference, on the examples, between the Y values of the polynomial and the corresponding Y_e values of the examples, is minimized. This amounts to asking that the model approximately reproduces the desired input-output relationships on the examples. A widely used and statistically motivated way can be to find the values of the parameters $a_0, a_1, \ldots, a_{(M-1)}$ which minimizes the average quadratic error on the examples as described in Section 5.1-5.2 (*least-squares fit* and *maximum-likelihood* estimation).

An alternative when the functional form is not known is to consider non-parametric models in machine learning.

Because the model is approximated, in Response Surface Methodology (RSM) for optimization, care must be taken to ensure that the optimal point determined by using the response surface indeed corresponds to the optimal point of the real system. In practice, this requires an iterative process, where a first model is built and used to determine a first candidate optimal X value, and then a second more localized model is built in the neighborhood of this first candidate optimizer for additional checking and refinement.

An example of a model (in this particular case Bayesian Locally-Weighted Regression (Section 9.2.1) applied to

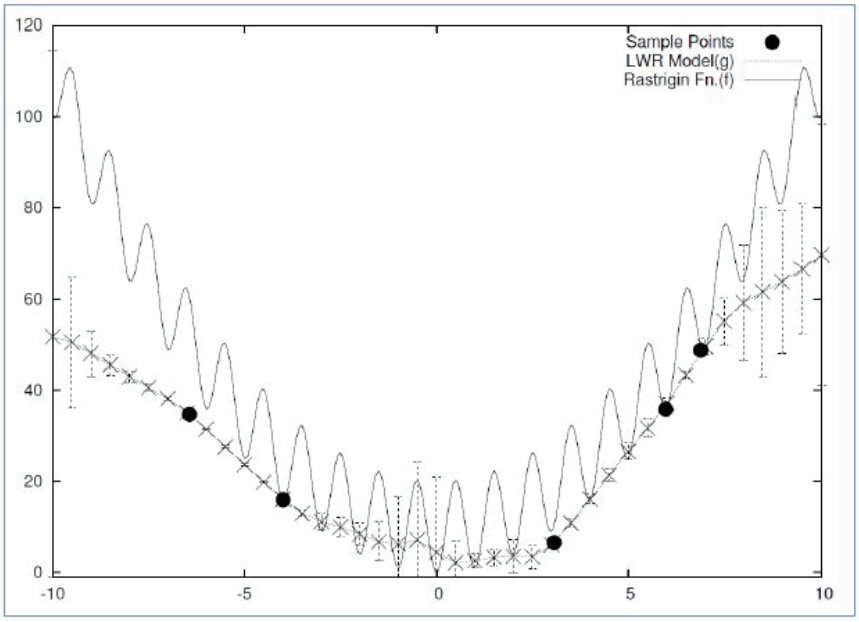


Figure 36.8: A response surface obtained by fitting the generated experimental points (black circles). The oscillating curve shows the original Rastrigin benchmark function. The surrogate model (crosses) in this case filters out the rapid oscillations and extracts the large-scale behavior of the system.

the Rastrigin benchmark function) is shown in Fig. 36.8. The sample points are used to build the model which in this case "irons out" the small oscillations and is therefore very useful to identify the area of x values corresponding to the optimal configurations.

36.3 Active or query learning

Acquiring training examples for supervised learning tasks is typically an expensive and time-consuming process. **Active learning** approaches attempt to reduce this cost by actively suggesting inputs for which supervision should be collected, in an iterative process **alternating learning and feedback elicitation**. At each iteration, active learning methods select the inputs maximizing an informativeness measure, which estimates how much an input would improve the current model if it was added to the current training set. The **informativeness of the query** inputs can be defined in different ways, and several active learning techniques exist [330].

The **uncertainty sampling (US)** principle [330] considers the input with highest predictive uncertainty as the most informative training example for the current model. This requires a learning model that can quantify its predictive uncertainty. The ability of **Gaussian Process Regression GPR** in estimating the confidence for individual predictions enables a suitable application of the uncertainty sampling principle. A Gaussian process is a statistical distribution X_t, $t \in T$, for which any finite linear combination of samples has a joint Gaussian distribution. This property permits the analytic calculation of relevant quantities, like the uncertainty in the prediction. A large variance σ^2 of the predictive distribution for a single test input $\mathbf{x}$ means that the test sample is not represented by the Gaussian Process (GP) model learned from the training data. The predictive variance quantifies the predictive uncertainty of the GP model. Therefore the input maximizing the predictive variance is selected by the uncertainty sampling principle. With GPR, the predictive uncertainty grows in regions away from training inputs. Active learning strategies more sophisticated than the US principle exist, but they usually demand more expensive computation.

The **query-by-committee strategy** maintains a committee of models, trained on the current set of examples and representing competing hypotheses. Each committee member votes on the output value of the candidate inputs, and the input considered most informative is the one on which they disagree most.

The **expected model change** principle considers as the most informative query the input that, when added to the training set would yield the greatest change in the current model (i.e., that has the greatest influence on the model parameters).

The **expected error reduction** criterion selects the input that is expected to yield the greatest reduction in the generalization error of the current model. Computing the expected generalization error is, however, computationally expensive, and, in general, it cannot be expressed analytically.

To overcome this limitation, the **variance reduction** approach queries the input that minimizes the model variance. The generalization error can in fact be decomposed into three components, refereed to as the noise, the bias, and the model variance. The noise component defines the variance of the output distribution given the input and is independent of the model and the training set. The bias error is introduced by the model class (e.g., a linear model class adopted to approximate a nonlinear function). The model variance estimates how much the model predictions vary when changing the training set. Because the model parameters can influence neither the noise nor the bias, the only way to reduce the future generalization error consists of minimizing the model variance. An effective application of the variance reduction approach is possible only under the assumption that the bias error is negligible.

An application of active learning principles in the area of multi-objective optimization, for the active learning of Pareto fronts, is presented in [74].

Gist

Contrary to some popular stories, the **experimental activity combines abundant creativity with a strategic focus on a goal to be reached**. If the focus is to model a system, often to identify improving configurations via automated optimization, sample input points need to be chosen carefully, depending on the goal and without wasting computational or real resources.

Samples generated by DOE can be used as *starting points* for explorations by optimization techniques based on local search.

Picking the appropriate design of experiments is not trivial and has consider carefully the type of model and the problem. If the form of the model is known and the model is parametric (e.g., the model is a polynomial of degree three), the experimental points can be generated to reduce the uncertainty in the estimated parameters, usually at the boundary of the admissible region. If the form is not known, like in the "non-parametric" modeling characterizing machine learning, a **randomized design** can be the most robust choice.

Chapter 37

Measuring problem difficulty in local search

A pessimist sees the difficulty in every opportunity; an optimist sees the opportunity in every difficulty.
(Winston Churchill)

37.1 Measuring and modeling problem difficulty

When using machine learning strategies in the area of heuristics, finding appropriate features to measure and appropriate metrics is a precious guide for the design of effective methods and for explaining and understanding.

Some challenging questions for the design of heuristics are:

- How can one predict *the future evolution of an optimization algorithm*? E.g., the running time to completion, the probability of finding a solution within given time bounds, etc.

- What is is *the most effective heuristic* for a given problem, or for a specific instance?
- What are the *intrinsically more difficult problems or instances* for a given search technique?

In this chapter we mention some interesting research issues related to measuring problem difficulty, measuring individual algorithm components, and selecting them through a diversification and bias metric.

Let us consider the issue of *understanding* why a problem is more difficult to solve for a stochastic local search method. One aims at discovering relationships between problem characteristics and problem difficulty. Because the focus is on local search methods, one would like to characterize statistical properties of the solution *landscape* causing poor or slow results by local search.

The effectiveness of a stochastic local search method is determined by how *microscopic local decisions* made at each search step interact to determine the *macroscopic global behavior* of the system. In particular, one studies how the function value f depends on the input configuration and changes after small and local modifications. **Statistical mechanics** has been very successful in the past at relating local and global behaviors of systems [184], for example starting from the molecule-molecule interaction to derive macroscopic quantities like pressure and temperature. Statistical mechanics builds upon statistics, by identifying appropriate statistical *ensembles* (configurations with their probabilities of occurrence) and deriving typical global behaviors of the ensemble members. When the number of system components is very large, the variance in the behavior is very small: most members of the ensemble will behave in a similar way. As an example in Physics, if one has two communicating containers of one liter and a gas with five flying molecules, the probability to find all molecules in one container is not negligible. On the other hand, if the containers are filled with air at normal pressure, the probability to observe more than 51% of the molecules in one container is very close to zero : even if the individual motion is very complex, the macroscopic behavior will produce a 50% subdivision with a very small and hardly measurable random deviation.

Unfortunately, the situation for combinatorial search problems is much more complicated than the situations for physics-related problems, so that the precision of theoretical results is more limited. Nonetheless, a growing body of literature exists, which sheds light onto different aspects of combinatorial problems and permits *a level of understanding and explanation which goes beyond the simple empirical models* derived from massive experimentation.

37.2 Phase transitions in combinatorial problems

Models inspired by statistical mechanics have been proposed for some well known combinatorial problems. An extensive review of models applied to constraint satisfaction problems, in particular the graph coloring problem, is present in [184]. The SAT problem, in particular the 3-SAT, has been the playground for many investigations [94, 104, 292, 334].

Phase-transitions have been identified as a mechanism to study and explain problem difficulty. A phase transition in a physical system is characterized by the abrupt change of its macroscopic properties at certain values of the defining parameters. For example, consider the transitions from ice to water to steam at specific values of temperature and pressure. Phenomena analogous to phase transitions have been studied for random graphs [125, 60]: as a function of the average node degree, some macroscopic property like connectivity change in a very rapid manner. The work in [198] predicts that large-scale artificial intelligence systems and cognitive models will undergo sudden phase transitions from disjointed parts into coherent structures as their topological connectivity increases beyond a critical value. Citing from the paper: “this phenomenon, analogous to phase transitions in nature, provides a new paradigm with which to analyze the behavior of large-scale computation and determine its generic features.”

Phase transitions in Constraint Satisfaction and SAT problems have been widely analyzed [82, 272, 326, 235, 337, 292]. A clear introduction to phase transitions and the search problem is present in [185]. A surprising result is that hard problem instances are concentrated near the same parameter values for a wide variety of common search heuristics. This location also corresponds to a **transition between solvable and unsolvable instances**. For example, when the control parameter that is changed is the number of clauses in SAT instances, different schemes like complete backtracking and local search show very long computing times in the same transition region.

For backtracking, this is due to a competition between two factors: number of solutions and facility of pruning many subtrees. A small number of clauses (*under-constrained* problem) implies many solutions, it is easy to find one of them. At the other extreme, a large number of clauses (*over-constrained* problem) implies that any tentative solution is quickly ruled out (pruned from the tree). It is fast to rule out all possibilities and conclude that there is no solution. The *critically constrained* instances in between are the hardest ones.

The local search method is not complete and one must limit the experimentation to solvable instances. One may naïvely expect that the search becomes harder with a smaller *number of solutions* but the situation is not so simple. At the limit, if only one solution is available but the attraction basin is very large, local search will easily find it. Not only the number of solutions but also the number and depth of *sub-optimal local minima* play a role. A large number of deep local minima is causing a waste of search time in a similar way to tentative solutions in backtracking, which fail only after descending very deeply in the search tree. Among a growing body of experimental research, [94] presents results on CSP and SAT, [104] results on the crossover point in random 3-SAT.

In addition to being of high scientific interest, identifying zones where the most difficult problems are is relevant for **generating difficult instances to challenge algorithms**. As strange as it may sound at the beginning, it is not so easy to identify difficult instances of NP-hard problems [326]. Computational complexity classes are defined through a *worst-case* analysis: in practice the worst cases may be very difficult to encounter or to generate.

37.3 Empirical models of fitness surfaces

More empirical *descriptive cost models* of problem difficulty aim at identifying measurable *instance characteristics (features) influencing the search cost*. A good descriptive model should account for a significant portion of the variance in search cost [94, 292, 334, 377].

The performance of search algorithms depends on the features of the search space. In particular, a useful measure of variability of the search landscape is given by the correlation between the values of the objective function f over all pairs of configurations at distance d. The proper distance to be used depends on the nature of the solving technique. In local search the distance between two configurations X and X' can be measured measured as the minimum number of local steps to transform one into the other. After choosing the distance function, one defines the **Landscape Correlation Function** [378]:

$$R(d) = \frac{\mathrm{E}_{\mathrm{dist}(X,X')=d}[(f(X)-\mu)(f(X')-\mu)]}{\sigma^2}, \tag{37.1}$$

where $\mu = \mathrm{E}[X]$ and $\sigma^2 = \mathrm{Var}[X]$. This measure captures the idea of *ruggedness* of a surface: a low correlation function implies high statistical independence between points at distance d. While it is expectable that for large values of d the correlation $R(d)$ goes to zero (unless the search landscape is very smooth), the value of $R(1)$ can be meaningful.

Intuitively, $R(1)$ tells us whether a local move from the current configuration changes the f value in a manner which is significantly different from a random restart. $R(1) \approx 0$ means that, on average, there is little correlation between the objective value at a given configuration and the value of its neighbors. This can signal of a poor choice of the neighborhood structure, or that the problem is particularly hard for local search. On the other hand, $R(1) \approx 1$ is a clear indication that the neighborhood structure produces a *smooth* fitness surface.

Computing equation (37.1) for large search spaces can be difficult. A common estimation technique uses random walks. In particular, Big-Valley models [59] (a.k.a. *massif central* models) have been considered to explain the success of local search, and the preference for continuing from a given local optimum instead of restarting from scratch. These models measure the **autocorrelation of the time-series of f values produced by a random walk**. The autocorrelation function (ACF) of a random process describes the correlation between the process at different points in time. Let $X^{(t)}$ be the search configuration at time t. If $f(X^{(t)})$ has mean μ and variance σ^2 then the ACF can be defined as

$$R'(d) = \frac{E_t[(f(X^{(t)})-\mu)(f(X^{(t+d)})-\mu)]}{\sigma^2}. \tag{37.2}$$

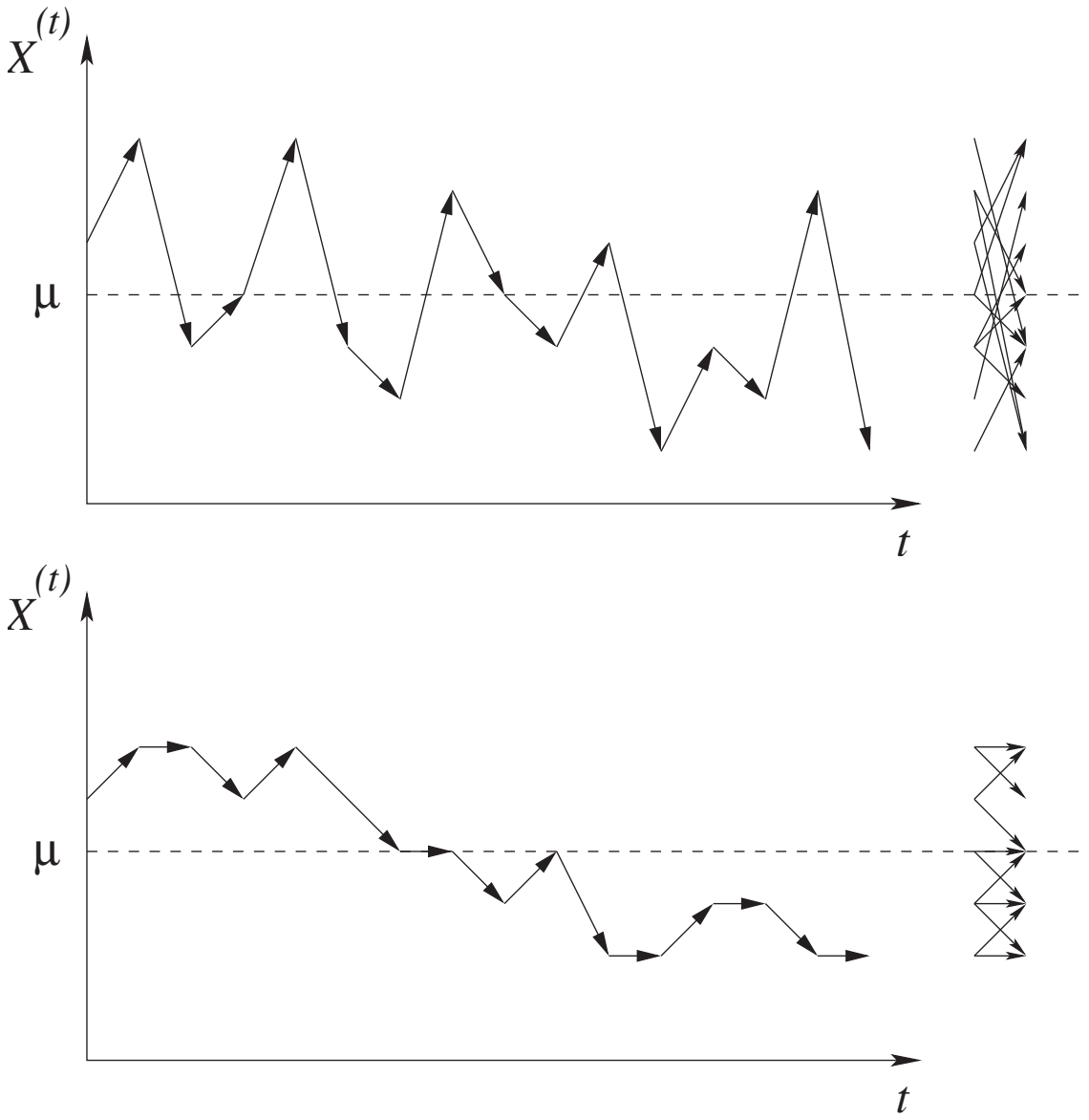


Figure 37.1: Estimating autocorrelation on a rugged (top) and a smooth (bottom) landscape by means of a random walk. Vectors between fitness values at subsequent steps are shown on the right; in particular, $R'(1)$ is determined as the correlation between the endpoints of these arrows with respect to the mean value (dashed line).

Fig. 37.1 provides a pictorial view of autocorrelation estimation by means of a random walk. In particular, the right side of each plot shows how subsequent moves are correlated: every arrow shows one step, so that correlation is computed between the head and the tail of each arrow with respect to the mean value (dashed line). It is apparent that the bottom walk represents a smoother landscape, and this translates to correlated arrow endpoints.

As noted above, it is expected that both $R(d)$ and $R'(d)$ become smaller and smaller as long as d increases: points separated by a small path are more correlated than separated ones. Empirical measurements show that $R'(d)$ often

follows an exponential decay law:

$$R'(d) \approx e^{-\frac{d}{\lambda}}. \tag{37.3}$$

The value of λ that best approximates the $R'(\cdot)$ sequence is called the **correlation length**, and it measures the range over which fluctuations of f in one region of space are correlated. An approximation of λ can be obtained by solving (37.3) when $d = 1$:

$$\lambda \approx -\frac{1}{\ln R'(1)}.$$

Clearly, we are assuming that correlation between nearby configurations is positive: otherwise, no significant approximation can be obtained and the whole correlation analysis loses its meaning.

Equation (37.3) defines the correlation length as the distance where autocorrelation decreases by a factor of e, so that the definition is somewhat arbitrary. Moreover, for many problems the correlation length is a function of the problem's size and it does not explain the variance in computational cost among instances of the same size [377]. However, it is possible to normalize it with respect to some measure of the problem's instance (e.g., number of dimensions, size of the neighborhood) in order to make it a useful tool for comparisons.

An example of fitness landscape analysis for the Quadratic Assignment Problem is presented in [267], where autocorrelation analysis and fitness-distance correlation analysis [223] are adopted for choosing suitable evolutionary operators.

A **fitness distance correlation** (FDC) c an be calculated by measuring the Hamming distances between sets of bit-strings and the global optimum, and comparing with their fitness. Of course, the technique can be generalized to deal with different distance measures. An instructive counterexample is presented in [9].

37.4 Tunable landscapes

The *NK landscape model* [230], proposed in the field of computational biology, provides a parametric and tunable landscape for the generation of problem instances. *NK* landscapes capture the intuition that every coordinate in the search space contributes to the overall fitness in a complex way, often by enabling or disabling the contribution of other variables in a process that is known in biology as *epistasis*.

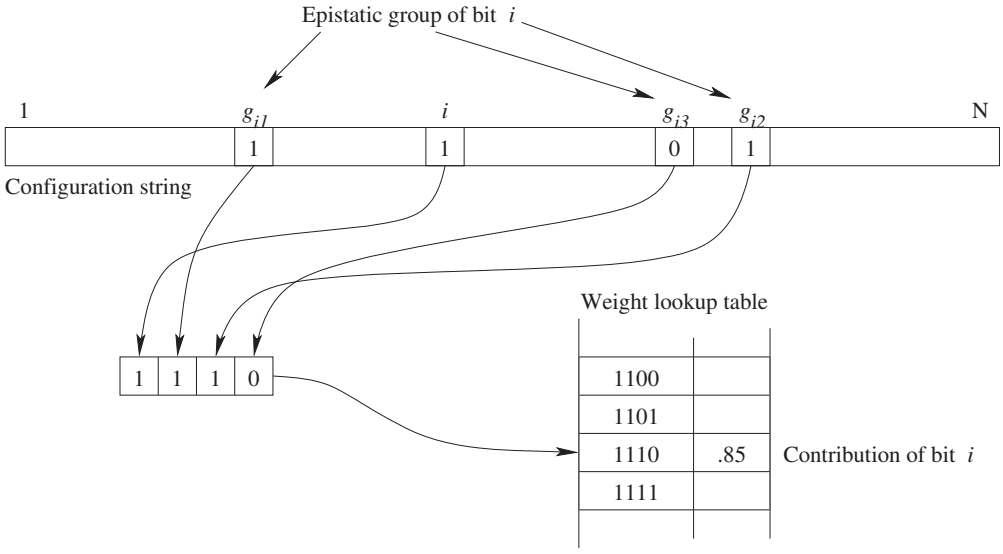


Figure 37.2: Epistatic contribution of bit i in an NK model for $K = 3$.

An NK system is defined by N binary components (e.g., bits in the configuration string) and by the number K of other components that interact with each component. In other words, each bit identifies a $(K+1)$-bit interaction group. A uniform random-distributed mapping from $\{0,1\}^{K+1}$ to $[0,1]$ determines the contribution of each interaction group to the total fitness, which is computed as the average of all contributions.

In mathematical terms, let the configuration space be the N-bit strings $\{0,1\}^N$. The current configuration is $S = s_1 \dots s_N$, $s_i \in \{0,1\}$. For every position $i = 1, \dots, N$, let us define its interaction group as a set of $K+1$ different indices $G_i = (g_{i0}, g_{i1}, \dots, g_{iK})$ so that $g_{i0} = i$, $g_i j \in \{1, \dots, N\}$ and if $l \neq m$ then $g_{il} \neq g_{im}$. Let $w : \{0,1\}^{K+1} \to [0,1]$, defined by random uniform distribution, represent the contribution of each interaction group. Then

$$f(S) = \frac{1}{N} \sum_{i=1}^{N} w(s_{g_{i0}} s_{g_{i1}} \dots s_{g_{iK}}).$$

Fig. 37.2 shows how every bit contributes to the fitness value by means of its epistatic interaction group as defined before. For small values of K the contribution function w can be encoded as a (2^{K+1})-entry lookup table.

The parameter K controls the so-called "ruggedness" of the landscape: $K = 0$ means that no bitwise interaction is present, so that every bit independently contributes to the overall fitness, leading to an embarrassingly simple optimization task. On the other hand, if $K = N-1$, then changing a single bit modifies all contributions to the overall fitness value: the ruggedness of the surface is maximized.

NK systems have been used as problem generators in the study of various combinatorial optimization algorithms [224, 175, 338].

37.5 Measuring local search components: diversification and bias

To ensure progress in algorithmic research it is not sufficient to have a horse-race of different algorithms on a set of instances and declare winners and losers. Actually, very little information can be obtained by these kinds of comparisons. In fact, if the number of instances for the benchmark is limited and if sufficient time is given to an intelligent researcher (...and very motivated to get publication!) be sure that some promising results will be eventually obtained, via a careful tuning of algorithm parameters.

A better method is to design a *generator of random instances* so that it can produce instances used during the development and tuning phase, while a different set of instances extracted from the same generator is used for the final test. This method mitigates the effect of "intelligent tuning done by the researcher on a finite set of instances," but still it does not explain *why* a method is better than another one. Explaining *why* is related to the **generality and prediction power** of the model. If one can predict the performance of a technique on a problem (or on a single instance) – of course before the run is finished, predicting the past is always easy! – then he takes some steps towards understanding.

This exercise takes different forms depending on what one is predicting, what are the starting data, what is the computational effort spent on the prediction, etc. The work in [33] dedicated to solving the MAX-SAT problem with non-oblivious local search aims at **relating the final performance to measures obtained after short runs of a method**. In particular, the average f value (*bias*) and the average speed in Hamming distance from a starting configuration (*diversification*) is monitored and related to the final algorithm performance.

Let us focus onto local-search based heuristics: it is well known that the basic compromise to be reached is that between **diversification and bias**. Given the obvious fact that only a negligible fraction of the admissible points can be visited for a non-trivial task, the search trajectory $X^{(t)}$ should be generated to visit preferentially points with large f values (*bias*) and to avoid the confinement of the search in a limited and localized portion of the search space (*diversification*). The two requirements are conflicting: as an extreme example, random search is optimal for diversification but not for bias. Diversification can be associated with different metrics. Here we adopt the *Hamming distance* as a measure of the distance between points along the search trajectory. The Hamming distance $H(X,Y)$ between two binary strings X and Y is given by the number of bits that are different.

The investigation follows this scheme:

- After selecting the metric (diversification is measured with the Hamming distance and bias with mean f values visited), the diversification of simple *random walk* is analyzed to provide a basic system against which more complex components are evaluated.
- The **diversification-bias plots (D-B plots)** of different basic components are investigated and a conjecture is formulated that the best components for a given problem are the *maximal elements* in the diversification-bias (D-B) plane for a suitable partial ordering (Section 37.5.1).
- The conjecture is validated by a competitive analysis of the components on a benchmark.

Let us now consider the **diversification properties of Random Walk**. Random Walk generates a Markov chain by selecting at each iteration a random move, with uniform probability:

$$X^{(t+1)} = \mu_{r(t)} X^{(t)} \text{ where } r(t) = \text{Rand}\{1, \ldots, n\}$$

Without loss of generality, let us assume that the search starts from the zero string: $X^{(0)} = (0, 0, ..., 0)$. In this case the Hamming distance at iteration t is:

$$H(X^{(t)}, X^{(0)}) = \sum_{i=1}^{n} x_i^{(t)}$$

and therefore the expected value of the Hamming distance at time t, defined as $\widehat{H}^{(t)} = \widehat{H}(X^{(t)}, X^{(0)})$, is:

$$\widehat{H}^{(t)} = \sum_{i=1}^{n} \widehat{x_i}^{(t)} = n\, \widehat{x}^{(t)} \tag{37.4}$$

The equation for $\widehat{x}^{(t)}$, the probability that a bit is equal to 1 at iteration t, is derived by considering the two possible events that i) the bit remains equal to 1 and ii) the bit is set to 1. In detail, after defining as $p = 1/n$ the probability that a given bit is changed at iteration t, one obtains:

$$\widehat{x}^{(t+1)} = \widehat{x}^{(t)}\,(1-p) \;+\; (1-\widehat{x}^{(t)})\,p = \widehat{x}^{(t)} \;+\; p\,(1-2\widehat{x}^{(t)}) \tag{37.5}$$

It is straightforward to derive the following theorem:

Theorem 4 *If $n > 2$ (and therefore $0 < p < \frac{1}{2}$) the difference equation (37.5) for the evolution of the probability $\widehat{x}^{(t)}$ that a bit is equal to one at iteration t, with initial value $\widehat{x}^{(0)} = 0$, is solved for t integer, $t \geq 0$ by:*

$$\widehat{x}^{(t)} \;=\; \frac{1-(1-2p)^t}{2} \tag{37.6}$$

The qualitative behavior of the average Hamming distance can be derived from the above. At the beginning $\widehat{H}^{(t)}$ has a linear growth in time:

$$\widehat{H}^{(t)} \approx t \tag{37.7}$$

For large t the expected Hamming distance $\widehat{H}^{(t)}$ tends to its asymptotic value of $n/2$ in an exponential way, with a "time constant" $\tau = n/2$

Let us now compare the evolution of the mean Hamming distance for different algorithms. The analysis is started as soon as the first local optimum is encountered by LS, when diversification becomes crucial. LS$^+$ has the same evolution as LS with the only difference that it *always* moves to the best neighbor, even if the neighbor has a worse solution value f. LS$^+$, and Fixed-TS with fractional prohibition T_f equal to 0.1, denoted as TS(0.1), are then run for $10\ n$ additional iterations. Fig. 37.3 shows the average Hamming distance as a function of the additional iterations after reaching the LS optimum, see [33] for experimental details.

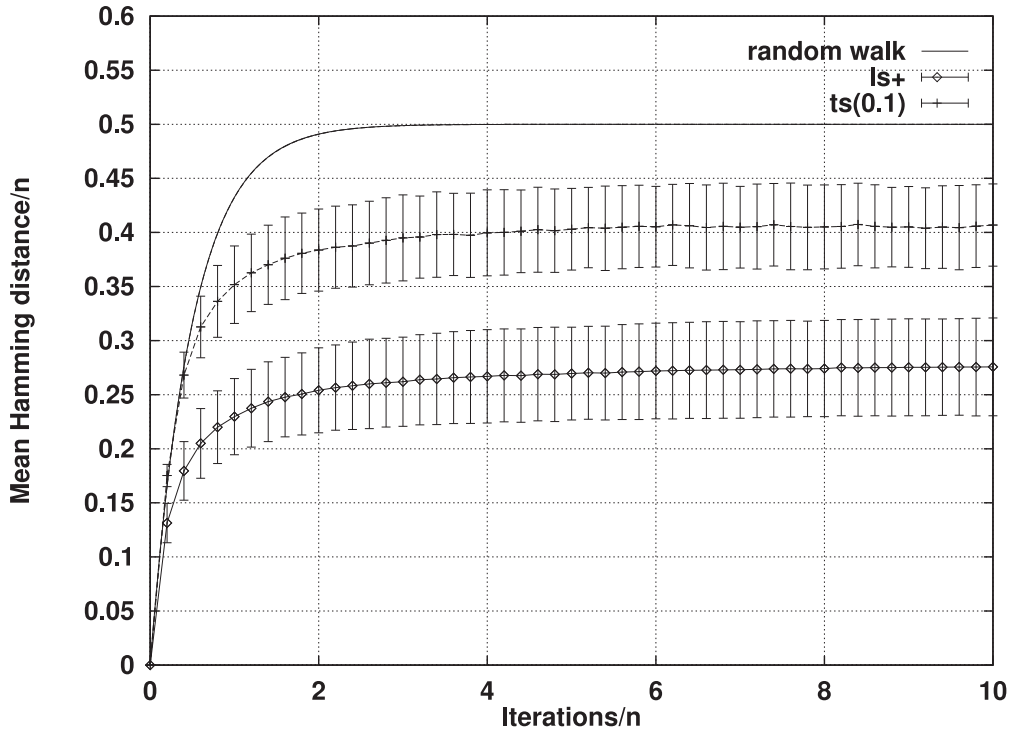


Figure 37.3: Average Hamming distance reached by Random Walk, LS$^+$ and TS(0.1) from the first local optimum of LS, with standard deviation (MAX-3-SAT). Random walk evolution is also reported for reference.

Although the initial linear growth is similar to that of Random Walk, the Hamming distance does not reach the asymptotic value $n/2$ and a remarkable difference is present for the two algorithms. The fact that the asymptotic value is not reached even for large iteration numbers implies that all visited strings tend to lie in a confined region of the search space, with bounded Hamming distance from the starting point.

Let's note that, for large n values, most binary strings are at distance of approximately $n/2$ from a given string. In detail, the Hamming distances are distributed with a binomial distribution with the same probability of success and failure ($p = q = 1/2$): the fraction of strings at distance H is equal to

$$\left(\begin{array}{c} n \\ H \end{array} \right) \times \frac{1}{2^n} \tag{37.8}$$

It is well known that the mean is $n/2$ and the standard deviation is $\sigma = \sqrt{n}/2$. The above coefficients increase up to the mean $n/2$ and then decrease. Because the ratio σ/n tends to zero for n tending to infinity, for large n values most strings are clustered in a narrow peak at Hamming distance $H = n/2$. As an example, one can use the Chernoff bound [168]:

$$Pr[H \leq (1-\theta)pn] \leq e^{-\theta^2 np/2} \tag{37.9}$$

the probability to find a point at a distance less than $np = n/2$ decreases in the above exponential way ($\theta \geq 0$). The distribution of Hamming distances for $n = 500$ is shown in Fig. 37.4.

Clearly, if better local optima are located in a cluster that is not reached by the trajectory, they will never be found. In other words, a robust algorithm demands that some stronger diversification action is executed. For example, an option is to activate a restart after a number of iterations that is a small multiple of the time constant $n/2$.

