UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38100 Povo — Trento (Italy), Via Sommarive 14
http://dit.unitn.it/

REACTIVE SEARCH AND INTELLIGENT OPTIMIZATION

Roberto Battiti, Mauro Brunato, and Franco Mascia

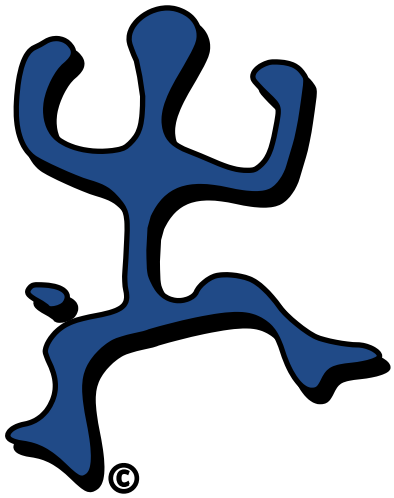
Technical Report # DIT-07-049

Reactive Search and Intelligent Optimization

Roberto Battiti, Mauro Brunato and Franco Mascia

Dipartimento di Informatica e Telecomunicazioni,
Università di Trento, Italy

Version 1.02, July 6, 2007
Technical Report DIT-07-049, Università di Trento, July 2007
Available at: `http://reactive-search.org/`
Email for correspondence: `battiti@dit.unitn.it`



Contents

Preface

Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza.

Li miei compagni fec'io sì aguti,
con questa orazion picciola, al cammino,
che a pena poscia li avrei ritenuti;

e volta nostra poppa nel mattino,
de' remi facemmo ali al folle volo,
sempre acquistando dal lato mancino.

Consider your origins:
you're not made to live as beasts,
but to follow virtue and knowledge.

My companions I made so eager,
with this little oration, of the voyage,
that I could have hardly then contained them;

that morning we turned our vessel,
our oars we made into wings for the foolish flight,
always gaining ground toward the left.

(Dante, Inferno Canto XXVI, translated by Anthony LaPorta)

We would like to thank our colleagues, friends and students for reading preliminary versions, commenting, and discovering mistakes. Of course we keep responsibility for the remaining ones. In particular we wish to thank Elisa Cilia and Paolo Campigotto for their fresh initial look at the book topics. Comments on different chapters have been submitted by various colleagues and friends, including: Matteo Gagliolo, Holger Hoos, Vittorio Maniezzo. This book is *version 1.0*, which means that we expect future releases in the next months as soon as we carve out reading and writing time from our daily chores. Writing a more detailed preface, including acknowledging all comments, also by colleagues who are reading version 1.0, is also on the stack. The Reactive Search website at `http://reactive-search.org/` is a good place to look for updated information. Finally, if you are working in areas related to Reactive Search and Intelligent Optimization and you do not find references here, we will be very happy to hear from you and to cite your work in the future releases.

Roberto, Mauro and Franco

Chapter 1

Introduction: Machine Learning for Intelligent Optimization

Errando discitur
Chi fa falla; e fallando s'impara.
You win only if you aren't afraid to lose. Rocky Aoki
Mistakes are the portals of discovery. James Joyce

This book is about learning for problem solving. Let's start with some motivation if you are not already an expert in the area, just to make sure that we talk about issues which are not far from everybody's human experience. Human problem solving is strongly connected to learning. Learning takes places when the problem at hand is not well known at the beginning, and its structure becomes more and more clear when more experience with the problem is available. For concreteness, let's consider skiing. What distinguishes an expert skier from a novice is that the novice knows some instructions but needs a lot experience to *fine tune* the techniques (with some falling down into local minima and restarts, so to speak) while the real expert jumps *seamlessly* from sensors to action, without effort and "symbolic" thinking. The knowledge accumulated from the previous experience has been *compiled into parameters of a neural system* working at very high speed. Think about you driving a car and try to explain in detail how you move your feet when driving: after so many years the knowledge is so compiled into your neural system that you hardly need any high-level thinking. Of course, this kind of fine tuning of problem-solving strategies and knowledge compilation into parameters of a dynamical system (our nervous system) is quite natural for us, while more primitive creatures are more rigid in their behavior. Think about a fly getting burnt by an incandescent light bulb (fooled because no light bulb was present during its genetic evolution apart from a distant one called "sun"). You know the rest of the story: the fly will get burnt again and again and again. No learning and fine tuning can have disastrous consequences. In addition to learning, search by trial-and-error, generation and test, repeated modifications of solutions by small local changes are also part of the human life.

What is critical for men is critical also in many human-developed problem solving strategies. It is not surprising that many methods for solving problems in Artificial Intelligence, Operations Research and related areas, follow the *search* scheme, for example *searching for an optimal configuration on a tree of possibilities* by adding one solution component at a time, and backtracking if a dead-end is encountered, or *searching by generating a trajectory of candidate solutions* on a landscape defined by the corresponding solution value.

For most of the relevant and difficult problems (see computational complexity at the voice "NP-hardness") researchers now believe that *the optimal solution cannot be found exactly in acceptable computing times*, which grow as a low-order polynomial of the input size. This is a well known negative result established in the last decades of theoretical computer science. Hardness of approximation, in addition to NP-hardness, is a kind of "affirmative action" for

heuristics.

Heuristics used to suffer from a bad reputation, citing from Papadimitriou and Steiglitz book [4]:

> *6. Heuristics* Any of the < five > approaches above without a formal guarantee of performance can be considered a "heuristic." However *unsatisfactory mathematically*, such approaches are certainly valid in practical situations.

Unfortunately, because of the "hardness of approximation" results [6], the hope of finding approximation algorithms with formal performance guarantees must be abandoned for many relevant problems. We are condemned to live with heuristics for very long times, maybe forever, and some effort is required to make them more satisfactory, both from a theoretical and from a practical point of view.

A particular delicate issue in many heuristics is their detailed tuning. Some schemes are not rigid but allow for the specification of choices in the detailed algorithm design, or values of internal parameters. Think about our novice skier, its detailed implementation ranging from world champion to ... the writers of this book: the difference in parameter tuning is evident. The term meta-heuristics is used in some cases to denote generic algorithmic schemes which can be specialized for different problems. We do not like the term too much because the boundary between the heuristic and the meta-heuristic is not always clear-cut.

Parameter tuning is a crucial issue both in the scientific development and in the practical use of heuristics. In some cases the detailed tuning is executed by a researcher or by a final user. As parameter tuning is user dependent the reproducibility of the heuristics results is difficult as is comparing different parametric algorithms. Algorithm A can be better than algorithm B if tuned by Roberto, while it can be worse if tuned by Mauro.

In this book we consider some machine learning methods which can be profitably used in order to **automate the tuning** process and make it an integral and **fully documented** part of the algorithm. In particular the focus is on *sub-symbolic learning* schemes, where the accumulated knowledge is compiled into the parameters of the method, or the parameters regulating a dynamical system to build or search for a solution. In many cases sub-symbolic learning schemes will work without giving a high-level symbolic explanation. Think about neural networks, think about the champion skier who cannot explain how he allocates forces to the different muscles during a slalom.

If learning acts *on-line*, i.e., while the algorithm is solving an instance of a problem, **task-dependent local properties** can be used by the algorithm to determine the appropriate balance between *diversification* and *intensification*. Deciding whether it is better to look for gold where the other miners are excavating (*intensification/exploitation*) or to go and explore other valleys and uncharted territories (*diversification/exploration*) is an issue which excruciated forty-niners and which we will meet over and over again in the following chapters. Citing for example from [5]"diversification drives the search to examine new regions, and intensification focuses more intently on regions previously found to be good. (Intensification typically operates by re-starting from high quality solutions, or by modifying choice rules to favor the inclusion of attributes of these solutions)".

1.1 Parameter tuning and intelligent optimization

As we mentioned, many state-of-the-art heuristics are characterized by a certain number of **choices and free parameters**, whose appropriate setting is a subject that raises issues of research methodology [1, 2, 3]. 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. The Machine

Learning community, with significant influx from Statistics, developed in the last decades a rich variety of "design principles" that can be used to develop machine learning methods and algorithms and that can be profitably used in the area of parameter tuning for heuristics. In this way one *eliminates the human intervention*. This 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 complication will increase, but the price is worth paying if the two following objectives are reached:

- **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 from the scientific point of view, where objective evaluations are crucial. The recent introduction of software archives further simplifies the test and **simple re-use** of heuristic algorithms.
- **Automation**. The time-consuming tuning phase is now substituted by an automated process. 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, with a possible preliminary phase of exploratory tests, followed by the above described exhaustive documentation of the tuning process when the algorithm is presented to the scientific community.

In particular, **Reactive Search** advocates the integration of sub-symbolic machine learning techniques into local search heuristics for solving complex optimization problems. The word reactive hints at a ready response to events during the search through an internal *online feedback loop for the self-tuning of critical parameters*. In Reactive Search the past history of the search is used for:

feedback-based parameter tuning the algorithm maintains the internal flexibility needed to cover many problems, but the tuning is automated, and executed while the algorithm runs and "monitors" its past behavior.

automated balance of diversification and intensification An automated heuristic balance for the the "exploration versus exploitation" dilemma can be obtained through feedback mechanisms, for example by starting with intensification, and by progressively increasing the amount of diversification only when there is evidence that diversification is needed.

1.2 Book outline

The book does not aim at a complete coverage of the widely expanding research area of heuristics, meta-heuristics, stochastic local search, etc. The task would be daunting and bordering on the myth of Sisyphus, condemned by the gods to ceaselessly rolling a rock to the top of a mountain, whence the stone would fall back of its own weight. The rolling stones are in this case caused by the rapid development of new heuristics for many problems which would render a book obsolete after a short span.

We aim at giving the main principles and at developing some fresh intuition for the approaches. We like mathematics but we also think that hiding the underlying motivations and sources of inspiration takes some color out of the scientific work ("Grau, teurer Freund, ist alle Theorie. Und grün des Lebens goldner Baum" — Johann Wolfgang von Goethe). On the other hand, pictures and hand-waving can be very dangerous in isolation and we try to avoid these pitfalls by giving also the basic equations when possible, or by at least directing the reader to the bibliographic references for deepening a topic.

The point of view of the book is to look at the zoo of different optimization beasts to underline *opportunities for learning and self-tuning strategies*.

The focus is definitely more on methods than on problems. We introduce some problems to make the discussion more concrete or when a specific problem has been widely studied by reactive search and intelligent optimization heuristics.

Intelligent optimization, the application of machine learning strategies in heuristics it in itself a very wide area, and the space dedicated to *reactive search* techniques (online learning techniques applied to local search) is wider because of personal interest. We plan to produce a more balanced version in the future.

The structure of the following chapters is as follows: i) the basic issues and algorithms are introduced, ii) the parameters critical for the success of the different methods are identified, iii) opportunities and schemes for the automated tuning of these parameters are presented and discussed.

Bibliography

[1] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. Stewart, *Designing and reporting on computational experiments with heuristic methods*, Journal of Heuristics **1** (1995), no. 1, 9–32.

[2] J.N. Hooker, *Testing heuristics: We have it all wrong*, Journal of Heuristics **1** (1995), no. 1, 33–42.

[3] Catherine C. McGeoch, *Toward an experimental method for algorithm simulation*, INFORMS Journal on Computing **8** (1996), no. 1, 1–28.

[4] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization, algorithms and complexity*, Prentice-Hall, NJ, 1982.

[5] Y. Rochat and E. Taillard, *Probabilistic diversification and intensification in local search for vehicle routing*, Journal of Heuristics **1** (1995), no. 1, 147–167.

[6] M. Sudan, *Efficient checking of polynomials and proofs and the hardness of approximation problems*, Ph.D. thesis, Computer Science Division, University of California at Berkeley, 1992.

Chapter 2

Reacting on the neighborhood

How many shoe-shops does one person visit before making a choice?
There is not a single answer, please specify whether male or female!

2.1 Local search based on perturbation

A basic problem-solving strategy consists of starting from an initial tentative solution and trying to improve it through repeated small changes. At each repetition the current configuration is slightly modified (*perturbed*), the function to be optimized is tested, the change is kept if the new solution is better, otherwise another change is tried. The function $f(X)$ to be optimized is called with more poetic names in some papers: *fitness* function, *goodness* function, *objective* function.

Fig 2.1 shows an example in the history of bike design (do not expect historical fidelity here, this book is about algorithms!). Model A is a starting solution with a single wheel, it works but it is not optimal yet. Model B is a randomized attempt to add some pieces to the original design, the situation is worse. One could revert back to model A and start other changes. But let's note that, if one insists and proceeds with a second addition, one may end up with Model C, clearly superior from a usability and safety point of view! This real life 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. Let's note in passing that everybody's life is an example of an optimization algorithm in action: most of the changes are localized, dramatic changes do happen, but not so frequently. The careful reader may notice that the goodness function of our life is not so clearly defined. To this we answer that this book is not about philosophy, let's stop here with far-fetched analogies and go down to the nitty-gritty of the algorithms.

Local search based on perturbing a candidate solution, which we assume already known to the reader, is a first paradigmatic case where simple learning strategies can be applied. Let's define the notation. $\mathcal{X}$ is the search space, $X^{(t)}$ is the current solution at iteration ("time") t. $N(X^{(t)})$ is the neighborhood of point $X^{(t)}$, obtained by applying a set of basic moves $\mu_0, \mu_1, ..., \mu_M$ to the current configuration:

$$N(X^{(t)}) = \{X \in \mathcal{X} \text{ such that } X = \mu_i(X^{(t)}), i = 0, ...M\}$$

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.

Local search starts from an admissible configuration $X^{(0)}$ and builds a *search trajectory* $X^{(0)}, ..., X^{(t+1)}$ where the successor of the current point is a point in the neighborhood with a lower value of the function f to be minimized. If no neighbor has this property, i.e., if the configuration is a local minimizer, the search stops.

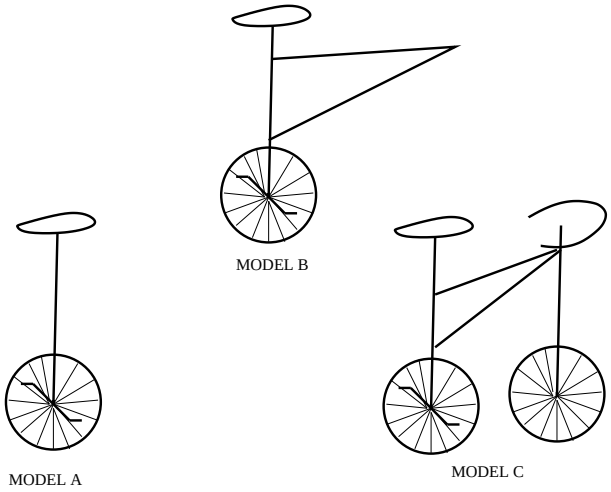


Figure 2.1: A local search example: how to build a better bike.

$$Y \leftarrow \text{IMPROVING-NEIGHBOR}(\ N(X^{(t)})\ ) \tag{2.1}$$

$$X^{(t+1)} = \begin{cases} Y & \text{if } f(Y) < f(X^{(t)}) \\ X^{(t)} & \text{otherwise (search stops)} \end{cases} \tag{2.2}$$

IMPROVING-NEIGHBOR returns an improving element in the neighborhood. In a simple case this is the element with the lowest f values, but other possibilities exist, as we will see in what follows.

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 $f(x)$ — continuous with continuous derivatives. If one stays in the neighborhood, the change is limited by the maximum value of the derivative multiplied by the displacement. Combinatorial optimization needs different measures to quantify the notion that a small change of the configuration is coupled, at least in a statistical way, to a small change of f, see also Chapter 9.

Now, a neighborhood is suitable for local search if it reflects the problem structure. For example, if the solution is given by a permutation (like in TSP, or in sorting) an improper neighborhood would be to consider single bit changes of a binary string describing the current solution, which would immediately cause illegal configurations. 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. By the way, 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.

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

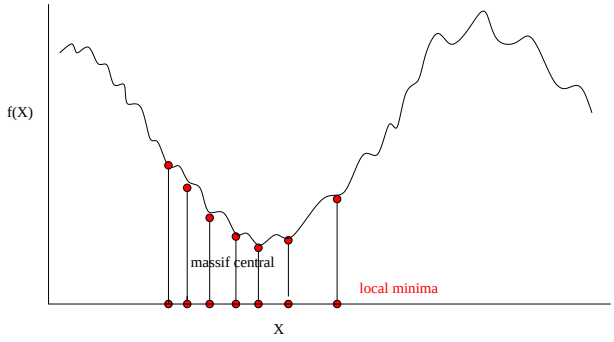


Figure 2.2: Structure in optimization problems: the "massif central" hypothesis.

minima. **Local minima like to be in good company!** Let us define as *attraction basin* associated to a local optimum the set of points X which are mapped to the 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 (analogies have limits and we stop at a lake otherwise we would end up trivially at the same global optimum, the ocean!).

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. Try to move from the bottom of the Grand Canyon to the Death Valley if not convinced. This structural property is also called *Big Valley*, or *massif central*, see also [25] for a probabilistic analysis of local minima distribution in the Traveling Salesman Problem (TSP), and Chapter 9 for additional comments.

To help the intuition, one may think about a smooth f surface in a continuous environment, with basins of attractions which tend to have a nested "fractal" structure, see Fig. 2.2. 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 expert about Fourier transforms, think about looking at a figure with de-focusing lenses. At first the large scale details will be revealed (for example a figure of 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. What is important is that, at each scale, the situation is not random noise and a pattern, a 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 Search (VNS), see Section 2.3 and Iterated Local Search (ILS), see Section 2.4.

2.2 Learning how to evaluate the neighborhood

It looks like there is little online or off-line learning to be considered for a simple technique like local search. A closer look reveals some possibilities. 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. It is well known that a neighborhood appropriate to a given problem is the most critical issue in the development of efficient strategies, and this is mostly related to off-line learning techniques, like those used formally or informally by researchers during the algorithm design process. But 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. A third option is of course to adapt both choices during a run.

Let's start from the first context where a neighborhood is chosen before the run is started, and only the process to select an improving neighbor is dynamic during the run. 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 here: 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 a second. 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 FIRSTMOVE. If no such candidate exists the trajectory is at a local optimum. A randomized examination order can be used to avoid spurious effect due to a specific examination order. 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 may expect that a small number of candidates needs to be evaluated in the early phase of the search, while identifying an improving move will become more and more difficult during the later phases, when the configuration will be 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 repeat, if the number of examined neighbors is adapted to the evaluation results (keep evaluating until either an improving neighbor is found or all neighbors have been examined) no user intervention is needed for the self-tuning of the appropriate number of neighbors.

Another possibility consists of examining a *a random subset* of neighbors, while ensuring that the sample is representative of the entire neighborhood (for example stopping the examination when the possibility of finding better improving values is not worth the additional computational effort).

A preliminary radical proposal which avoids analyzing any neighborhood and which chooses a neighbor according to *utility values determined by reinforcement learning* [24] is presented in [20]. The neighbor choice is dictated by the best-utility one among a set of *repair* heuristics associated to constraints in Constraint Satisfaction Problems. The purpose is to "switch between different heuristics during search in order to adapt to specific regions of the search space."

In the next Section we will consider more structured strategies where the neighborhood is not fixed at the beginning and the appropriate neighborhood to use at a given iteration is picked *from a set of different neighborhoods*.

2.3 Learning the appropriate neighborhood in variable neighborhood search

Consider the three possible sources of information to adapt a proper neighborhood: problem, specific instance, and current position in the search space. There are cases when no optimal and fixed neighborhood is defined for a problem because of lack of information, or cases when adaptation of the neighborhood to the local configuration is beneficial.

To design an automated neighborhood tuning technique one has to specify the amount of variability among the possible neighborhood structures. A possible way is to consider **a set**

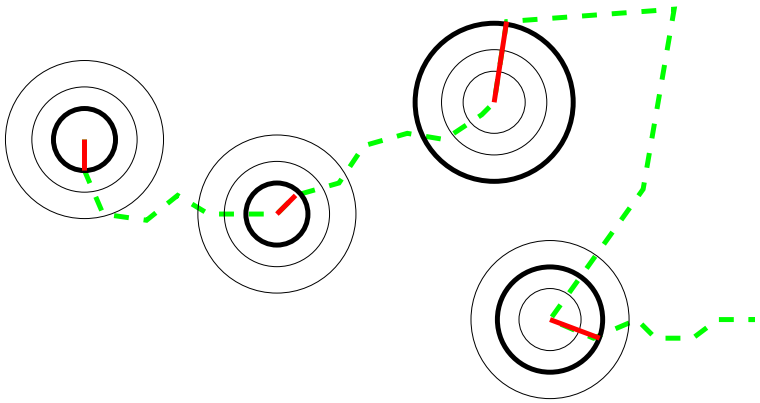


Figure 2.3: Variable neighborhood search: the used neighborhood ("circle" around the current configuration) varies along the search trajectory.

of neighborhoods, defined *a priori* at the beginning of the search, and then aim at using the most appropriate one during the search, as illustrated in Fig. 2.3. This is the seminal idea of the Variable Neighborhood Search (VNS) technique, see [8].

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.)

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 [3]. One orders the neighborhoods according to their "diameter" (or to the *strength* of the perturbation executed, or to the distance from the starting configuration to the neighbors measured with an appropriate metric). For example, if the search space is given by binary strings and the distance is the Hamming distance, one may consider as N_1 changes of a single bit, N_2 changes of two bits, etc. The default neighborhood N_1 is the one with the least diameter, if local search makes progress one sticks to this default neighborhood. As soon as a local minimum with respect to N_1 is encountered one tries to go to greater distances from the current point aiming at discovering a nearby attraction basin, possibly leading to a better local optimum.

Fig. 2.4 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

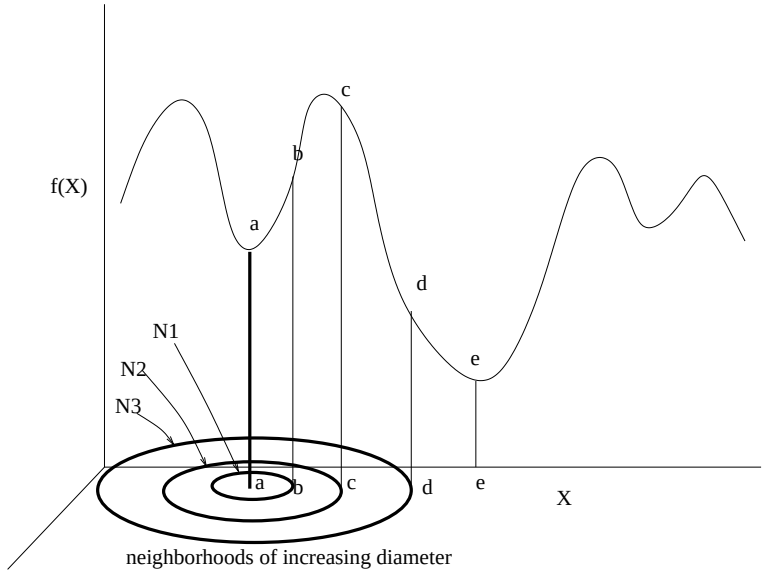


Figure 2.4: Variable neighborhoods of different diameters. Neighboring points are on the circumferences at different distances. The figure is intended to help the intuition: the actual neighbors considered in the text are discrete.