When a local search component is started, new configurations are obtained at each iteration until the first local optimum is encountered, because the number of satisfied clauses increases by at least one. During this phase additional

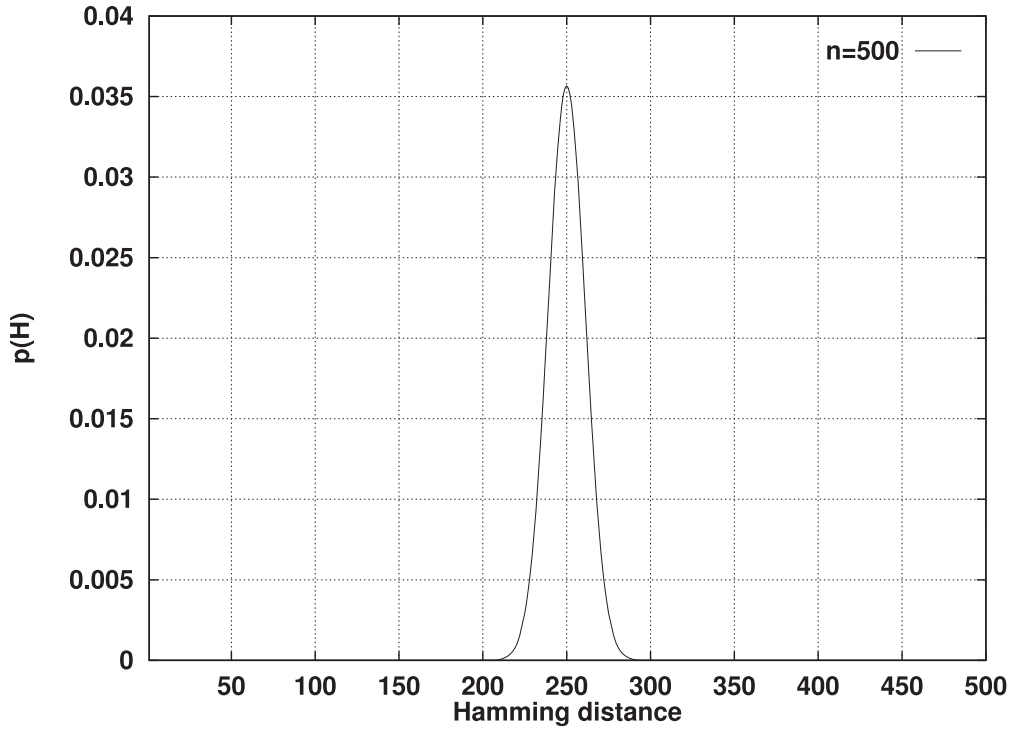


Figure 37.4: Probability of different Hamming distances for n = 500.

diversification schemes are not necessary and potentially dangerous, because they could lead the trajectory astray, away from the local optimum.

The compromise between bias and diversification becomes critical after the first local optimum is encountered. In fact, if the local optimum is strict, the application of a move will worsen the f value, and an additional move could be selected to bring the trajectory back to the starting local optimum.

The mean bias and diversification depend on the value of the internal parameters of the different components. All runs proceed as follows: as soon as the first local optimum is encountered by LS, it is stored and the selected component is then run for additional $4n$ iterations. The final Hamming distance H from the stored local optimum and the final value of the number of unsatisfied clauses u are collected. The values are then averaged over different tasks and different random number seeds.

Different diversification-bias (D-B) plots are shown in Fig. 40.3. Each point gives the D-B coordinates $(\widehat{H_n}, \widehat{u})$, i.e., average Hamming distance divided by n and average number of unsatisfied clauses, for a specific parameter setting in the different algorithms. The Hamming distance is normalized with respect to the problem dimension n, i.e., $\widehat{H_n} \equiv \widehat{H}/n$. Three basic algorithms are considered: GSAT-with-walk, Fixed-TS, and HSAT. For each of these, two options about the guiding functions are studied: one adopts the "standard" oblivious function, the other the non-oblivious f_{NOB} introduced in Section 35.5.2. Finally, for GSAT-with-walk one can change the probability parameter p, while for Fixed-TS one can change the fractional prohibition T_f: parametric curves as a function of a single parameter are therefore obtained.

GSAT, Fixed-TS(0.0), and GSAT-with-walk(0.0) coincide: no prohibitions are present in TS and no stochastic choice is present in GSAT-with-walk. The point is marked with "0.0" in Fig. 40.3. By considering the parametric curve for GSAT-with-walk(p) (label "gsat" in Fig. 40.3) one observes a gradual increase of $\widehat{u}$ for increasing p, while the mean Hamming distance reached at first decreases and then increases. The initial decrease is unexpected because it contradicts the intuitive argument that more stochasticity implies more diversification. The reason for the above result

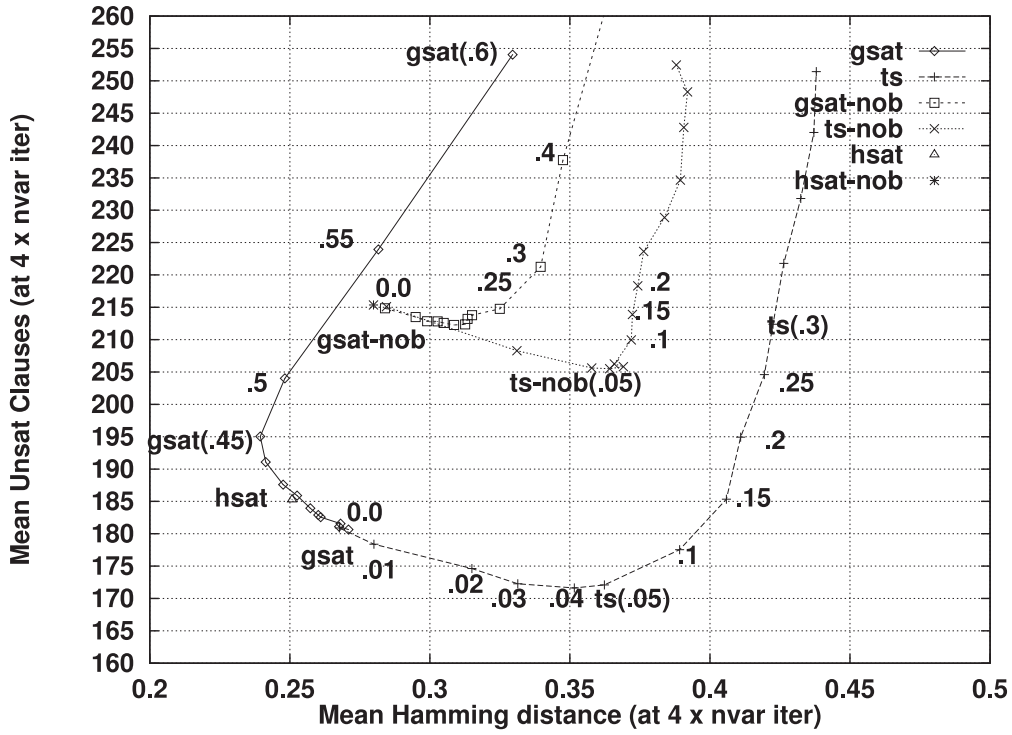


Figure 37.5: Diversification-bias plane. Mean number of unsatisfied clauses after $4\ n$ iterations versus mean Hamming distance. MAX-3-SAT tasks. Each run starts from GSAT local optimum, see [33].

is that there are two sources of "randomness" in the GSAT-with-walk algorithm (see Fig. 35.16), one deriving from the random choice among variables in unsatisfied clauses, active with probability p, the other one deriving from the random breaking of ties if more variables achieve the largest Δf.

Because the first randomness source increases with p, the decrease in $\widehat{H_n}$ could be explained if the second source decreases. This conjecture has been tested and confirmed [33]. The larger amount of stochasticity implied by a larger p keeps the trajectory on a rough terrain at higher values of f, where flat portions tend to be rare. *Vice versa*, almost no tie is present when the non-oblivious function is used. The algorithm on the optimal frontier of Fig. 40.3 is Fixed-TS(T_f), and the effect of a simple **aspiration criterion** [148], and a **tie-breaking rule** for it is studied in [33].

The advantage of the D-B plot analysis is clear: it suggests possible causes for the behavior of different algorithms, leading to a more focused investigation.

37.5.1 A conjecture: better algorithms are Pareto-optimal in D-B plots

A conjecture about the relevance of the diversification-bias metric is proposed in [33]. A relation of partial order, denoted by the symbol $\geq$ and called "domination," is introduced in a set of algorithms in the following way: given two component algorithms A and B, A *dominates* B ($A \geq B$) if and only if it has a larger or equal diversification and bias: $\widehat{f}_A \geq \widehat{f}_B$ and $\widehat{H_n}_A \geq \widehat{H_n}_B$.

By definition, component A is a *maximal element* of the given relation if the other components in the set *do not* possess both a higher diversification, and a better bias. In the graph one plots the number of *un*satisfied clauses versus the Hamming distance, therefore the *maximal* components are in the lower–right corner of the set of $(\widehat{H_n}, \widehat{u})$ points. The points are characterized by the fact that no other point has both a larger diversification and a smaller number of

satisfied clauses.

Conjecture
If local-search based components are used in heuristic algorithms for optimization, the components producing the best f values during a run, on the average, are the maximal elements *in the diversification-bias plane for the given partial order.*

The conjecture produces some "falsifiable" predictions that can be tested experimentally. In particular, a partial ordering of the different components is introduced: component A is better than component B if $\widehat{H}_{nA} \geq \widehat{H}_{nB}$ *and* $\widehat{u}_A \leq \widehat{u}_B$. The ordering is partial because no conclusions can be reached if, for example, A has better diversification but worse bias when compared with B.

Clearly, when one applies a technique for optimization, one wants to maximize the best value found during the run. This value is affected by both the *bias* and the *diversification*. The search trajectory must visit preferentially points with large f values but, as soon as one of this point is visited, the search must proceed to visit new regions. The above conjecture is tested experimentally in [33], with fully satisfactory results. We do not expect this conjecture to be always valid but it is a useful guide when designing and understanding component algorithms.

A definition of three metrics is used in [318] [342] for studying algorithms for SAT and CSP. The first two metrics *depth* (average unsatisfied clauses) and *mobility* (Hamming distance speed) correspond closely to the above used *bias* and *diversification*. The third measure (*coverage*) takes a more global view at the search progress. In fact, one may have a large mobility but nonetheless remain confined in a small portion of the search space. A two-dimensional analogy is that of bird flying at high speed along a circular trajectory: if fresh corn is not on the trajectory it will never discover it. Coverage is intended to measure *how systematically* the search explores the entire space. In other words, coverage is what one needs to ensure that eventually the optimal solution will be identified, no matter how it is camouflaged in the search landscape. The motivation for a *speed of coverage* measure is intuitively clear, but the detailed definition and implementation is somewhat challenging. In [318] a worst-case scenario is considered and *coverage* is defined as the size of the *largest unexplored gap in the search space*. For a binary string, this is given by the maximum Hamming distance between any unexplored assignment and the nearest explored assignment.

Unfortunately, measuring the speed of coverage it is not so fast, actually it can be NP-hard for problems with binary strings, and one has to resort to approximations [318]. For an example, one can consider sample points given by the negation of the visited points along the trajectory and determine the maximum minimum distance between these points and points along the search trajectory. The rationale for this heuristic choice is that the negation of a string is the farthest point from a *given* string (one tries to be on the safe side to estimate the real coverage). After this estimate is available one divides by the number of search steps. Alternatively, one could consider how fast coverage decreases during the search (a discrete approximation of the coverage speed). Dual measures on the constraints are studied in [342].

Gist

A motivated researcher may use junk heuristics and get positive results after carefully tuning their meta-parameters on single benchmark instances. Science and technology will not advance with this kind of brutal horse-racing but they require predictive and explanatory models.

In a manner similar to machine learning, to get usable flexible algorithms, tuning should be done in an automated manner on some "training instances", and performance should be tested on novel instances characterized by similar statistical properties.

Measures of problem difficulty are a first step in this scientific process. **Phase-transitions** can explain why some instances are very easy and other instances almost impossible to solve.

Empirical models of fitness surfaces can explain if local search will be effective and can help in selecting the proper local moves.

Tunable landscapes and problem generators are useful probes to validate different choices in algorithm design.

Diversification-bias plots (D-B plots) can measure the effectiveness of single building blocks and deliver insight on their balance of exploration versus exploitation.

For comparable results, the simplest algorithms (with less meta-parameters) should always be chosen because they are more robust and simpler to study and understand. Complexity in optimization heuristics should grow in stages, after careful motivations of each addition and demonstrations of the gained performance benefits. Less is more, also in heuristics.

Part VI

Cooperation and multiple objectives in optimization

Chapter 38

Cooperative Learning And Intelligent Optimization (C-LION)

Whenever any important business has to be done in the monastery,
let the Abbot call together the whole community and state the matter to be acted upon.
Then, having heard the advice of the monks, let him turn the matter over in his own mind
and do what he shall judge to be most expedient.
The reason why we said that all should be consulted is that the Lord often reveals to the younger what is best.
(The Rule of Saint Benedict, 530-550, Montecassino)

As we have seen in the previous chapters, for each problem there are often **many possible algorithms** for solving it. Their effectiveness depends on the characteristics of the specific instances. Even if the performance of a single algorithm is dominating, when the algorithm has **meta-parameters or stochastic components**, there are opportunities to improve performance or to reduce the risk of suboptimal results by considering **different runs**. The objective of

C-LION is to **increase the level of automation** by designing an additional intelligent coordination layer after starting from individual algorithmic building blocks.

When considering local search (LS), we have seen that **learning from the previous phase of the stochastic search** by modifying the probabilities of generating new sample points can lead to more efficient schemes w.r.t. simple Pure Random Search, like Markov Chain Montecarlo, a.k.a. Simulated Annealing. The local minima traps can be cured by Reactive Search Optimization (RSO). Up to now the discussion was mostly dedicated to single solution schemes. One can now generalize: instead of information from a single running algorithm, one can **use information from many solution processes** running in parallel, with *complex interaction and coordination*. In particular, one can keep a sample (population) of points (current good solutions) and modify it with a generation probability that depends on all points in the population. **Population-based algorithms** do exactly this: they transform one group of points (current **generation**) to a new group of points (next generation) by probabilistic rules. Some more robustness and diversification can be obtained. The population can be seen as the concrete representation of **a probability distribution on the input space, which is dynamically changing** to reflect the accumulated knowledge and the interesting regions for future sampling.

Cooperative Learning and Intelligent Optimization (C-LION) denotes a framework for solving problems by a strategic use of memory and *cooperation* among a team of self-adaptive solution processes. Widely popular genetics and evolution-based analogies will be covered in Chapter 39. This chapter consider mostly analogies based on the organization of human society. Our species is particularly remarkable in this capability to coordinate individual human problem solvers. Our culture and civilization have a lot to do with cooperation and coordination of intelligent human beings. Coming closer to our daily work, the history of science shows a steady advancement caused by the creative interplay of many individuals, both during their lifetimes and through the continuation of work by the future generations. We are all dwarfs standing on the shoulders of giants (*nanos gigantium humeris insidentes*, Bernard of Chartres).

C-LION is a paradigm and not a single technique, so that different names have been used for specific techniques [41], but for the sake of brevity the following discussion reflects on some long-term goals of this effort, with a more focussed example for the coordination of Local Search streams. In this context, the three pillars of *C-LION* are: multiple local searchers in charge of districts (portions of input space), mutual coordination, and continuous "reactive" learning and adaptation.

C-LION adopts a **sociological/political paradigm**. Each local searcher takes care of a *district* (an input area), generates samples and decides when to fire local search in coordination with the other members. The **organized subdivision of the configuration space** is adapted in an online manner to the characteristics of the problem instance. **Coordination by use of globally collected information** is crucial to identify promising areas in configuration space and allocate search effort. Analogies are to be used only to guide intuition, the final algorithm does not have to use the terminology of the field of the analogy [341].

38.1 Intelligent and reactive solver teams

The appetite for more effective and efficient basic solvers can be satisfied by adopting more complex higher-level techniques, built on top of basic mechanisms and embodying elements of meta-optimization and self-tuning. **Meta-optimization** is the process of optimizing parameters which define a flexible optimization algorithm (e.g., the prohibition parameter in RSO based on prohibitions in Section 27.2).

When one considers the relevant issues in designing solver teams, one encounters striking analogies with sociological and behavioral theories related to human teams, with a similar presence of apparently contradictory conclusions. Let us mention some of the basic issues from a qualitative and analogical point of view, deferring more specific algorithmic implementations in the next sections.

We tend to prefer the term **solver teams** to underline that an individual solver can be an arbitrarily intelligent agent capable of collecting information, developing models, exchanging the relevant part of the obtained information with his fellow searchers, and responding in a strategic manner to the information collected. Teamwork is the concept of people working together cooperatively, as in a sports team.

Why does it make sense to consider the team members separately from the team? After all, one could design a complex entity where the boundary between the individual team members and the coordination strategy is fuzzy, a kind of Star Trek *borg hive* depicted as an amalgam of cybernetically enhanced humanoid drones of multiple species, organized as an inter-connected collective with a hive mind and assimilating the biological and technological distinctiveness of other species to their own, in pursuit of perfection.

The answer lies in the simpler design of more advanced and effective search strategies and in the better possibility to *explain* the success of a particular team by separating its members' capabilities from the coordination strategy (*divide et impera*). Let's note that alternative points of view exist and are perfectly at home in the scientific arena: for example, one may be interested in explaining how and why a collection of very simple entities manages to solve problems not solvable by the individuals. For example how simple ants manage to transport enormous weights by joining forces. In all cases, *solver team* existence cannot motivated by its sexiness or by its correspondence to poetic biological analogies [341], but only by a demonstrated superiority w.r.t. the state of the art of the individual solvers.

Individual quality of the team members. A basic issue is the relationship between the quality of the members and the collective quality of the team. If the team members are poor performers one should not expect an exceptional team. Nonetheless, it is of scientific and cultural interest to assess the potential for problem solving through the interaction of a multitude of simple members. An inspiring book [353] deals with the *wisdom of crowds*: "why the many are smarter than the few and how collective wisdom shapes business, economies, societies, and nations." Of course, acid comments go back ages, like in Nietzsche's citation "I do not believe in the collective wisdom of individual ignorance." Adapting a conclusion from the review [306], "in matters for which true expertise can be identified, one would much rather rely on the best judgments of the most knowledgeable specialists than a crowd of laymen. But in matters for which no expertise or training is genuinely involved, in dealing with fields of study whose principles are ambiguous, contentious, and rarely testable, ...then yes, there is sense to polling a group of people." In optimization, one expects a much bigger effectiveness by starting from the most competitive single-thread techniques, but some dangers lurk depending on how the team is integrated and information is shared and used.

Diversity of the team members. If all solvers act in same manner the advantage of a team is lost (see lower panes of Fig. 39.3). Diversity can be obtained in many possible ways, for example by using different solvers, or identical solvers but with different random initializations. In some cases, the effects of an initial randomization is propagated throughout a search process that shows a sensitive dependence on initial conditions that is characteristic for chaotic processes [360].

Diversity means that, in some cases, combining simpler and inferior performers with more effective ones can increase the overall performance and/or improve robustness. By the way, diversity is also crucial for ensembles of learning machines [364, 29].

Information sharing and cooperation. When designing a *solver team* one must decide about the way in which information collected by the various solvers is shared and used to modify the individual member decisions. An extreme design principle consists of **complete independence and periodic reporting** of the best solution to a coordinator. Simplicity and robustness make this extreme solution the first to try. More complex interaction schemes should always be compared against this baseline, see for example [30] where independent parallel walks of tabu search are considered.

More complex sharing schemes involve periodic collection of the best-so-far (record) values and configurations, the current configurations, more complex summaries of the entire search history of the individual solvers, for example the list of local minima encountered.

After the information is shared, a decision process modifies the individual future search in a strategic manner. Here complexities and open research issues arise. Let's consider a simple case: if a solver is informed by a team member about a new best value obtained for a configuration which is far from the current search region, is it better for it to move to the new and promising area or to keep visiting the current region? See also Fig. 39.3.

An example of the dangerous effects of interaction in the social arena is called *groupthink* [213], a mode of thinking that people engage in when they are deeply involved in a cohesive group, when the members' strivings for unanimity override their motivation to realistically appraise alternative courses of action. *Design by committee* is a second term referring to the poor results obtained by a group, particularly in the presence of poor and incompetent leadership. A camel is a horse designed by a committee.

An example of a pragmatic use of memory in cooperation is [295], where experiments highlight that "memory is useful but its use from the very beginning of the search is not recommended."

Centralized versus distributed management schemes This issue is in part related to computing hardware and communication networks, in part related to the software abstraction adopted for programming. The centralized schemes, often related to some form of synchronous communication, see a central coordinator acting in a **master-slave** relationship w.r.t. the individual solvers. The team members are given instructions (for example initialization points, parameters, detailed strategies) by the coordinator and periodically report about search progress and other parameters. The opposite design point consists of distributed computation by peers, which periodically and often asynchronously exchange information and autonomously decide about their future steps. **Gossiping optimization** schemes fall in this category. The design alternatives are related to efficiency but also to simplicity of programming and understanding. For example, a synchronous parallel machine may be handled more efficiently through a central coordination, while a collection of computers distributed in the world, connected by internet and prone to disconnections, may find a more natural coordination scheme by *gossiping*. Reviews of parallel strategies for local search and meta-heuristics are presented in [368] (see for example the *multiple walks* parallelism), in [160] ("controlled pool maintenance") and [103].

Reactive versus non-reactive schemes Last but not least comes the issue of learning on the job and self-tuning of algorithm parameters. In addition to the adaptation of individual parameters based on individual search history, which is by now familiar to the reader, new possibilities for a reactive adaptation arise by considering the search history of other team members and the overall coordination scheme. As examples, adaptation in evolutionary computation is surveyed in [178, 124]. Adaptation can act on the representation of the individuals, the evaluation function, the variation, selection and replacement operators and their probabilities, the population (size, topology, etc.). In their taxonomy, parameter tuning coincides with our "off-line tuning," while parameter control coincides with "on-line tuning," adaptive parameter control has a reactive flavor, while self-adaptive parameter control means that one want to use the same golden hammer (GA) for all nails, including meta-optimization (the parameters are encoded into chromosomes). Strategic design embodying intelligence more than randomization is also advocated in the *scatter search and path relinking approach* [151, 154]. Scatter search is a general framework to maintain a reference set of solutions, determined by their function values but also by their level of diversity, and to build new ones by "linearly interpolating and extrapolating" between subsets of these solutions. Of course, interpolation must be interpreted to work in a discrete setting (the uniform cross-over in GA is a form of interpolation) and adapted to the problem structure. *Path relinking* generalizes the concept: instead of creating a new solution from a set of two parents, an entire path between them is created by starting from one extreme and progressively modifying the solution to reduce the distance from the other point. The approach is strongly pragmatic, alternatives are tried and judged in a manner depending on the final results and not on the adherence to biological and evolutionary principles.

38.2 Portfolios and restarts

Let us consider *Las Vegas* algorithms, which always terminate with a correct solution and have a stochastic distribution of the runtime, the time required to terminate. We are interested both in the expected value of the runtime and in its standard deviation. The standard deviation is related to the **risk**; in some cases having a larger average CPU time with a small deviation is preferable to having a smaller average but with some instances requiring enormous times. There are two simple ways to combine the execution of different algorithms or of different versions of the same algorithm

(with different random seeds) to obtain different expected runtimes and standard deviations: one is based on **restarting** an algorithm if it does not terminate within a given time limit, the other one is based on combining more runs in a time-sharing interleaving manner: the **portfolio** approach.

Algorithm portfolios, first proposed in [199], follow the standard practice in economics to obtain different return-risk profiles in the stock market by combining stocks characterized by individual return-risk values. Risk is related to the standard deviation of return. An algorithm portfolio runs more algorithms concurrently on a sequential computer, in a time-sharing manner, by allocating a fraction of the total CPU cycles to each of them. The first algorithm to finish determines the termination time of the portfolio, while the other algorithms are stopped immediately after one reports the solution, see Fig. 38.1.

It is intuitive that the CPU time can be radically reduced in this manner. To clarify ideas consider an extreme example where, depending on the initial random seed, the runtime can be of 1 second or of 1000 seconds, with the same probability. If one runs a single process, the expected runtime is approximately of 500 seconds. If one runs more copies, the probability that at least one of them is lucky (i.e., that it terminates in 1 second) increases very rapidly towards one. Even if termination is now longer than 1 second because more copies share the same CPU, it is intuitive that the expected time will be much shorter than 500.

A portfolio can consist of different algorithms but also of different runs of the same algorithm, with different random seeds. In the case of more runs of the same algorithm, there is a different way to have more runs share a given CPU, by terminating a run prematurely and *restarting* the algorithm.

In the above example, a run can be stopped if it does not terminate within 1 second. Because the probability to have a sequence of unlucky cases rapidly goes to zero, again the expected runtime of the restart strategy will be much less than 500 seconds.

As an example in web surfing, the response time to deliver a page can vary a lot. The customary behavior of clicking again on the same link after patience is finished can save the user from an "endless" waiting time.

38.3 Predicting the performance of a portfolio from its component algorithms

To make the above intuitive arguments precise let $T_{\mathcal{A}}$ be the random variable describing the time of arrival of process $\mathcal{A}$ when the whole CPU time is allocated to it. Let $p_{\mathcal{A}}(t)$ be its probability distribution. The *survival function* $S_{\mathcal{A}}(t)$ is the probability that process $\mathcal{A}$ takes longer that t to complete:

$$S_{\mathcal{A}}(t) = \Pr(T_{\mathcal{A}} > t) = \int_{\tau>t} p_{\mathcal{A}}(\tau)\, \mathrm{d}\tau = 1 - F_{\mathcal{A}}(t)$$

where $F_{\mathcal{A}}(t)$ is the corresponding cumulative distribution function. If only a fraction α of the total CPU time is dedicated to it in a time-sharing fashion, with arbitrarily small time quanta and no process swapping overhead, we can model the new system as a process $\mathcal{A}'$ whose time of completion is described by random variable $T_{\mathcal{A}'} = \alpha^{-1} T_{\mathcal{A}}$. Its probability distribution and cumulative distribution function are respectively:

$$p_{\mathcal{A}'}(t) = p_{\mathcal{A}}(\alpha t), \qquad F_{\mathcal{A}'}(t) = F_{\mathcal{A}}(\alpha t), \qquad S_{\mathcal{A}'}(t) = S_{\mathcal{A}}(\alpha t).$$

Consider a portfolio of two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$. To simplify the notation, let T_1 and T_2 be the random variables associated with their termination times (each being executed on the whole CPU), with survival functions S_1 and S_2. Let α_1 be the fraction of CPU time allocated to process running algorithm $\mathcal{A}_1$. Then the fraction dedicated to $\mathcal{A}_2$ is $\alpha_2 = 1 - \alpha_1$. The completion time of the two-process portfolio system is therefore described by the random variable

$$T = \min\{\alpha_1^{-1} T_1, \alpha_2^{-1} T_2\}. \tag{38.1}$$

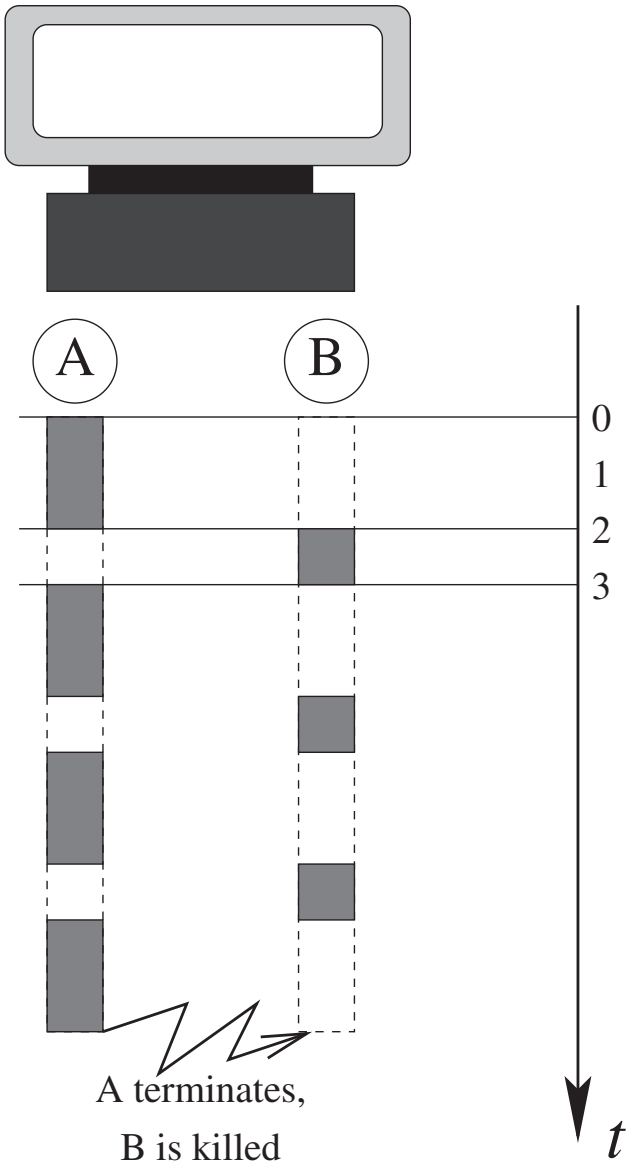


Figure 38.1: A sequential portfolio strategy.

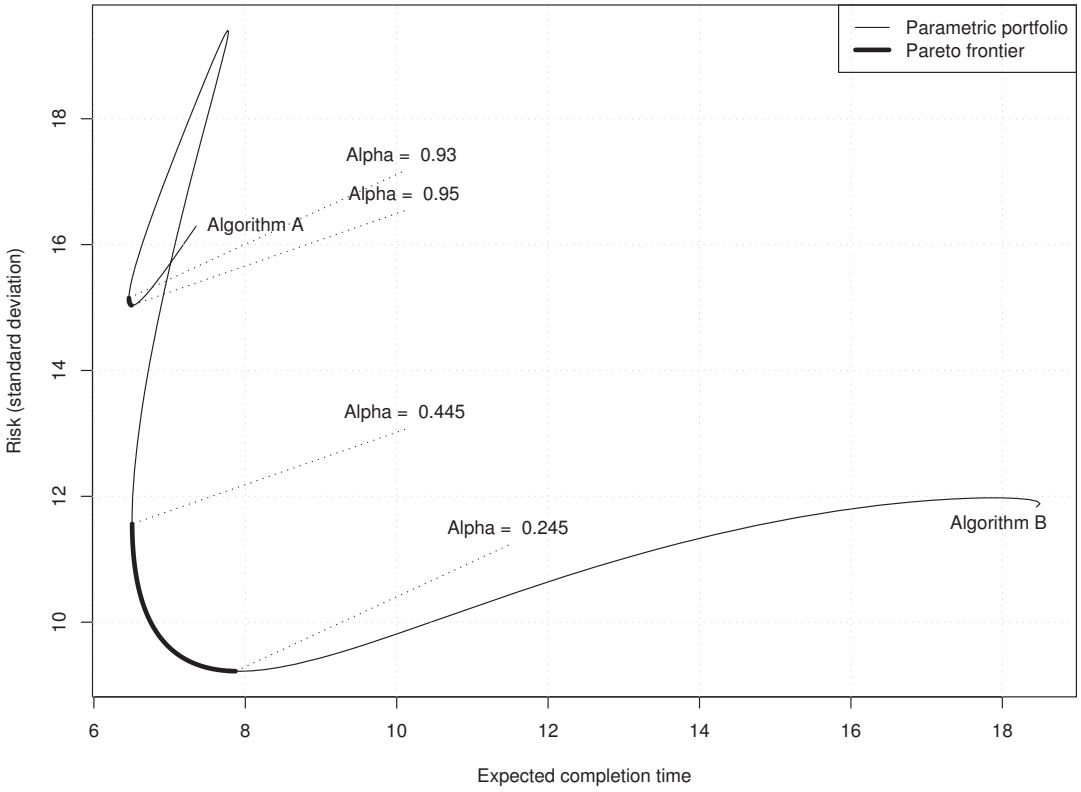


Figure 38.2: Expected runtime versus standard deviation (risk) plot. The efficient frontier contains the set of non-dominated configurations (a.k.a. Pareto-optimal or extremal points).

The survival function of the portfolio is

$$\begin{aligned} S(t) &= \Pr(T > t) = \Pr(\min\{\alpha_1^{-1}T_1, \alpha_2^{-1}T_2\} > t) \\ &= \Pr(\alpha_1^{-1}T_1 > t \wedge \alpha_2^{-1}T_2 > t) = \Pr(\alpha_1^{-1}T_1 > t)\Pr(\alpha_2^{-1}T_2 > t) \\ &= \Pr(T_1 > \alpha_1 t)\Pr(T_2 > \alpha_2 t) \\ &= S_1(\alpha_1 t)S_2(\alpha_2 t). \end{aligned}$$

The probability distribution of T can be obtained by differentiation:

$$p(t) = -\frac{\partial S(t)}{\partial t}.$$

Finally, the expected termination value $\mathrm{E}(T)$ and the standard deviation $\sqrt{\mathrm{Var}(T)}$ can be calculated.