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? For example by running local search from it and by looking 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 Chapter 4, 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 schemes for using the set of different neighborhoods in an organized way are possible [10]. Variable Neighborhood Descent (VND), see Fig. 2.5, uses the default neighborhood first, and the ones with a higher number 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 one 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. 2.5 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

```
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_{textmax}:
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 2.5: 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 SkewedVariableNeighborhoodDescent(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))                        shake
6.     X'' ← LocalSearch(X')                              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 2.6: 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.

Fig. 2.6. This version requires a suitable distance function $\rho(X, X')$ between two solutions, e.g., the Hamming distance for binary strings, and a *skewness* parameter α to regulate the trade-off between movement distance and willingness to accept worse values. By looking at Fig. 2.4, 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.

To determine an empirical α value [10] one could resort to a preliminary investigation about the distribution of local minima, by using a multi-start version of VNS: one repeatedly generates random initial configurations and runs VNS to convergence. Then one studies the behavior of their expected f values as a function of the distance from the best known solution. After collecting the above experimental data one at least knows some reasonable ranges for α. For example one may pick α such that the probability of success of the escape operation is different from zero. In any case a more principled way of determined α is a topic worth additional investigation. *Valley profiles* are a second useful source of statistical information for designing an appropriate VNS strategy. They are obtained by extracting random starting points from different neighborhoods around a given point and by executing local search. Then one estimates the probabilities that the trajectories go back to the initial point or to other attraction basins and derives suggestions about the "strength of the kick" which is needed to escape with a certain probability.

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 [19, 17], where "kicks" of appropriate strength are used to exit from local minima, see also Section 2.4 about Iterated Local Search.

An explicitly reactive-VNS is considered in [5] for the VRPTW problem (vehicle routing 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 [11]. 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 Affine Shaker algorithm in [2], is mainly considered to speed-up convergence to a local minimizer, not to jump to nearby valleys.

2.4 Iterated local search

If a local search "building block" is available, for example as a concrete software library, how can it be used by some upper layer coordination mechanism as a black box to get better results? An answer is given by *iterating* calls to the local search routine each time starting from a properly chosen configuration. Of course, if the starting configuration is random, one starts from scratch and knowledge about the previous searches is lost. This trivial form actually is called simply **repeated local search**.

Learning implies that the previous history (for example information 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 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

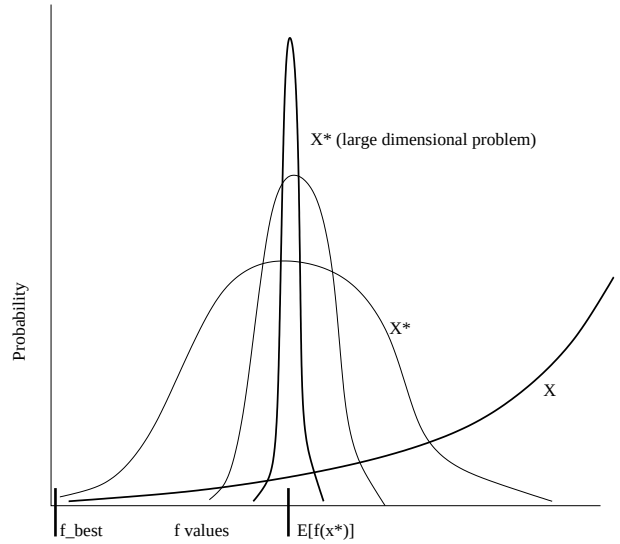


Figure 2.7: Probability of low f values is larger for local minima in $\mathcal{X}^*$ than for a point randomly extracted from $\mathcal{X}$. Large-dimensional problems tend to have a very peaked distribution for $\mathcal{X}^*$ values.

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 [4] a local minimum of the depot location problem is perturbed by placing artificial "hazards" zones. As soon as a new point is reached, the hazard is eliminated and the usual local search starts again. Let's note that hazard zone are penalizing routes inside them: the f value is increased by a penalty value. In other words, the perturbation is obtained by temporarily modifying the objective function so that the old local minimum will now be suboptimal with the modified function. A quarter of a century ago one already finds seminal ideas related to iterated local search and guided local search [26].

One may also argue that iterated local search, at least its more advanced implementations, shares many design principles with variable neighborhood search. A similar principle is active in the iterated Lin-Kernighan algorithm of [12], 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 [19, 17, 18, 25] contains very interesting results coupled with a clear description of the principles.

Our description follows mainly [15]. 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, objective function values 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. 2.7.

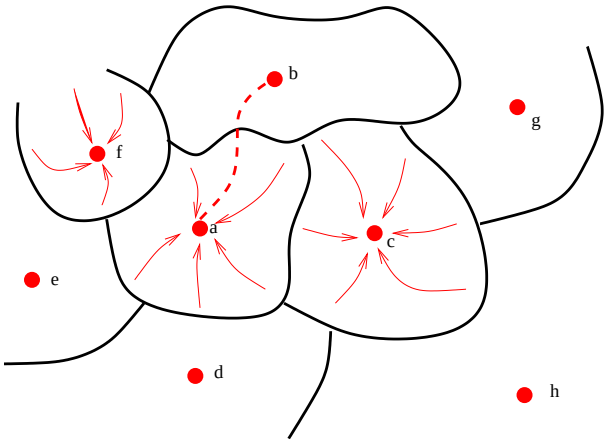


Figure 2.8: Neighborhood among attraction basins induced a neighborhood definition on local minima in $\mathcal{X}^*$.

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 [21] 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 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 is 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 mapped to the given optimum by local search, one says that 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., there are points on the boundary where a point in the other attraction basin is a neighbor in the original space $\mathcal{X}$. For example, in Fig. 2.8, 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. 2.9 and Fig. 2.10: generate a sufficiently strong perturbation leading to a new point and then apply local search until convergence at a local minimum.

One has to determine the appropriate *strength* of the perturbation, furthermore one has

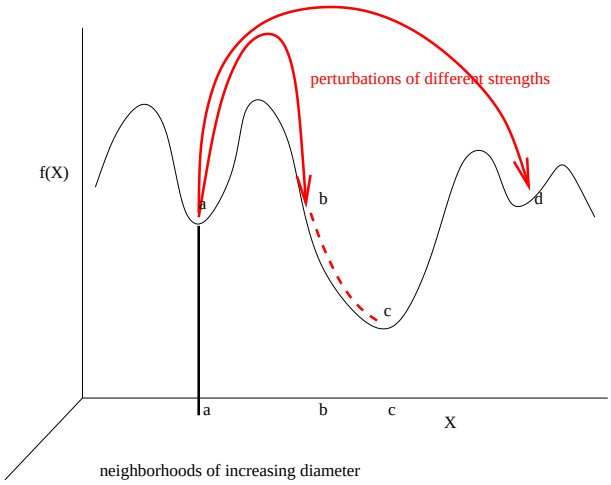


Figure 2.9: ILS: a perturbation leads from a to b, then local search to c. If perturbation is too strong one may end up at d therefore missing the closer local minima.

1. **function IteratedLocalSearch** ()
2. $X^0 \leftarrow$ **InitialSolution**()
3. $X^* \leftarrow$ **LocalSearch** (X^0)
4. **repeat**
5. $k \leftarrow 1$ *default neighborhood*
6. **while** $k \leq k_{\max}$
7. $X' \leftarrow$ **Perturb** (X^*, history)
8. $X^{*'} \leftarrow$ **LocalSearch** (X')
9. $X^* \leftarrow$ **AcceptanceDecision** (X^*, $X^{*'}$, history)
10. **until** (no improvement or termination condition)

Figure 2.10: Iterated Local Search

to avoid cycling: if the perturbation is too small there is the risk that the solution returns back to the starting local optimum. After this happens, an endless cycle is produced if the perturbation and local search are deterministic.

Learning based on the previous search history is of paramount importance also in this case [23]. 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.

While simple implementations of ILS are often adequate to improve on local search results, and do not require opening the "black box" local search, high performance can be obtained by *jointly optimizing the four basic components*: INITIALSOLUTION, LOCALSEARCH, PERTURB, and ACCEPTANCEDECISION. Greedy construction is often recommended in INITIALSOLUTION to identify quickly low-cost solutions, while the effect of the initial point is less relevant for long runs provided that sufficient exploration is maintained by the algorithm. Memory and reactive learning can be used in a way similar to that of [1] to adapt the *strength* of the perturbation to the local characteristics in the neighborhood of the current solution for the considered instance. Creative perturbations can be obtained by temporarily changing the objective function with penalties so that the current local minimum is displaced, like in [4, 6], or by *fixing* some configuration variables and by optimizing sub-parts of the problem [16]. The ACCEPTANCEDECISION in its different realizations can range from a strict requirement of improvement, which accepts a move only if it improves the f value, to a very relaxed *random walk* which always accepts the randomly generated moves to favor diversification, to an intermediate *simulated annealing* test, which accepts a move depending on the f difference and on a *temperature* parameter T, with probability: $\exp\{(f(X^*) - f(X^{*\prime}))/T\}$, leading to the *large-step Markov chain* implementation of [17, 18].

It is already clear, and it will hopefully become more so in the following chapters, 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 robustness and permit more *hands-off* usage of software for optimization.

In particular, the Simulated Annealing method [14] is a popular technique to allow also worsening moves in a stochastic way. In other schemes, diversification can be obtained through the temporary *prohibition* of some local moves. Glover [7] and, independently, Hansen and Jaumard [9] popularized the Tabu Search (TS) approach. Similar ideas have been used in the past for the Traveling Salesman [22] and graph partitioning [13] problems.

We will encounter these more complex techniques in the following chapters.

Bibliography

[1] R. Battiti and M. Protasi, *Reactive search, a history-sensitive heuristic for MAX-SAT*, ACM Journal of Experimental Algorithmics **2** (1997), no. ARTICLE 2, http://www.jea.acm.org/.

[2] R. Battiti and G. Tecchiolli, *Learning with first, second, and no derivatives: a case study in high energy physics*, Neurocomputing **6** (1994), 181–206.

[3] _____, *The reactive tabu search*, ORSA Journal on Computing **6** (1994), no. 2, 126–140.

[4] John Baxter, *Local optima avoidance in depot location*, The Journal of the Operational Research Society **32** (1981), no. 9, 815–819.

[5] O. Braysy, *A reactive variable neighborhood search for the vehicle-routing problem with time windows*, INFORMS JOURNAL ON COMPUTING **15** (2003), no. 4, 347–368.

[6] B. Codenotti, G. Manzini, L. Margara, and G. Resta, *Perturbation: An efficient technique for the solution of very large instances of the euclidean tsp*, INFORMS JOURNAL ON COMPUTING **8** (1996), no. 2, 125–133.

[7] F. Glover, *Tabu search - part i*, ORSA Journal on Computing **1** (1989), no. 3, 190–260.

[8] N. Mladenovic P. Hansen, *Variable neighborhood search*, Computers and Operations Research **24** (1997), no. 11, 1097–1100.

[9] P. Hansen and B. Jaumard, *Algorithms for the maximum satisfiability problem*, Computing **44** (1990), 279–303.

[10] P. Hansen and N. Mladenovic, *A tutorial on variable neighborhood search*, Tech. Report ISSN: 0711-2440, Les Cahiers du GERAD, Montreal, Canada, July 2003.

[11] Bin Hu and Gnther R. Raidl, *Variable neighborhood descent with self-adaptive neighborhood-ordering*, Proceedings of the 7th EU/MEeting on Adaptive, Self-Adaptive, and Multi-Level Metaheuristics, malaga, Spain (Carlos Cotta, Antonio J. Fernandez, and Jose E. Gallardo, eds.), 2006.