By turning the α_1 knob, therefore, a series of possible combinations of expected completion time $\mathrm{E}(T)$ and risk $\sqrt{\mathrm{Var}(T)}$ becomes available. Fig. 38.2 illustrates an interesting case where two algorithms $\mathcal{A}$ and $\mathcal{B}$ are given. Algorithm $\mathcal{A}$ has a fairly low average completion time, but it suffers from a large standard deviation, because the distribution

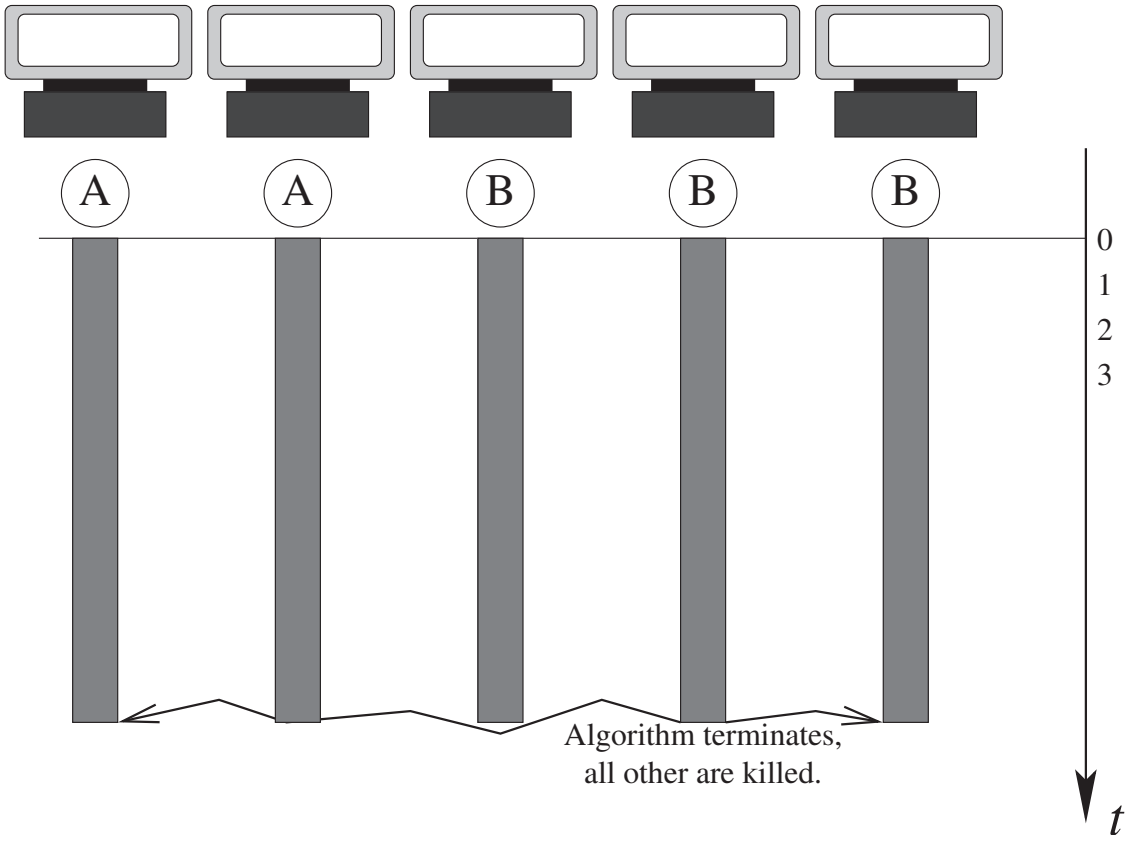


Figure 38.3: A portfolio strategy on a parallel machine.

is bimodal or heavy-tailed, while algorithm $\mathcal{B}$ has a higher expected runtime, but with the advantage of a lower risk of having a longer computation. By combining them as described above, we obtain a parametric distribution whose expected value and standard deviation are plotted against each other for α_1 going from 0 (only $\mathcal{B}$ executed) to 1 (pure $\mathcal{A}$). Some of the obtained distributions are *dominated* (there are parameter values that yield distributions with lower mean time *and* lower risk) and can be eliminated from consideration in favor of better alternatives, while the choice among the non-dominated possibilities (on the *efficient frontier* shown in black dots in the figure) has to be specified depending on the user preferences between lower expected time or lower risk. The choice along the Pareto frontier is similar when investing in the stock market: while some choices can be immediately discarded, there are no free meals and a higher return comes with a higher risk.

38.3.1 Parallel processing

Let us consider a different context [159] and assume that N equal processors and two algorithms are available so that one has to decide how many copies n_i to run of the different algorithms, as illustrated in Fig. 38.3. Of course no processor should remain idle, therefore $n_1 + n_2 = N$.

Consider time as a discrete variable (clock ticks or fractions of second), let T_i be the discrete random variable associated with the termination time of algorithm i having probability $p_i(t)$, the probability that process i halts precisely

at time t. As in the previous case, we can define the corresponding cumulative probability and survival functions:

$$F_i(t) = \Pr(T \leq t) = \sum_{\tau=0}^{t} p_i(\tau), \qquad S_i(t) = \Pr(T > t) = \sum_{\tau=t+1}^{\infty} p_i(\tau).$$

To calculate the probability $p(t)$ that the portfolio terminates exactly at time $T = t$, we must sum probabilities for different events: the event that one processor terminates at t while the other ones take more than t, the event that two processors terminate at t while the other ones take more than t, and so on. The different runs are independent and therefore probabilities are multiplied. If $n_1 = N$ (all processors are assigned to the same algorithm), this leads to:

$$p(t) = \sum_{i=1}^{N} \binom{N}{i} p_1(t)^i S_1(t)^{N-i} \tag{38.2}$$

The portfolio survival function $S(t)$ is easier to compute on the basis of the survival function of the single process $S_1(t)$:

$$S(t) = S_1(t)^N \tag{38.3}$$

When two algorithms are considered, the probability computation has to be modified to consider the different ways to distribute i successes at time t among the two sets of copies such that $i_1 + i_2 = i$ (i_1 and i_2 being non-negative integers).

$$p(t) = \sum_{\substack{0 \leq i_1 \leq n_1 \\ 0 \leq i_2 \leq n_2 \\ i_1 + i_2 \geq 1}} \binom{n_1}{i_1} p_1(t)^{i_1} S_1(t)^{n_1 - i_1} \binom{n_2}{i_2} p_2(t)^{i_2} S_2(t)^{n_2 - i_2}. \tag{38.4}$$

Similar although more complicated formulas hold for more algorithms. As before, the complete knowledge about $p(t)$ can then be used to calculate the mean and variance of the runtimes. **Portfolios can be effective to cure the heavy-tailed behavior** of $p_i(t)$ in many complete search methods, where very long runs occur more frequently than one may expect, in some cases leading to infinite mean or infinite variance [158]. Heavy-tailed distributions are characterized by a power-law decay, also called tails of Pareto-Lévy form, namely:

$$P(X > x) \approx Cx^{-\alpha}$$

where $0 < \alpha < 2$ and C is a constant.

Experiments with the portfolio approach [199, 159] show that, in some cases, a slight **mixing of strategies** can be beneficial provided that one component has a relatively high probability of finding a solution fairly quickly. Portfolios are also particularly effective when negatively correlated strategies are combined: one algorithm tends to be good on the cases which are more difficult for the other one, and *vice versa*. In branch-and-bound applications [159] one finds that ensembles of risky strategies can outperform the more conservative best-bound strategies. In a suitable portfolio, a depth-first strategy which often quickly reaches a solution can be preferable to a breadth first strategy with lower expected time but longer time to obtain a first solution.

Portfolios can also be applied to component routines inside a single algorithm, for example to determine an acceptable move in a local-search based strategy.

38.4 Reactive portfolios

The assumption in the above analysis is that the statistical properties of the individual algorithms are known beforehand, so that the expected time and risk of the portfolio can be calculated, the efficient frontier determined and the final choice executed depending on the risk-aversion nature of the user. The strategy is therefore *off-line*: a preliminary exhaustive study of the components precedes the portfolio definition.

Variable	Scope	Meaning
$\mathcal{A}_i$	(input)	i-th algorithm ($i = 1, \ldots, n$)
b_k	(input)	k-th problem instance ($k = 1, \ldots, m$)
f_P	(input)	Function deciding time slice according to expected completion time
f_τ	(input)	Function estimating the expected completion time based on history
τ_i	(local)	Expected remaining time to completion of current run of algorithm $\mathcal{A}_i$
α_i	(local)	Fraction of CPU time dedicated to algorithm $\mathcal{A}_i$
history	(local)	Collection of data about execution and status of each process

1. **function AOTA**($\mathcal{A}_1, \ldots, \mathcal{A}_n, b_1, \ldots, b_m, f_P, f_\tau$)
2. **repeat** $\forall b_k$
3. ⌜ initialize $(\tau_1, \ldots, \tau_n)$
4. **while** (b_k **not** solved)
5. ⌜ update $(\alpha_1, \ldots, \alpha_n) \leftarrow f_P(\tau_1, \ldots, \tau_n)$
6. **repeat** $\forall \mathcal{A}_i$
7. ⌜ run $\mathcal{A}_i$ for a slot of CPU time $\alpha_i \Delta t$
8. update history of $\mathcal{A}_i$
9. ⌞ update estimated termination $\tau_i \leftarrow f_\tau$(history)
10. ⌞ update model f_τ considering also the complete history of the last solved instance

Figure 38.4: The inter-problem AOTA framework.

If the distributions $p_i(t)$ are unknown, or if they are only partially known, one has to resort to **reactive portfolios, where the strategy is dynamically changed in an online manner** when more information is obtained about the task(s) being solved and the algorithm status. For example, one may derive a maximum-likelihood estimate of $p_i(t)$, use it to define the first values α_i of the CPU time allocations, and then refine the estimate of $p_i(t)$ when more information is received and use it to define subsequent allocations. A preliminary suggestion of dynamic online strategies is present in [199].

Dynamic strategies for search control mechanisms in a portfolio of algorithms are considered in [91, 92]. In this framework, statistical models of the quality of solutions generated by each algorithm are computed online and used as a control strategy for the algorithm portfolio, to determine how many cycles to allocate to each of the interleaved search strategies.

A "life-long learning" approach for dynamic algorithm portfolios is considered in [140]. The general approach of "dropping the artificial boundary between training and usage, exploiting the mapping *during* training, and including training time in performance evaluation," also termed Adaptive Online Time Allocation [139], is fully reactive. In the inter-problem AOTA framework, see Fig. 38.4, a set of algorithms $\mathcal{A}_i$ is given, together with a sequence of problem instances b_k, and the goal is to minimize the runtime of the whole set of instances. The model used to predict the runtime $p_i(t)$ of algorithm $\mathcal{A}_i$ is updated after each new instance b_k is solved. The portion of CPU time α_i is allocated to each algorithm $\mathcal{A}_i$ in the portfolio through a heuristic function which is decreasing for longer estimated runtimes τ_i.

An Extreme Reactive Portfolio (XRP) is proposed in [71]. It is based on simple performance indicators: record value and iterations elapsed from the last record. The two indicators are used for a combined ranking and a stochastic replacement of the worst-performing members with a new searcher with random parameters or a perturbed version of a well-performing member.

38.5 Defining an optimal restart time

Restarting an algorithm at time τ is beneficial if its expected time to convergence is less than the expected additional time to converge, given that it is still running at time τ [366]:

$$E[T] < E[T - \tau | T > \tau]. \tag{38.5}$$

Whether restart is beneficial or not depends of on the distribution of runtimes. As a trivial example, if the distribution is exponential, restarting the algorithm does not modify the expected runtime.

If the distribution is heavy-tailed, restart easily cures the problem. For example, heavy tails can be encountered if a stochastic local search algorithm like simulated annealing is trapped in the neighborhood of a local minimizer. Although eventually the probability of visiting the optimal solution will be one, an enormous number of iterations can be spent in the attraction basin around the local minimizer before escaping. Restart is a direct method to escape deep local minima!

If the algorithm is always restarted at time τ, each run corresponds to a Bernoulli trial which succeeds with probability $P_\tau = \Pr(T \leq \tau)$ — remember that T is the random variable associated with termination time of an unbounded run of the algorithm. The number of runs executed by the restart strategy before success follows a geometric distribution with parameter P_τ, in fact the probability of a success at repetition k is $(1 - P_\tau)^{k-1} P_\tau$. The distribution of the termination time T_τ of the restart strategy with restart time τ can be derived by observing that at iteration t one has had $\lfloor t/\tau \rfloor$ restarts and $(t \bmod \tau)$ remaining iterations. Therefore, the survival function of the restart strategy is

$$S_\tau(t) = \Pr(T_\tau > t) = (1 - P_\tau)^{\lfloor t/\tau \rfloor} \Pr(T > t \bmod \tau). \tag{38.6}$$

The tail decays now in an exponential manner: the restart portfolio is *not* heavy-tailed.

In general, a restart strategy consists of executing a sequence of runs of a randomized algorithm, to solve a given instance, stopping each run k after a time $\tau(k)$ if no solution is found, and restarting an independent run of the same algorithm, with a different random seed. The optimal restart strategy is uniform, i.e., one in which a constant $\tau_k = \tau$ is used to bound each run [255]. In this case, the expected value of the total runtime T_τ , i.e., the sum of runtimes of the successful run, and all previous unsuccessful runs is equal to:

$$E(T_\tau) = \frac{\tau - \int_0^\tau F(t)\ \mathrm{d}t}{F(\tau)} \tag{38.7}$$

where $F(\tau)$ is the cumulative distribution function of the runtime T for an unbounded run of the algorithm, i.e., the probability that the problem is solved before time τ. The demonstration is simple. For a given cutoff τ, each run succeeds with probability $F(\tau)$ (Bernoulli trials) and the mean number of trials before a successful run is encountered is $1/F(\tau)$. The expected length of each run is:

$$\int_0^\tau t p(t)\ \mathrm{d}t + \tau(1 - F(\tau))$$

Consider the cases when termination is within τ or later, so that the run is terminated prematurely. Because $p(t) = F'(t)$, this is equal to:

$$\int_0^\tau t F'(t)\ \mathrm{d}t.$$

The result follows from the fact that:

$$\frac{\mathrm{d}}{\mathrm{d}t}(tF(t)) = tF'(t) + F(t)$$

and therefore:

$$\int_0^\tau t F'(t)\ \mathrm{d}t + \int_0^\tau F(t)\ \mathrm{d}t = \tau F(\tau)$$

giving (38.7).

In the discrete case:

$$E(T_\tau) = \frac{\tau - \sum_{t<\tau} F(t)}{F(\tau)} \tag{38.8}$$

If the distribution is known, an optimal cutoff time can be determined by minimizing (38.7). If the distribution is not known, a universal non-uniform strategy, with cutoff sequence: $(1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, \dots)$ achieves a performance within a logarithmic factor of the expected runtime of the optimal policy, see [255] for details.

Calculating the runtime distribution can require large amounts of CPU time in case of heavy tails because one has to wait for the termination of very long runs. In this case the **censored sampling** approach can be used. Censored sampling allows to bound the duration of each experimental run and still exploit the information obtained from the runs which converge before the censoring threshold [280]. Let us model the probability density function as $g(t|\theta)$, θ being the parameter to be identified from the experiments. Without censoring one can determine g by maximizing the likelihood $\mathcal{L}$ of the obtained sequence of termination times $\mathcal{T} = (t_1, t_2, \dots, t_k)$ given θ:

$$\mathcal{L}(\mathcal{T}|\theta) = \prod_{i=1}^{k} \mathcal{L}(t_i|\theta) = \prod_{i=1}^{k} g(t_i|\theta) \tag{38.9}$$

With censoring, some experimental runs will exceed the cutoff time t_c. In these cases the corresponding multiplicative term in (38.9) is substituted with

$$\mathcal{L}_c(t_c|\theta) = \int_{t_c}^{\infty} g(\tau|\theta)\, \mathrm{d}\tau = 1 - G(t_c|\theta) \tag{38.10}$$

where $G(t|\theta)$ is the conditional cumulative distribution function corresponding to g.

One has to decide about a proper cutoff threshold t_c. A way to determine it is to ask for target u on the fraction of terminated runs (uncensored samples), run k experiments in parallel (or with interleaving) and stop as soon as the desired target is reached.

The final receipt is therefore: i) choose an appropriate parametric model for the runtime distribution, ii) determine the best parameters of the model by maximizing the likelihood, where some terms are substituted with the censored likelihood of (38.10), iii) use the estimated runtime distribution to determine the optimal restart time. Some examples of parametric models are considered in [141].

38.6 Reactive restarts

Up to now the assumption has been that the only observation which can be used is given by the *length of a run* and that the runs are *independent*. Let us now consider more advanced strategies where at least one of these assumptions is relaxed. Given the results mentioned in the previous section, it looks as if the problem is solved for the complete knowledge case and the zero knowledge case (within a multiplicative constant and logarithmic factor which can be large for practical applications). Actually, the most interesting case is between the two situations, when a partial knowledge is available which is increasing as soon as more data become available during a run or during a sequence of runs on related instances. Real-time observations about the characteristics of a specific instance and about the state of the solver *during a run* permit better results.

In [191, 231] features capturing the state of a solver during the initial phase of the run are used to **predict the length of a run**, so that the prediction can be used by dynamic restart policies. Bayesian models to predict the runtime starting from both structural evidence available at the beginning of the run, and execution evidence available during the run (in a reactive manner) are trained with supervised machine learning. To be more precise, the discrimination is between long and short runs, i.e., runs longer or shorter than the median. The dynamic policy considered in [191] is as follows:

1. observe a run for O steps (observation horizon)

2. if the run is not terminated predict whether it will converge in a total of L steps

3. if the prediction is negative, restart immediately, otherwise run up to a total of L steps before restarting.

Because the model is not perfect, an important parameter is the model accuracy A, the probability of a correct prediction. If p_i is the probability of a run ending within i steps, the probability of convergence during a single run is therefore $p_O + A(p_L - p_O)$ and the expected number of runs until a solution is found is $E(n) = 1/(p_O + A(p_L - p_O))$. An upper bound on the expected number of steps in a single run can be derived by assuming that runs ending within O steps take exactly O steps, while runs terminating between $O+1$ and $O+L$ steps take exactly L steps. The probability of continuation, taking the limited accuracy into account, is $Ap_L + (1-A)(1-p_L)$. An upper bound on the length of a single run is therefore $E_{\text{ub}}(R) = O + (L-O)(Ap_L + (1-A)(1-p_L))$, and an upper bound on the expected time to solve a problem with the above policy is $E(n)E_{\text{ub}}(R)$. The estimate can be now minimized by varying L and the observation horizon. The model is rude; for example, no observations during the steps after O are used, only a bound and not the exact expected number of steps is minimized. In spite of its roughness, significantly superior results of the dynamic policy w.r.t. the static one are demonstrated. Three different contexts are defined: in the *single instance* context one has to solve a specific instance as soon as possible, in the *multiple instance* context one draws cases from a distribution of instances and has to solve either *any instance* as soon as possible, or *as many instances as possible* for a given amount of time allocated (*max instances* problem) [191].

The assumption of independence among runs is relaxed in [310]. For example, independence is not valid if more runs are on the same instance picked at the beginning from one of several probability distributions. As an example, consider two distributions, one consisting of instances which are solved in 10 iterations, the other one of instances which are solved in 100 iterations. If an instance is not solved in 10 iterations we know that 100 iterations are needed and restarting would only waste computing cycles. Compare this with the situation of a single distribution with probability 0.5 of converging at iteration 10, probability 0.5 of converging at iteration 1000, with independence among the runs. Here restarting is clearly useful as shown in Section 38.5. The work in [310] considers the context where one among several RTDs is picked at the beginning - without informing the user - and a new sample is extracted at each run from the same distribution (e.g., consider two different distributions corresponding to satisfiable or unsatisfiable instances of SAT). The task is to find the optimal restart policy $(t_1, t_2, \dots)$ but now, after each unsuccessful run, the solver's belief about the source distribution can be updated. The problem of finding the optimal restart policy is formulated as a Markov decision process and solved with dynamic programming, considering both the case in which only the termination time is observed, and the case when other predictors of the distribution can be used, for example the evidence obtained during the run about the fact that a SAT instance is or is not satisfiable.

38.7 Racing: Exploration and exploitation of candidate algorithms

Portfolios and restarts are simple ways to combine more algorithms, or more runs of a given randomized strategy, to obtain either a lower expected convergence time, or a lower risk (variance), or both.

We have already seen that more advanced reactive strategies can be obtained by using a reactive learning loop *while* the portfolio or restart scheme runs. In this way, some of the portfolio parameters or the restart threshold can take fresh information into account.

A related strategy using a "life-long learning" loop to optimize the allocation of time among a set of alternative algorithms for solving a specific instance is termed **racing**. Running algorithms are like horses: after the competition is started one gets more and more information about the relative performance and periodically updates the bets on the winning horses, which are assigned a growing fraction of the available future computing cycles, see Fig. 38.5.

A racing strategy is characterized by two components: i) the estimate of the future potential given the current state of the search, i.e., given the history of the previous iterations and the corresponding results, ii) the allocation of the future CPU cycles to speedup the overall objective of minimizing a function.

Racing is related to a paradigmatic problem in machine leaning and intelligent heuristics known as the **k-armed bandit problem**. One is faced with a slot machine with k arms which, when pulled, yield a payoff from a fixed but unknown distribution. One wants to maximize the expected total payoff over a sequence of n trials. If the distribution

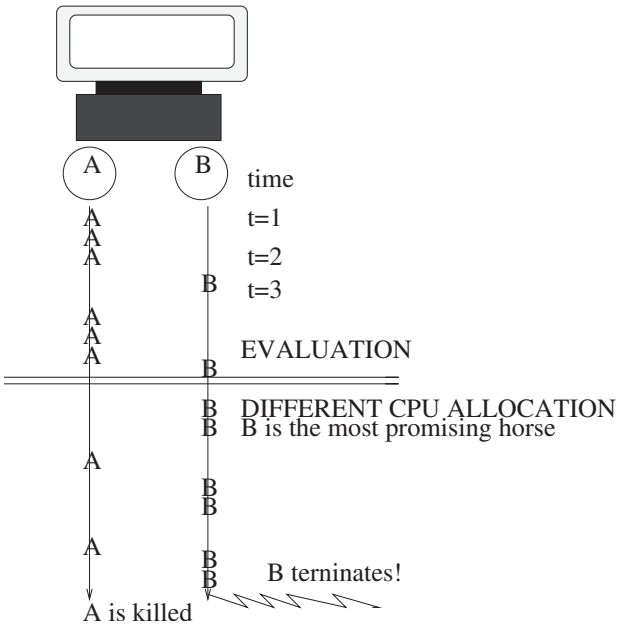


Figure 38.5: A racing strategy. The different horses (algorithms) are evaluated periodically to reallocate the CPU time shares.

is known one would immediately pull only the best performing arm. What makes the problems intriguing is that one has to split the effort between **exploration** to learn the different distributions and **exploitation** to pull the best arm, once the winner becomes clear. One is reminded of the critical exploration-versus-exploitation dilemma observed in optimization heuristics, but there is an important difference: in optimization one is not interested in maximizing the total payoff but in maximizing *the best pull* (the maximum value obtained by a pull in the sequence). The paper [128] is dedicated to determining a sufficient number of pulls to select with a high probability an arm (an hypothesis) whose *average* payoff is near-optimal. The max version of the bandit problem is considered in [93, 92]. An asymptotically optimal algorithm is presented in [350], in the assumption of a generalized extreme value (GEV) payoff distribution for each arm. Our explanation follows closely [349], which presents a simple distribution-free approach.

38.7.1 Racing to maximize cumulative reward by interval estimation

The first algorithm CHERNOFF-INTERVAL-ESTIMATION is for the classical bandit problem, which is then used as a starting point for the THRESHOLD-ASCENT algorithm dedicated to the max k-armed bandit problem. The assumption is that pulling an arm produces a random variable $X_i \in [0, 1]$. Because some effort is spent in exploration to determine (in an approximated manner) the best arm, of course the performance is less than that obtainable by knowing the best arm and pulling it all the time. What one misses by not having the information about the winning horse at the beginning is called **regret**. Precisely, regret is the difference between the payoff obtained by always pulling the best arm on a specific instance minus the cumulative payoff actually received during the racing strategy.

CHERNOFF-INTERVAL-ESTIMATION pulls arms and keeps an estimate of: the number of times n_i of pulls of the i-th arm, the expected reward $\bar{\mu}_i = \frac{x_i}{n_i}$ and an upper bound (with a specific minimum probability) on the reward $U(\bar{\mu}_i, n_i)$. At each iteration, the arm with the **highest upper bound** is pulled (Fig. 38.6 and Fig. 38.7). The upper

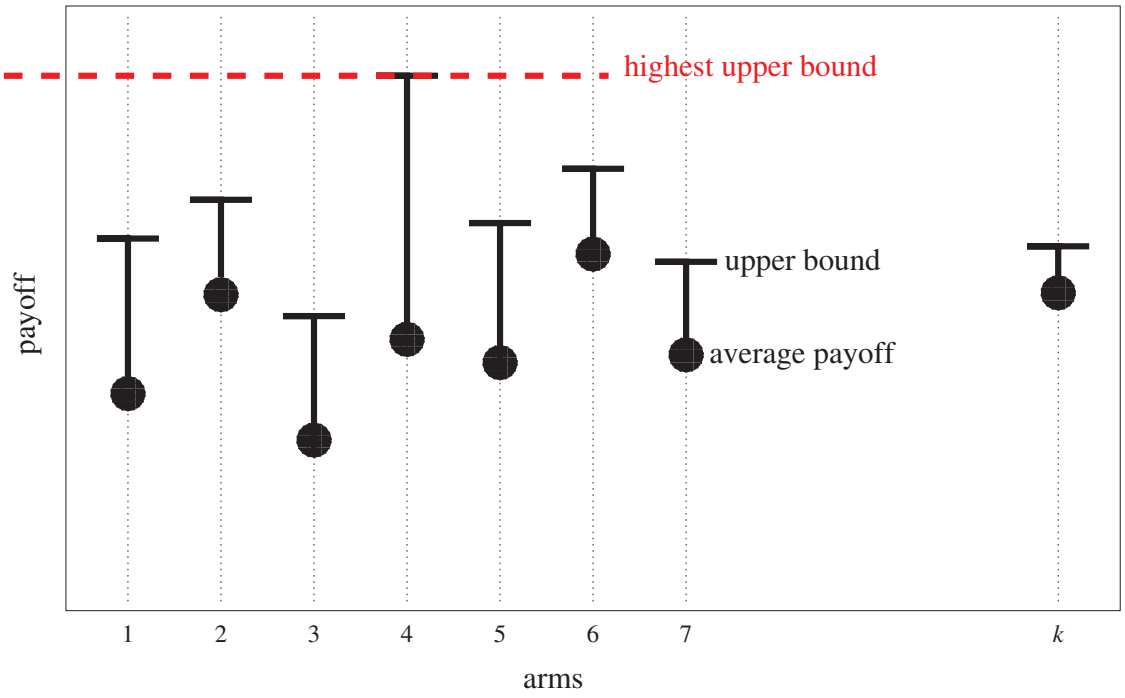


Figure 38.6: Racing with interval estimation. At each iteration an estimate of the expected payoff of each arm as well as its "error bar" are available.

1. **function Chernoff_Interval_Estimation**(n, δ)
2. **forall** $i \in \{1, 2, ..., k\}$ Initialize $x_i \leftarrow 0,\ n_i \leftarrow 0$
3. **repeat** n times:
4. $\quad \hat{i} \leftarrow \arg\max_i U(\bar{\mu}_i, n_i)$
5. $\quad$ pull arm $\hat{i}$, receive payoff R
6. $\quad x_i \leftarrow x_i + R,\ n_i \leftarrow n_i + 1$

Figure 38.7: The CHERNOFF-INTERVAL-ESTIMATION routine.

bound is derived from Chernoff's inequality and is as follows:

$$U(\mu, n) = \begin{cases} \mu + \frac{\alpha + \sqrt{2n\mu\alpha + \alpha^2}}{n} & \text{if } n > 0 \\ \infty & \text{otherwise} \end{cases} \tag{38.11}$$

where $\alpha = \ln\left(\frac{2nk}{\delta}\right)$ and δ regulates our confidence requirements, see later.

Chernoff's inequality estimates how much the empirical average can be different from the real average. Let $X = \sum_{i=1}^{n} X_i$ be the sum of independent identically distributed random variables with $X_i \in [0, 1]$, and $\mu = E[X_i]$ be the real expected value. The probability of an error of the estimate greater than $\beta\mu$ decreases in the following exponential way:

$$P\left[\frac{X}{n} < (1 - \beta)\mu\right] < e^{-\frac{n\mu\beta^2}{2}} \tag{38.12}$$

From this basic inequality, which does not depend on the particular distribution, one derives that, if arms are pulled according to the algorithm in Fig. 38.7, with probability at least $(1 - \delta/2)$, for all arms and for all n repetitions the upper bound is not wrong: $U(\bar{\mu_i}, n_i) > \mu_i$. Therefore each suboptimal arm (with $\mu_i < \mu^*$, μ^* being the best arm expected reward) is not pulled many times and the expected *regret* is limited to at most:

$$(1 - \delta)2\sqrt{3\mu^* n(k - 1)\alpha} + \delta\mu^* n \tag{38.13}$$

A similar algorithm based on Chernoff-Hoeffding's inequality has been presented in a previous work [12]. In their simple UCB1 deterministic policy, after pulling each arm once, one then pulls the arm with the highest bound $U(\bar{\mu}, n_i) = \bar{\mu} + \sqrt{\frac{2\ln n}{n_i}}$, see [12] for more details and experimental results.

38.7.2 Aiming at the maximum with threshold ascent

Our optimization context is characterized by a set of horses (different stochastic algorithms) aiming at discovering the maximum value for an instance of an optimization problem, for example different greedy procedures characterized by different ordering criteria, see [349] for an application to the Resource Constrained Project Scheduling Problem. The "reward" is the final result obtained by a single run of an algorithm. Racing is a way to allocate more runs to the algorithms which tend to get better results on the given instance.

We are therefore not interested in cumulative reward, but in the *maximum* reward obtained at any pull. A way to estimate the potential of different algorithms is to put a threshold $Thres$, and to estimate the probability that each algorithm produces a value above threshold. The estimate is the corresponding empirical frequency. Unfortunately, the appropriate threshold is not known at the beginning, and one may end up with a trivial threshold - so that all algorithms become indistinguishable - or with an impossible threshold, so that no algorithm will reach it. The heuristic solution presented in [349] reactively learns the appropriate threshold while the racing scheme runs (Fig. 38.9 and Fig. 38.8).

The threshold starts from zero (remember that all values are bounded in $[0, 1]$), and it is progressively raised raised until a selected number s of experimented payoffs above threshold is left. For simplicity, but it is easy to generalize, one assumes that payoffs are integer multiples of a suitably small Δ, $R \in \{0, \Delta, 2\Delta, ..., 1 - \Delta, 1\}$. In the figure, $\bar{\nu_i}$ is the frequency with which arm i received a value greater than $Thres$ in the past, an estimate of the probability that it will do so in the future. This quantity is easily calculated from $n_{i,R}$, the number of payoffs equal to R received by horse i. The upper bound U is the same as before.

The parameter s controls the tradeoff between intensification and diversification. If $s = 1$ the threshold becomes so high that no algorithm reaches it: the bound is determined only by n_i and the next algorithm to run is the one with the lowest n_i (Round Robin). For larger values of s one start differentiating between the individual performance. A larger s means a more robust evaluation of the different strategies (not based on pure luck - so to speak), but a very large value means that the threshold gets lower and lower so that even poor performers have a chance of being selected. The specific setting of s is therefore not so obvious and it looks like more work is needed.

function Threshold_Ascent(s, n, δ)
Thres $\leftarrow 0$
forall $i \in \{1, 2, ..., k\}$
  forall R values
    Initialize $n_{i,R} \leftarrow 0$
repeat n times:
  while (number of payoffs received above threshold $\geq s$)
    $Thres \leftarrow Thres + \Delta$ (raise threshold)
  $\hat{i} \leftarrow \arg\max_i U(\bar{\nu}_i, n_i)$
  pull arm $\hat{i}$, receive payoff R
  $n_{i,R} \leftarrow n_{i,R} + 1$

Figure 38.8: The THRESHOLD-ASCENT routine.

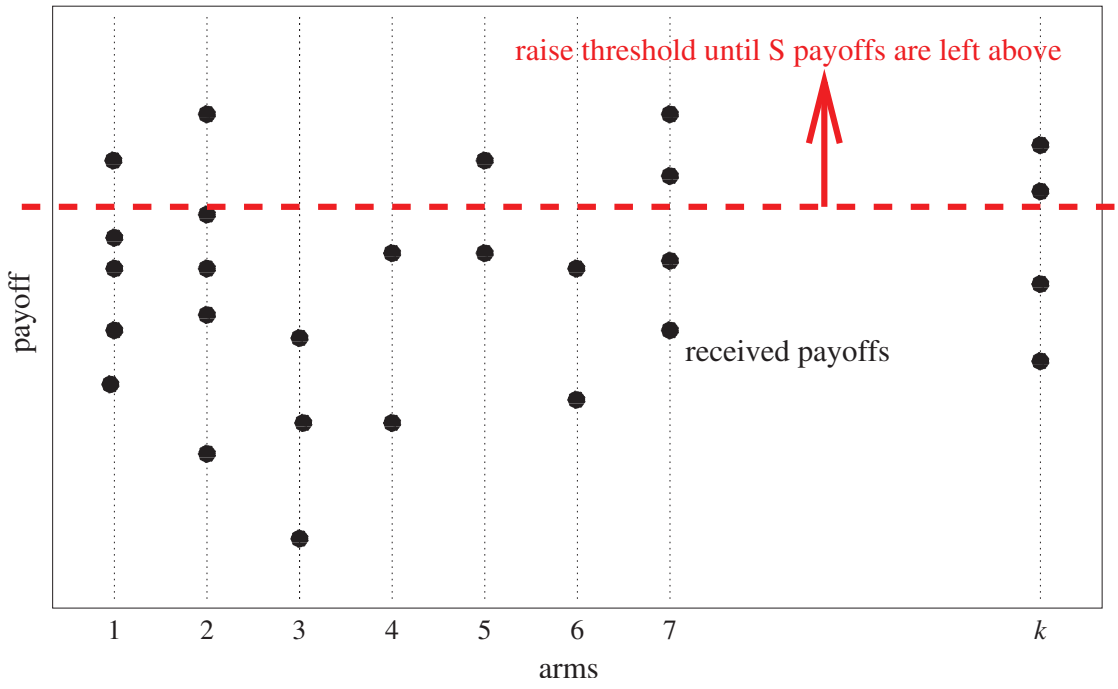


Figure 38.9: Threshold ascent: the threshold is progressively raised until a selected number of experimented payoffs is left.

38.7.3 Racing for off-line configuration of heuristics

The context here is that of selecting in an off-line manner the best configuration of parameters θ for a heuristic solving repetitive problems [54]. Let's assume that the set of possible θ values is finite. For example, a pizza delivery service receives orders and, at regular intervals, has to determine the best route to serve the last customers. In this case an off-line algorithm tuning (or "configuration"), even if expensive, is worth the effort because it is going to be used for a long time in the future.

There are two sources of randomness in the evaluation: the stochastic occurrence of an instance, with a certain probability distribution, and the intrinsic stochasticity in the randomized algorithm while solving a given instance. Given a criterion $\mathcal{C}(\theta)$ to be optimized with respect to θ, for example the average cost of the route over different instances and different runs, the ideal solution of the configuration problem is:

$$\theta^* = \arg\min_{\theta} \mathcal{C}(\theta) \tag{38.14}$$

where $\mathcal{C}(\theta)$ is the following Lebesgue integral (I is the set of instances, C is the range for the cost of the best solution found in a run, depending on the instance i and the configuration θ):

$$\mathcal{C}(\theta) = E_{I,C}\left[c(\theta, i)\right] = \int_I \int_C c(\theta, i) dP_C(c|\theta, i) dP_I(i) \tag{38.15}$$

The probability distributions are not known at the beginning. Now, to calculate the expected value for each of the finite configurations, ideally one could use a brute force approach, considering a very large number of instances and runs, tending to infinity. Unfortunately, this approach is tremendously costly, usually each run to calculate $c(\theta, i)$ implies a non-trivial CPU cost, and one has to resort to smarter methods.

Firstly, the above integral in equation (38.15) is estimated in a Monte Carlo fashion by considering a set of instances. Secondly, as soon as the first estimates become available, the manifestly poor configurations are discarded so that the **estimation effort is more concentrated onto the most promising candidates**. This process is actually a bread-and-butter issue for researchers in heuristics, with racing one aims at **a statistically sound hands-off approach**. In particular, one needs a sound criterion to determine that a candidate configuration θ_j is *significantly* worse than the current best configuration available, given the current state of the experimentation.

The situation is illustrated in Fig. 38.10, at each iteration a new test instance is generated and the surviving candidates are run on the instance. The expected performance and error bars are updated. Afterwards, if some candidates have error bars that show a clear inferior performance, they are eliminated from further consideration. Before deciding for elimination, a candidate is checked to see whether its optimistic value (top error bar) can beat the pessimistic value of the best performer.

The advantage is clear: costly evaluation cycles to get better estimates of performance are dedicated only to the most promising candidates. Racing is terminated when a single candidate emerges as the winner or when a certain maximum number of evaluations have been executed, or when a target error bar ϵ has been obtained, depending on available CPU time and application requirements.

The variations of the off-line racing technique depend on the way in which **error bars** are derived from the experimental data.

In [261], racing is used to select models in a supervised learning context, in particular for *lazy* or memory-based learning. Two methods are proposed for calculating error bars. One is based on Hoeffding's bound which makes the only assumption of independence of the samples: the probability that the true error E_{true} being more than ϵ away from the estimate E_{est} is:

$$Prob(\|E_{true} - E_{est}\| > \epsilon) < 2e^{\frac{-n\epsilon^2}{B^2}} \tag{38.16}$$

where B bound the largest possible error. In practice, this can be heuristically estimated as some multiple of the estimated standard deviation. Given the confidence parameter δ for the right-hand side of equation (38.16) (we want the probability of a large error to be less than δ), one easily solves for the error bar $\epsilon(n, \delta)$:

$$\epsilon(n, \delta) = \sqrt{\frac{B^2 \log(2/\delta)}{2n}} \tag{38.17}$$

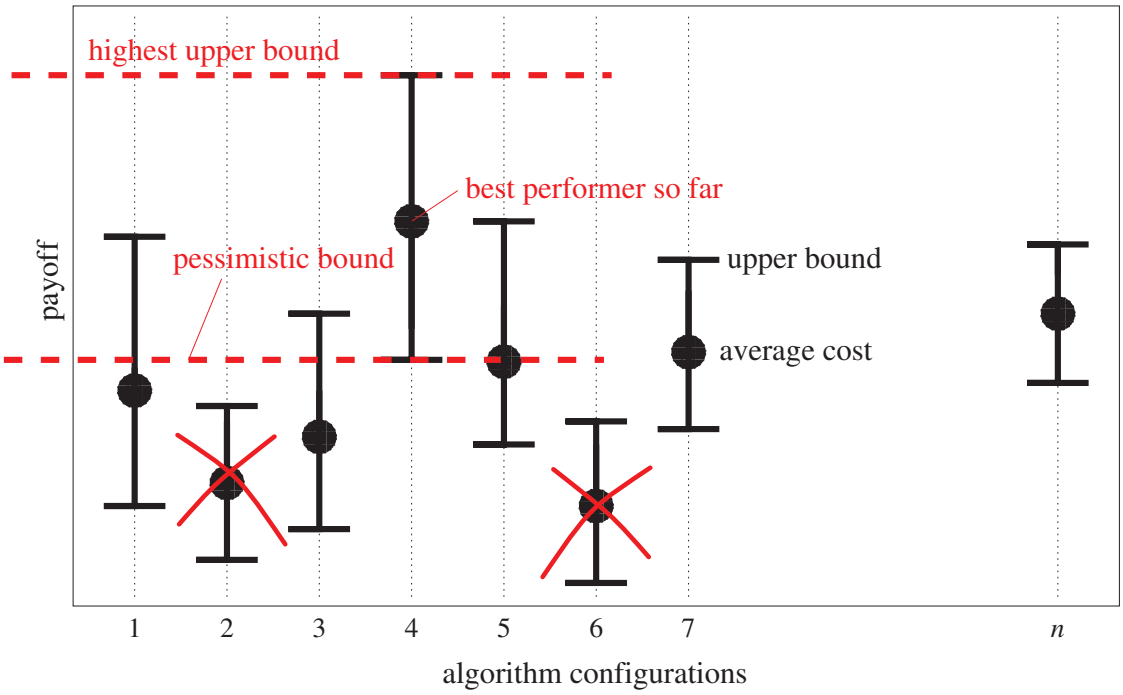


Figure 38.10: Racing for off-line optimal configuration of meta-heuristics. At each iteration an estimate of the expected performance with error bars is available. Error bars are reduced when more tests are executed, their values depend also on confidence parameter δ. In the figure, configurations 2 and 6 perform significantly worse than the best performer 4 and can be immediately eliminated from consideration: even if the real value of their performance is at the top of the error bar they cannot beat number 4.

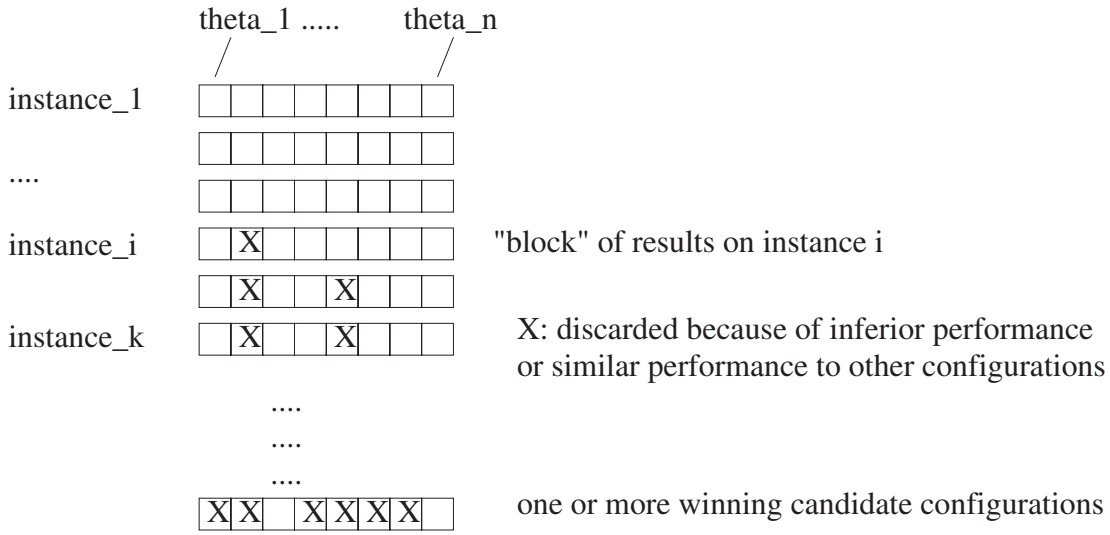


Figure 38.11: Racing for off-line optimal configuration of meta-heuristics. The most promising candidate algorithm configurations are identified asap so that these can be evaluated with a more precise estimate (more test instances). Each block corresponds to results of the various configurations on the same instance.

If the accuracy ϵ and the confidence δ are fixed, one can solve for the required number of samples n. The value $(1-\delta)$ is the confidence in the bound for a single model during a single iteration, additional calculations provide a confidence $(1-\Delta)$ of selecting the best candidate after the entire algorithm is terminated [261].

Tighter error bounds can be derived by making more assumptions about the statistical distribution. If the evaluation errors are normally distributed one can use Bayesian statistics, the second method proposed in [261]. One candidate model is eliminated if the probability that a second model has a better expected performance is above the usual confidence threshold:

$$Prob(E^{j}_{true} > E^{j'}_{true} \| e_j(1), ..., e_j(n), e_{j'}(1), ..., e_{j'}(n)) > 1-\delta \tag{38.18}$$

Additional methods for shrinking the intervals, as well as suggestions for using a statistical method known as *blocking* are explained in [261]. Model selection in continuous space is considered in [121].

In [54] the focus is explicitly on meta-heuristics configuration. Blocking through ranking is used in the F-RACE algorithm, based on the Friedman test, in addition to an aggregate test over all candidates performed before considering pairwise comparisons. Each block (Fig. 38.11) consists of the results obtained by the different candidate configurations θ_j on an additional instance i. From the results one gets a ranking R_{lj} of θ_j within block l, from the smallest to the largest, and $R_j = \sum_{l=1}^{k} R_{lj}$ the sum of the ranks over all instances. The Friedman test [98] considers the statistics T:

$$T = \frac{(n-1)\sum_{j=1}^{n}\left(R_j - \frac{k(n+1)}{2}\right)^2}{\sum_{l=1}^{k}\sum_{j=1}^{n} {R_{lj}}^2 - \frac{kn(n+1)^2}{4}} \tag{38.19}$$

Under the null hypothesis that the candidates are equivalent so that all possible rankings are equally likely, T is χ^2 distributed with $(n-1)$ degrees of freedom. If the observed t value exceeds the $(1-\delta)$ quantile of the distribution, the null hypothesis is rejected in favor of the hypothesis that at least one candidate tends to perform better than at least another one. In this case one proceed with a pairwise comparison of candidates. Configurations θ_j and θ_h are

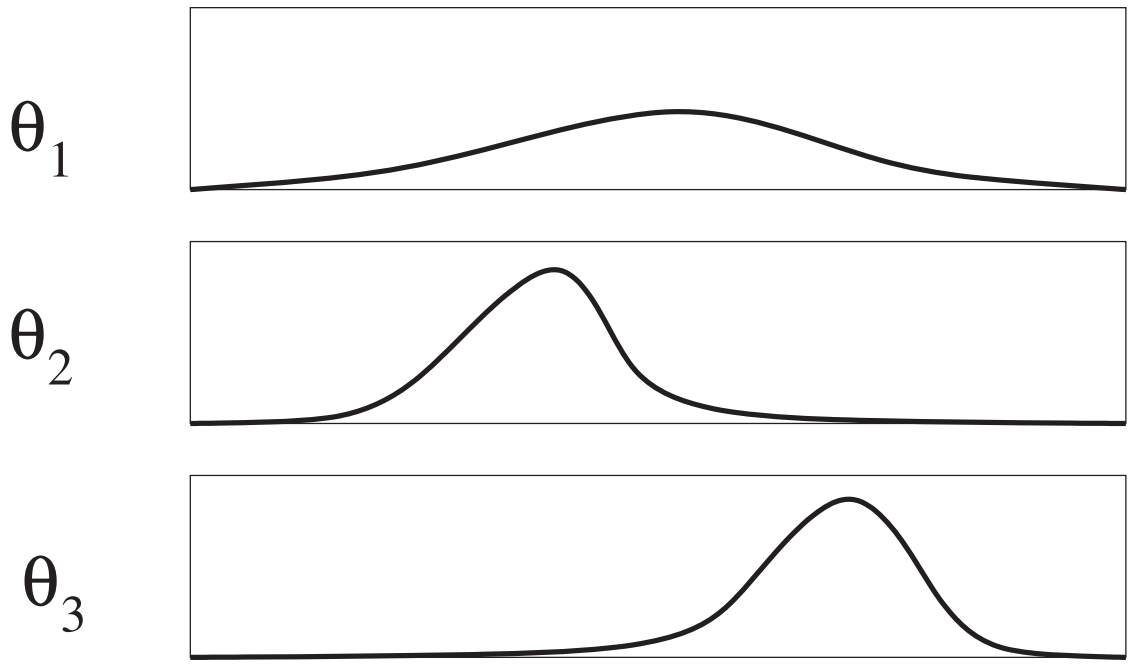


Figure 38.12: Bayesian elimination of inferior models, from the posterior distribution of costs of the different models one can eliminate the models which are inferior in a statistically significant manner, for example model θ_3 in the figure, in favor of model θ_2, while the situation is still undecided for model θ_1.

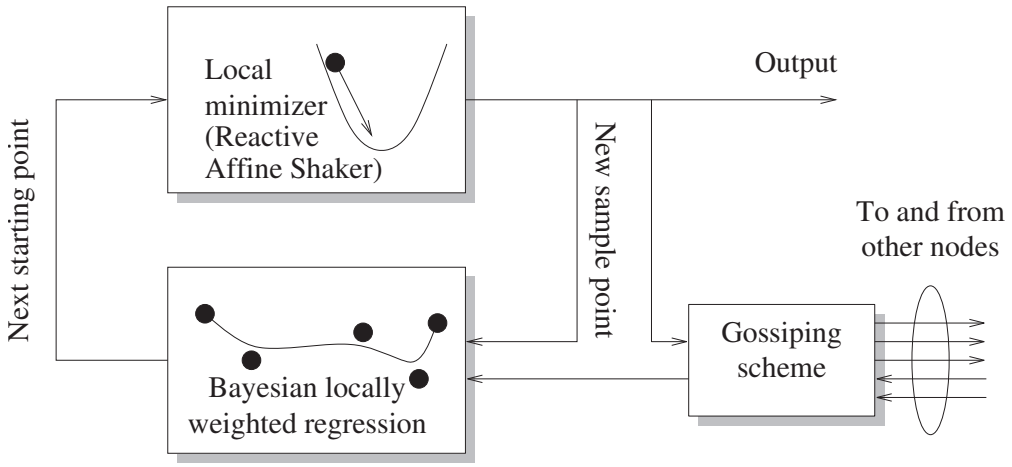


Figure 38.13: The distributed setting for the memory-based Reactive Affine Shaker.

considered different if:

$$\frac{\|R_j - R_h\|}{\sqrt{\frac{2k(1-\frac{T}{k(n-1)})\left(\sum_{l=1}^{k}\sum_{j=1}^{n} {R_{ij}}^2 - \frac{kn(n+1)^2}{4}\right)}{(k-1)(n-1)}}} > t_{1-\delta/2} \tag{38.20}$$

where $t_{1-\delta/2}$ is the $(1-\delta/2)$ quantile of the Student's t distribution. In this case the worse configuration is eliminated from further consideration.

38.8 Gossiping Optimization

Let's consider the following scenario: a set of intelligent searchers is spread on a number of computers, possibly throughout the world. While every searcher executes a local search heuristic, it takes advantage from the occasional injection of new information coming from its partners on other machines.

For instance, let's consider the field of continuous optimization, for which the RAS heuristic has been introduced in Section 32.1. In memory-based RAS [72] a fast local minimizer, the Reactive Affine Shaker, interacts with a model of the search space by feeding it with new data about the search and retrieving suggestions about the best starting point for a new run.

While MRAS has been devised as a sequential heuristic [72], it can be extended to a distributed one as described in Fig. 38.13: a gossiping component, described below, communicates model information to other nodes, and feeds information coming from other nodes to the model.

Sharing information among nodes is a delicate issue. The algorithm aims at function optimization, so it should spend most of its time doing actual optimization, not just broadcasting information to other nodes. Let's now discuss some communications issues arising in this context.

38.8.1 Epidemic communication for optimization

The use of parallel and distributed computing for solving complex optimization tasks has been investigated extensively in the last decades [356, 30]. Most works assume the availability of either a dedicated parallel computing facility or of a specialized clusters of networked machines that are coordinated in a centralized fashion (master-slave, coordinator-cohort, etc.). While these approaches simplify management, they have limitations with respect to scalability and robustness and require a dedicated investment in hardware.

Recently, the **peer-to-peer (P2P)** paradigm for distributed computing has demonstrated that networked applications can scale far beyond the limits of traditional distributed systems without sacrificing efficiency and robustness.

A well known problem with P2P systems is their high level of dynamism: nodes join and leave the system continuously, in many cases unexpectedly and without following any "exit protocol." This phenomenon, called *churn*, together with the large number of computational nodes, are the two most prominent research challenges posed by P2P systems: no node has an up-to-date knowledge of the entire system, and the maintenance of consistent distributed information may as well be impossible.

On the other hand, a clear advantage of P2P architectures is the exploitation of unused computational resources, such as personal desktop machines, volunteered by people who keep using their computers while participating to a shared optimization effort. The systems based on a central coordinator repeat a simple loop: every involved machine receives from a central server a subset of the search space (samples, parameter intervals), performs an exhaustive coverage of the subset and reports the results, receiving another search subset.

More distributed schemes originated in the context of databases [114], where **epidemic protocols** have been able to deal with the high levels of unpredictability associated with P2P systems. Apart from the original goal of information dissemination (messages are "broadcasted" through random exchanges between nodes), epidemic protocols are now used to solve several different problems, from membership and topology management to resource sharing.

We focus our attention onto stochastic local search schemes based on memory, where little or no information about the function to be optimized is available at beginning of the search. In this context, the knowledge acquired from function evaluations at different input points during the search can be mined to build models so that the future steps of the search process can be optimized. An example is the online adaptive self-tuning of parameters while solving a specific instance proposed by Reactive Search Optimization (RSO). Recent developments of interest consider the integration of multiple techniques and the feedback obtained by preliminary phases of the execution for a more efficient allocation of the future effort.

In the P2P scenario the crucial issues and tradeoffs to be considered when designing distributed optimization strategies are:

Coordination and interaction One has a choice of possibilities ranging from independent search processes reporting the end results, to fully coordinated "teams" of searchers exchanging new information after each step of the search process.

Synchronization In a peer-to-peer environment the synchronization must be very loose to avoid wasting computational cycles while waiting for synchronization events.

Type and amount of exchanged information It ranges from the periodic exchange of current configurations and related function values, see for example particle swarm [95] and genetic algorithms, to the exchange of more extensive data about past evaluations, possibly condensed into local heuristic models of the function [72].

Frequency of gossiping, convergence issues We consider a simple basic interaction where a node picks a random neighbor, exchanges some information and updates its internal state (memory). The spreading of the information depends both on the gossiping frequency and on the interconnection topology. Tradeoffs between a more rapid information exchange and a more rapid advancement of each individual search process are of interest.

Effects of delays on "distributed snapshots" Because of communication times, congestion and possible temporary disconnections, the received information originated from a node may not reflect accurately the current state, so that decisions are made in a suboptimal manner.

The distributed realization of a gossiping optimization scheme ranges between two extremes:

- **Independent execution of stochastic processes** — Some global optimization algorithms are stochastic by nature; in particular, the first evaluation is not driven by prior information, so the earliest stages of the search often require some random decision. Different runs of the same algorithm can evolve in a very different way, so that the parallel independent execution of identical algorithms with different random seeds permits to explore the tail of the outcome distribution towards lower values.

- **Complete synchronization and sharing of information** — Some optimization algorithms can be modeled as parallel processes with shared memory. Processes can be coordinated in such a way that every single step of each process, i.e., decision on the next point to evaluate, is performed while considering information about all processes.

Between the two extremal cases, a wide spectrum of algorithms can be designed to perform individual searches with some form of loose coordination. An example is the "GOSH!" paradigm (Gossiping Optimization Search Heuristics) proposed in [52]. In this proposal, in order to distribute the Memory-Based Affine Shaker (MRAS) algorithm, every node maintains its own past history and uses it to model the function landscape and locate the best suitable starting point. Occasionally, pairs of nodes communicate and share relevant information about their past history in order to build a better common model.

38.9 Intelligent coordination of local search processes

Models of cultural evolution inspire a set of powerful optimization techniques known as **Memetic Algorithms (MAs)**. According to a seminal paper [275], memetic algorithms are population-based approaches that combine a fast heuristic to improve a solution (and even reach a local minimum) with a recombination mechanism that creates new individuals.

The fast heuristic to improve a solution is some form of **local search** (LS) already explained in Chapter 24. As explained, the motivation for the effectiveness of stochastic local search for many real-world optimization tasks lies in the *correlation between function values at nearby points*. The probability to find points with lower values is larger for neighbors of points which are *already* at low function values.

Although powerful, LS finds *locally optimal* points, which are not necessarily globally optimal. The paradigm is that of starting from a **basin of attraction** around a locally optimal point (or region) and generating a trajectory in the configuration space through a discrete dynamical system which is "flowing like a drop of water towards the bottom of the basin." Repeated LS, starting from different initial points, is a partial cure of the local minima problem but is completely *memory-less*: no information about previous searches influences future efforts.

In many cases a given optimization instance is characterized by *structure at different levels*, as explained with the big valley property of Fig. 24.7 in Chapter 24. If we reduce the initial search space to a set of attractors (the local minima), again it may be the case that nearby attractors – having an attraction basin close to each other – tend to have correlated values. This means that *knowledge* of previously found local optima can be used to direct the future investigation efforts. Starting from initial points close to promising attractors favors the discovery of other good quality local optima, provided of course that a sufficient diversification mechanism avoids falling back to previously visited ones.

In sequential local search the knowledge accumulated about the fitness surface flows from past to future searches, while in parallel processes with more local searchers active at the same time, knowledge is transferred by mutual sharing of partial results. We argue that the relevant subdivision is not between sequential and parallel processes (one can easily simulate a parallel process on a sequential machine) but between different ways of **using the knowledge accumulated by set of local search streams to influence the strategic allocation of computing resources** to the different LS streams, which will be activated, terminated, or modified depending on a shared knowledge base, either accumulated in a central storage, or in a distributed form but with a periodic exchange of information.

MAs fit in this picture, a set of individuals described by genes and subjected to genetic evolution scouts the fitness surface to search for successful initial points, while LS mechanisms (analogous to life-time learning) lead selected individuals to express their full potential by reaching local optima through local search. The **Genetic Algorithms** used in standard MAs follow the biological paradigms of selection/reproduction, cross-over and mutation. While GAs are effective for many problems, there is actually no guarantee that specific biologically-motivated genetic operators must be superior to **human-made direct mechanisms to share the knowledge accumulated** about the fitness surface by a set of parallel search streams (a.k.a. population). Alternative coordination mechanisms have been proposed for example in [363].

38.10 *C-LION*: a political analogy

We like analogies derived from the human experience more that analogies based on animals or genetics. Politics is a process by which groups of people make collective decisions. Groups can be governments, but also corporate, academic, and religious institutions. The issue is one of finding deliberate plans of action to guide decisions and achieve rational outcome(s). In politics one aims at making important organizational decisions, including the identification of spending priorities, and choosing among them on the basis of the impact they will have.

Local search is an effective building block for starting from an initial configuration of a problem instance and progressively building better solutions by moving to neighboring configurations. In an organized institution, like a corporation composed of individuals with intelligent problem-solving capabilities, each expert, when working on a tentative solution in his competence area, will after some time come up with an improved solution. The objective is to strategically allocate the work so that, depending on the accumulated performance of the different experts and competencies, superior solutions are obtained.

Memetic Algorithms start from local search and consider a hybridized genetic mechanism to implicitly accumulate knowledge about past local search performance by the traditional biologically-motivated mechanisms of selection/reproduction, mutation and cross-over. The first observation is that an individual can exploit its initial genetic content (its initial position) in a more directed and determined way. This is effected by considering the initial string as a *starting point* and by initiating a run of local search from this initial point, for example scouting for a local optimum. The term **memetic algorithms** [241, 275] has been introduced for models which combine the evolutionary adaptation of a population with individual learning within the lifetime of its members. The term derives from Dawkins' concept of a *meme* which is a unit of cultural evolution that can exhibit local refinement [112]. Actually, there are two obvious ways in which individual learning can be integrated: a first way consists of replacing the initial genotype with the better solution identified by local search (*Lamarckian evolution*), a second way can consist of modifying the fitness function by taking into account not the initial value but the final one obtained through local search. In other words, the fitness does not evaluate the initial state but the value of the "learning potential" of an individual, measured by the result obtained after local search. This evaluation changes the fitness landscape, while the evolution is still Darwinian in nature.

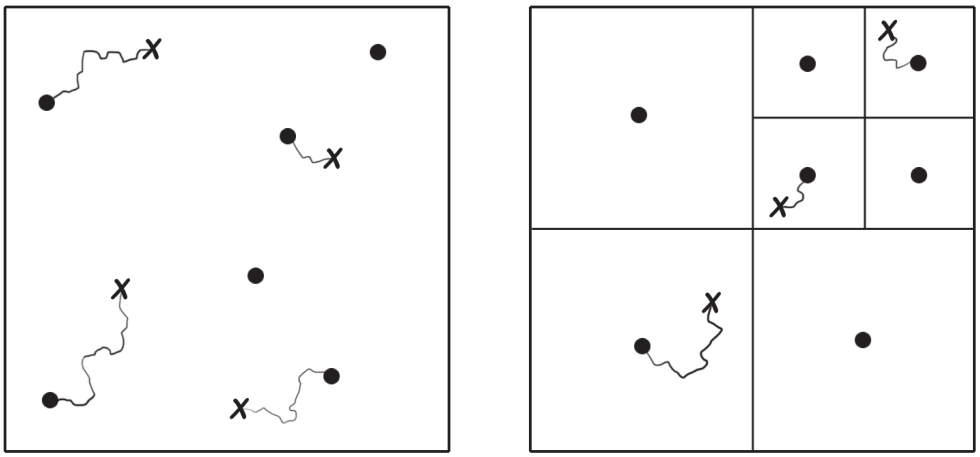


Figure 38.14: Different ways of allocating local searchers: by Memetic Algorithms (left) and by a political analogy (right). Crosses represent starting points, circles local optima reached after running local search. In the second case each individual is responsible for an area of configuration space. The political analogy is with territorial subdivisions given by electoral districts, or areas assigned to different marketing managers in a business example. Some local search streams are shown.

When the road of cultural paradigms is followed, it is natural to consider models derived from organizations of intelligent individuals equipped with individual learning and social interaction capabilities also in the **strategic allocation of resources** to the different search streams. In particular, the work in [41] presents a hybrid algorithm for the global optimization of functions (called continuous reactive tabu search), in which a fast combinatorial component (the Reactive Search Optimization based on prohibitions) identifies promising *districts* (boxes) in a tree-like partition of the initial search space, and a stochastic local search minimizer (the Reactive Affine Shaker — RAS — algorithm) finds the local minimum in a promising attraction basin. The social analogy can be that of organizing a marketing effort in a large company, see also Fig. 38.14: each individual (in the analogy a marketing manager) is responsible for a geographical area, the size of the geographical area is adapted to the interest of the different regions and the budget allocated to the different individuals is related to results obtained during their previous campaigns. A second political analogy is with a territorial subdivision given by electoral districts, again adapted to the interest of the different areas (population density) and again fighting for resources according to the area potentials.

But now it is time to stop with analogies and to consider the algorithms. The development of the *C-LION* framework is guided by the following design principles.

- **General-purpose optimization:** no requirements of differentiability or continuity are placed on the function f to be optimized.
- **Global optimization:** while the local search component identifies a local optimum in a given attraction basin, the combinatorial component favors jumps between different basins, with a *bias* toward regions that plausibly contain good local optima.
- **Multi-scale search:** the use of grids at different scales in a tree structure is used to spare CPU time in slowly-varying regions of the search space and to intensify the search in critical regions.
- **Simplicity, reaction and adaptation:** the algorithmic structure of *C-LION* is simple, the few parameters of the method are adapted in an automated way during the search, by using the information derived from memory. The dilemma between intensification and diversification is solved by using intensification until there is evidence that diversification is needed (when too many districts are repeated excessively often along the search trajectory). The tree-like discretization of the search space in districts is activated by evidence that the current district contains more than one attraction basin.
- **Tunable precision:** the global optimum can be located with high precision both because of the local adaptation of the grid size and because of the decreasing sampling steps of the stochastic RAS when it converges.

C-LION is characterized by an efficient use of *memory* during the search, as advocated by the Reactive Search Optimization (RSO). In addition, simple *adaptive* (*feedback*) mechanisms are used to tune the space discretization, by growing a *tree* of search districts, and to adapt the prohibition period of RSO acting on prohibitions. This adaptation limits the amount of user intervention to the definition of an initial search region, by setting upper and lower bounds on each variable, no parameters need to be tuned.

The *C-LION* framework based on Local Search **fuses Reactive Search Optimization with a problem-specific Local Search component**. An instance of an optimization problem is a pair $(\mathcal{X}, f)$, where $\mathcal{X}$ is a set of feasible points and f is the cost function to be *minimized*: $f : \mathcal{X} \to \mathbb{R}$. In the following we consider *continuous optimization* tasks where $\mathcal{X}$ is a compact subset of $\mathbb{R}^N$, defined by bounds on the N independent variables x_i, where $B_{Li} \leq x_i \leq B_{Ui}$ (B_L and B_U are the lower and upper bounds, respectively).

In many popular algorithms for continuous optimization one identifies a "local minimizer" that locates a local minimum by descending from a starting point, and a "global" component that is used to diversify the search and to reach the global optimum. We define as *attraction basin* of a local minimum X_l the set of points that will lead to X_l when used as starting configurations for the local minimizer.

In some cases, as we noted in our starting assumptions, an effective problem-specific local search component is available for the problem at hand, and one is therefore motivated to consider a **hybrid strategy**, whose local minimizer

has the purpose of finding the local minimum with adequate precision, and whose combinatorial component has the duty of discovering promising attraction basins for the local minimizer to be activated. Because the local minimizer is costly, it is activated only when the plausibility that a region contains a good local optimum is high. On the contrary, a fast evaluation of the search districts is executed by the combinatorial component, and the size of the candidate districts is adapted so that it is related to that of a single attraction basin. A district is split when there is evidence that at least two different local minima are located in the same district.

38.11 A C-LION example: RSO cooperating with RAS

We now briefly summarize a concrete example of the *C-LION* framework based on Local Search, called *continuous reactive tabu search (C-RTS)* in the original publication [41].