[12] D.S. Johnson, *Local optimization and the travelling salesman problem*, Proc. 17th Colloquium on Automata Languages and Programming, LNCS, vol. 447, Springer Verlag, Berlin, 1990.

[13] B. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Technical J. **49** (1970), 291–307.

[14] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), 671–680.

[15] H. R. Lourenco, O. C. Martin, and T. Stutzle, *Iterated local search*, ISORMS **57** (2002), 321.

[16] H.R. Lourenco, *Job-shop scheduling: Computational study of local search and large-step optimization methods*, European Journal of Operational Research **83** (1995), 347–364.

[17] Olivier Martin, Steve W. Otto, and Edward W. Felten, *Large-step markov chains for the traveling salesman problem*, Complex Systems **5:3** (1991), 299.

[18] _____, *Large-step markov chains for the tsp incorporating local search heuristics*, Operation Research Letters **11** (1992), 219–224.

[19] Olivier C. Martin and Steve W. Otto, *Combining simulated annealing with local search heuristics*, ANNALS OF OPERATIONS RESEARCH **63** (1996), 57–76.

[20] Alexander Nareyek, *Choosing search heuristics by non-stationary reinforcement learning*, (2004), 523–544.

[21] G. R. Schreiber and O. C. Martin, *Cut size statistics of graph bisection heuristics*, SIAM JOURNAL OF OPTIMIZATION **10** (1999), no. 1, 231–251.

[22] K. Steiglitz and P. Weiner, *Algorithms for computer solution of the traveling salesman problem*, Proceedings of the Sixth Allerton Conf. on Circuit and System Theory, Urbana, Illinois, IEEE, 1968, pp. 814–821.

[23] T. Stuetzle, *Local search algorithms for combinatorial problems - analysis, improvements, and new applications*, Ph.D. thesis, Darmstadt University of Technology, Dept. of Computer Science, 1998.

[24] Richard S. Sutton and Andrew G. Barto, *Reinforcement learning: An introduction*, MIT Press, 1998.

[25] 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.

[26] Christos Voudouris and Edward Tsang, *Guided local search and its application to the traveling salesman problem*, European Journal of Operational Research **113** (1999), 469–499.

Chapter 3

Reacting on the annealing schedule

Temperature schedules are cool

3.1 Stochasticity in local moves and controlled worsening of solution values

As mentioned previously, local search stops at a locally optimal point and for problems with a rich internal structure encountered in many applications, *searching in the vicinity of good local minima* may lead to the discovery of better minima. In the previous chapter 2 one changed the neighborhood structure in order to push the trajectory away from the local minimum. In this chapter the neighborhood structure is *fixed*, but the move generation and acceptance are stochastic and one permits also a "controlled worsening" of solution values aiming at escaping from the local attractor. Let's note that a "controlled worsening" has already been considered in the SKEWED-VNS routine of Section 2.3, where one accepted worsening moves provided that they lead the trajectory sufficiently far.

Now, if one extends local search by *accepting also 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 starting local minimum will be chosen at the second iteration, leading to an endless cycle of "trying to escape and falling back immediately to the starting point". This situation surely happens if the local minimum is strict (all neighbors have worse f values) and if more than one step is needed before points with f values better than that of the local minimum become accessible (until they become neighbors of the current solution point).

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 Simulated Annealing (SA) method has been investigated. We will summarize the technique, with a hint at mathematical results, and then underline opportunities for self-tuning.

3.2 Simulated Annealing and Asymptotics

The Simulated Annealing method [17] 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.

$$
\begin{aligned}
Y &\leftarrow \text{NEIGHBOR}(\ N(X^{(t)})\ ) \\
X^{(t+1)} &\leftarrow \begin{cases} Y & \text{if } f(Y) \leq f(X^{(t)}) \\ Y & \text{if } f(Y) > f(X^{(t)}), \text{ with probability } p = e^{-(f(Y)-f(X^{(t)}))/T} \\ X^{(t)} & \text{if } f(Y) > f(X^{(t)}), \text{ with probability } (1-p). \end{cases}
\end{aligned} \tag{3.1}
$$

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. The rule in (3.1) 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, 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 [19] and in optimization [24, 7, 17, 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 [11], and even random search [8] have better asymptotic results for some problems.

Given the limited practical relevance of the theoretical results, a practitioner has to place asymptotic results in the background and develop heuristics where high importance is attributed to learning from a task before the search is started and from the current local characteristics during the search. In the following, after a brief introduction about some asymptotic convergence results, we will concentrate on the more recent developments in SA considering adaptive and learning implementations.

3.2.1 Asymptotic convergence results

Let $(\mathcal{S}, f)$ be an instance of a combinatorial optimization problem, $\mathcal{S}$ being the search space and f being the objective function. Let $\mathcal{S}^*$ be the set of optimal solutions. One starts from an initial configuration $X^{(0)}$ and repeatedly applies (3.1) to generate a trajectory $X^{(t)}$. Under appropriate conditions, the probability of finding one of the optimal solutions when the number of iterations go to infinity tends to one:

$$\lim_{k\to\infty} \Pr(X^{(k)} \in \mathcal{S}^*) = 1. \tag{3.2}$$

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:

$$P_{ij}(k) = \Pr(X^{(k)} = j | X^{(k-1)} = i). \tag{3.3}$$

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

$$q_i = \lim_{k\to\infty} \Pr(X^{(k)} = i | X^{(0)} = j), \text{ for all } j \in \mathcal{O} \tag{3.4}$$

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:

$$\sum_{j\in\mathcal{O}} q_j P_{ji} = q_i \tag{3.5}$$

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:

$$q_i(T) = \frac{e^{-f(i)/T}}{\sum_{j\in\mathcal{S}} e^{-f(j)/T}} \tag{3.6}$$

and

$$\lim_{T\to 0} q_i(T) = q_i^* = \frac{1}{|\mathcal{S}^*|}\chi_{\mathcal{S}^*}(i) \tag{3.7}$$

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

It follows that:

$$\lim_{T\to 0}\lim_{k\to\infty} \Pr_T(X^{(k)} \in \mathcal{S}^*) = 1 \tag{3.8}$$

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 T_k must be lowered before converging: one has 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:

$$T_k \geq \frac{A}{\log(k + k_0)} \tag{3.9}$$

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

$$\lim_{k\to\infty} \Pr(X^{(k)} \in \mathcal{S}^*) = 1 \tag{3.10}$$

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!).

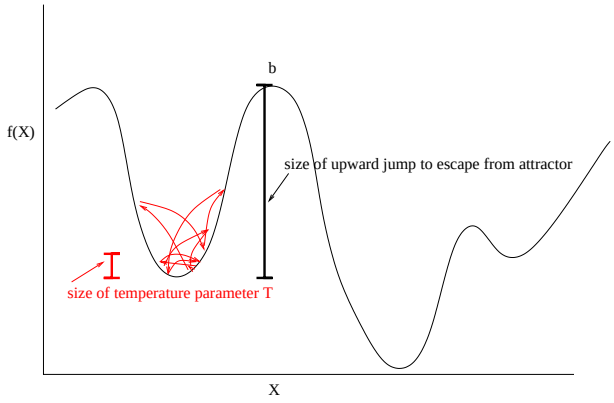


Figure 3.1: Simulated Annealing: if the temperature is very low w.r.t. the jump size SA risks a practical entrapment close to a local minimizer.

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 [11], and even random search [8] 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 [26].

More details about cooling schedules can be found in [20, 13]. Extensive experimental results of SA for graph partitioning, coloring and number partitioning are presented in [15, 16]. A comparison of SA and Reactive Search has been presented in [5, 6].

3.3 Online learning strategies in simulated annealing

If one wants to be poetic, the main feature of simulated annealing lies in its asymptotic convergence properties, the main drawback lies in the *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, by developing simple models and by activating more direct *escape* strategies aiming at a better time-management than the "let's go to infinite time" principle. In the following sections we will summarize the main memory-based approaches developed in the years to make SA a competitive strategy.

3.3.1 Combinatorial optimization problems

Even if a vanilla version of 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 proper scale of the temperature is wrong, extremely poor results are to be expected. The work [27] suggests to estimate: the mean of the distribution of f values (f is usually called "energy" when exploiting physical analogies) to define a maximum energy scale of the system, its standard deviation to define the maximum-temperature scale, and the minimum change in energy to define the minimum-temperature scale. These temperature scales tell us where to begin and end an annealing schedule (this is the same as the cooling schedule introduced before).

The analogy with physics is pursued in [18], where concepts related to *phase transitions* and *specific heat* are used. While we avoid entering physics, the idea is that 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 signalled by wide variations of the f values. In detail, 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 heuristically a slower decrease of the temperature. 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} (T_{msp} is the temperature corresponding to the maximum specific heat, when the scaled variance reaches its maximal value) one switches from a faster temperature decrease given by α to the slower one given by β.

Up to now we discussed only about a monotonic decrease of the temperature. This process 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 slow 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 3.1. In other words, given a set of parameters T_{start} and $alpha$, 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. Now, for a new instance, it is not so simple to guess appropriate parameter values. Furthermore, in many cases one would like to use 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 the stopping criterion in many cases should be decided *a posteriori*, for example after determining that additional time has little probability to improve significantly on the result.

Because this problem is related to a monotonic temperature decrease, a motivation arises to consider *non-monotonic cooling schedules*, see [9, 23, 3]. A very simple proposal [9] suggests resetting 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 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) *increase the temperature* to diversify the solution and visit other attraction basins, iii) decrease 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 of 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 [23]. 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 [4] see Chapter 4, and in variable neighborhood search, see Chapter 2.

Modifications departing from the exponential acceptance rule and adaptive stochastic local search methods for combinatorial optimization are considered in [21, 22]. Experimental evidence shows that the *a priori* determination of SA parameters and acceptance function does not lead to efficient implementations. 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 [22] 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 can be seen as deriving evidence of "entrapment" in a local minimum and activating reactively 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" [22].