As local minimizer, the Reactive Affine Shaker described in Section 32.1 is used in this case, although the *C-LION* method can of course be developed with other local searchers with demonstrated efficiency on the problem at hand. In the hybrid scheme, *RSO* identifies promising regions for the local minimizer to be activated.

The initial search region is specified by bounds on each independent variable x_i, where $B_{Li} \leq x_i \leq B_{Ui}$, for $i = 1, ..., N$. The basic structure through which the initial search region is partitioned consists of a *tree of districts* (boxes with axes parallel to the coordinate axes). The tree is born with 2^N equal-size leaves, obtained by dividing in half the initial range on each variable. Each district is then subdivided into 2^N equally-sized children, as soon as two different local minima are found in it. Because the subdivision process is triggered by the local properties of f, after some iterations of *C-RTS* the tree will be of varying depth in the different regions, with districts of smaller sizes being present in regions that require an intensification of the search. Only the *leaves* of the tree are admissible search points for the combinatorial component of *C-RTS*. The leaves partition the initial region: the intersection of two leaves is empty, the union of all leaves coincides with the initial search space. A typical configuration for a two-dimensional task is shown in Fig. 38.15, where each leaf-district is identified by thick borders and a bold binary string.

Each existing district for a problem of dimension N is identified by a unique binary string B_S with $n \times N$ bits: $B_S = [g_{11}, ..., g_{1n}, ..., g_{N1}, ..., g_{Nn}]$. The value n is the depth of the district in the tree: $n = 0$ for the *root* district, $n = 1$ for the leaves of the initial tree (and therefore the initial string has N bits), n increases by one when a given district is subdivided. The length of the district edge along the i-th coordinate is therefore equal to $(B_{Ui} - B_{Li})/2^n$. The position of the district origin B_{Oi} along the i-th coordinate is

$$B_{Oi} = B_{Li} + (B_{Ui} - B_{Li}) \sum_{j=1}^{n} \frac{g_{ij}}{2^j}.$$

The evaluated neighborhood of a given district consists only of existing leaf-districts: no new districts are created during the neighborhood evaluation. Now, after applying the elementary moves to the identifying binary string B_S of a given district B, one obtains $N \times n$ districts of the same size distributed over the search space as illustrated in Fig. 38.15, for the case of $B_S = (1010, 1011)$. Because the tree can have different depth in different regions, it can happen that some of the obtained strings do *not* correspond to leaf-districts, others can cover *more* than a single leaf-district. In the first case one evaluates the smallest *enclosing* leaf-district, in the second case one evaluates a randomly-selected *enclosed* leaf-district. The random selection is executed by generating a point with uniform probability in the original district, and by selecting the leaf that contains the point. This assures that the probability for a leaf to be selected is proportional to its volume.

Evaluating opportunities for the different districts

While the RSO algorithm for combinatorial optimization generates a search trajectory consisting of points $X^{(t)}$, *C-RTS* generates a trajectory consisting of *leaf-districts* $B^{(t)}$. There are two important changes to be underlined: firstly, the function $f(X)$ must be substituted with a routine measuring the potential that the current district contains good local optima, secondly, the tree is *dynamic* and the number of existing districts grows during the search.

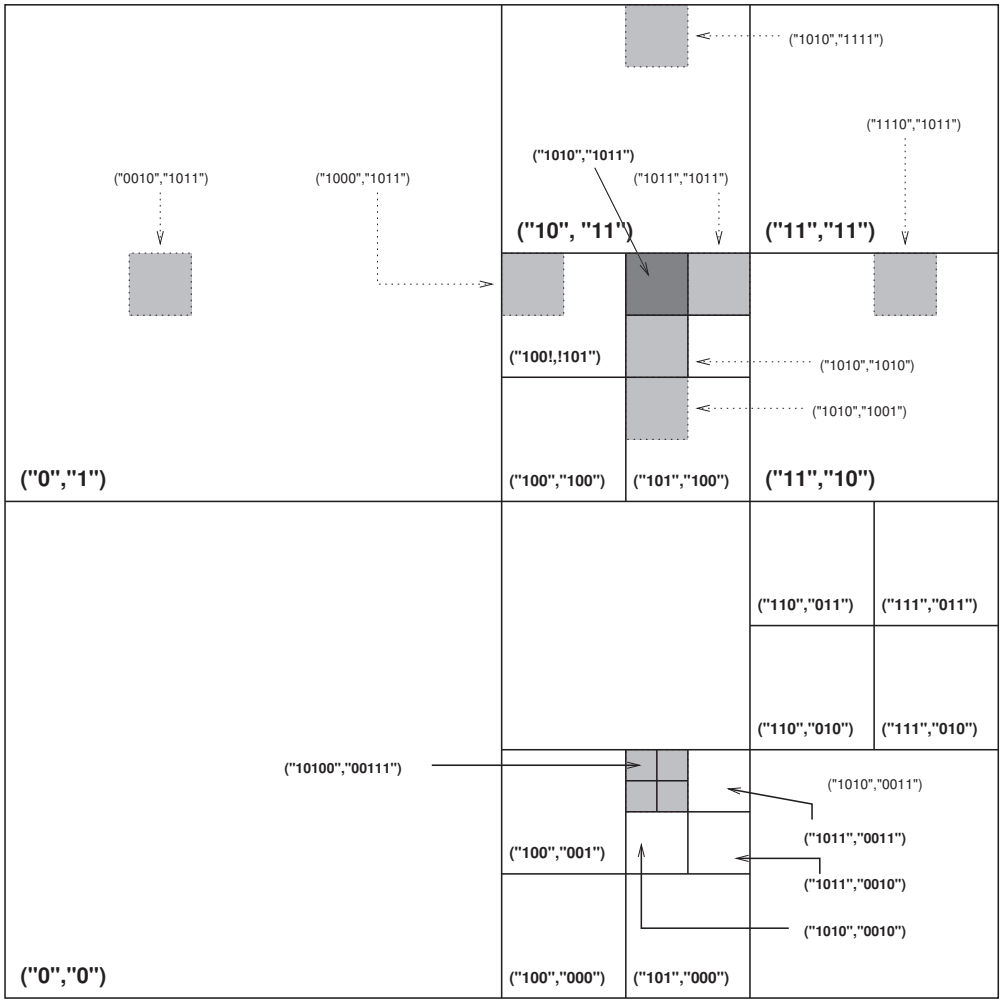


Figure 38.15: *C-RTS*: tree of search *districts*. Thick borders and bold strings identify existing leaf-districts, hatched districts show the neighborhood of district (1010,1011).

The combinatorial component must identify promising districts quickly. In the absence of detailed models about the function f to be minimized, a simple evaluation of a district B can be obtained by generating a point X with a uniform probability distribution inside its region and by evaluating the function $f(X)$ at the obtained point. Let us use the same function symbol, the difference being evident from its argument: $f(B) \equiv f(rand\ X \in B)$. The potential drawback of this simple evaluation is that the search can be strongly biased in favor of a district in the case of a "lucky" evaluation (e.g., $f(X)$ close to the minimum in the given district), or away from a district in the opposite case. To avoid this drawback, when a district is encountered again during the search, a *new* point X is generated and evaluated and some collective information is returned. The value $f(B)$ returned is then the *average* of the evaluated X_i: $f(B) \equiv (1/N_B) \sum_{i=1}^{N_B} f(X_i)$, where N_B is the number of points.

Let us consider the example of Fig. 38.15. The current district (1010,1011) has the neighbors shown with a hatched pattern. The neighbor (0010,1011) in the upper left part is not an existing leaf-district, it is therefore transformed into the enclosing existing leaf-district (0,1). *Vice versa* the neighbor (1010,0011) in the lower right part contains four leaves, one of them (10100,00111) is the output of a random selection. Fig. 38.16 specifies the complete final

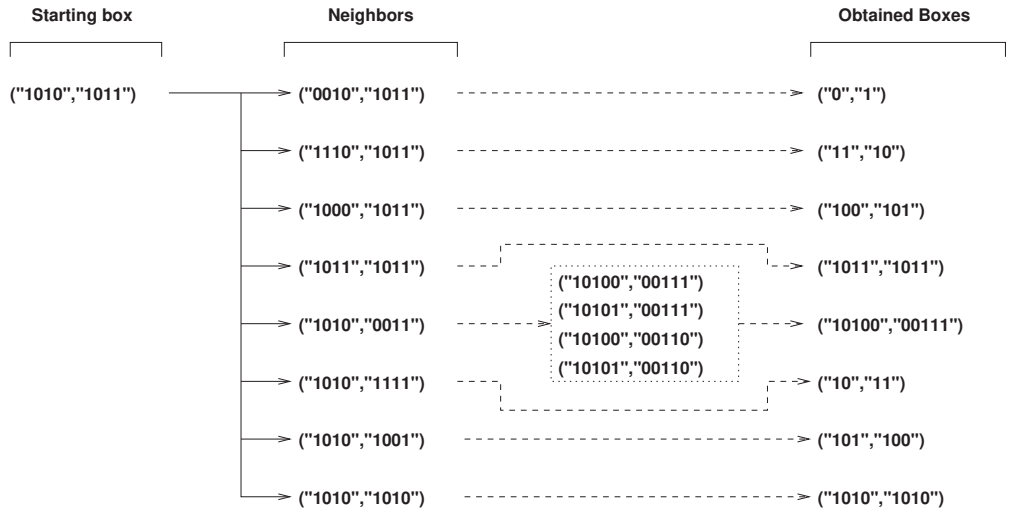


Figure 38.16: *C-RTS*: Evaluation of the neighborhood of district (1010,1011).

neighborhood obtained for the given example.

Decision about activating Local Search in a given region

According to the RSO dynamics the neighborhood districts obtained starting from a current district are evaluated only if the corresponding basic move from the current point is not prohibited. Only if the evaluation $f(B^{(t)})$ of the current district is less than all evaluations executed in the neighborhood, a decision is taken about the possible triggering of the Local Search component (the Reactive Affine Shaker). In other words, a necessary condition for activating high-precision and expensive searches with Local Search is that there is a high plausibility — measured by $f(B)$ — that the current region can produce local minima that are better with respect to the given neighborhood of candidate districts. Given the greedy nature of the combinatorial component, the current district $B^{(t)}$ on the search trajectory moves toward non-tabu locally optimal districts, therefore it will eventually become locally optimal and satisfy the conditions for triggering RAS. Let us note that, if a given district B loses the above contest (i.e., it is not locally optimal for RSO), it *maintains* the possibility to win when it is encountered again during the search, because the evaluation of a different random point X can produce a better $f(B)$ value. Thanks to the evaluation method *C-RTS* is *fast in optimal conditions*, when the f surface is smooth and $f(B)$ is a reliable indicator of the local minimum that can be obtained in region B, but it is *robust in harder cases*, when the $f(B)$ values have a high standard deviation or when they are unreliable indicators of good local minima obtainable with the Reactive Affine Shaker.

The local optimality of the current district B is necessary for activating RAS but it is not sufficient, unless B is locally optimal for the first time, a case in which RAS is always triggered. Otherwise, if $r > 1$ is the number of times that district B has been locally optimal during the search, an additional RAS run must be justified by a sufficient probability to find a *new* local minimum in B. Bayesian rules to estimate the probability that all local optima have been visited can be applied in the context of a single district, where a multi-start technique is realized with repeated activations of RAS from uniformly distributed starting points. Because of our splitting criterion, at most one local optimum will be associated to a given district (a district is split as soon as two different local optima are found, see Section 38.11). In addition, some parts of the district can be such that RAS will exit the borders if the initial point belongs to these portions. One can therefore partition the district region into W components, the attraction basins of the local minima contained in the district and a possible basin that leads RAS outside, so that the probabilities of the basins sum up to one ($\sum_{w=1}^{W} P_w = 1$).

According to [58], if $r > W + 1$ restarts have been executed and W different cells have been identified, the total

relative volume of the "observed region" (i.e., the posterior expected value of the relative volume Ω) can be estimated by

$$E(\Omega|r,W) = \frac{(r-W-1)\,(r+W)}{r\,(r-1)}\,; \qquad r > W+1. \tag{38.21}$$

The Reactive Affine Shaker is always triggered if $r \leq W+1$, because the above estimate is not valid in this case, otherwise the RAS is executed again with probability equal to $1 - E(\Omega|r,W)$. In this way, additional runs of RAS tend to be spared if the above estimate predicts a small probability to find a new local optimum, but a new run is never completely prohibited for the sake of robustness: it can happen that the Bayesian estimate of equation (38.21) is unreliable, or that the unseen portion $(1 - E(\Omega|r,W))$ contains a very good minimum with a small attraction basin.

The initial conditions for RAS (described in Fig. 32.1) are that the initial search point is extracted from the uniform distribution inside B, the initial search frame is $\vec{b}_i = \vec{e}_i \times (1/4) \times (B_{Ui} - B_{Li})$ where $\vec{e}_i$ are the canonical basis vectors of $\mathbb{R}^N$. The Reactive Affine Shaker generates a trajectory that must be contained in the district B enlarged by a border region of width $(1/2) \times (B_{Ui} - B_{Li})$, and it must converge to a point contained in B. If RAS exits the enlarged district or the root-district, it is terminated, the function evaluations executed by RAS are discarded. If it converges to a point outside the original district but inside the enlarged district, the point location is saved. In both cases the *C-RTS* combinatorial component continues in the normal way: the next district $B^{(t+1)}$ is the best one in the admissible neighborhood of $B^{(t)}$. In any case the "best so far" value is always updated by considering all admissible points evaluated (those that are inside of the root-district).

A possible exception to the normal *C-RTS* evolution can happen only in the event that RAS converges inside $B^{(t)}$ to a local minimum X_l. If X_l is the first local minimum found, it is saved in a memory structure associated to the district. If a local minimum Y_l was already present, and X_l corresponds to the same point, it is discarded, otherwise the current district is *split* until the "siblings" in the tree divide the two points. After the splitting is completed, the current district $B^{(t)}$ does not correspond to an existing leaf anymore: to restore legality a point is selected at random with uniform distribution in $B^{(t)}$ and the legal $B^{(t)}$ becomes the leaf-district that contains the random point. Therefore each leaf-district in the partition of the initial district has a probability of being selected that is proportional to its volume. The splitting procedure is explained in the following section.

Adapting the district area to the local fitness surface

As soon as two different local minima X_l and Y_l are identified in a given district B, the current district is subdivided into 2^N equal-sized boxes. If X_l and Y_l belong to two different leaf-districts of the new partition, the splitting is terminated, otherwise the splitting is applied to the district containing X_l and Y_l, until their separation is obtained.

In all cases the old district ceases to exist and it is substituted with the collection obtained through the splitting. The local minima X_l and Y_l are associated with their new boxes. Numerically, the criterion used in the tests for considering *different* two local minima X_l and Y_l is that $\|X_l - Y_l\| < \epsilon$, where ϵ is a user-defined precision requirement.

All local minima identified are saved and reported when *C-RTS* terminates.

An example of the tree structure produced during a run of *C-RTS* is shown in Fig. 38.17, for the case of a two-dimensional function (a "Strongin" function described in [41]). The local optima are clearly visible as "mountain tops." One notices that the points evaluated (the points used to calculate $f(B)$) are distributed quasi-uniformly over the search space: this is a result of the volume-proportional selection, and it guarantees that all regions of the search space are treated in a fair manner. The RAS trajectories either converge to a local minimum (bullet) or are terminated when they exit from the enlarged district, as explained in Section 38.11. Because of our splitting criterion, each district contains at most one local minimum. Although it is not visible from the figure, most points (about 85% in the example) are evaluated during the local search phases, that are the most expensive parts of the *C-RTS* algorithm.

38.12 Other C-LION algorithms

C-LION is **a paradigm to increase automation**, by using different optimization building blocks (or different instances of randomized algorithms), and by coordinating and managing them in an intelligent adaptive manner.

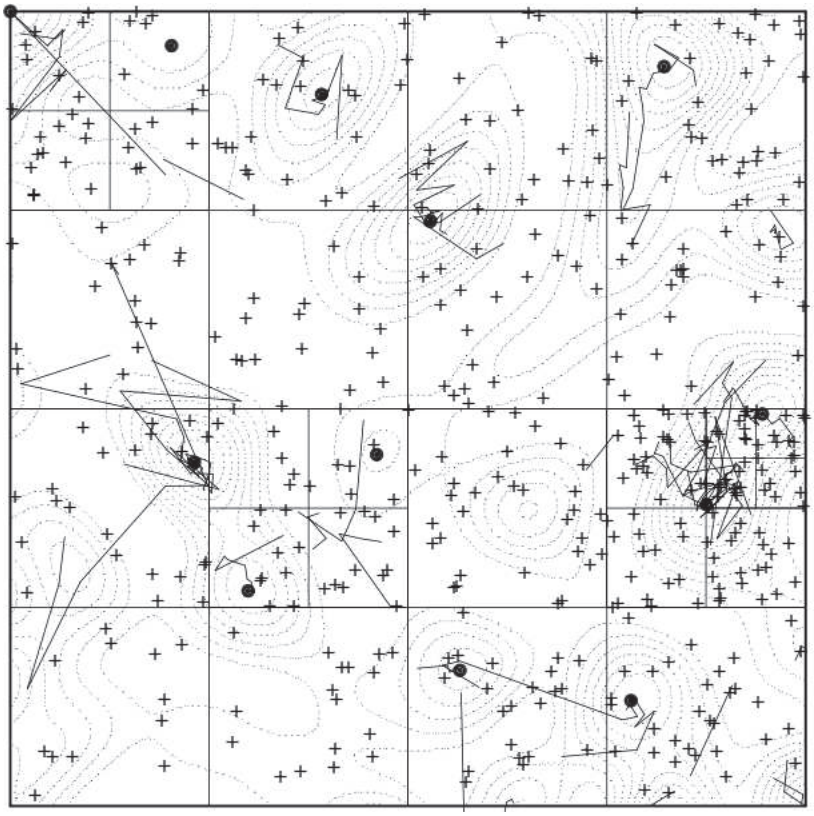


Figure 38.17: A *C-RTS* tree structure adapted to a fitness surface and calculated points. Evaluated points (crosses), local minima (bullets), LS trajectories (jagged lines). Figure derived from [41].

The cooperation structure exemplified by C-RTS can be used with different local search components, or different details about splitting the original space and firing local searches.

The work [70] proposes a variation of C-RTS (called *CoRSO*)in which the Inertial Shaker method generates candidate points in an adaptive search box and a moving average of the steps filters out evaluation noise and high-frequency oscillations. Finally, a portfolio of independent search streams (*P-CoRSO*) is proposed to increase the robustness of the algorithm.

Other notable examples are strategies like the '**Divide-the-Best**' algorithms based on Lipschitz assumptions and on efficient diagonal partitions and the **P-algorithm with simplicial partitioning** [390].

If $f(x)$ satisfies the Lipschitz condition over the search hyper-interval with an unknown Lipschitz constant K, a deterministic 'Divide-the-Best' algorithm based on efficient diagonal partitions of the search domain and smooth auxiliary functions is proposed in [329]. The method adaptively estimates the unknown Lipschitz constant K and the objective function and its gradient are evaluated only at two vertices corresponding to the main diagonal of the generated hyperintervals.

The second case assumes that the functions satisfies some statistical model, so that theoretically justified methods can be developed, in the framework of rational decision making under uncertainty, to generate new sample points based on information derived from previous samples, and to study convergence properties [390]. The P-algorithm [393] generates the next point to be evaluated as the one maximizing the probability to improve the current record, given the previously observed samples. In multiple dimensions, if y_{on} is the current record value and (x_i, y_i) are the

previous evaluated points and corresponding values, the next $(n+1)$-th optimization step is defined as:

$$x_{n+1} = \underset{x}{\text{argmax}} \Pr\Big\{\xi(x) \leqslant (y_{on} - \epsilon) \mid \xi(x_1) = y_1, ..., \xi(x_n) = y_n\Big\}.$$

The P-algorithm with simplicial partitioning is proposed in [394] to obtain a practical algorithm. The observation points coincide with the vertices of the simplices and different strategies for defining an initial covering and subdividing the interesting simplices are proposed and considered. The P^*-algorithm, combining the P-algorithm with local search, is related to the algorithm presented in this paper, which is also based on a combination of global models and efficient local searches when the current area is deemed sufficiently interesting.

Gist

Many optimization problems of real-world interest are complex and need enormous computing times for their solution. Luckily, there are often **many different algorithms** to try, different by their design or by the value of their meta-parameters. In addition, the use of many **computers working in parallel** (maybe living in the cloud, rented when they are needed) comes to the rescue to reduce the clock time to produce acceptable solutions.

In some cases, one can consider **independent search streams**, periodically reporting the best solutions found so far to some central coordinator. You should never ever underestimate the power of simple solutions!

In other cases, more **intelligent schemes of coordination** among the various running algorithms lead to higher automation, and better efficiency and effectiveness. The C-LION paradigm proposes to coordinate and manage a team of interacting optimization algorithms through adaptation based on intelligent reflection on their current and past results.

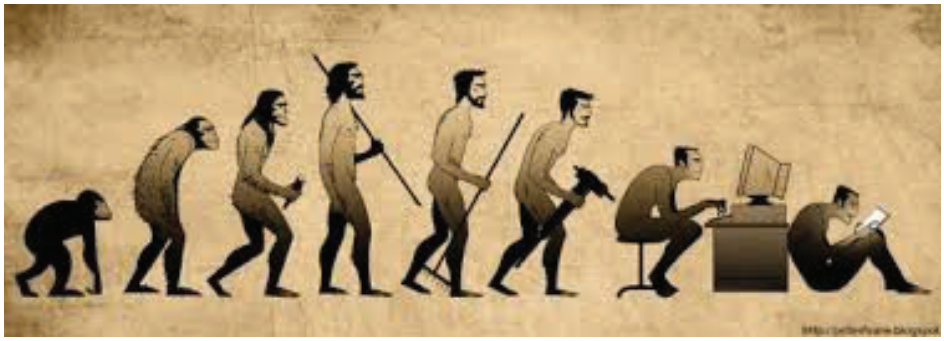
Paradigms derived from human organizations, characterized by "learning on the job" capabilities, can lead to superior results with respect to paradigms derived from simpler living organisms or genetic principles.

Sane human people solve complex problems better than viruses, normally with less deadly consequences. Flies do not learn a lot during their lives, and easily end up burnt by incandescent light bulbs. Kids need to touch a hot bulb once to become aware and avoid doing that again in the future.

Chapter 39

Genetics, evolution and nature-inspired analogies

What a book a devil's chaplain might write on the clumsy, wasteful, blundering, low, and horribly cruel work of nature!
(Charles Darwin)



Population-based algorithms using multiple search streams became popular by using analogies derived from genetics, nature, and evolution. This emphasis on the imitation of very simple living organism is not always justified and more effective schemes can be developed by considering more explicit learning and self-tuning schemes. But if you are selling optimization, be aware of the surprising effect of nature-inspired analogies. In this area, biological analogies derived from the behavior of different species abound [341]. Elegant flocks of birds search for food or migrate in effective manners, herds of sheep get better guidance and protection than isolated members. Groups of hunting animals can often prevail over much bigger and powerful but isolated ones.

Analogies from nature can be inspiring but also misleading when they are translated directly into procedures for problem solving and optimization. Let's consider a flock of birds or an ant colony searching for food. If an individual finds food, it makes perfect sense for the survival of the species to inform other members so that they can also get their share of nutrients. The analogy between food and good solutions of an optimization problems is not only far-fetched but quite simply wrong. If one searcher already found a good suboptimal solution, attracting other searchers in the same attraction basin around the locally optimal point only means wasting precious computational resources which could be spent by exploring different regions of the search space. One encounters here the basic tradeoff between intensification and diversification.

The adoption of a set of interacting search streams has a long history, not only when considering natural evolution, but also heuristics and learning machines. It is not surprising to find very similar ideas under different names, including ensembles, pools, agents, committees, multiple threads, mixture of experts, genetic algorithms, particle swarms,

evolutionary strategies. The terms are not synonymous because each parish church has specific strong beliefs and true believers.

39.1 Genetic algorithms and evolution strategies

A rich source of inspiration for adopting a set of evolving candidate solutions is derived from the theory natural evolution of the species, dating back to the original book by Charles Darwin [108] "On the Origin of Species by Means of Natural Selection, or the preservation of favored races in the struggle for life." It introduced the theory that populations evolve over the course of generations through a process of **natural selection**: individuals more suited to the environment are more likely to survive and more likely to reproduce, leaving their inheritable traits to future generations. After the more recent discovery of the genes, the connection with optimization is as follows: each individual is a candidate solution described by its genetic content (genotype). The genotype is randomly changed by mutations, the suitability for the environment is described by a *fitness function*, which is related to the function to be optimized. Fitter individuals in the current population produce a larger offspring (new candidate solutions), whose genetic material is a recombination of the genetic material of their parents.

Seminal works include [17, 303, 131, 319, 186]. A complete presentation of different directions in this huge area is out of the scope of this section, let's concentrate on some basic ideas, and on the relationships between GA and intelligent search strategies.

Let's consider an optimization problem where the configuration is described by a binary string, mutation consists of randomly changing bit values with a fixed probability Π_{mute} and recombination consists of the so called *uniform cross-over*: starting from two binary strings X and Y a third string Z is derived where each individual bit is copied from X with probability $1/2$, from Y otherwise

The pseudo-code for a slightly more general version of a genetic algorithm, with an additional parameter for cross-over probability Π_{cross} is shown in Fig. 39.1, and illustrated in Fig. 39.2. After the generation of an initial population P, the algorithm iterates through a sequence of basic operations: first the fitness of each individual is computed and, if goals are met, the algorithm stops. Some individuals are chosen by a random selection process that favors elements with a high fitness function; a crossover is applied to randomly selected pairs in order to combine their features, then some individuals undergo a random mutation. The algorithm is then repeated on the new population.

Let the population of candidate solutions be a set of configurations scattered on the fitness surface. Such configurations explore their neighborhoods through the mutation mechanism: usually the mutation probability is very low, so that in the above example a small number of bits is changed. After this very primitive form of local (perturbative) search move the population is substituted by a new one, where the better points have a larger probability to survive and a larger probability to generate offspring points. With a uniform cross-over the offspring is generated through a kind of "linear interpolation" between the two parents. Because a real interpolation is excluded for binary values, this combination is obtained by the random mechanism described: if two bits at corresponding positions in X and Y have the same value, this value is maintained in Z - as it is in a linear interpolation of real-valued vectors - otherwise a way to define a point in between is to pick randomly from either X or Y if they are different.

There are at least three critical issues when adopting biological mechanisms for solving optimization problems: first one should demonstrate that they are effective - not so easy because defining the function that biological organisms are optimizing is far from trivial. One risks a circular argument: survival of the fittest means that the fittest are the ones who survived. Second, one should demonstrate that they are efficient. Just consider how many generations were needed to adapt our species to the environment. Third, even assuming that GA are effective, one should ask whether natural evolution proceeds in a Darwinian way because it is intrinsically superior or because of **hard biological constraints**, which may be masochistic to keep in a human-designed algorithm.

For example, it is now believed that Lamarck was wrong in assuming that the experience of an individual could be passed to his descendants: **genes do not account for modifications caused by lifelong learning**. Airplanes do not flap their wings and, in a similar manner, when a technological problem has to be solved, one is free to depart from the biological analogy and to design the most effective method with complete freedom.

Initialization — Compute a random population with M members $P = \{s^{(j)} \in S^\ell, j = 0, \ldots, M-1\}$, where each string is built by randomly choosing ℓ symbols of S.

Repeat :

Evaluation — Evaluate the fitness $f^{(i)} = f(s^{(i)})$; compute the rescaled fitness $\bar{f}^{(j)}$:

$$f_{\text{min}} = \min_j f^{(j)}; \quad f_{\text{max}} = \max_j f^{(j)}; \quad \bar{f}^{(j)} = \frac{f^{(j)} - f_{\text{min}}}{f_{\text{max}} - f_{\text{min}}}$$

Test — If the population P contains one or more individuals achieving the optimization goal within the requested tolerance, stop the execution.

Stochastic selection — Build a new population $Q = \{q^{(j)}, j = 0, \ldots, N-1\}$ such that the probability that an individual $q \in P$ is member of Q is given by $f(q)/\sum_{p\in P} f(p)$:

Reproduction — Choose $N/2$ distinct pairs $(q^{(i)}, q^{(j)})$ using the N individuals of Q. For each pair build, with probability Π_{cross}, a new pair of offsprings mixing the parents' genes, otherwise copy the original genes:

for $i \leftarrow 0, \ldots, (M-1)/2$
  if Rand(1) < Π_{cross}
    for $j \leftarrow 0, \ldots, l-1$
      if Rand(1) < .5
        $\bar{q}_j^{(2i)} \leftarrow q_j^{(2i)}$; $\bar{q}_j^{(2i+1)} \leftarrow q_j^{(2i+1)}$
      else
        $\bar{q}_j^{(2i)} \leftarrow q_j^{(2i+1)}$; $\bar{q}_j^{(2i+1)} \leftarrow q_j^{(2i)}$
  else
    $\bar{q}^{(2i)} \leftarrow q^{(2i)}$; $\bar{q}^{(2i+1)} \leftarrow q^{(2i+1)}$

Mutation — In each new individual $\bar{q}^{(j)}$ change, with probability Π_{mutate}, each gene $\bar{q}_i^{(j)}$ with a randomly chosen gene of S. Let us denote the new population Q'.

Replacement — Replace the population P with the newly computed one Q'.

Figure 39.1: Pseudocode for a popular version of GA.

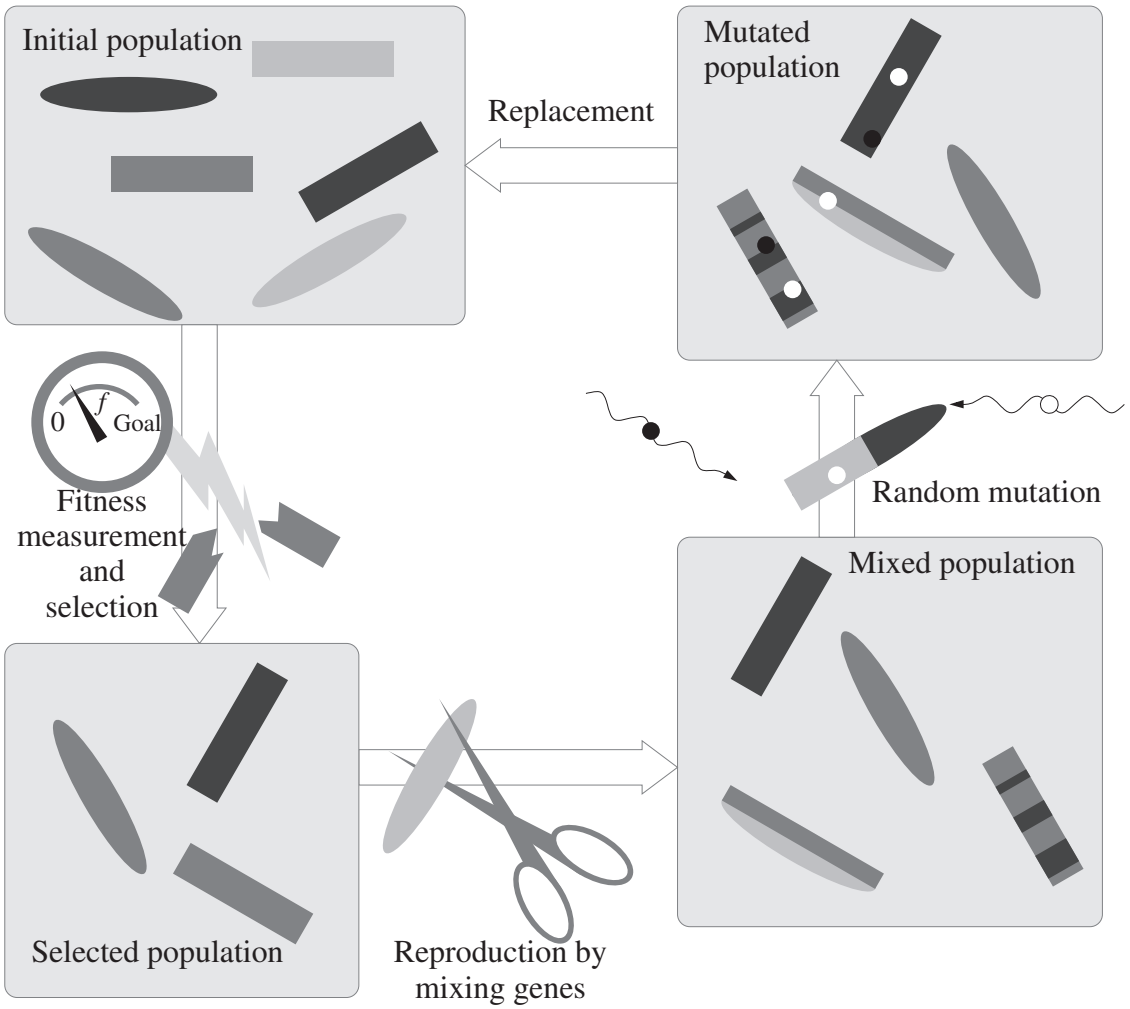


Figure 39.2: Representation of a genetic algorithm framework. Counter-clockwise from top left: starting from an initial population, stochastic selection is applied, then genes are mixed and random mutations occur to form a new population.