"Cybernetic" optimization is proposed in [12] as a way to use probabilistic information for feedback during a run of SA. The idea is to consider more runs of SA running in parallel and to aim at *intensifying the search* (by lowering the temperature parameter) when there is evidence that the search is converging to the optimum value. If one looks for gold, one should spend more time "looking where all the other prospectors are looking" [12] (but let's note that actually one may argue differently depending on the luck of the other prospectors!). The empirical evidence is taken from the similarity between current configurations of different parallel runs. For sure, if more solutions are close to the same optimum point, they are also close to each other. The contrary is not necessarily true, nonetheless this is taken as evidence of a (possible) closeness to the optimum point, implying intensification and causing in a reactive manner a gradual reduction of the temperature.

3.3.2 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 [10]. The basic method is to generate a new point with a random step along a direction $\boldsymbol{e}_h$, evaluate the function and accept the move with the probability given in (3.1). 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. In particular, after a number N_{S} of random moves cycling along the coordinates directions, the step-range vector $\boldsymbol{v}$ is updated: each component is increased if the number of accepted moves is greater than 60%, reduced if it is less than 40%. The speed of increase/decrease could be different for the different coordinate dimensions (in practice it is fixed to 2 and 1/2 in the above paper). After $N_{\text{S}}N_{\text{T}}$ cycles, (N_{T} being a second fixed parameter), the temperature is reduced in a multiplicative manner: $T_{k+1} \leftarrow r_T T_k$, and the current point is reset to the best-so-far point found during the previous search. 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 [14] the parameters that control temperature cooling schedule and 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 late stages [14].

Bibliography

[1] E. H. L. Aarts, J.H.M. Korst, and P.J. Zwietering, *Deterministic and randomized local search*, Mathematical Perspectives on Neural Networks (M. Mozer P. Smolensky and D. Rumelhart (Eds.), eds.), 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** (1989), 79–95.

[3] D. Abramson, H. Dang, and M. Krisnamoorthy, *Simulated annealing cooling schedules for the school timetabling problem*, Asia-Pacific Journal of Operational Research **16** (1999), 1–22.

[4] R. Battiti and G. Tecchiolli, *The reactive tabu search*, ORSA Journal on Computing **6** (1994), no. 2, 126–140.

[5] ———, *Simulated annealing and tabu search in the long run: a comparison on qap tasks*, Computer and Mathematics with Applications **28** (1994), no. 6, 1–8.

[6] ———, *Local search with memory: Benchmarking rts*, Operations Research Spektrum **17** (1995), no. 2/3, 67–86.

[7] V. Cherny, *A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm*, Journal of Optimization Theory and Applications **45** (1985), 41–45.

[8] T.-S. Chiang and Y. Chow, *On the convergence rate of annealing processes*, SIAM Journal on Control and Optimization **26** (1988), no. 6, 1455–1470.

[9] D.T. Connolly, *An improved annealing scheme for the qap*, European Journal of Operational Research **46** (1990), no. 1, 93–100.

[10] 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** (1987), no. 3, 262–280.

[11] A.G. Ferreira and J. Zerovnik, *Bounding the probability of success of stochastic methods for global optimization*, Computers Math. Applic. **25** (1993), no. 10/11, 1–8.

[12] Mark A. Fleischer, *Cybernetic optimization by simulated annealing: Accelerating convergence by parallel processing and probabilistic feedback control*, Journal of Heuristics **1** (1996), no. 2, 225–246.

[13] Bruce Hajek, *Cooling schedules for optimal annealing*, Math. Oper. Res. **13** (1988), no. 2, 311–329.

[14] L. Ingber, *Very fast simulated re-annealing*, Mathl. Comput. Modelling **12** (1989), no. 8, 967–973.

[15] 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** (1989), no. 6, 865–892.

[16] _____, *Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning*, Oper. Res. **39** (1991), no. 3, 378–406.

[17] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), 671–680.

[18] P. J. M. Laarhoven and E. H. L. Aarts (eds.), *Simulated annealing: theory and applications*, Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[19] 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** (1953), no. 6, 10871092.

[20] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vincentelli, *Convergence and finite-time behavior of simulated annealing*, Advances in Applied Probability **18** (1986), no. 3, 747–771.

[21] Surendra Nahar, Sartaj Sahni, and Eugene Shragowitz, *Experiments with simulated annealing*, DAC '85: Proceedings of the 22nd ACM/IEEE conference on Design automation (New York, NY, USA), ACM Press, 1985, pp. 748–752.

[22] _____, *Simulated annealing and combinatorial optimization*, DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation (Piscataway, NJ, USA), IEEE Press, 1986, pp. 293–299.

[23] Ibrahim Hassan Osman, *Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem*, Ann. Oper. Res. **41** (1993), no. 1-4, 421–451.

[24] Martin Pincus, *A monte carlo method for the approximate solution of certain types of constrained optimization problems*, Operations Research **18** (1970), no. 6, 1225–1228.

[25] Patrick Siarry, Gérard Berthiau, François Durdin, and Jacques Haussy, *Enhanced simulated annealing for globally minimizing functions of many-continuous variables*, ACM Trans. Math. Softw. **23** (1997), no. 2, 209–228.

[26] P. N. Strenski and Scott Kirkpatrick, *Analysis of finite length annealing schedules*, Algorithmica **6** (1991), 346–366.

[27] S.R. White, *Concepts of scale in simulated annealing*, AIP Conference Proceedings, vol. 122, 1984, pp. 261–270.

Chapter 4

Reactive prohibitions

It is a good morning exercise for a research scientist to discard a pet hypothesis every day before breakfast. It keeps him young.
Konrad Lorenz

4.1 Prohibitions for diversification (Tabu Search)

The Tabu Search meta-heuristic [13] is based on the use of *prohibition-based* techniques and "intelligent" schemes as a complement to basic heuristic algorithms like local search, with the purpose of guiding the basic heuristic *beyond local optimality*. It is difficult to assign a precise date of birth to these principles. For example, ideas similar to those proposed in TS can be found in the *denial* strategy of [24] (once common features are detected in many suboptimal solutions, they are *forbidden*) or in the opposite *reduction* strategy of [19] (in an application to the Travelling Salesman Problem, all edges that are common to a set of local optima are fixed). In very different contexts, prohibition-like strategies can be found in *cutting planes* algorithms for solving integer problems through their Linear Programming relaxation (inequalities that cut off previously obtained fractional solutions are generated) and in branch and bound algorithms (subtrees are not considered if the leaves cannot correspond to better solutions), see the textbook [22].

The renaissance and full blossoming of "intelligent prohibition-based heuristics" starting from the late eighties is greatly due to the role of F. Glover in the proposal and diffusion of a rich variety of meta-heuristic tools [13, 14], but see also [17] for an independent seminal paper.

It is evident that Glover's ideas have been a source of inspiration for many approaches based on the intelligent use of memory in heuristics. Let's only cite the four "principles" about using long-term memory in tabu-search of *recency, frequency, quality and influence*, also cited in [6]. A growing number of TS-based algorithms has been developed in the last years and applied with success to a wide selection of problems [15]. It is therefore difficult, if not impossible, to characterize a "canonical form" of TS, and classifications tend to be short-lived. Nonetheless, at least two aspects characterize many versions of TS: the fact that TS is used to complement *local (neighborhood) search*, and the fact that the main modifications to local search are obtained through the *prohibition* of selected moves available at the current point. TS acts to continue the search beyond the first local minimizer without wasting the work already executed, as it is the case if a new run of local search is started from a new random initial point, and to enforce appropriate amounts of diversification to avoid that the search trajectory remains confined near a given local minimizer.

In our opinion, the main competitive advantage of TS with respect to alternative heuristics based on local search like Simulated Annealing (SA) [18] lies in the intelligent use of the past history of the search to influence its future steps.

Table 4.1: A classification based on discrete dynamical systems.

Discrete Dynamical System (search trajectory generation)	
Deterministic	Stochastic
Strict TS	Probabilistic TS
Fixed TS	Robust TS
	Fixed TS with stochastic tie breaking
Reactive TS	Reactive TS with stochastic tie breaking
	Reactive TS with neighborhood sampling
	(stochastic candidate list strategies)

Let us assume that the feasible search space is the set of binary strings with a given length L: $\mathcal{X} = \{0,1\}^L$). $X^{(t)}$ is the current configuration and $N(X^{(t)})$ the previously introduced neighborhood. In prohibition-based search (Tabu Search) some of the neighbors are *prohibited*, a subset $N_A(X^{(t)}) \subset N(X^{(t)})$ contains the *allowed* ones. The general way of generating the search trajectory that we consider is given by:

$$X^{(t+1)} = \text{BEST-NEIGHBOR}(\ N_A(X^{(t)})\ ) \quad (4.1)$$

$$N_A(X^{(t+1)}) = \text{ALLOW}(N(X^{(t+1)}), X^{(0)}, ..., X^{(t+1)}) \quad (4.2)$$

The set-valued function ALLOW selects a subset of $N(X^{(t+1)})$ in a manner that depends on the entire search trajectory $X^{(0)}, ..., X^{(t+1)}$.

4.1.1 Forms of Tabu Search

There are several vantage points from which to view heuristic algorithms. By analogy with the concept of *abstract data type* in Computer Science [1], and with the related *object-oriented* software engineering techniques [8], it is useful to separate the abstract concepts and operations of TS from the detailed implementation, i.e., realization with specific data structures. In other words, *policies* (that determine which trajectory is generated in the search space, what the balance of intensification and diversification is, etc.) should be separated from *mechanisms* that determine *how* a specific policy is realized. An essential abstract concept in TS is given by the *discrete dynamical system* of (4.1)–(4.2) obtained by modifying local search.

Before starting the discussion on the use of memory to generate prohibitions in the next section, let's note in passing that other softer possibilities exist. For example the HSAT [12] variation of GSAT 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 [12].

4.1.2 Dynamical systems

A classification of some TS-related algorithms that is based on the underlying dynamical system is illustrated in Fig 4.1.

A first subdivision is given by the *deterministic* versus *stochastic* nature of the system. Let us first consider the deterministic versions. Possibly the simplest form of TS is what is called *strict-TS*: a neighbor is prohibited if and only if it has already been visited during the previous part of the search [13] (the term "strict" is chosen to underline the rigid enforcement of its simple prohibition rule). Therefore (4.2) becomes:

$$N_A(X^{(t+1)}) = \{X \in N(X^{(t+1)}) \quad \text{s. t.} \quad X \notin \{X^{(0)}, ..., X^{(t+1)}\}\} \quad (4.3)$$

Let us note that strict-TS is parameter-free.

Two additional algorithms can be obtained by introducing a *prohibition parameter* T that determines how long a move will remain prohibited after the execution of its inverse. The *fixed-TS* algorithm is obtained by fixing T throughout the search [13]. A neighbor is allowed if and only if it is obtained from the current point by applying a move such that its inverse has not been used during the last T iterations. In detail, if LASTUSED(μ) is the last usage time of move μ (LASTUSED(μ) $= -\infty$ at the beginning):

$$N_A(X^{(t)}) = \{X = \mu \circ X^{(t)} \text{ s. t. } \textsc{LastUsed}(\mu^{-1}) < (t - T)\} \tag{4.4}$$

If T changes with the iteration counter depending on the search status (in this case the notation will be $T^{(t)}$), the general dynamical system that generates the search trajectory comprises an additional evolution equation for $T^{(t)}$, so that the three defining equations are now:

$$T^{(t)} = \textsc{React}(T^{(t-1)}, X^{(0)}, ..., X^{(t)}) \tag{4.5}$$

$$N_A(X^{(t)}) = \{X = \mu \circ X^{(t)} \text{ s. t. } \textsc{LastUsed}(\mu^{-1}) < (t - T^{(t)}) \tag{4.6}$$

$$X^{(t+1)} = \textsc{Best-Neighbor}(N_A(X^{(t)}))\} \tag{4.7}$$

Let us note that $\mu = \mu^{-1}$ for the basic moves acting on binary strings. Now, possible rules to determine the prohibition parameter by reacting to the repetition of previously-visited configurations have been proposed in [4] (*reactive-TS*, *RTS* for short). In addition, there are situations where the single reactive mechanism on T is not sufficient to avoid long cycles in the search trajectory and therefore a second reaction is needed [4]. The main principles of RTS are briefly reviewed in Sec. 4.2.

Stochastic algorithms related to the previously described deterministic versions can be obtained in many ways. For example, prohibition rules can be substituted with *probabilistic generation-acceptance rules* with large probability for allowed moves, small for prohibited ones, see for example the *probabilistic-TS* [13]. Stochasticity can increase the robustness of the different algorithms, in addition [13] "randomization is a means for achieving diversity without reliance on memory," although it could "entail a loss in efficiency by allowing duplications and potentially unproductive wandering that a more systematic approach would seek to eliminate." Incidentally, asymptotic results for TS can be obtained in probabilistic TS [11]. In a different proposal (*robust-TS*) the prohibition parameter is randomly changed between an upper and a lower bound during the search [25]. Stochasticity in fixed-TS and in reactive-TS can be added through a *random breaking of ties*, in the event that the same cost function decrease is obtained by more than one winner in the BEST-NEIGHBOR computation. At least this simple form of stochasticity should always be used to avoid external search biases, possibly caused by the ordering of the loop indices.

If the neighborhood evaluation is expensive, the exhaustive evaluation can be substituted with a partial *stochastic sampling*: only a partial list of candidates is examined before choosing the best allowed neighbor.

4.1.3 A worked out example of Fixed Tabu Search

Let us assume that the search space $\mathcal{X}$ is the set of 3-bit strings ($\mathcal{X} = [b_1, b_2, b_3]$) and the cost function is:

$$f([b_1, b_2, b_3]) = b_1 + 2\ b_2 + 3\ b_3 - 7\ b_1\ b_2\ b_3$$

The feasible points (the edges of the 3-dimensional binary cube) are illustrated in Fig. 4.1 with the associated cost function. The neighborhood of a point is the set of points that are connected with edges.

The point $X^{(0)} = [0, 0, 0]$ with $f(X^{(0)}) = 0$ is a local minimizer because all moves produce a higher cost value. The best of the three admissible moves is μ_1, so that $X^{(1)} = [1, 0, 0]$. Note that the move is applied even if $f(X^{(1)}) = 1 \geq f(X^{(0)})$, so that the system abandons the local minimizer.

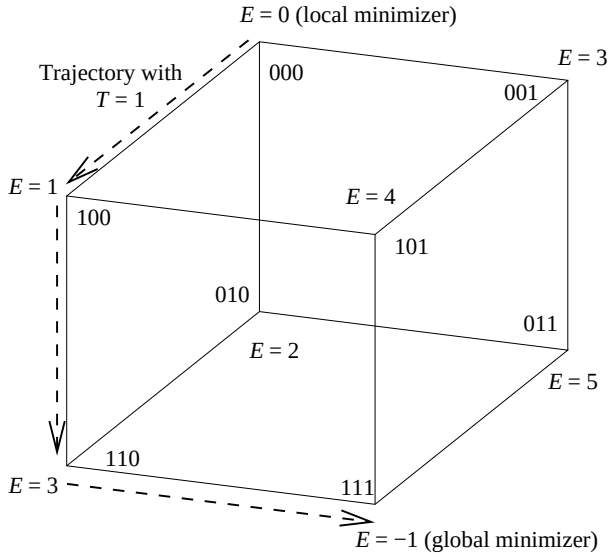


Figure 4.1: Search space, f values and search trajectory.

If $T^{(1)} = 0$, the best move from $X^{(1)}$ will again be μ_1 and the system will return to its starting point: $X^{(1)} = X^{(0)}$. If $T^{(t)}$ remains equal to zero the system is trapped forever in the limit cycle $[0,0,0] \rightarrow [1,0,0] \rightarrow [0,0,0] \rightarrow [1,0,0]$.....

On the contrary, if $T^{(t)} = 1$, μ_1 is prohibited at $t = 1$ because it was used too recently, i.e. its most recent usage time $\Lambda(\mu_1)$ satisfies $\Lambda(\mu_1) = 0 \geq (t - T^{(t)}) = 0$. The neighborhood is therefore limited to the points that can be reached by applying μ_2 or μ_3 ($N([1,0,0]) = \{[1,1,0],[1,0,1]\}$). The best *admissible* move is μ_2, so that $X^{(2)} = [1,1,0]$ with $f(X^{(2)}) = 3$.

At $t = 2$ μ_2 is prohibited, μ_1 is admissible again because $\Lambda(\mu_1) = 0 < (t - T^{(t)}) = 1$, and μ_3 is admissible because it was never used. The best move is μ_3 and the system reaches the global minimizer: $X^{(3)} = [1,1,1]$ with $f(X^{(3)}) = -1$.

4.1.4 Relation between prohibition and diversification

The prohibition parameter T used in eq. 4.4 is related to the amount of *diversification*: the larger T, the longer the distance that the search trajectory must go before it is allowed to come back to a previously visited point. In particular, the following propositions can be demonstrated:

- The Hamming distance H between a starting point and successive point 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 relations are immediately demonstrated if one considers that, as soon as the value of a variable is flipped, this value remains unchanged for the next T iterations. In order to come back to the starting configuration, all $T + 1$ variables changed in the first phase must be changed back to their original values.

But large values of T imply that only a limited subset of the possible moves can be applied to the current configuration. In particular, T must be less than or equal to $n - 2$ to assure

that at least two moves can be applied (clearly, if only one move can be applied the choice is not influenced by the f values in the neighborhood). It is therefore appropriate to set T to the smallest value that guarantees diversification.

4.1.5 How to escape from an attractor

Local minima points are *attractors* of the search trajectory generated by deterministic local search. If the cost function is integer-valued and lower bounded it can be easily shown that a trajectory starting from an arbitrary point will terminate at a local minimizer. All points such that a deterministic local search trajectory starting from them terminates at a specific local minimizer make up its *attraction basin*. Now, as soon as a local minimizer is encountered, its entire attraction basin is not of interest for the optimization procedure, in fact its points do not have smaller cost values. It is nonetheless true that better points could be close to the current basin, whose boundaries are not known. One of the problems that must be solved in heuristic techniques based on local search is how to continue the search beyond the local minimizer and how to *avoid the confinement* of the search trajectory. Confinements can happen because the trajectory tends to be biased toward points with low cost function values, and therefore also toward the just abandoned local minimizer. The fact that the search trajectory remains close to the minimizer for some iterations is clearly a desired effect in the hypothesis that better points are preferentially located in the neighborhood of good suboptimal point rather than among randomly extracted points.

Simple confinements can be *cycles* (endless repetition of a sequence of configurations during the search) or more complex trajectories with no clear periodicity but nonetheless such that only a limited portion of the search space is visited (they are analogous to *chaotic attractors* in dynamical systems).

An heuristic prescription is that the search point is kept close to a discovered local minimizer at the beginning, snooping about better attraction basins. If these are not discovered, the search should gradually progress to larger distances (therefore progressively enforcing longer-term diversification strategies).

Some very different ways of realizing this general prescription are here illustrated for a "laboratory" test problem defined as follows. The search space is the set of all binary strings of length L. Let us assume that the search has just reached a (strict) local minimizer and that the cost f in the neighborhood is strictly increasing as a function of the number of different bits with respect to the given local minimizer (i.e., as a function of the Hamming distance). Without loss of generality, let us assume that the local minimizer is the zero string ($[00...0]$) and that the cost is precisely the Hamming distance. Although artificial, the assumption is not unrealistic in many cases. An analogy in continuous space is the usual positive-definite quadratic approximation of the cost in the neighborhood of a strict local minimizer of a differentiable function. In the following parts the discussion is mostly limited to deterministic versions.

Strict-TS

In the deterministic version of strict-TS, if more than one basic move produce the same cost decrease at a given iteration, the move that acts on the right-most (least significant) bit of the string is selected.

The set of obtained configuration for $L = 4$ is illustrated in Fig 4.2. Let us now consider how the Hamming distance evolves in time, in the optimistic assumption that the search always finds an allowed move until all points of the search space are visited. If $H(t)$ is the Hamming distance at iteration t, the following holds true:

$$H(t) \leq \lfloor \log_2(t) \rfloor + 1 \tag{4.8}$$

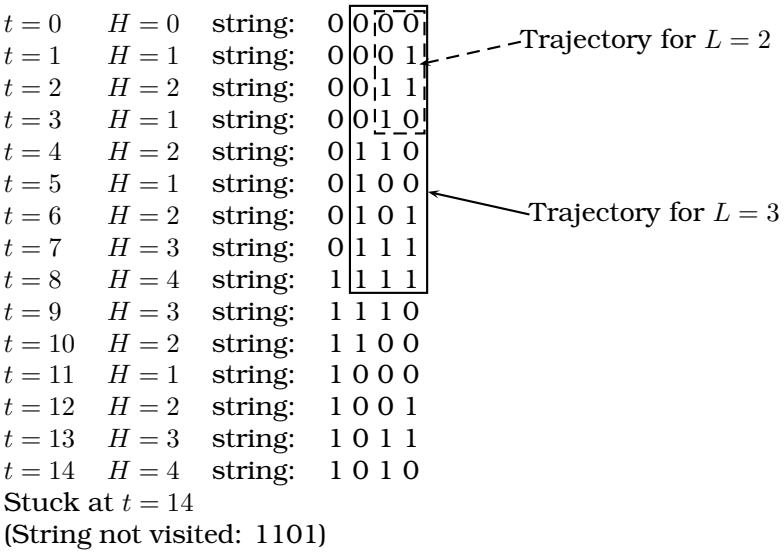


Figure 4.2: Search trajectory for deterministic strict-TS: iteration t, Hamming distance H and binary string.

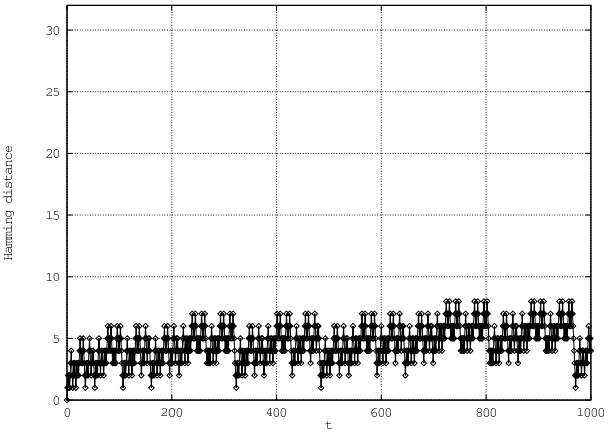


Figure 4.3: Evolution of the Hamming distance for deterministic strict-TS ($L = 32$).

This can be demonstrated after observing that a complete trajectory for an $(L-1)$-bit search space becomes a legal *initial part* of the trajectory for L-bit strings after appending zero as the most significant bit (see the trajectory for $L = 3$ in Fig 4.2). Now, a certain Hamming distance H can be reached only as soon as or after the H-th bit is set (e.g, H=4 can be reached only at or after $t = 8$ because the fourth bit is set at this iteration). Equation (4.8) trivially follows.