A second departure from the biological world is as follows: in biology one is interested in the convergence of the entire population/species to a high fitness value, in optimization one aims at having at least *one* high-fitness solutions *during* the search, not necessarily at the end, and one could not care less whether most individuals are far from the best when the search is terminated.

When using the stochastic local search language adopted in this book, the role of the mutation/selection and recombination operators cannot be explained in a clear-cut manner. When the mutation rate is small, the effect of the combined **mutation and selection of the fittest** can be interpreted as searching in the neighborhood of a current point, and accepting (selecting) the new point in a manner proportional to the novel fitness value. The behavior is of intensification/exploitation provided that the mutation rate is small, otherwise one ends up with a random restart, but not too small, otherwise one is stuck with the starting solution. The explanation of **cross-over** is more dubious. Let's consider uniform cross-over: if the two parents are very different, the distance between parents and offspring is large so that cross-over has the effect of moving the points rapidly on the fitness surface, while keeping the most stable bits constant and concentrating the exploration on the most undecided bits, the ones varying the most between members of the population. But if the similarity between parents is large, the cross-over will have little effect on the offspring, the final danger being that of a *premature convergence* of the population to a suboptimal point. The complexity inherent in explaining Darwinian metaphors for optimization makes one think whether more direct terms and definitions should be used [107] ("metaphors are not always rhetorically innocent").

At this point, you may wonder whether the term "team member" is justified in a basic GA algorithm. After all, each member is a very simple individual indeed, it comes to life through some randomized recombination of its parents and does a little exploration of its neighborhood. If it is lucky by encountering a better fitness value in the neighborhood, it has some probability to leave some of its genetic material to its offspring, otherwise it is terminated. No memory is kept of the individuals, only **a collective form of history is kept through the population**. Yes, "team member" *is* exaggerated. But now let's come back to the issue "airplanes do not flap their wings" to remember that as computer scientists and problem solvers our design freedom is limited only by our imagination.

We may imagine at least two different forms of *hybridized genetic algorithms*. The first observation is that, in order to deserve its name, a team member can execute a more directed and determined exploitation of its initial genetic content (its initial position). This is effected by considering the initial string as a *starting point* and by initiating a run of local search from this initial point, for example scouting for a local optimum. Lamarck can have his revenge here, now nobody prohibits substituting the initial individual with its ameliorated version after the local search. The term **memetic algorithms** [275, 241] has been introduced for models which combine the evolutionary adaptation of a population with individual learning within the lifetime of its members. The term derives from Dawkins' concept of a *meme* which is a unit of cultural evolution that can exhibit local refinement [112].

Actually, there are two obvious ways in which individual learning can be integrated: a first way consists of replacing the initial genotype with the better solution identified by local search (**Lamarckian evolution**), a second way can be of modifying the fitness by taking into account not the initial value but the final one obtained through local search. In other words, the fitness does not evaluate the initial state but the value of the "learning potential" of an individual, measured by the result obtained after the local search. This has the effect of changing the fitness landscape, while the resulting form of evolution is still Darwinian in nature. This and related forms of combinations of learning and evolution are known as the **Baldwin effect** [179, 381].

An interesting observation in [390] is that having two parents to generate a descendant may actually lead to less efficient algorithms with respect to simple versions of population-based algorithms in which each descendant has only one parent. Airplaned do not flap their wings to fly, in a similar manner efficient population-based algorithms do not need to imitate nature for the sake of imitating it, therefore leaving more freedom to the algorithm designer to experiment with effective novel ideas.

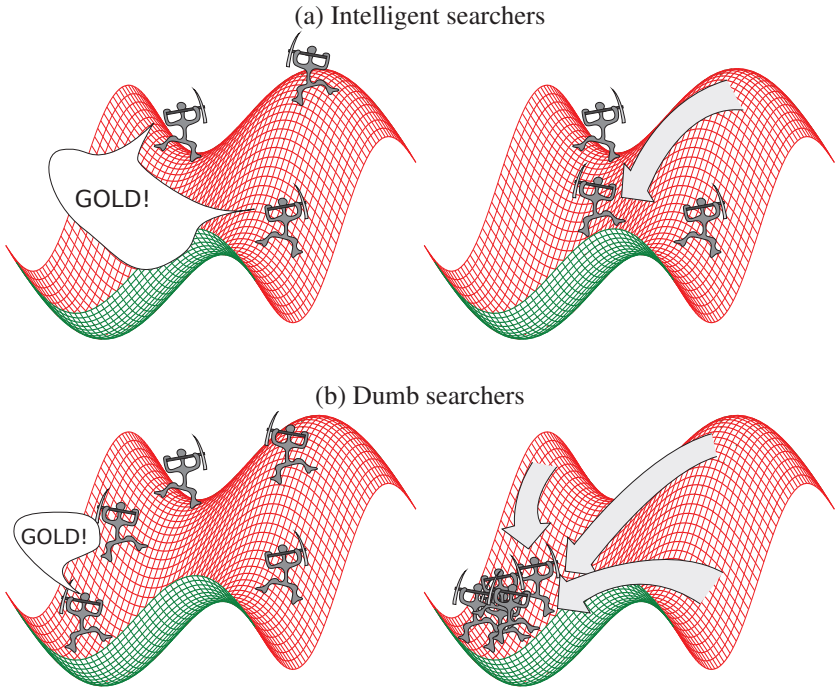


Figure 39.3: Opportunities and pitfalls of teams of searchers: an intelligent searcher may decide to move between two successful ones (top panes), while dumb searchers (bottom) swarm to the same place, so that diversification can be lost. If you were a 49er, which model would you choose (and, besides, would you scream your findings)?

Gist

There is little doubt that considering **more search processes during optimization with some form of coordination** is a useful higher level, going beyond a single search process.

There are more **doubts on the intrinsic usefulness of specific analogies from nature**, genetics and evolution to develop effective state-of-the-art algorithms.

For sure, these analogies have a positive effect on the marketing of optimization and may help in converting new researchers to the growing area of optimization heuristics.

On the other hand, if one cares about scientific progress, one should be completely free to abandon analogies - based on nature or culture - in order to design the most effective algorithms.

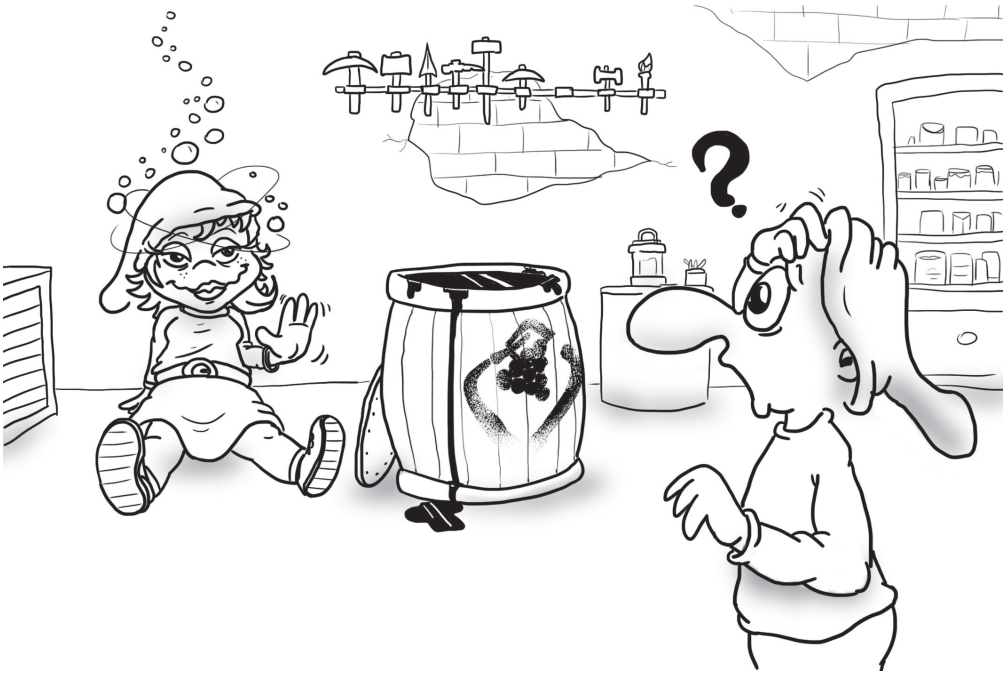
In particular, one should remember that viruses do not learn a lot while they are living, and that by using memory and machine learning strategies, one can greatly improve most primitive techniques, even if the final method does not use sex for mating individuals and is therefore less sexy.

Everybody would like that math studied during lifetime could be passed to his children via genes, but biological constraints so far failed to implement a Lamarckian mechanism: the passage occurs by more direct training mechanisms! Feel free to improve this sad state of affairs in your algorithms.

Chapter 40

Multi-Objective Optimization

Non si può avere la botte piena e la moglie ubriaca.
One cannot have a full wine-barrel and a drunk wife.
(Italian proverb)



Life is full of compromises, popular wisdom says that one cannot "ride two horses with one ass." Most of the real-world problem cannot be cast into the simple and mathematically crystal-clear form of minimizing a function $f(x)$.

There are two crucial difficulties. First, most problems have **more than one objective** to reach, to be maximized. This is the case in **multi-objective optimization problems** (**MOOP**), for which many *conflicting* objectives have to be traded off in selecting the preferred choice. Most real-world problems are of this kind. When you buy a car, you have different objectives in mind: speed, cost, size, etc. and you have your own way of weighing the different objectives. If you bought a Ferrari, your preferences are probably different from someone who bought a city car. If you are searching for a partner, different combinations of beauty and intelligence are available for the possible candidates

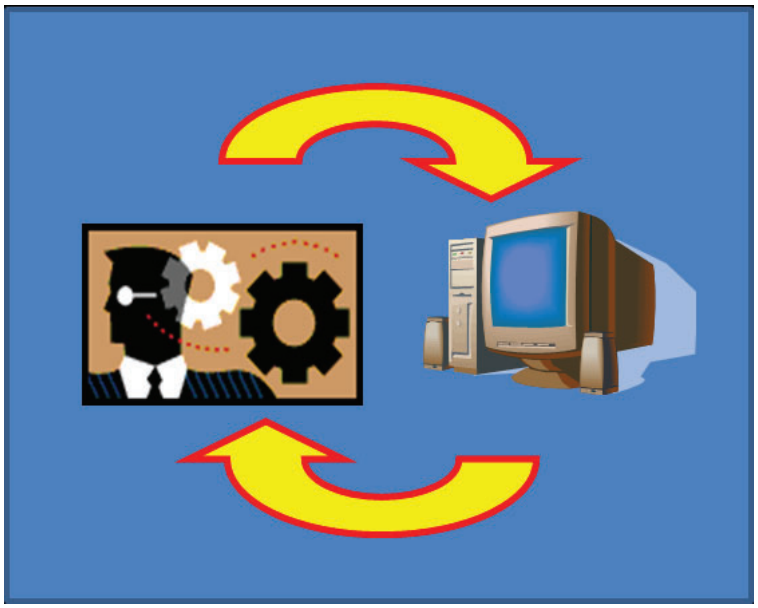


Figure 40.1: Problem solving and optimization frequently involves an iterative process with crucial learning steps.

(agreed, this is a coarse simplification!). Unfortunately, it is rare that a candidate simultaneously maximizes both parameters, **compromises and tradeoffs** must be accepted.

Even if an optimal abstract combination of two or more objectives into an overall *utility function* to be maximized exists, **obtaining a function in a closed mathematical form can be very difficult**, if not impossible. Try asking your best friend (better not to ask directly your partner): "Can you give me the utility function describing your best combination of beauty and intelligence?", or, if you limit your model to linear combinations, "Can you tell me your weights to combine beauty and intelligence?" Problem solving and optimization techniques often deliver a large number of potential solutions. Design process innovation, virtual prototyping, business process engineering are some examples. The decision maker is left with the crucial task of identifying a preferred solution, taking into account explicitly defined objectives (one or more mathematical functions to maximize), hard and soft constraints, and **preferences which are not explicit but often crucial for an intelligent decision**.

Problem-solving is often an **iterative process with learning**, as illustrated in Fig. 40.1. A learning path occurs between two entities: the decision maker and the supporting software system. The decision maker analyzes some representative solutions, learning about concrete possibilities and updating his objectives. The software memorizes the user preferences and shifts the focus of attention to regions of the design/solution space which are deemed more relevant by the final user. The iterative process is continued until a satisfactory solution is found or patience is exhausted.

MORSO (multi-objective Reactive Search Optimization) denotes solution methods intended for **multi-objective optimization characterized by incremental and learning paths**. Learning takes places in the user mind and in the solution algorithms. A closely related term is that of interactive multi-objective optimization, but we intend to stress systematic, automated and online learning techniques in a more direct manner.

In the following sections, the concept of Pareto-optimality is formally defined in Sec. 40.1 and the main solution techniques are presented in Sec. 40.2. Finally, a way to keep the final decision maker in the loop as a provider of learning signals is illustrated in Sec. 40.4 about brain-computer optimization.

Figure 40.2: Individual objectives in Pareto-optimization are building blocks, but the preferred combination is not given (image courtesy of LEGO).

40.1 Multi-objective optimization and Pareto optimality

In the classic case of multi-objective optimization problems (MOOPs), the user specifies a set of m desirable objectives, but he does not spell out the tradeoffs, the relative importance of the different objectives, the proper combination of them into an overall utility function. A MOOP can be stated as:

$$\begin{aligned} \text{minimize} \quad & \mathbf{f}(\mathbf{x}) = \{f_1(\mathbf{x}), \ldots, f_m(\mathbf{x})\} \\ \text{subject to} \quad & \mathbf{x} \in \Omega, \end{aligned}$$

where $\mathbf{x} \in \mathbb{R}^n$ is a vector of n decision variables; $\Omega \subset \mathbb{R}^n$ is the *feasible region*, typically specified as a set of constraints on the decision variables. In the above example of looking for a suitable partner, Ω could be the set of all persons of a given sex (male or female) who can at least read and write. You will not even consider a partner who does not satisfy these constraints.

The vector $\mathbf{f} : \Omega \to \mathbb{R}^m$ is made of m objective functions which need to be jointly minimized[1]. Objective vectors are images of decision vectors and can be written as $\mathbf{z} = \mathbf{f}(\mathbf{x}) = \{f_1(\mathbf{x}), \ldots, f_m(\mathbf{x})\}$. The above problem is ill-posed whenever **objective functions are conflicting**, a frequent situation in real-world contexts. In these cases, an objective vector is considered optimal if none of its components can be improved without worsening at least one of the others. An objective vector $\mathbf{z}$ is said to *dominate* $\mathbf{z}'$, denoted as $\mathbf{z} \prec \mathbf{z}'$, if $z_k \leq z'_k$ for all k and there exist at least one h such that $z_h < z'_h$. A point $\hat{\mathbf{x}}$ is **Pareto-optimal** if there is no other $\mathbf{x} \in \Omega$ such that $\mathbf{f}(\mathbf{x})$ dominates $\mathbf{f}(\hat{\mathbf{x}})$. Fig. 40.3 illustrates the concept. The **Pareto frontier** (or Pareto front, **PF** for short) consists of all Pareto-optimal points. In the example, a partner is Pareto-optimal if no other partner is at the same time nicer *and* more intelligent, or nicer with the same intelligence level, etc. As you immediately recognize, considering only Pareto-optimal candidates makes sense: no rational person would ever prefer a dominated partner! By restricting attention to the Pareto frontier (the set of choices that are Pareto-efficient), a designer can make tradeoffs within this set, rather than considering the full range of every parameter. "Rob Peter to pay Paul" well expresses the concept related to modifying a solution in a way that makes some aspect better but some aspect worse, producing no net gain for all objectives.

Vilfredo Pareto had the courage to cross boundaries between disciplines. After graduating in Civil Engineering while working in Florence, he was among the first to analyze economic problems with mathematical tools [291]. In

[1] In the case of levels of beauty and intelligence, which must clearly be *maximized*, just minimize their opposites.

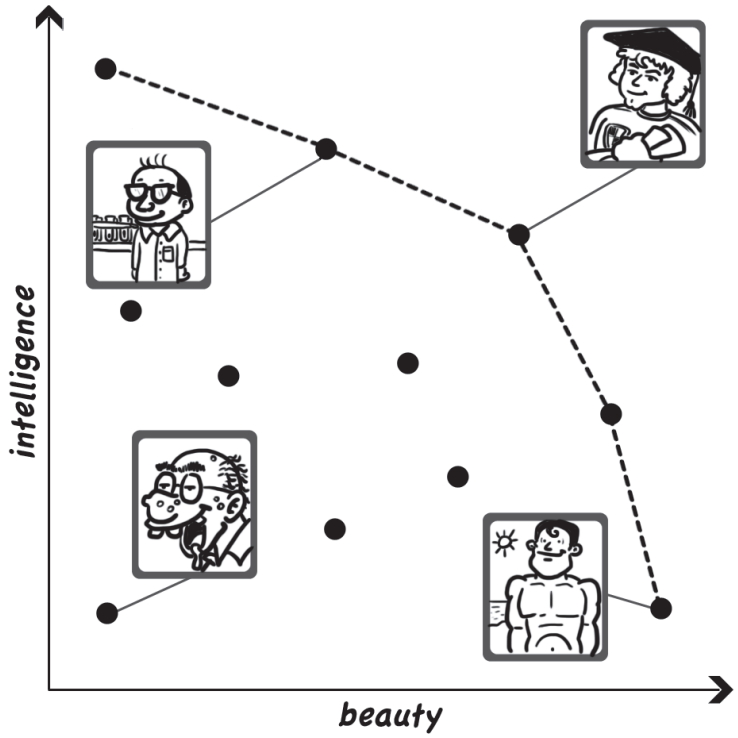


Figure 40.3: Pareto optimality. All dominated points like the persons in the middle are not considered as potential candidates for the final choice. On the Pareto frontier, shown with a dashed line, tradeoffs need to be considered.

1893, he became the Chair of Political Economy at the University of Lausanne in Switzerland, where he defined his concept of Pareto-optimality: "The optimum allocation of the resources of a society is not attained so long as it is possible to make at least one individual better off in his own estimation while keeping others as well off as before in their own estimation." After the translation of Pareto's Manual of Political Economy into English, Stadler [344] applied the notion of Pareto Optimality to the fields of engineering and science in the middle 1970's. Then the applications of multi-objective optimization in engineering design grew rapidly [123] and is now a tool used in most engineering companies.

Notable examples of Pareto-optimization are in economics (consumer demand and indifference curves, production possibilities frontier, macroeconomic policy-making), in finance (optimal portfolios maximizing return and minimizing risk, or variance of return), in engineering (engine design, controller design, product and process optimization, radio resource management, electric power systems, etc.).

40.2 Pareto-optimization: main solution techniques.

Let's mention some solution techniques to solve the MOOP defined in equation (40.1).

As mentioned, a Pareto-optimal solutions is one that cannot be improved in any of the objectives without degrading at least one of the other objectives. In mathematical terms, a feasible solution $x^1 \in \Omega$ is said to (Pareto) dominate another solution $x^2 \in \Omega$, if

$$f_i(x^1) \leq f_i(x^2) \text{for all indices} i \in \{1, 2, \ldots, k\}$$

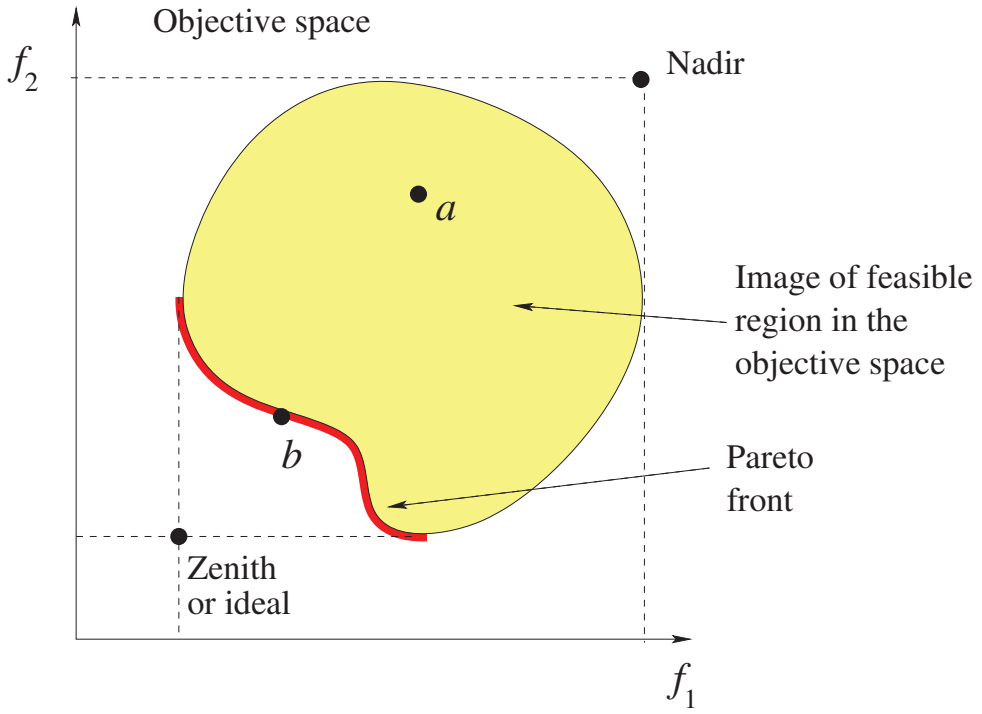


Figure 40.4: Point b belongs to the Pareto front: no other feasible solution is strictly better in one objective and at least as good for the other ones (here minimization is assumed). Point a doesn't.

and

$$f_j(x^1) < f_j(x^2) \text{for at least one index} j \in \{1, 2, \ldots, k\}\,.$$

A solution $x^1 \in \Omega$ is called Pareto optimal, if there does not exist another solution that dominates it.

The Pareto front of a multi-objective optimization problem is bounded by a so-called **nadir objective vector** z^{nad} and a **zenith** (or ideal) **objective vector** z^{zen}, defined as follows:

$$z_i^{\text{nad}} = \sup_{x \in X \text{ is Pareto optimal}} f_i(x) \text{ for all } i = 1, \ldots, k \tag{40.1}$$

$$z_i^{\text{zen}} = \inf_{x \in X} f_i(x) \text{ for all } i = 1, \ldots, k. \tag{40.2}$$

When finite, the components of a *nadir* and an *ideal* objective vector define upper and lower bounds for the objective function values of Pareto optimal solutions. The nadir objective vector can only be approximated as, typically, the whole Pareto optimal set is unknown. In addition, a **utopian objective vector** z^{utopian} can be defined because of numerical reasons as:

$$z_i^{\text{utopian}} = z_i^{\text{zen}} - \epsilon \text{ for all } i = 1, \ldots, k, \tag{40.3}$$

where $\epsilon > 0$ is a small constant.

A standard way to deal with multiple objective is to obtain a single one by combining them (this is called **aggregation by utility functions**, utility being the combined objective). **Linear scalarization** (also called **weighted-sum**) build a linear utility function. The aggregated problem to solve is defined as:

$$\min_{x \in X} \sum_{i=1}^{k} \lambda_i \, f_i(x), \tag{40.4}$$

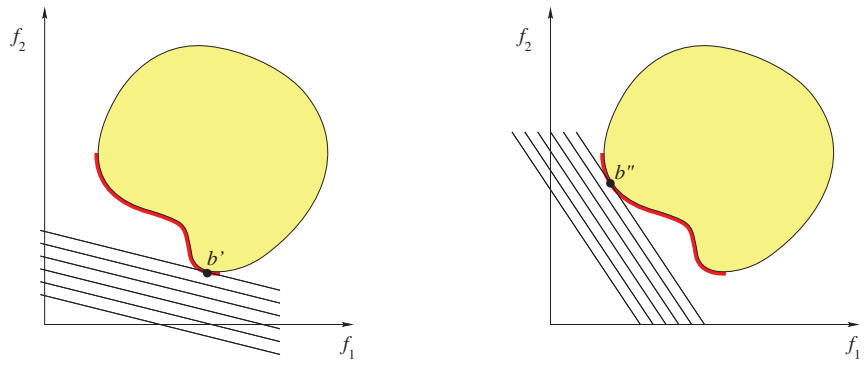


Figure 40.5: Two different scalarizations identify different preferred solutions on the PF.

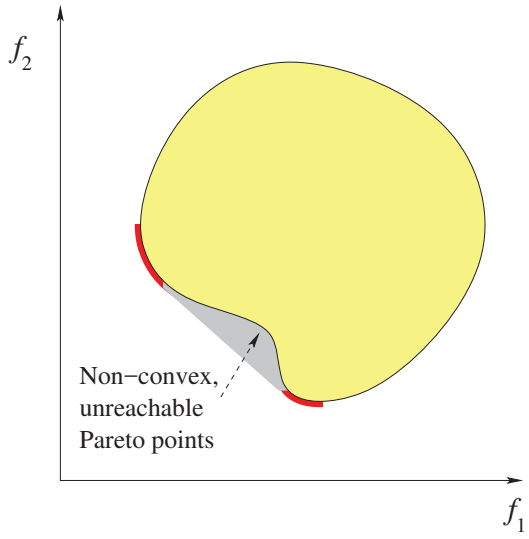


Figure 40.6: Scalarization cannot identify some Pareto-optimal solutions (unless the PF is convex).

where the weights of the objectives $\lambda_i > 0$ are the parameters ("weights") of the scalarization. Needless to say, the solution depends on the weights. Because what matters are relative sizes and not absolute values, weights are usually normalized: $\sum_{i=1}^{k} \lambda_i = 1$.

To visualize the process, one is approaching the Pareto front with a line (a plane, a hyperplane) oriented in a certain manner. When the line touches a Pareto-optimal solution, the contact point is picked as the preferred solution (Fig. 40.5). A Pareto optimal solution is called **supported-efficient** if it is an optimal solution to equation (40.4) with a particular weight vector [232]. If the Pareto-front is not convex, some Pareto-optimal solutions *cannot* be identified by solving (40.4) with some weights, so that the scalarization technique must be used with some care.

One can define a neighborhood of a locally or globally optimal solution to (40.4), and then identify more Pareto optimal solutions via some form of local search, with a reasonable amount of computational effort

In **Tchebycheff approach** [269] the scalar utility function is in the form:

$$\min_{x \in X} \max_{i=1}^{k} \{\lambda_i | f_i(x) \; - \; z^{ideal}{}_i |\}, \tag{40.5}$$

where z^{ideal} is the reference point defined above. For each Pareto-optimal point x^* there exists a weight vector such

that x^* is the optimal solution of (40.5) and each optimal solution of (40.5) is a Pareto optimal solution. One weakness with this approach is that its aggregation function is not smooth for a continuous MOOP, and the simpler scalarization is often preferred.

To obtain an approximation to the PF via a certain number of Pareto-optimal points, one can solve **a set of the above single objective problems** with different carefully selected weight coefficient vectors. This **decomposition** idea has been successfully used in MOEA/D [389]. MOEA/D decomposes a multiobjective optimization problem into a number of scalar optimization subproblems, using a set of different weight vectors λ, one for each scalarized problem, and optimizes them simultaneously (Fig.40.7). By modifying local search to consider Pareto-optimality and the existence of more Pareto-optimal solutions, each subproblem can be improved in an iterative manner by using information from its several neighboring subproblems. A subproblem is a neighbor if their defining weight vectors are close.

Pareto local search (PLS) is a natural extension of single objective local search methods [290, 257]. It works with a set of mutually nondominated solutions, explores some or all of the neighbors of these solutions to find new solutions for updating this set at each iteration. PLS exploits local search algorithms for single-objective problems, and solves a multi-objective problem by **chains of related aggregations** and **chains of good solutions** for aggregations. Good solutions provide starting points for a search regarding a next aggregation. After finding an initial set of supported efficient solutions, the objective is to identify *non-supported* efficient solutions located between the supported efficient ones. In PLS, the neighborhood of every solution of a population is explored, and *if the neighbor is not dominated* by a solution of the list, the neighbor is added to the population. The exclusion of dominated neighbors is the basic modification w.r.t. standard LS.

The work [232] combines ideas from evolutionary algorithms, decomposition approaches, and **Pareto local search**. It suggests a simple yet efficient memetic algorithm for combinatorial multiobjective optimization problems: **memetic algorithm based on decomposition (MOMAD)**. It decomposes a combinatorial multiobjective problem into a number of single objective optimization problems using an aggregation method. MOMAD evolves three populations: 1) population PL for recording the current solution to each subproblem; 2) population PP for storing starting solutions for Pareto local search; and 3) an external population PE for maintaining all the nondominated solutions found so far during the search. At each generation, a Pareto local search method is first applied to search a neighborhood of each solution in PP to update PL and PE. Then a single objective local search is applied to each perturbed solution in PL for improving PL and PE, and reinitializing PP. MOMAD provides a generic hybrid multiobjective algorithmic framework in which problem specific knowledge, well developed single objective local search and Pareto local search methods can be hybridized. It is a population-based iterative method and thus an anytime algorithm.

40.3 MOOPs: how to get missing information and identify user preferences

Pareto-optimization is caused by **lack of complete information** in the definition of the problem. Some information is present, given by a set of positive objectives to reach (the individual $f_i(x)$ functions, a sort of "building blocks"), but information about how the different objectives have to be combined is missing. The tradeoffs are not yet solved, otherwise one would come up with a single combined objective.

When discussing about **the role of the final user (decision maker) in providing information to pick a preferred solution** among the Pareto front, one distinguishes the three following possibilities.

- **A priori methods** require that sufficient preference information is expressed before the solution process for example by picking weights in the the linearly-scalarized utility function method of equation (40.4), or by setting the goal in **goal programming** [251, 81]. In goal programming, a desired target is set for the solution vector, the single-objective problem associated becomes that of minimizing the deviation between the obtained solution and the target (deviation measured with a suitable metric). The drawback of *a priori* methods is that the DM often does not know before how realistic his expectations are, and that there are no subsequent learning possibilities.

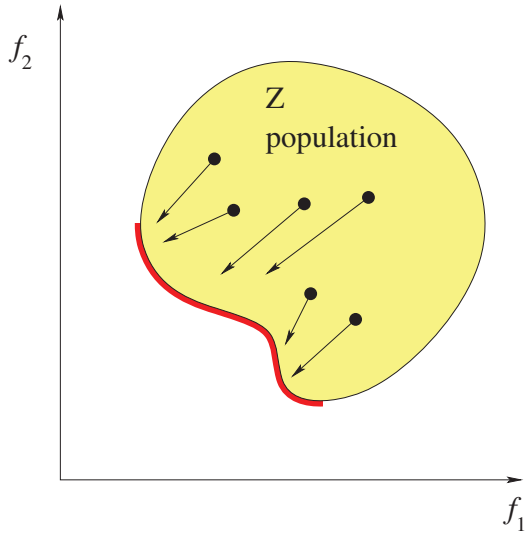


Figure 40.7: Population-based approaches (e.g., based on decomposition) can be used to obtain a set of representative solutions along the Pareto-front. The differens subproblems are iteratively improved to move them closer to the PF.

- **A posteriori methods** aim at producing all the Pareto optimal solutions or a representative subset thereof. The set is then presented to the decision maker (DM) for selecting the preferred solution. These methods demand a lot of computing and place a heavy burden on the shoulders of the DM. A human person can typically choose one among a limited set (say 10) solution, but he has difficulty in choosing one among one million of alternatives!
- In **Interactive methods**, the solution process is iterative and the decision maker continuously interacts with the method when searching for the most preferred solution. The decision maker is expected to express preferences at each iteration in order to get Pareto optimal solutions that are of interest to him and **learn what kind of solutions are attainable**.

40.4 Brain-computer optimization (BCO): the user in the loop

As mentioned, asking a user to quantify his **utility function** (for example by choosing weights of a linear combination of the different objectives) *a priori*, before seeing the actual optimization results, is challenging. If the final user is cooperating with an optimization expert, misunderstanding between the persons may arise. This problem can become dramatic when the number of the conflicting objectives in MOOP increases. As a consequence, the final user may be dissatisfied with the solutions, because some of the objectives remain hidden in his mind. Different drawbacks are present in *a posteriori* approaches, which deliver to the final user a representative set of solutions on the entire Pareto front so that he can pick his most preferred solution.

Although a user may have difficulties in providing explicit weights and mathematical formulas, for sure he can *evaluate the returned solutions*. In most cases, the situation improves with **an interactive process between the final user and the optimizer to change the definition of the problem**. The optimizer will then be run over the new version of the problem. This process may be iterated an arbitrary number of times, as shown in Fig. 40.1.

We are now entering the most advanced and exciting topic of this book: the integration of analytics, visualization and optimization. Abstract solution points are vectors of numbers which convey a specific **meaning**.

In **interactive problem-solving** sessions, the user can get information about a specific solution by calling a problem-specific routine. This routine, which has to be provided for the different applications, can be used to visualize detailed information about a specific point, e.g., through a graphical display of a solution.

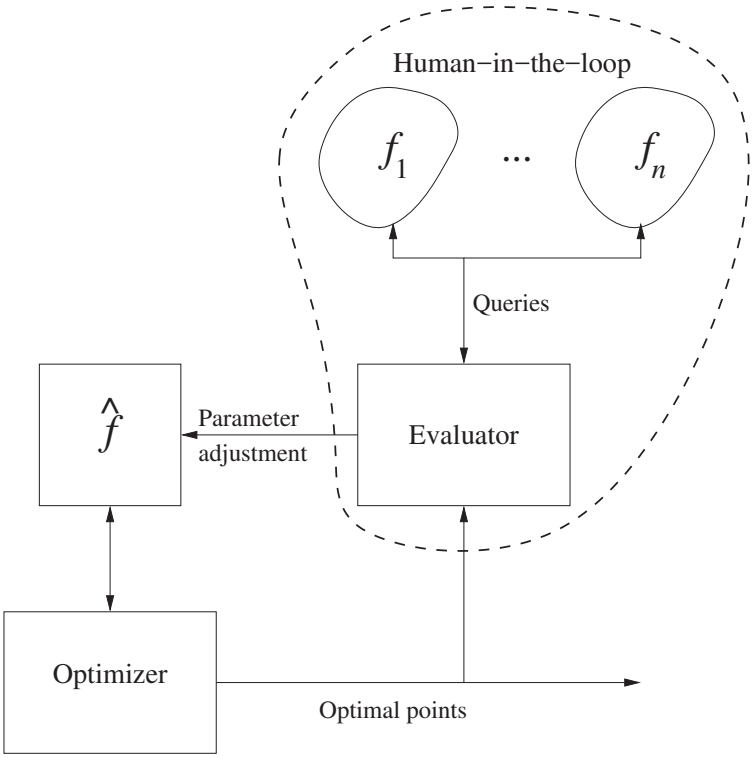


Figure 40.8: Brain-Computer Optimization: learning the problem definition from the final user in the case of interactive multi-objective optimization (adapted from [28]).

The same problem-specific routine can be used to accept **feedback** about the specific solution, such as a personal evaluation, as illustrated in Fig. 40.8.

As a rule of thumb, most of the problem-solving effort in the real world is spent on **defining the problem**, on specifying in a computable manner the function to be optimized. After this modeling work is completed, optimization becomes in certain cases a commodity. The implication for researchers and developers is that much more effort should be devoted to design supporting techniques and tools to help the final user, often without expertise in mathematics and in optimization, to define and refine the function to be optimized so that it corresponds to his real objectives. Think about defining your favorite weights for beauty versus intelligence while searching for a partner. If somebody asks you for quantitative ways to specify the tradeoff before starting the search, you may feel very embarrassed (we hope!). Only after seeing some examples you may clarify your weights and objectives.

Reactive Search Optimization is dedicated to online learning techniques to support the quest for a solution by **self-adapting a local search method in a manner depending on the previous history of the search**. The learning signals consist of data about the structural characteristics of the instance collected while the algorithm is running. For example, data about sizes of basins of attraction, entrapment of trajectories, repetitions of previously visited configurations. The algorithm learns by interacting with a previously unknown environment given by an existing (and fixed) problem definition.

We argue that there is a second interesting online learning loop, where learning signals originate from a final user and are intended to **modify and refine the problem definition** itself. This context is potentially very wide, depending on the amount of knowledge about the problem given *a priori*, on the allowed modifications, on the kind of questions asked.

An example of Interactive Multi-Objective Optimization with RSO is described in [28]. The methods share with

the work in [395, 113, 346] the interaction with the final user realized via pairwise comparison of solutions, but tackle a broader class of problems with nonlinear preference functions. This case is of interest because many (maybe most) decision problems are *nonlinear*, to reflect our preference for reasonable compromise solutions. A presentation of the state of the art for learning arbitrary (*nonlinear*) models is in [32].

We focus here on the simpler and classical linear case [28] and the goal consists of **learning the non-dominated solution preferred by the final user**. Assume that the user provides the different objectives in the MOOP problem, but he cannot quantify the **weights** of the different objectives before seeing the actual optimization results. The system aims at learning the weights vector $\boldsymbol{w} = (w_1, w_2, \ldots, w_m)$ optimizing the linear combination g:

$$g(\boldsymbol{x}, \boldsymbol{w}) = w_1 f_1(\boldsymbol{x}) + w_2 f_2(\boldsymbol{x}) + \cdots + w_m f_m(\boldsymbol{x}).$$

In a more compact form:

$$g(x_1, x_2, \ldots, x_n, \boldsymbol{w}) = \boldsymbol{f}(\boldsymbol{x})^T \boldsymbol{w},$$

where $\boldsymbol{f} = (f_1, f_2, \ldots, f_m)$. Without loss of generality, assume that g must be minimized.

The feedback from the decision maker at each iteration is obtained by presenting two solutions and asking him to indicate his favorite solution between them, if any. This is the simplest question to be asked, a qualitative overall preference. If the decision maker cannot answer, he should probably get another job. The preferences stated by the final user are translated into *constraints* that the weights must satisfy. This guarantees that the obtained utility function is consistent with the user's judgments. If $\boldsymbol{a} = (a_1, a_2, \ldots, a_n)$ and $\boldsymbol{b} = (b_1, b_2, \ldots, b_n)$ are the two solutions provided by the system, the preference $\boldsymbol{a} \prec \boldsymbol{b}$ of the final user for the solution **a** with respect to the solution **b** is represented by the following constraint:

$$g(\boldsymbol{a}, \boldsymbol{w}) < g(\boldsymbol{b}, \boldsymbol{w}).$$

Therefore, a new linear constraint on the weights is generated for each question asked to the final user. The problem of learning the user preference then becomes that of finding a solution $\boldsymbol{w}$ for the set of constraints on the weights generated by the user's feedback.

The weights are initialized with random values in the interval (0,1), and then normalized so that their sum is 1. At each iteration, two non-dominated solutions $\boldsymbol{a}$ and $\boldsymbol{b}$ are compared by the final user. Both solutions are obtained by minimizing a linear combination of the objective functions of the input problem:

$$\min_{\boldsymbol{x}} g(\boldsymbol{x}, \boldsymbol{w}).$$

In particular, the first solution **a** is obtained by using the current weights vector $\boldsymbol{w}^{\text{curr}}$ found by applying the *middlemost weights* technique [297], by solving the following linear programming problem:

$$\begin{aligned} \max_{\boldsymbol{w}} \quad & \gamma \\ \text{subject to} \quad & \begin{cases} g(\boldsymbol{a}, \boldsymbol{w}) \leq g(\boldsymbol{b}, \boldsymbol{w}) - \gamma & \forall \boldsymbol{a} \prec \boldsymbol{b} \\ w_i \geq \gamma & \forall i = 1, \ldots, m \\ \gamma \geq 0. \end{cases} \end{aligned}$$

The meaning of the above is to search for a weight which is *consistent* but also *far from the boundaries* of the consistent region. The bigger the positive γ parameter, the safer the inequalities. Even if limited noise is added (for example caused by physical quantities with measurements errors) and the g values are slightly changed, there is still a *safety margin* before the direction of the inequality is changed.

The second solution **b** can be obtained by using the weights vector $\boldsymbol{w}^{\text{pert}}$, generated by perturbing $\boldsymbol{w}^{\text{curr}}$, and by ensuring that the two generated solutions are sufficiently dissimilar. Issues related to considering possible infeasible sets of linear constraints (which can be generated by a confused decision maker, or because a linear approximation is a too rough) are considered in [28]. The more complex non-linear case is solved with machine learning techniques based on the Support Vector Machine method in [32].

A novel approach based on active learning of Pareto fronts (ALP) is presented in [74]. ALP casts the identification of the Pareto front into a supervised machine learning task. The computational effort in generating the supervised information is reduced by an active learning strategy. In particular, the model is learned from a set of informative training objective vectors.

To finish this introductory presentation, remember, if you have a challenging problem to solve, the *source of power of intelligent optimization* will help you both to *define* exactly what you want to accomplish and to actually *compute* one or more solutions.

In many cases some decisions can be, and maybe should be, deferred until some initial possible solutions are evaluated by an expert user.

Gist

When visiting a business as a consultant, a healthy carrier of the traditional math-oriented approach to optimization will ask the typical question "What is the function that you want to optimize in your business?" By "function" he means an explicit mathematical model, a formula relating inputs (decisions to be made) to output (like profit) without any ambiguity. This attitude and the lack of clearly defined models for most businesses probably explains why the power of optimization is still stifled in the real world.

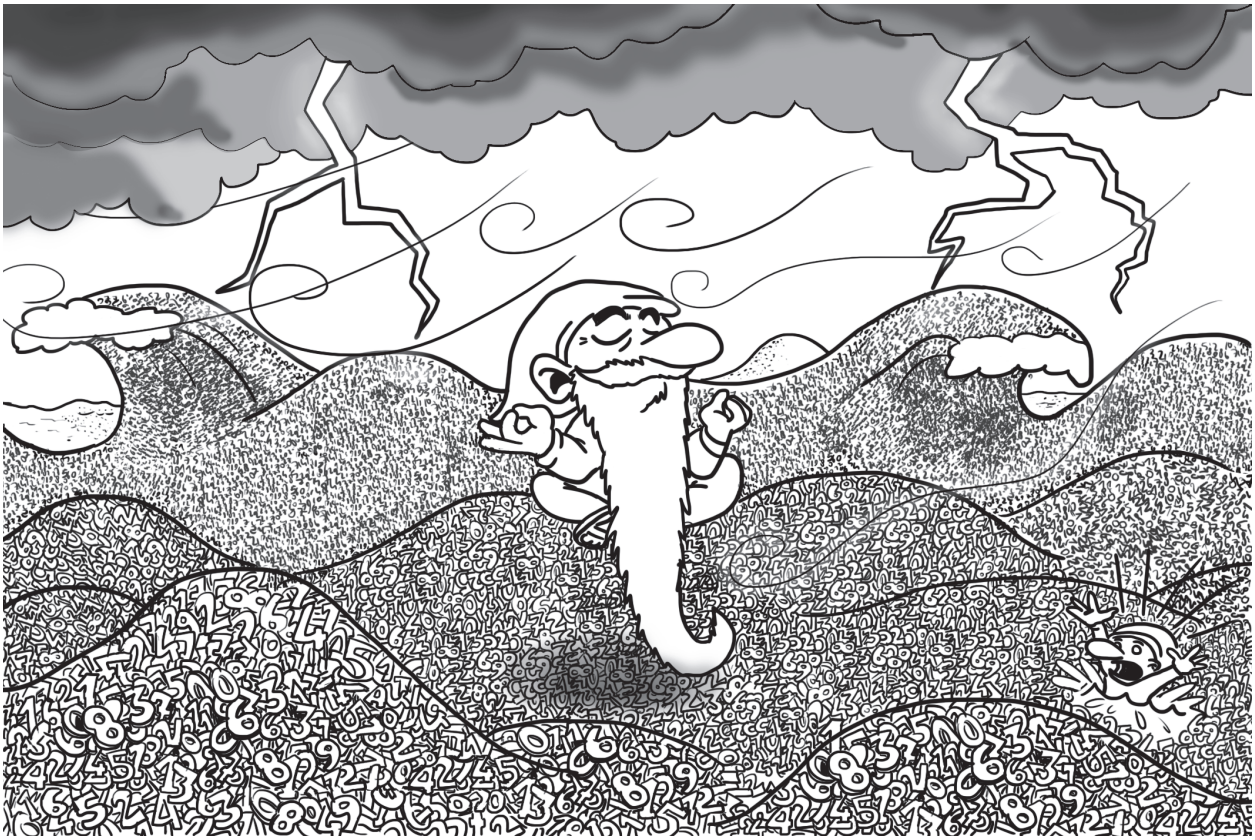
After the business owner tells him "Sorry, I have no mathematical function," the **LION way** opens a window of hope and opportunity to unleash the power of optimization. He can reply with "Do not worry, even if you cannot give me your model, I can build it for you **from your data and your feedback**." Using a personal computer to support decision-making should not lead you to scrap your expert personal brain.

Most problem-solving and optimization efforts are intrinsically **iterative processes with learning involved**, learning from data, and learning from the decision makers. When this is acknowledged, a bright new era of opportunities is ushered in. A lot of effort is still required, which is *good* news for data scientists, but the road ahead is mapped.

Chapter 41

Conclusion

Rem tene, verba sequentur
Master the data, verbal interpretations will follow
(attributed to Cicero and Cato)



Congratulations for reaching this point. You have now in your pockets powerful tools to build models from data, to understand and explain, to identify improvements and to create disruptive new solutions, in a never-ending loop leading to better products and better services.

It is time to read the introduction again, and check if it means something different now. Contact or visit us, we are

happy to know that there are other *sane* people with a *insane* passion for data mining, building models, optimizing, and developing new ideas and business methods in the process.

Exit your building and use your knowledge to build a better world.

Figure 41.1: Ciclo dei mesi, Trento, January, circa 1397

Bibliography

[1] E. H. L. Aarts, J.H.M. Korst, and P.J. Zwietering. Deterministic and randomized local search. In M. Mozer P. Smolensky and D. Rumelhart, editors, *Mathematical Perspectives on Neural Networks*. Lawrence Erlbaum Publishers, Hillsdale, NJ, 1995, to appear.

[2] E.H.L. Aarts and J.H.M. Korst. Boltzmann machines for travelling salesman problems. *European Journal of Operational Research*, 39:79–95, 1989.

[3] D. Abramson, H. Dang, and M. Krisnamoorthy. Simulated annealing cooling schedules for the school timetabling problem. *Asia-Pacific Journal of Operational Research*, 16:1–22, 1999.

[4] Y.S. Abu-Mostafa. Learning from hints in neural networks. *Journal of Complexity*, 6(2):192–198, 1990.

[5] D. Achlioptas, L. M. Kirousis, E. Kranakis, and D. Krinzac. Rigorous results for random (2+p)-SAT. Technical report, Dept. of Computer Engineering and Informatics, University of Patras, Greece, 1997.

[6] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[7] Hirotugu Akaike. Akaike's information criterion. In *International Encyclopedia of Statistical Science*, pages 25–25. Springer, 2011.

[8] P. Alimonti. New local search approximation techniques for maximum generalized satisfiability problems. In *Proc. Second Italian Conf. on Algorithms and Complexity*, pages 40–53, 1994.

[9] L. Altenberg. Fitness distance correlation analysis: An instructive counterexample. *Proceedings of the 7th International Conference on Genetic Algorithms (ICGA'97)*, pages 57–64, 1997.

[10] Alexandr Andoni and Piotr Indyk. E 2 lsh 0.1 user manual. 2005.

[11] P. Asirelli, M. de Santis, and A. Martelli. Integrity constraints in logic databases. *Journal of Logic Programming*, 3:221–232, 1985.

[12] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256, 2002.

[13] G. Ausiello, P. Crescenzi, and M. Protasi. Approximate solution of NP optimization problems. *Theoretical Computer Science*, 150:1–55, 1995.

[14] G. Ausiello and M. Protasi. Local search, reducibility and approximability of NP-optimization problems. *Information Processing Letters*, 54:73–79, 1995.

[15] J. Bahren, V. Protopopescu, and D. Reister. Trust: a deterministic algorithm for global optimization. *Science*, 276:10941097, 1997.

[16] Horace B Barlow. Summation and inhibition in the frog's retina. *The Journal of physiology*, 119(1):69–88, 1953.

[17] N.A. Barricelli. Numerical testing of evolution theories. *Acta Biotheoretica*, 16(1):69–98, 1962.

[18] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neurolike adaptive elements that can solve difficult learning problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13:834–846, 1983.

[19] R. Battiti. Accelerated back-propagation learning: Two optimization methods. *Complex Systems*, 3(4):331–342, 1989.

[20] R. Battiti. First-and second-order methods for learning: Between steepest descent and newton's method. *Neural Computation*, 4:141–166, 1992.

[21] R. Battiti. Using the mutual information for selecting features in supervised neural net learning. *IEEE Transactions on Neural Networks*, 5(4):537–550, 1994.

[22] R. Battiti. Partially persistent dynamic sets for history-sensitive heuristics. Technical Report UTM-96-478, Dip. di Matematica, Univ. di Trento, 1996. Revised version, Presented at the Fifth DIMACS Challenge, Rutgers, NJ, 1996.

[23] R. Battiti. Reactive search: Toward self-tuning heuristics. In V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors, *Modern Heuristic Search Methods*, chapter 4, pages 61–83. John Wiley and Sons Ltd, 1996.

[24] R. Battiti. Time- and space-efficient data structures for history-based heuristics. Technical Report UTM-96-478, Dip. di Matematica, Univ. di Trento, 1996.

[25] R. Battiti and M. Brunato. Reactive search: machine learning for memory-based heuristics. In Teofilo F. Gonzalez, editor, *Approximation Algorithms and Metaheuristics*, chapter 21, pages 21–1 – 21–17. Taylor and Francis Books (CRC Press), Washington, DC, 2007.

[26] R. Battiti and M. Brunato. Reactive Search Optimization: Learning while Optimizing. *Handbook of Metaheuristics*, 146:543–571, 2010.

[27] R. Battiti, M. Brunato, and F. Mascia. *Reactive Search and Intelligent Optimization*, volume 45 of *Operations research/Computer Science Interfaces*. Springer Verlag, 2008.

[28] R. Battiti and P. Campigotto. Reactive search optimization: Learning while optimizing. an experiment in interactive multi-objective optimization. In S. Voss and M. Caserta, editors, *Proceedings of MIC 2009, VIII Metaheuristic International Conference*, Lecture Notes in Computer Science. Springer Verlag, 2010.

[29] R. Battiti and A. M. Colla. Democracy in neural nets: Voting schemes for accuracy. *Neural Networks*, 7(4):691–707, 1994.

[30] R. Battiti and G.Tecchiolli. Parallel biased search for combinatorial optimization: Genetic algorithms and tabu. *Microprocessor and Microsystems*, 16:351–367, 1992.

[31] R. Battiti and F. Masulli. BFGS optimization for faster and automated supervised learning. In *Proceedings of the International Neural Network Conference (INNC 90)*, pages 757–760, 1990.

[32] R. Battiti and A. Passerini. Brain-Computer Evolutionary Multiobjective Optimization (BC-EMO): A Genetic Algorithm Adapting to the Decision Maker. *IEEE Transactions on Evolutionary Computation*, 14(15):671 – 687, 2010.

[33] R. Battiti and M. Protasi. Reactive search, a history-sensitive heuristic for MAX-SAT. *ACM Journal of Experimental Algorithmics*, 2(ARTICLE 2), 1997. http://www.jea.acm.org/.

[34] R. Battiti and M. Protasi. Solving MAX-SAT with non-oblivious functions and history-based heuristics. In D. Du, J. Gu, and P. M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, number 35 in DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, pages 649–667. American Mathematical Society, Association for Computing Machinery, 1997.

[35] R. Battiti and M. Protasi. Approximate algorithms and heuristics for MAX-SAT. In D.Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 1, pages 77–148. Kluwer Academic Publishers, 1998.

[36] R. Battiti and G. Tecchiolli. Learning with first, second, and no derivatives: a case study in high energy physics. *Neurocomputing*, 6:181–206, 1994.

[37] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[38] R. Battiti and G. Tecchiolli. Simulated annealing and tabu search in the long run: a comparison on QAP tasks. *Computer and Mathematics with Applications*, 28(6):1–8, 1994.

[39] R. Battiti and G. Tecchiolli. Local search with memory: Benchmarking rts. *Operations Research Spektrum*, 17(2/3):67–86, 1995.

[40] R. Battiti and G. Tecchiolli. Training neural nets with the reactive tabu search. *IEEE Transactions on Neural Networks*, 6(5):1185–1200, 1995.

[41] R. Battiti and G. Tecchiolli. The continuous reactive tabu search: blending combinatorial optimization and stochastic search for global optimization. *Annals of Operations Research – Metaheuristics in Combinatorial Optimization*, 63:153–188, 1996.

[42] Roberto Battiti and Alan Albert Bertossi. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48(4):361–385, Apr 1999.

[43] Roberto Battiti and Anna Maria Colla. Democracy in neural nets: Voting schemes for classification. *Neural Networks*, 7(4):691–707, 1994.

[44] A.B. Baum. On the capabilities of multilayer perceptrons. *Journal of Complexity*, 4:193–215, 1988.

[45] John Baxter. Local optima avoidance in depot location. *The Journal of the Operational Research Society*, 32(9):815–819, Sep 1981.

[46] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[47] Yoshua Bengio. Learning deep architectures for AI. *Foundations and trends in Machine Learning*, 2(1):1–127, 2009.

[48] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[49] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

[50] J.L. Bentley. Experiments on traveling salesman heuristics. *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 91–99, 1990.

[51] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[52] Marco Biazzini, Mauro Brunato, and Alberto Montresor. Towards a decentralized architecture for optimization. In *Proc. 22nd IEEE International Parallel and Distributed Processing Symposium*, Miami, FL, USA, April 2008.

[53] M. Bilenko, S. Basu, and R.J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *Proceedings of the twenty-first international conference on Machine learning*, page 11. ACM, 2004.

[54] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In W.B. Langdon *et al.*, editor, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers. Also available as: AIDA-2002-01 Technical Report of Intellektik, Technische Universität Darmstadt, Darmstadt, Germany.

[55] C. E. Blair, R. G. Jeroslow, and J. K. Lowe. Some results and experiments in programming for propositional logic. *Computers and Operations Research*, 13(5):633–645, 1986.

[56] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[57] M. Boehm and E. Speckenmeyer. A fast parallel sat solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17:381–400, 1996.

[58] C. Boender and A. Rinnooy Kan. A bayesian analysis of the number of cells of a multinomial distribution. *The Statistician*, 32:240–249, 1983.

[59] KD Boese, AB Kahng, and S. Muddu. On the big valley and adaptive multi-start for discrete global optimizations. *Operation Research Letters*, 16(2), 1994.

[60] Bollobás. *Random Graphs*. Cambridge University Press, 2001.

[61] Ingwer Borg and Patrick JF Groenen. *Modern multidimensional scaling: Theory and applications*. Springer Science & Business Media, 2005.

[62] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[63] Chernoff Bound. Probability of error, equivocation, and the. *IEEE Transactions on Information Theory*, 16(4), 1970.

[64] O. Braysy. A reactive variable neighborhood search for the vehicle-routing problem with time windows. *INFORMS JOURNAL ON COMPUTING*, 15(4):347–368, 2003.

[65] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[66] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and regression trees*. Chapman&Hall / CRC press, 1993.

[67] R. P. Brent. *Algorithms for Minimization without Derivatives*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1973.

[68] C. G. Broyden, J. E. Dennis, and J. J. More'. On the local and superlinear convergence of quasi-newton methods. *J.I.M.A*, 12:223–246, 1973.

[69] M. Brunato and R. Battiti. RASH: A self-adaptive random search method. In Carlos Cotta, Marc Sevaux, and Kenneth Sörensen, editors, *Adaptive and Multilevel Metaheuristics*, volume 136 of *Studies in Computational Intelligence*. Springer, 2008.

[70] Mauro Brunato and Roberto Battiti. Corso (collaborative reactive search optimization): Blending combinatorial and continuous local search. *Informatica, Lith. Acad. Sci.*, 27(2):299–322, 2016.

[71] Mauro Brunato and Roberto Battiti. Extreme reactive portfolio (xrp): Tuning an algorithm population for global optimization. In *International Conference on Learning and Intelligent Optimization*, pages 60–74. Springer, 2016.

[72] Mauro Brunato, Roberto Battiti, and Srinivas Pasupuleti. A memory-based rash optimizer. In Ariel Felner Robert Holte Hector Geffner, editor, *Proceedings of AAAI-06 workshop on Heuristic Search, Memory Based Heuristics and Their applications*, pages 45–51, Boston, Mass., 2006. ISBN 978-1-57735-290-7.

[73] JB Butcher, David Verstraeten, Benjamin Schrauwen, CR Day, and PW Haycock. Reservoir computing and extreme learning machines for non-linear time-series data analysis. *Neural networks*, 38:76–89, 2013.

[74] Paolo Campigotto, Andrea Passerini, and Roberto Battiti. Active learning of pareto fronts. *IEEE Transactions on Neural Networks and Learning Systems*, 25(3):506 – 519, March 2014.

[75] Soumen Chakrabarti. *Mining the Web: discovering knowledge from hypertext data*. Morgan Kaufmann, 2003.

[76] S. Chakradar, V. Agrawal, and M. Bushnell. Neural net and boolean satisfiability model of logic circuits. *IEEE Design and Test of Computers*, pages 54–57, 1990.

[77] M.-T. Chao and J. Franco. Probabilistic analysis of two heuristics for the 3-satisfiability problem. *SIAM J. Comput.*, 15:1106–1118, 1986.

[78] O. Chapelle, M. Chi, and A. Zien. A continuation method for semi-supervised SVMs. In *Proceedings of the 23rd international conference on Machine learning*, page 192. ACM, 2006.

[79] O. Chapelle, B. Schölkopf, and A. Zien. *Semi-supervised learning*. The MIT Press, Cambridge, MA, 2006.

[80] Olivier Chapelle. Training a support vector machine in the primal. *Neural Computation*, 19(5):1155–1178, 2007.

[81] Abraham Charnes and William Wager Cooper. Goal programming and multiple objective optimizations: Part 1. *European Journal of Operational Research*, 1(1):39–54, 1977.

[82] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. *Proceedings of the 12th IJCAI*, pages 331–337, 1991.

[83] J. Chen, D. Friesen, and H. Zheng. Tight bound on johnson's algorithm for MAX-SAT. In *Proc. 12th Annual IEEE conf. on Computational Complexity, Ulm, Germany*, pages 274–281, 1997.

[84] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *arXiv preprint arXiv:1603.02754*, 2016.

[85] J. Cheriyan, W. H. Cunningham, T Tuncel, and Y. Wang. A linear programming and rounding approach to MAX 2-SAT. In M. Trick and D. S. Johson, editors, *Proceedings of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability*, number 26 in DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pages 395–414, 1996.

[86] Kevin J Cherkauer. Human expert-level performance on a scientific image analysis task by a system using combined artificial neural networks. In *Working notes of the AAAI workshop on integrating multiple learned models*, pages 15–21. Citeseer, 1996.

[87] V. Cherny. A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–45, 1985.

[88] T.-S. Chiang and Y. Chow. On the convergence rate of annealing processes. *SIAM Journal on Control and Optimization*, 26(6):1455–1470, 1988.

[89] V. Chvatal and B. Reed. Mick gets some (the odds are on his side). In *Proc. 33th Ann. IEEE Symp. on Foundations of Comput. Sci.*, pages 620–627. IEEE Computer Society, 1992.

[90] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35:759–768, 1988.

[91] V.A. Cicirello. *Boosting Stochastic Problem Solvers Through Online Self-Analysis of Performance*. PhD thesis, Carnegie Mellon University, Also available as technical report CMU-RI-TR-03-27., 2003.

[92] Vincent Cicirello and Stephen Smith. The max k-armed bandit: A new model for exploration applied to search heuristic selection. In *20th National Conference on Artificial Intelligence (AAAI-05)*, July 2005. Best Paper Award.

[93] Vincent A. Cicirello and Stephen F. Smith. *Principles and Practice of Constraint Programming CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, chapter Heuristic Selection for Stochastic Search Optimization: Modeling Solution Quality by Extreme Value Theory, pages 197–211. Springer Berlin / Heidelberg, 2004.

[94] David A. Clark, Jeremy Frank, Ian P. Gent, Ewan MacIntyre, Neven Tomov, and Toby Walsh. Local search and the number of solutions. In *Principles and Practice of Constraint Programming*, pages 119–133, 1996.

[95] M. Clerc and J. Kennedy. The particle swarm – explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[96] Pierre Comon. Independent component analysis, a new concept? *Signal processing*, 36(3):287–314, 1994.

[97] D.T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, 46(1):93–100, 1990.

[98] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, December 1999.

[99] S. A. Cook and D. G. Mitchell. Finding hard instances of the satisfiability problem: a survey. In D.-Z. Du, J. Gu, and P.M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Association for Computing Machinery, 1997.

[100] S.A. Cook. The complexity of theorem-proving procedures. In *Proc. of the Third Annual ACM Symp. on the Theory of Computing*, pages 151–158, 1971.

[101] A. Corana, M. Marchesi, C. Martini, and S. Ridella. Minimizing multimodal functions of continuous variables with the simulated annealing algorithm. *ACM Trans. Math. Softw.*, 13(3):262–280, 1987.

[102] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press Cambridge, 6 edition, 2001.

[103] T. Crainic and M. Toulouse. Parallel strategies for metaheuristics. In F. Glover and G. Kochenberger, editors, *State-of-the-Art Handbook in Metaheuristics*, chapter 1. Kluwer Academic Publishers, 2002.

[104] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3-sat. *Artif. Intell.*, 81(1-2):31–57, 1996.

[105] Antonio Criminisi. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends in Computer Graphics and Vision*, 7(2-3):81–227, 2011.

[106] Paul Dagum and Michael Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artificial intelligence*, 60(1):141–153, 1993.

[107] J.M. Daida, S.P. Yalcin, P.M. Litvak, G.A. Eickhoff, and J.A. Polito. Of metaphors and darwinism: Deconstructing genetic programming's chimera. In *Proceedings CEC-99: Congress in Evolutionary Computation, Piscataway*, pages 453–462. IEEE Press, 1999.

[108] C. Darwin. *On The Origin of Species*. Signet Classic, reprinted 2003, 1859.

[109] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[110] M. Davis, G. Logemann, and D. Loveland. A machien program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[111] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[112] R. Dawkins. *The selfish gene.* Oxford. Oxford University, 1976.

[113] R.F. Dell and M.H. Karwan. An interactive MCDM weight space reduction method utilizing a Tchebycheff utility function. *Naval Research Logistics*, 37(2):403–418, 1990.

[114] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, British Columbia, Canada, August 1987. ACM Press.

[115] J. E. Dennis and R. B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. Prentice Hall, Englewood Cliffs, NJ, 1983.

[116] Thomas G Dietterich. Ensemble methods in machine learning. In *Multiple classifier systems*, pages 1–15. Springer, 2000.

[117] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *arXiv preprint cs/9501101*, 1995.

[118] Thomas G Dietterich, Pedro Domingos, Lise Getoor, Stephen Muggleton, and Prasad Tadepalli. Structured machine learning: the next ten years. *Machine Learning*, 73(1):3–23, 2008.

[119] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, Berkeley, CA, May 28-30 1986. ACM.

[120] Dingzhu Du, Jun Gu, and Panos M. Pardalos. *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Association for Computing Machinery, 1997.

[121] Artur Dubrawski and Jeff Schneider. Memory based stochastic optimization for validation and tuning of function approximators. In *Conference on AI and Statistics*, 1997.

[122] Yvan Dumas, Jacques Desrosiers, and François Soumis. The pickup and delivery problem with time windows. *European Journal of Operational Research*, 54(1):7 – 22, 1991.

[123] James S Dyer, Peter C Fishburn, Ralph E Steuer, Jyrki Wallenius, and Stanley Zionts. Multiple criteria decision making, multiattribute utility theory: the next ten years. *Management science*, 38(5):645–654, 1992.

[124] A.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, Jun 1999.

[125] P. Erdos and A. Renyi. On random graphs. *Publ. Math. Debrecen*, 6:290–297, 1959.

[126] A.G. Ferreira and J. Zerovnik. Bounding the probability of success of stochastic methods for global optimization. *Computers Math. Applic.*, 25(10/11):1–8, 1993.

[127] Sir Ronald Aylmer Fisher, Ronald Aylmer Fisher, Statistiker Genetiker, Ronald Aylmer Fisher, Statistician Genetician, Ronald Aylmer Fisher, and Statisticien Généticien. *The design of experiments*, volume 12. Oliver and Boyd Edinburgh, 1960.

[128] Philip W. L. Fong. A quantitative study of hypothesis selection. In *International Conference on Machine Learning*, pages 226–234, 1995.

[129] J. Franco and M. Paull. Probabilistic analysis of the davis-putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.

[130] J. Frank. Learning short-term weights for GSAT. In *Proc. INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 15, pages 384–391. LAWRENCE ERLBAUM ASSOCIATES LTD, USA, 1997.

[131] A. Fraser and D.G. Burnell. *Computer models in genetics*. McGraw-Hill New York, 1970.

[132] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.

[133] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[134] Jerome H Friedman. Exploratory projection pursuit. *Journal of the American statistical association*, 82(397):249–266, 1987.

[135] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[136] Jerome H Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002.

[137] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[138] J.H. Friedman and J.W. Tukey. A projection pursuit algorithm for exploratory data analysis. *Computers, IEEE Transactions on*, C-23(9):881–890, Sept 1974.

[139] M. Gagliolo and J. Schmidhuber. A neural network model for inter-problem adaptive online time allocation. In W. Duch et al., editor, *Proceedings Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005, 15th Int. Conf.*, volume 2, pages 7–12, Warsaw, 2005. Springer, Berlin.

[140] M. Gagliolo and J. Schmidhuber. Dynamic algorithm portfolios. In *Proceedings AI and MATH '06, Ninth International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, Jan 2006.