In practice the above optimistic assumption is not true: strict-TS can be stuck (trapped) at a configuration such that all neighbors have already been visited. In fact, the smallest L such that this event happens is $L = 4$ and the search is stuck at $t = 14$, so that the string $[1101]$ is not visited. The problem worsens for higher-dimensional strings. For $L = 10$ the search is stuck after visiting 47 % of the entire search space, for $L = 20$ it is stuck after visiting only 11 % of the search space.

If the trajectory must reach Hamming distance H with respect to the local minimum point before escaping (i.e., before encountering a better attraction basin) the necessary number of iterations is at least exponential in H. Fig. 4.3 shows the actual evolution of the Hamming distance for the case of $L = 32$. The detailed dynamics is complex, as "iron curtains" of visited points (that cannot be visited again) are created in the configuration space and the trajectory

must obey the corresponding constraints. The slow growth of the Hamming distance is related to the "basin filling" effect [4] of strict-TS: all points at smaller distances tend to be visited before points at larger distances (unless the iron curtains prohibit the immediate visit of some points). On the other hand, let us note that the exploration is not as "intensifying" as an intuitive picture of strict-TS could lead one to believe: some configurations at small Hamming distance are visited only in the last part of the search. As an example let us note that the point at $H = 4$ is visited at $t = 8$ while one point at $H = 1$ is visited at $t = 11$ (t is the iteration). This is caused by the fact that, as soon as a new bit is set for the first time, all bits to the right are progressively cleared (because new configurations with lower cost are obtained). In particular, the second configuration at $H = 1$ is encountered at $t > 2$, the third at $t > 4$,... the n-th at $t > 2^{(n-1)}$. Therefore, at least a configuration at $H = 1$ will be encountered only after $2^{(L-1)}$ have been visited.

Let us note that the relation of (4.8) is valid only in the assumption that strict-TS is deterministic and that it is not stuck for any configuration. Let us now assume that one manages to obtain a more "intensifying" version of strict-TS, i.e., that all configurations at Hamming distance less than or equal to H are visited before configurations at distance greater than H. The initial growth of the Hamming distance is in this case much slower. In fact, the number of configurations C_H to be visited is:

$$C_H = \sum_{i=0}^{H} \left(\begin{array}{c} L \\ i \end{array} \right) \tag{4.9}$$

It can be easily derived that $C_H >> 2^H$, if $H << L$. As an example[1], for $L = 32$ one obtains from (4.9) a value $C_5 = 242\ 825$, and therefore this number of configurations have to be visited before finding a configuration at Hamming distance greater than 5, while $2^5 = 32$. An explosion in the number of iterations spent near a local optimum occurs unless the nearest attraction basin is very close. The situation worsens in higher-dimensional search spaces: for $L = 64$, $C_5 = 8\ 303\ 633$, $C_4 = 679\ 121$. This effect can be seen as a manifestation of the "curse of dimensionality:" a technique that works in very low-dimensional search space can encounter dramatic performance reductions as the dimension increases. In particular, there is the danger that the entire search span will be spent while visiting points at small Hamming distances, unless additional diversifying tools are introduced.

Fixed-TS

The analysis of fixed-TS is simple: as soon as a bit is changed it will remain prohibited ("frozen") for additional T steps. Therefore (see Fig. 4.4), the Hamming distance with respect to the starting configuration will cycle in a regular manner between zero and a maximal value $H = T + 1$ (only at this iteration the ice around the first frozen bit melts down and allows changes that are immediately executed because H decreases). All configurations in a cycle are different (apart from the initial configuration). The cycling T behavior is the same for both the deterministic and the stochastic version, the property of the stochastic version is that different configurations have the possibility of being visited in different cycles. In fact all configurations at a given Hamming distance H have the same probability of being visited if $H \leq T + 1$, zero probability otherwise.

The effectiveness of fixed-TS in escaping from the local attractor depends on the size of the T value with respect to the minimal distance such that a new attraction basin is encountered. In particular, if T is too small the trajectory will never escape, but if T is too big an "over-constrained" trajectory will be generated.

[1] Computations have been executed by Mathematica©.

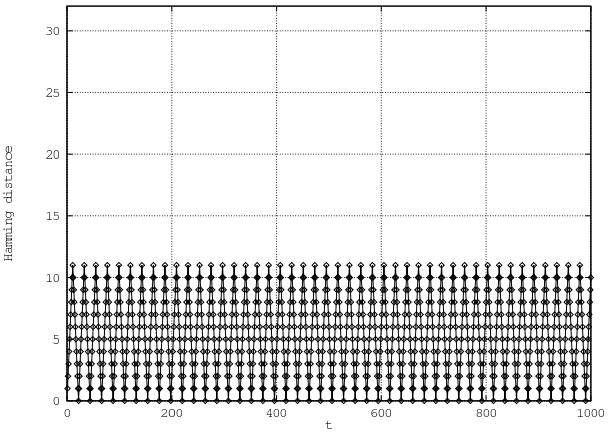


Figure 4.4: Evolution of the Hamming distance for both deterministic and stochastic fixed-TS ($L = 32$, $T = 10$).

Reactive-TS

The behavior of reactive-TS depends on the specific reactive scheme used. Given the previously illustrated relation between the prohibition T and the diversification, a possible prescription is that of gradually increasing T if there is evidence that the system is confined near a local attractor, until a new attractor is encountered. In particular, the evidence for a confinement can be obtained from the repetition of a previously visited configuration, while the fact that a new attraction basin has been found can be postulated if repetitions disappear for a suitably long period. In this last case, T is gradually decreased. This general procedure was used in the design of the RTS algorithm in [4], where specific rules are given for the entire feedback process.

In the present discussion we consider only the initial phase of the escape from the attractor, when increases of T are dominant over decreases. In fact, to simplify the discussion, let us assume that $T = 1$ at the beginning and that the reaction acts to increase T when a local minimum point is repeated, in the following manner:

$$\textsc{React}(T) \quad = \quad \min\{\max\{T \times 1.1, T + 1\}, L - 2\} \tag{4.10}$$

The initial value (and lower bound) of one implies that the system does not come back immediately to a just left configuration. The upper bound is used to guarantee that at least two moves are allowed at each iteration. Non-integral values of T are cast to integers before using them (the largest integer less than or equal to T).

The evolution of T for the deterministic version is shown in Fig. 4.5, repetitions of the local minimum point cause a rapid increase up to its maximal value. As soon as the value is reached the system enters a cycle. This limit cycle is caused by the fact that no additional attraction basins exist in the test case considered, while in real-world *fitness surfaces* the prohibition T tends to be small with respect to its upper bound, both because of the limited size of the attraction basins and because of the complementary reaction that decreases T if repetitions disappear.

The behavior of the Hamming distance is illustrated in Fig. 4.6. The maximal Hamming distance reached increases in a much faster way compared to the strict-TS case.

Now, for a given $T^{(t)}$ the maximum Hamming distance that is reached during a cycle is $H_{max} = T^{(t)} + 1$ and the cycle length is $2(T^{(t)} + 1)$. After the cycle is completed the local minimizer is repeated and the reaction occurs. The result is that $T^{(t)}$ increases monotonically, and therefore the cycle length does also, as illustrated in Fig. 4.7 that expands the initial part of the graph.

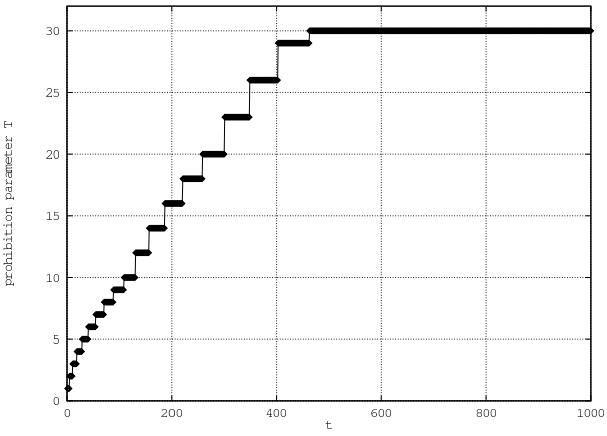


Figure 4.5: Evolution of the prohibition parameter T for deterministic reactive-TS with reaction at local minimizers ($L = 32$).

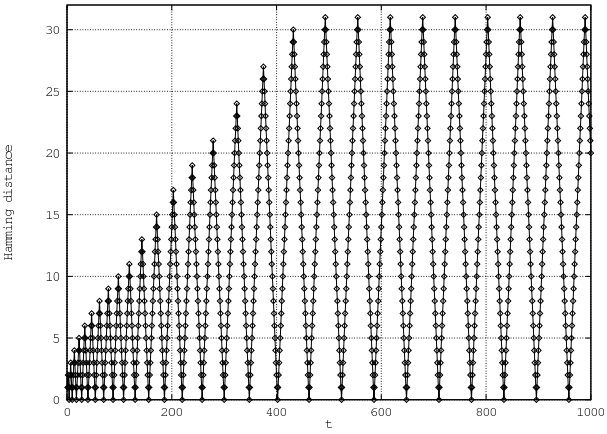


Figure 4.6: Evolution of the Hamming distance for simplified reactive-TS ($L = 32$).

Let us now consider a generic iteration t at which a reaction occurs (like $t = 4, 10, 18, ...$ in Fig. 4.7). At the beginning (4.10) will increase T by one unit at each step. If the prohibition is T just before the reaction, the total number t of iterations executed is:

$$t(T) = \sum_{i=1}^{T} 2(i+1) = 3T + T^2 \tag{4.11}$$

$$t(H_{max}) = \left(H_{max}^2 + H_{max} - 2\right) \tag{4.12}$$

$$H_{max}(t) = \frac{1}{2}\left(\sqrt{9+4t} - 1\right) \tag{4.13}$$

where the relation $T = H_{max} - 1$ has been used. Therefore the increase of the maximum reachable Hamming distance is approximately $O(\sqrt{t})$ during the initial steps. The increase is clearly faster in later steps, when the reaction is multiplicative instead of additive (when $\lfloor T \times 1.1 \rfloor > T + 1$ in (4.10)), and therefore the above estimate of H_{max} becomes a lower bound in the following phase.

Let us note that the difference with respect to strict-TS is a crucial one: one obtains an (optimistic) logarithmic increase in the strict algorithm, and a (pessimistic) increase that behaves like the square root of the number of iterations in the reactive case. In this last case bold tours at increasing distances are executed until the prohibition T is sufficient to escape

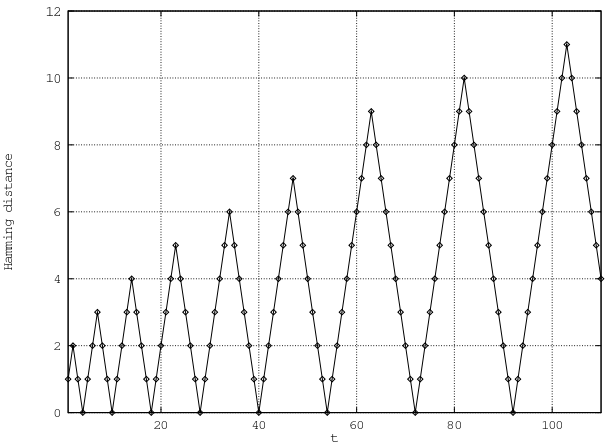


Figure 4.7: Evolution of the Hamming distance for reactive-TS, first 100 iterations ($L = 32$).

from the attractor. In addition, if the properties of the fitness surface change slowly in the different regions, and RTS reaches a given local minimizer with a T value obtained during its previous history, the chances are large that it will escape even faster.

4.2 Reactive Tabu Search (RTS)

Some problems arising in TS that are worth investigating are:

1. the determination of an appropriate prohibition T for the different tasks,
2. the robustness of the technique for a wide range of different problems,
3. the adoption of minimal computational complexity algorithms for using the search history.

The three issues are briefly discussed in the following sections, together with the RTS methods proposed to deal with them.

4.2.1 Self-adjusted prohibition period

In RTS the prohibition T is determined through feedback (i.e., *reactive*) mechanisms during the search. T is equal to one at the beginning (the inverse of a given move is prohibited only at the next step), it increases only when there is *evidence* that diversification is needed, it decreases when this evidence disappears. In detail: the evidence that diversification is needed is signaled by the repetition of previously visited configurations. All configurations found during the search are stored in memory. After a move is executed the algorithm checks whether the current configuration has already been found and it reacts accordingly (T increases if a configuration is repeated, T decreases if no repetitions occurred during a sufficiently long period).

Let us note that T is not fixed during the search, but is determined in a dynamic way depending on the *local structure* of the search space. This is particularly relevant for "inhomogeneous" tasks, where the statistical properties of the search space vary widely in the different regions (in these cases a fixed T would be inappropriate).