[141] M. Gagliolo and J. Schmidhuber. Impact of censored sampling on the performance of restart strategies. In *CP 2006 - Twelfth International Conference on Principles and Practice of Constraint Programming - Nantes, France*, pages 167–181. Springer, Berlin, Sep 2006.

[142] H. Gallaire, J. Minker, and J. M. Nicolas. Logic and databases: a deductive approach. *Computing Surveys*, 16(2):153–185, 1984.

[143] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):721–741, 1984.

[144] I.P. Gent and T. Walsh. An empirical analysis of search in GSAT. *Journal of Artificial Intelligence Research*, 1:47–59, 1993.

[145] I.P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for sat. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 28–33. AAAI Press / The MIT Press, 1993.

[146] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *Proceedings of the ninth ACM conference on Hypertext and hypermedia: links, objects, time and space—structure in hypermedia systems: links, objects, time and space—structure in hypermedia systems*, pages 225–234. ACM, 1998.

[147] P.E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, 1981.

[148] F. Glover. Tabu search - part i. *ORSA Journal on Computing*, 1(3):190–260, 1989.

[149] F. Glover. Tabu search - part ii. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[150] F. Glover. Tabu search, Part I1. *ORSA journal on Computing*, 2(1):4–32, 1990.

[151] F. Glover. Scatter search and star-paths: beyond the genetic metaphor. *Operations Research Spektrum*, 17(2/3):125–138, 1995.

[152] F. Glover. Tabu search — Uncharted domains. *Annals of Operations Research*, 149(1):89–98, 2007.

[153] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39(3):653–684, 2000.

[154] F. Glover, M. Laguna, and R. Marti. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39(3):653–684, 2000.

[155] M.X. Goemans and D.P. Williamson. New $\frac{3}{4}$-approximation algorithms for the maximum satisfiability problem. *SIAM Journal on Discrete Mathematics*, 7(4):656–666, 1994.

[156] A. Goerdt. A threshold for unsatisfiability. *Journal of Computer and System Sciences*, 53:469–486, 1996.

[157] A. A. Goldstein. *Constructive Real Analysis*. Harper and Row, New York, 1967.

[158] Carla Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. of Automated Reasoning*, 24((1/2)):67–100, 2000.

[159] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.

[160] P. Greistorfer and S. VoS. *Controlled Pool Maintenance in Combinatorial Optimization*, volume 30 of *Operations Research/Computer Science Interfaces*, chapter 18, pages 387–424. Springer verlag, 2005.

[161] J. Gu. *Parallel Algorithms and Architectures for very fast AI Search*. PhD thesis, University of Utah, 1989.

[162] J. Gu, Q.-P. Gu, and D.-Z.Du. Convergence properties of optimization algorithms for the SAT problem. *IEEE Transactions on Computers*, 45(2):209–219, 1996.

[163] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In D.-Z. Du, J. Gu, and P.M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Association for Computing Machiner y, 1997.

[164] Jun Gu. Efficient local search for very large-scale satisfiability problem. *ACM SIGART Bulletin*, 3(1):8–12, 1992.

[165] Liyanaarachchi Lekamalage Chamara Kasun Guang-Bin Huang, Zuo Bai and Chi Man Vong. Local receptive fields based extreme learning machine. *IEEE COMPUTATIONAL INTELLIGENCE MAGAZINE*, 10(2), 2015.

[166] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of the 19th Ann. Symp. on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[167] Isabelle Guyon and André Elisseeff. An introduction to feature extraction. In *Feature extraction*, pages 1–25. Springer, 2006.

[168] T. Hagerup and C. Rueb. A guided tour of chernoff bounds. *Information Processing Letters*, 33:305–308, 1989/90.

[169] Bruce Hajek. Cooling schedules for optimal annealing. *Math. Oper. Res.*, 13(2):311–329, 1988.

[170] P. L. Hammer, P. Hansen, and B. Simeone. Roof duality, complementation and persistency in quadratic 0-1 optimization. *Mathematical Programming*, 28:121–155, 1984.

[171] N. Mladenovic P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11):1097–1100, 1997.

[172] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.

[173] P. Hansen and N. Mladenovic. Variable neighborhood search. In E.K. Burke and G. Kendall, editors, *Search methodologies: introductory tutorials in optimization and decision support techniques*, pages 211–238. Springer, 2005.

[174] Babak Hassibi, David G Stork, et al. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, pages 164–164, 1993.

[175] R. B. Heckendorn, S. Rana, and D. L. Whitey. Test function generators as embedded landscapes. In W. Banzhaf and C. Reeves, editors, *Foundations of Genetic Algorithms 5*, pages 183–198. Morgan Kaufmann, San Francisco, USA, 1999.

[176] David Heckerman. *A tutorial on learning with Bayesian networks*. Springer, 1998.

[177] J.A. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, Inc., Redwood City, CA, 1991.

[178] R. Hinterding, Z. Michalewicz, and AE Eiben. Adaptation in evolutionary computation: a survey. In *IEEE International Conference on Evolutionary Computation*, pages 65–69, 1997.

[179] G.E. Hinton and S.J. Nowlan. How learning can guide evolution. *Complex Systems*, 1(1):495–502, 1987.

[180] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[181] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[182] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[183] Tin Kam Ho. The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):832–844, 1998.

[184] Tad Hogg. *Applications of Statistical Mechanics to Combinatorial Search Problems*, volume 2, pages 357–406. World Scientific, Singapore, 1995.

[185] Tad Hogg, Bernardo A. Huberman, and Colin P. Williams. Phase transitions and the search problem. *Artif. Intell.*, 81(1-2):1–15, 1996.

[186] J.H. Holland. Adaptation in Nature and Artificial Systems. *Ann Arbor, MI: University of Michigan Press*, 1975.

[187] J.N. Hooker. Resolution vs. cutting plane solution of inference problems: some computational experience. *Operations research letters*, 7(1):1–7, 1988.

[188] H.H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the national conference on artificial intelligence*, volume 18, pages 655–660. AAAI Press; MIT Press, 1999.

[189] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Biophysics*, 79:2554–2558, 1982.

[190] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[191] E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and D. M. Chickering. A bayesian approach to tackling hard computational problems. In *Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 235–244, Seattle,USA, Aug 2001.

[192] Bin Hu and Gnther R. Raidl. Variable neighborhood descent with self-adaptive neighborhood-ordering. In Carlos Cotta, Antonio J. Fernandez, and Jose E. Gallardo, editors, *Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics, malaga, Spain*, 2006.

[193] Gao Huang, Guang-Bin Huang, Shiji Song, and Keyou You. Trends in extreme learning machines: A review. *Neural Networks*, 61:32–48, 2015.

[194] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: A new learning scheme of feedforward neural networks. In *Proceedings of International Joint Conference on Neural Networks (IJCNN2004)*, July 2004.

[195] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1):489–501, 2006.

[196] MD Huang, F. Romeo, and A. Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. *IEEE International Conference on Computer Aided Design*, pages 381–384, 1986.

[197] Peter J Huber. Projection pursuit. *The annals of Statistics*, pages 435–475, 1985.

[198] B.A. Huberman and T. Hogg. Phase transitions in artificial intelligence systems. *Artificial Intelligence*, 33(2):155–171, 1987.

[199] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, January 3 1997.

[200] F. Hutter, Y. Hamadi, H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. *Principles and Practice of Constraint Programming-CP 2006*, pages 213–228, 2006.

[201] Aapo Hyvärinen. Fast and robust fixed-point algorithms for independent component analysis. *Neural Networks, IEEE Transactions on*, 10(3):626–634, 1999.

[202] Aapo Hyvärinen. Independent component analysis: recent advances. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 371(1984), 2012.

[203] Aapo Hyvärinen and Erkki Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4):411–430, 2000.

[204] Ronald L Iman, JE Campbell, and JC Helton. An approach to sensitivity analysis of computer models. i-introduction, input, variable selection and preliminary variable assessment. *Journal of quality technology*, 13:174–183, 1981.

[205] L. Ingber. Very fast simulated re-annealing. *Mathl. Comput. Modelling*, 12(8):967–973, 1989.

[206] A. Ishtaiwi, J. R. Thornton, Sattar A. Anbulagan, and D. N. Pham. Adaptive clause weight redistribution. In *Proceedings of the 12th International Conference on the Principles and Practice of Constraint Programming, CP-2006, Nantes, France*, pages 229–243, 2006.

[207] Gu J. Global optimization for satisfiability (sat) problem. *IEEE Transactions on Data and Knowledge Engineering*, 6(3):361–381, 1994.

[208] Gu J. and Du B. A multispace search algorithm (invited paper). *DIMACS Monograph on Global Minimization of Nonconvex Energy Functions*. to appear.

[209] Gu J. and Puri R. Asynchronous circuit synthesis with boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 14(8):961–973, 1995.

[210] Herbert Jaeger. The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148:34, 2001.

[211] Manfred Jaeger. Relational bayesian networks. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, UAI'97, pages 266–273, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[212] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[213] I.L. Janis. *Victims of groupthink*. Houghton Mifflin, 1972.

[214] Kevin Jarrett, Koray Kavukcuoglu, M Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE, 2009.

[215] B. Jaumard, M. Stan, and J. Desrosiers. Tabu search and a quadratic relaxation for the satisfiability problem. In M. Trick and D. S. Johson, editors, *Proceedings of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability*, number 26 in DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pages 457–477, 1996.

[216] T. Joachims. Making large-scale SVM learning practical. In Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11. MIT-Press, Cambridge, Mass., 1999.

[217] Thorsten Joachims. Making large scale svm learning practical. Technical report, Universität Dortmund, 1999.

[218] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[219] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892, 1989.

[220] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning. *Oper. Res.*, 39(3):378–406, 1991.

[221] D.S. Johnson. Local optimization and the travelling salesman problem. In *Proc. 17th Colloquium on Automata Languages and Programming*, volume 447 of *LNCS*. Springer Verlag, Berlin, 1990.

[222] J. L. Johnson. A neural network approach to the 3-satisfiability problem. *J. Parallel and Distributed Computing*, 6:435–449, 1989.

[223] T. Jones and S. Forrest. Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms. *Proceedings of the 6th International Conference on Genetic Algorithms table of contents*, pages 184–192, 1995.

[224] K. A. D. Jong, M. A. Potter, and W. M. Spears. Using problem generators to explore the effects of epistasis. In T. Bäck, editor, *Proc. 7th intl. conference on genetic algorithms*, San Francisco, USA, 2007. Morgan Kaufmann.

[225] Daniel Kahneman. Thinking, fast and slow, farrar, straus and giroux, 2011.

[226] A. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. *Random Structures and Algorithms*, 7:59–80, 1995.

[227] A. P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G. Resende. Computational exprience with an interior point algorithm on the satisfiability problem. *Annals of operations research*, 25:43–58, 1990.

[228] A. P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G. Resende. A continuous approach to inductive inference. *Mathematical programming*, 57:215–238, 1992.

[229] Jagat Narain Kapur. *Measures of information and their applications*. Wiley-Interscience, 1994.

[230] S. A. Kauffman and S. Levin. Towards a general theory of adaptive walks on rugged landscapes. *Journal of Theoretical Biology*, 128:11–45, 1987.

[231] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman. Dynamic restart policies. In *Eighteenth national conference on Artificial intelligence*, pages 674–681, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[232] Liangjun Ke, Qingfu Zhang, and Roberto Battiti. Hybridization of decomposition and local search for multiobjective optimization. *Cybernetics, IEEE Transactions on*, 44(10):1808–1820, 2014.

[233] Kenji Kira and Larry A Rendell. The feature selection problem: Traditional methods and a new algorithm. In *AAAI*, volume 2, pages 129–134, 1992.

[234] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[235] Scott Kirkpatrick and Bart Selman. Critical behavior in the satisfiability of random boolean expressions. *Science*, 264:1297–1301, 1994.

[236] L. M. Kirousis, E. Kranakis, and D. Krizanc. Approximating the unsatisfiability threshold of random formulas. In *Proceedings of the Fourth Annual European Symposium on Algorithms, ESA'96*, pages 27–38, Barcelona, Spain, September 1996. Springer-Verlag, LNCS.

[237] Igor Kononenko. Estimating attributes: analysis and extensions of relief. In *Machine Learning: ECML-94*, pages 171–182. Springer, 1994.

[238] Y. Koren and L. Carmel. Robust linear dimensionality reduction. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):459–470, 2004.

[239] E. Koutsoupias and C. H. Papadimitriou. On the greedy algorithm for satisfiability. *Information Processing Letters*, 43:53–55, 1992.

[240] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Physical review E*, 69(6):066138, 2004.

[241] N. Krasnogor and J. Smith. A tutorial for competent memetic algorithms: model, taxonomy, and design issues. *IEEE Transactions on Evolutionary Computation*, 9(5):474–488, Oct 2005.

[242] D.G. Krige. A statistical approach to some mine valuations and allied problems at the witwatersrand. Master's thesis, University of Witwatersrand, 1951.

[243] Joseph B Kruskal. Toward a practical method which helps uncover the structure of a set of multivariate observations by finding the linear transformation which optimizes a new 'index of condensation'. In *Statistical Computation*, pages 427–440. Academic Press, New York, 1969.

[244] Joseph B Kruskal and Myron Wish. *Multidimensional scaling*, volume 11. Sage, 1978.

[245] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing for scalable image search. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2130–2137. IEEE, 2009.

[246] O. Kullmann and H. Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Technical Report 1596, JohannWolfgang Goethe-Universitat, Fachbereich Mathematik, 60054 Frankfurt, Germany, January 1997.

[247] P. J. M. Laarhoven and E. H. L. Aarts, editors. *Simulated annealing: theory and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[248] G. Lebanon. Metric learning for text documents. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 497–508, 2006.

[249] Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361:310, 1995.

[250] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *NIPs*, volume 2, pages 598–605, 1989.

[251] Sang M Lee et al. *Goal programming for decision analysis*. Auerbach Philadelphia, 1972.

[252] A. Levy and A. Montalvo. The tunneling algorithm for the global minimization of functions. *SIAM Journal on Scientific and Statistical Computing*, 6:15–29, 1985.

[253] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[254] H. R. Lourenco, O. C. Martin, and T. Stutzle. Iterated local search. In F. Glover and G.A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321–353. Springer, 2003.

[255] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173180, 1993.

[256] Mantas LukošEvičIus and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[257] Thibaut Lust and Andrzej Jaszkiewicz. Speed-up techniques for solving large-scale biobjective tsp. *Computers & Operations Research*, 37(3):521–533, 2010.

[258] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

[259] D. Maier and S. C. Salveter. Hysterical b-trees. *Information Processing Letters*, 12(4):199–202, 1981.

[260] M. Marchiori. The quest for correct information on the web: Hyper search engines. *Computer Networks and ISDN Systems*, 29(8-13):1225–1235, 1997.

[261] Oden Maron and Andrew W. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11(1-5):193–225, 1997.

[262] Olivier Martin, Steve W. Otto, and Edward W. Felten. Large-step Markov chains for the traveling salesman problem. *Complex Systems*, 5:3:299, 1991.

[263] Olivier Martin, Steve W. Otto, and Edward W. Felten. Large-step Markov chains for the tsp incorporating local search heuristics. *Operation Research Letters*, 11:219–224, 1992.

[264] Olivier C. Martin and Steve W. Otto. Combining simulated annealing with local search heuristics. *ANNALS OF OPERATIONS RESEARCH*, 63:57–76, 1996.

[265] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the national conference on artificial intelligence*, number 14, pages 321–326. John Wiley & sons LTD, USA, 1997.

[266] Michael D McKay, Richard J Beckman, and William J Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61, 2000.

[267] P. Merz and B. Freisleben. Fitness landscape analysis and memetic algorithms for the quadratic assignment problem. *IEEE Transactions on Evolutionary Computation*, 4(4):337–352, Nov 2000.

[268] N. Metropolis, A. N. Rosenbluth, M. N. Rosenbluth, and A. H. Tellerand E. Teller. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21(6):10871092, 1953.

[269] Kaisa Miettinen. *Nonlinear multiobjective optimization*, volume 12. Springer Science & Business Media, 2012.

[270] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings AAAI-90*, pages 17–24, 1990.

[271] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[272] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, San Jose, Ca, July 1992.

[273] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. Convergence and finite-time behavior of simulated annealing. *Advances in Applied Probability*, 18(3):747–771, Sep 1986.

[274] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the national conference on artificial intelligence*, number 11, page 40. John Wiley & sons LTD, USA, 1993.

[275] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech Concurrent Computation Program, C3P Report*, 826, 1989.

[276] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.

[277] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

[278] Surendra Nahar, Sartaj Sahni, and Eugene Shragowitz. Simulated annealing and combinatorial optimization. In *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 293–299, Piscataway, NJ, USA, 1986. IEEE Press.

[279] Dana S Nau, Vipin Kumar, and Laveen Kanal. General branch and bound, and its relation to a* and ao*. *Artificial Intelligence*, 23(1):29–58, 1984.

[280] W. Nelson. *Applied Life Data Analysis*. Jon Wiley, New York, 1982.

[281] T. A. Nguyen, W. A. Perkins, T. J. Laffrey, and D. Pecora. Checking an expert system knowledge base for consistency and completeness. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 375–378, Los Altos, CA, 1985.

[282] E.M. Oblow. Pt: a stochastic tunneling algorithm for global optimization. *Journal of Global Optimization*, 20(2):191–208, June 2001.

[283] Ibrahim Hassan Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Ann. Oper. Res.*, 41(1-4):421–451, 1993.

[284] E. Osuna, R. Freund, and F. Girosi. Support vector machines: Training and applications. Technical Report AIM-1602, MIT Artificial Intelligence Laboratory and Center for Biological and Computational Learning, 1997.

[285] M. H. Overmars. Searching in the past ii: general transforms. Technical report, Dept. of Computer Science, Univ. of Utrecht, Utrecht, The Netherlands, 1981.

[286] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanfor University, 1998.

[287] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity*. Prentice-Hall, NJ, 1982.

[288] Christos H. Papadimitriou. On selecting a satisfying truth assignment (extended abstract). In *Proc. of the 32th annual symposium on foundations of computer science - FOCS*, pages 163–169, 1991.

[289] Athanasios Papoulis and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. Tata McGraw-Hill Education, 2002.

[290] Luis Paquete and Thomas Stützle. A two-phase local search for the biobjective traveling salesman problem. In *Evolutionary Multi-Criterion Optimization*, pages 479–493. Springer, 2003.

[291] Vilfredo Pareto. *Manuale di economia politica*, volume 13. Societa Editrice, 1906.

[292] Andrew J. Parkes. Clustering at the phase transition. In *AAAI/IAAI*, pages 340–345, 1997.

[293] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*, 2012.

[294] D.J. Patterson and H. Kautz. Auto-walksat: A self-tuning implementation of walk-sat. Electronic Notes in Discrete Mathematics (ENDM), 2001.

[295] D. Pelta, A. Sancho-Royo, C. Cruz, and J.L. Verdegay. Using memory and fuzzy rules in a co-operative multi-thread strategy for optimization. *Information Sciences*, 176(13):1849–1868, 2006.

[296] Dinh Tuan Pham and Philippe Garat. Blind separation of mixture of independent sources through a quasi-maximum likelihood approach. *Signal Processing, IEEE Transactions on*, 45(7):1712–1725, 1997.

[297] Selcen (Pamuk) Phelps and Murat Köksalan. An interactive evolutionary metaheuristic for multiobjective combinatorial optimization. *Management Science*, 49(12):1726–1738, 2003.

[298] Martin Pincus. A monte carlo method for the approximate solution of certain types of constrained optimization problems. *Operations Research*, 18(6):1225–1228, 1970.

[299] John Platt et al. Fast training of support vector machines using sequential minimal optimization. *Advances in kernel methods—support vector learning*, 3, 1999.

[300] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[301] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[302] Puri R. and Gu J. A bdd sat solver for satisfiability testing: an industrial case study. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):315–337, 1996.

[303] I. Rechenberg. *Evolutionsstrategie*. Frommann-Holzboog, 1973.

[304] M.G.C. Resende and T. A. Feo. A grasp for satisfiability. In M. Trick and D. S. Johson, editors, *Proceedings of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability*, number 26 in DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pages 499–520, 1996.

[305] M.G.C. Resende, L.S. Pitsoulis, and P.M. Pardalos. Approximate solution of weighted MAX-SAT problems using GRASP. In *DIMACS workshop on Satisfiability, Rutgers, NJ*. American Mathematical Society, 1996. in press.

[306] M.P. Silverman (reviewer). The Wisdom of Crowds. *American Journal of Physics*, 75:190, 2007.

[307] J. A. Robinson. A machine–oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[308] Frank Rosenblatt. Principles of neurodynamics. 1962.

[309] Peter J Rousseeuw and Annick M Leroy. *Robust regression and outlier detection*, volume 589. John Wiley & Sons, 2005.

[310] Y. Ruan, E. Horvitz, and H. Kautz. Restart policies with dependence among runs: A dynamic programming approach. 2002.

[311] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In D.E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations*. MIT Press, 1986.

[312] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[313] Craig Saunders, Alexander Gammerman, and Volodya Vovk. Ridge regression learning algorithm in dual variables. In *(ICML-1998) Proceedings of the 15th International Conference on Machine Learning*, pages 515–521. Morgan Kaufmann, 1998.

[314] Andrew Saxe, Pang W Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, and Andrew Y Ng. On random weights and unsupervised feature learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1089–1096, 2011.

[315] Robert E Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[316] G. R. Schreiber and O. C. Martin. Cut size statistics of graph bisection heuristics. *SIAM JOURNAL OF OPTIMIZATION*, 10(1):231–251, 1999.

[317] D. Schuurmans, F. Southey, and R.C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *Proceedings of the international joint conference on artificial intelligence*, volume 17, pages 334–341. Lawrence Erlbaum associates LTD, USA, 2001.

[318] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete sat procedures. *Artif. Intell.*, 132(2):121–150, 2001.

[319] H.P. Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, Inc. New York, NY, USA, 1981.

[320] H. Scudder III. Probability of error of some adaptive pattern-recognition machines. *IEEE Transactions on Information Theory*, 11(3):363–371, 1965.

[321] B. Selman and H.A. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the eleventh national Conference on Artificial Intelligence (AAAI-93)*, Washington, D. C., 1993.

[322] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the national conference on artificial intelligence*, volume 12. John Wiley & sons LTD, USA, 1994.

[323] B. Selman, H.A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In M. Trick and D. S. Johson, editors, *Proceedings of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability*, number 26 in DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pages 521–531, 1996.

[324] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, San Jose, Ca, July 1992.

[325] Bart Selman and Henry Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of IJCAI-93*, pages 290–295, 1993.

[326] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artif. Intell.*, 81(1-2):17–29, 1996.

[327] Ya D Sergeyev. Efficient strategy for adaptive partition of n-dimensional intervals in the framework of diagonal algorithms. *Journal of Optimization Theory and Applications*, 107(1):145–168, 2000.

[328] Yaroslav D Sergeyev and Dmitri E Kvasov. Global search based on efficient diagonal partitions and a set of lipschitz constants. *SIAM Journal on Optimization*, 16(3):910–937, 2006.

[329] Yaroslav D Sergeyev and Dmitri E Kvasov. A deterministic global optimization using smooth diagonal auxiliary functions. *Communications in Nonlinear Science and Numerical Simulation*, 21(1):99–111, 2015.

[330] Burr Settles. Active learning literature survey. *University of Wisconsin, Madison*, 52(55-66):11, 2010.

[331] D. F. Shanno. Conjugate gradient methods with inexact searches. *Mathematics of Operations Research*, 3(3):244–256, 1978.

[332] Joseph Sill, Gábor Takács, Lester Mackey, and David Lin. Feature-weighted linear stacking. *arXiv preprint arXiv:0911.0460*, 2009.

[333] V. Sindhwani, S.S. Keerthi, and O. Chapelle. Deterministic annealing for semi-supervised kernel machines. In *Proceedings of the 23rd international conference on Machine learning*, page 848. ACM, 2006.

[334] Josh Singer, Ian Gent, and Alan Smaill. Backbone Fragility and the Local Search Cost Peak. *Journal of Artificial Intelligence Research*, 12:235–270, 2000.

[335] S.Khanna, R.Motwani, M.Sudan, and U.Vazirani. On syntactic versus computational views of approximability. In *Proc. 35th Ann. IEEE Symp. on Foundations of Computer Science*, pages 819–836, 1994.

[336] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *Signal Processing Magazine, IEEE*, 25(2):128–131, 2008.

[337] B.M. Smith. Phase transition and the mushy region in constraint satisfaction problems. *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 100–104, 1994.

[338] R. E. Smith and J. E. Smith. New methods for tunable random landscapes. In W. N. Martin and W. M. Spears, editors, *Foundations of Genetic Algorithms 6*. Morgan Kaufmann, 2001.

[339] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. Technical Report NeuroCOLT NC-TR-98-030, Royal Holloway College, University of London, UK, 1998.

[340] F. J. Solis and R. J-B. Wets. Minimization by random search techniques. *Mathematics of Operations Research*, 6(1):19–30, February 1981.

[341] Kenneth Sörensen. Metaheuristics—the metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, 2015.

[342] Finnegan Southey. *Theory and Applications of Satisfiability Testing*, chapter Constraint Metrics for Local Search, pages 269–281. Springer Verlag, 2005.

[343] W. M. Spears. Simulated annealing for hard satisfiability problems. In M. Trick and D. S. Johnson, editors, *Proceedings of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability*, number 26 in DIMACS Series on Discrete Mathematics and Theoretical Computer Science, pages 533–555, 1996.

[344] Wolfram Stadler. A survey of multicriteria optimization or the vector maximum problem, part i: 1776–1960. *Journal of Optimization Theory and Applications*, 29(1):1–52, 1979.

[345] Ingo Steinwart, Don Hush, and Clint Scovel. Training svms without offset. *The Journal of Machine Learning Research*, 12:141–202, 2011.

[346] R.E. Steuer and E. Choo. An interactive weighted Tchebycheff procedure for multiple objective programming. *Mathematical Programming*, 26(1):326–344, 1983.

[347] Harald Stögbauer, Alexander Kraskov, Sergey A Astakhov, and Peter Grassberger. Least-dependent-component analysis based on mutual information. *Physical Review E*, 70(6):066123, 2004.

[348] James V Stone. *Independent component analysis*. Wiley Online Library, 2004.

[349] M. J. Streeter and S.F. Smith. A simple distribution-free approach to the max k-armed bandit problem. In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP 2006)*, 2006.

[350] Matthew J. Streeter and Stephen F. Smith. An asymptotically optimal algorithm for the max k-armed bandit problem. In *AAAI*, 2006.

[351] P. N. Strenski and Scott Kirkpatrick. Analysis of finite length annealing schedules. *Algorithmica*, 6:346–366, 1991.

[352] Roman G Strongin and Yaroslav D Sergeyev. *Global optimization with non-convex constraints: Sequential and parallel algorithms*, volume 45. Springer Science & Business Media, 2013.

[353] James Surowiecki. *The wisdom of crowds*. Random House Digital, Inc., 2005.

[354] Johan AK Suykens, Jos De Brabanter, Lukas Lukas, and Joos Vandewalle. Weighted least squares support vector machines: robustness and sparse approximation. *Neurocomputing*, 48(1):85–105, 2002.

[355] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[356] E.-G. Talbi. *Parallel Combinatorial Optimization*. John Wiley and Sons, USA, 2006.

[357] R. E. Tarjan. Updating a balanced search tree in o(1) rotations. *Information Processing Letters*, 16:253–257, 1983.

[358] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[359] Kai Ming Ting and Ian H Witten. Issues in stacked generalization. *Journal of Artificial Intelligence Research*, 10:271–289, 1999.

[360] D.A.D. Tompkins and H.H. Hoos. Warped landscapes and random acts of SAT solving. *Proc. of the Eighth Int'l Symposium on Artificial Intelligence and Mathematics (ISAIM-04)*, 2004.

[361] F. Hutter D.A.D. Tompkins and H.H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *Proc. Principles and Practice of Constraint Programming - CP 2002 : 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13*, volume 2470 of *LNCS*, pages 233–248. Springer Verlag, 2002.

[362] Kari Torkkola. Feature extraction by non parametric mutual information maximization. *The Journal of Machine Learning Research*, 3:1415–1438, 2003.

[363] Aimo Törn and Sami Viitanen. Topographical global optimization. *Recent advances in global optimization*, pages 384–398, 1992.

[364] G. Valentini and F. Masulli. Ensembles of learning machines. In M. Marinaro and R. Tagliaferri, editors, *Neural Nets WIRN Vietri-02*, Lecture Notes in Computer Sciences. Springer-Verlag, Heidelberg (Germany), 2002.

[365] Tony Van Gestel, Johan AK Suykens, Bart Baesens, Stijn Viaene, Jan Vanthienen, Guido Dedene, Bart De Moor, and Joos Vandewalle. Benchmarking least squares support vector machine classifiers. *Machine Learning*, 54(1):5–32, 2004.

[366] A. van Moorsel and K. Wolter. Analysis and algorithms for restart, 2004.

[367] Moshe Y Vardi. Is information technology destroying the middle class? *Communications of the ACM*, 58(2):5–5, 2015.

[368] M.G.A. Verhoeven and E.H.L. Aarts. Parallel local search. *Journal of Heuristics*, 1(1):43–65, 1995.

[369] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 9999:3371–3408, 2010.

[370] T.W.M. Vossen, M.G.A. Verhoeven, H.M.M. ten Eikelder, and E.H.L. Aarts. A quantitative analysis of iterated local search. Computing Science Reports 95/06, Department of Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, Netherlands, 1995.

[371] C. Voudouris and E. Tsang. Partial constraint satisfaction problems and guided local search. In *Proceedings of 2nd Int. Conf. on Practical Application of Constraint Technology (PACT 96), London*, pages 337–356, April 1996.

[372] Chris Voudouris and Edward Tsang. The tunneling algorithm for partial CSPs and combinatorial optimization problems. Technical Report CSM-213, 1994.

[373] Christos Voudouris and Edward Tsang. Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113:469–499, 1999.

[374] B.W. Wah and Z. Wu. Penalty Formulations and Trap-Avoidance Strategies for Solving Hard Satisfiability Problems. *Journal of Computer Science and Technology*, 20(1):3–17, 2005.

[375] C.J. Wang and E.P.K. Tsang. Solving constraint satisfaction problems using neural networks. In *Proc. Second International Conference on Artificial Neural Networks*, pages 295–299, 1991.

[376] R. Watrous. Learning algorithms for connectionist networks: applied gradient methods of nonlinear optimization. Technical Report MS-CIS-87-51, Univ. of Penn, 1987.

[377] Jean-Paul Watson, J. Christopher Beck, Adele E. Howe, and L. Darrell Whitley. Problem difficulty for tabu search in job-shop scheduling. *Artif. Intell.*, 143(2):189–217, 2003.

[378] E. Weinberger. Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biologial Cybernetics*, 63:325–336, 1990.

[379] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[380] S.R. White. Concepts of scale in simulated annealing. In *AIP Conference Proceedings*, volume 122, pages 261–270, 1984.

[381] D. Whitley, V.S. Gordon, and K. Mathias. Lamarckian Evolution, The Baldwin Effect and Function Optimization. In *Parallel Problem Solving from Nature–PPSN III: International Conference on Evolutionary Computation, Jerusalem, Israel*. Springer, 1994.

[382] Bernard Widrow, Aaron Greenblatt, Youngsik Kim, and Dookun Park. The no-prop algorithm: A new learning algorithm for multilayer neural networks. *Neural Networks*, 37:182–188, 2013.

[383] Bernard Widrow and Marcian E. Hoff. *Adaptive switching circuits*. Defense Technical Information Center, 1960.

[384] Bernard Widrow and Samuel D Stearns. Adaptive signal processing. *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1985, 491 p.*, 1, 1985.

[385] David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

[386] Margaret H Wright. Direct search methods: Once scorned, now respectable. *Pitman Research Notes in Mathematics Series*, pages 191–208, 1996.

[387] M. Yannakakis. On the approximation of maximum satisfiability. *Journal of Algorithms*, 17:475–502, 1994.

[388] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 2002.

[389] Q. Zhang and H. Li. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.

[390] Anatoly Zhigljavsky and Antanas Žilinskas. *Stochastic global optimization*, volume 9. Springer Science & Business Media, 2007.

[391] X. Zhu. Semi-supervised learning literature survey. *Computer Science, University of Wisconsin-Madison*, 2006.

[392] Xiaojin Zhu, Zoubin Ghahramani, John Lafferty, et al. Semi-supervised learning using Gaussian fields and harmonic functions. In *International Conference on Machine Learning*, volume 3, pages 912–919, 2003.

[393] Antanas Žilinskas. Axiomatic characterization of a global optimization algorithm and investigation of its search strategy. *Operations Research Letters*, 4(1):35–39, 1985.

[394] Antanas Žilinskas and J Žilinskas. Global optimization based on a statistical model and simplicial partitioning. *Computers & Mathematics with Applications*, 44(7):957–967, 2002.

[395] S. Zionts and J. Wallenius. An interactive multiple objective linear programming method for a class of underlying nonlinear utility functions. *Management Science*, 29(5):519–529, 1983.

Index