An example of the behavior of T during the search is illustrated in Fig. 4.8, for a Quadratic Assignment Problem task [4]. T increases in an exponential way when repetitions are encountered, it decreases in a gradual manner when repetitions disappear.

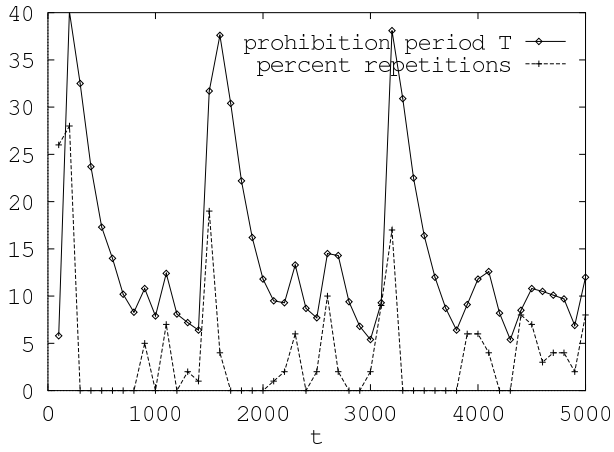


Figure 4.8: Dynamics of the the prohibition period T on a QAP task.

4.2.2 The escape mechanism

The basic tabu mechanism based on prohibitions is not sufficient to avoid long cycles (e.g., for binary strings of length L, T must be less than the length of the string, otherwise all moves are eventually prohibited, and therefore cycles longer than $2 \times L$ are still possible). In addition, even if "limit cycles" (endless cyclic repetitions of a given set of configurations) are avoided, the first reactive mechanism is not sufficient to guarantee that the search trajectory is not confined in a limited region of the search space. A "chaotic trapping" of the trajectory in a limited portion of the search space is still possible (the analogy is with *chaotic attractors* of dynamical systems, where the trajectory is confined in a limited portion of the space, although a limit cycle is not present).

For both reasons, to increase the robustness of the algorithm a second more radical diversification step (*escape*) is needed. The escape phase is triggered when too many configurations are repeated too often [4]. A simple escape action consists of a number of random steps executed starting from the current configuration (possibly with a bias toward steps that bring the trajectory away from the current search region).

With a stochastic escape, one can easily obtain the *asymptotic convergence* of RTS (in a finite-cardinality search space, escape is activated infinitely often: if the probability for a point to be reached after escaping is different from zero for all points, eventually all points will be visited - clearly including the globally optimal points). The detailed investigation of the asymptotic properties and finite-time effects of different escape routines to enforce long-term diversification is an open research area.

4.3 Implementation: storing and using the search history

While the above classification deals with dynamical systems, a different classification is based on the detailed data structures used in the algorithms and on the consequent realization of the needed operations. Different data structures can possess widely different computational complexities so that attention should be spent on this subject before choosing a version of TS that is efficient on a particular problem.

Some examples of different implementations of the same TS dynamics are illustrated in Fig. 4.9. Strict-TS can be implemented through the reverse elimination method (REM) [14, 9], a term that refers to a technique for the storage and analysis of the ordered list of all moves performed throughout the search (called "running list"). The same dynamics can be obtained in all cases through standard *hashing* methods and storage of the configurations [27, 4], or, for the case of a search space consisting of binary strings, through the *radix tree* (or "digital

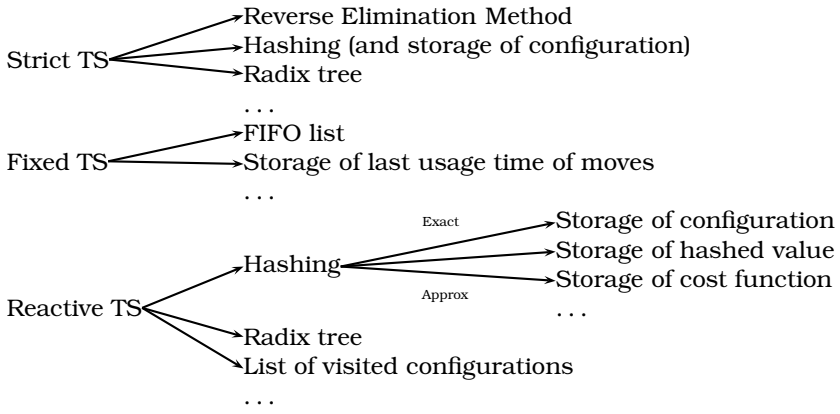


Figure 4.9: The same search trajectory can be obtained with different data structures.

tree") technique [4]. Hashing is an old tool in Computer Science: different hashing functions – possibly with incremental calculation – are available for different domains [7]. REM is not applicable to all problems (the "sufficiency property" must be satisfied [14]), in addition its computational complexity per iteration is proportional to the number of iterations executed, while the average complexity obtained through incremental hashing is $O(1)$, a small and constant number of operations. The worst-case complexity per iteration obtained with the radix tree technique is proportional to the number of bits in the binary strings, and constant with respect to the iteration. If the memory usage is considered, both REM and approximated hashing use $O(1)$ memory per iteration, while the actual number of bytes stored can be less for REM, because only changes (moves) and not configurations are stored.

Trivially, fixed-TS (alternative terms [13, 14] are: simple TS, static tabu search – STS – or tabu navigation method) can be realized with a first-in first-out list where the prohibited moves are located (the "tabu list"), or by storing in an array the last usage time of each move and by using (4.4).

Reactive-TS can be implemented through a simple list of visited configurations, or with more efficient hashing or radix tree techniques. At a finer level of detail, hashing can be realized in different ways. If the entire configuration is stored (see also Fig. 4.10) an exact answer is obtained from the memory lookup operation (a repetition is reported if and only if the configuration has been visited before). On the contrary, if a "compressed" item is stored, like a hashed value of a limited length derived from the configuration, the answer will have a limited probability of error (a repetition can be reported even if the configuration is new, because the compressed items are equal by chance – an event called "collision"). Experimentally, small collision probabilities do not have statistically significant effects on the use of reactive-TS as heuristic tool, and hashing versions that need only a few bytes per iteration can be used. The effect of collision probabilities when hashing is used in other schemes is a subject worth investigating.

4.3.1 Fast algorithms 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. 4.10. 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

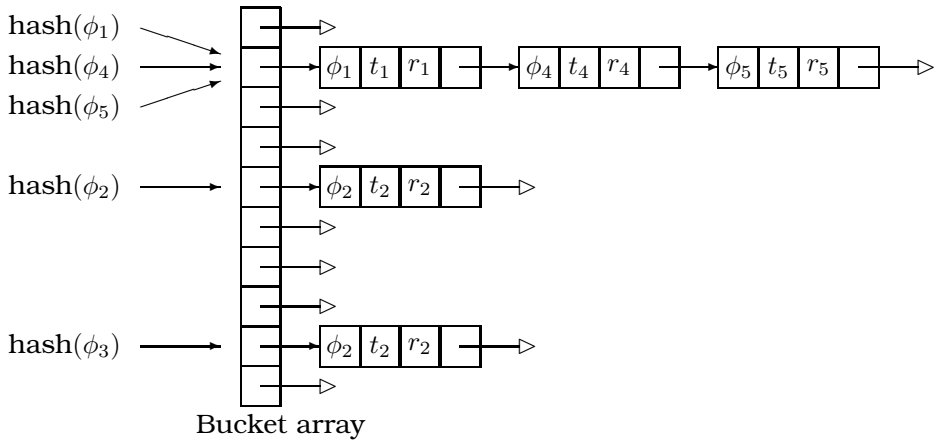


Figure 4.10: Open hashing scheme: items (configuration, or compressed hashed value, etc.) are stored in "buckets". The index of the bucket array is calculated from the configuration.

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 the average.

4.3.2 Persistent dynamic sets

Persistent dynamic sets are proposed to support memory–usage operations in history-sensitive heuristics in [3, 2].

Ordinary data structures are *ephemeral* [10], 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 [10] that is dedicated to a systematic study of persistence, continuing the previous work of [21].

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). 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 [5, 16] 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. Rebalancing the tree after insertion or deletion can be done in $O(1)$ rotations and $O(\log L)$ color changes [26]. In addition, the amortized number of color changes per update is $O(1)$, see for example [20].

Now, the REM method [13, 14] is closely reminiscent of a method studied in [21] 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 [23, 10]. Let us now summarize from [23] how a partially persistent red-black tree can be realized. An example of the realizations that we consider is presented in Fig. 4.11.

The trivial way is that of keeping in memory all copies of the ephemeral tree (see the top part of Fig. 4.11), each copy requiring $O(L)$ space. A smarter realization is based on *path copying*, independently proposed by many researchers, see [23] 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 [23]. 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 iff 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. 4.11 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 of $O(\log L)$, but the important result derived in [23] 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. 4.12. 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.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data structures and algorithms*, Addison-Wesley, 1983.

[2] R. Battiti, *Partially persistent dynamic sets for history-sensitive heuristics*, Tech. Report UTM-96-478, Dip. di Matematica, Univ. di Trento, 1996, Revised version, Presented at the Fifth DIMACS Challenge, Rutgers, NJ, 1996.

[3] ———, *Time- and space-efficient data structures for history-based heuristics*, Tech. Report UTM-96-478, Dip. di Matematica, Univ. di Trento, 1996.

[4] R. Battiti and G. Tecchiolli, *The reactive tabu search*, ORSA Journal on Computing **6** (1994), no. 2, 126–140.

[5] R. Bayer, *Symmetric binary b-trees: Data structure and maintenance algorithms*, Acta Informatica **1** (1972), 290–306.

[6] Christian Blum and Andrea Roli, *Metaheuristics in combinatorial optimization: Overview and conceptual comparison*, ACM Comput. Surv. **35** (2003), no. 3, 268–308.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, McGraw-Hill, New York, 1990.

[8] B. J. Cox, *Object oriented programming, an evolutionary approach*, Addison-Wesley, 1990.

[9] F. Dammeyer and S. Voss, *Dynamic tabu list management using the reverse elimination method*, Annals of Operations Research **41** (1993), 31–46.

[10] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, *Making data structures persistent*, Proceedings of the 18th Annual ACM Symposium on Theory of Computing (Berkeley, CA), ACM, May 28-30 1986.

[11] U. Faigle and W. Kern, *Some convergence results for probabilistic tabu search*, ORSA Journal on Computing **4** (1992), no. 1, 32–37.

[12] I.P. Gent and T. Walsh, *Towards an understanding of hill-climbing procedures for sat*, Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI Press / The MIT Press, 1993, pp. 28–33.

[13] F. Glover, *Tabu search - part i*, ORSA Journal on Computing **1** (1989), no. 3, 190–260.

[14] ———, *Tabu search - part ii*, ORSA Journal on Computing **2** (1990), no. 1, 4–32.

[15] ———, *Tabu search: Improved solution alternatives*, Mathematical Programming, State of the Art (J. R. Birge and K. G. Murty, eds.), The Univ. of Michigan, 1994, pp. 64–92.

[16] L. J. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, Proc. of the 19th Ann. Symp. on Foundations of Computer Science, IEEE Computer Society, 1978, pp. 8–21.

[17] P. Hansen and B. Jaumard, *Algorithms for the maximum satisfiability problem*, Computing **44** (1990), 279–303.

[18] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), 671–680.

[19] S. Lin, *Computer solutions of the travelling salesman problems*, Bell Systems Technical J. **44** (1965), no. 10, 2245–69.

[20] D. Maier and S. C. Salveter, *Hysterical b-trees*, Information Processing Letters **12** (1981), no. 4, 199–202.

[21] M. H. Overmars, *Searching in the past ii: general transforms*, Tech. report, Dept. of Computer Science, Univ. of Utrecht, Utrecht, The Netherlands, 1981.

[22] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization, algorithms and complexity*, Prentice-Hall, NJ, 1982.

[23] N. Sarnak and R. E. Tarjan, *Planar point location using persistent search trees*, Communications of the ACM **29** (1986), no. 7, 669–679.

[24] K. Steiglitz and P. Weiner, *Algorithms for computer solution of the traveling salesman problem*, Proceedings of the Sixth Allerton Conf. on Circuit and System Theory, Urbana, Illinois, IEEE, 1968, pp. 814–821.

[25] E. Taillard, *Robust taboo search for the quadratic assignment problem*, Parallel Computing **17** (1991), 443–455.

[26] R. E. Tarjan, *Updating a balanced search tree in o(1) rotations*, Information Processing Letters **16** (1983), 253–257.

[27] D. L. Woodruff and E. Zemel, *Hashing vectors for tabu search*, Annals of Operations Research **41** (1993), 123–138.

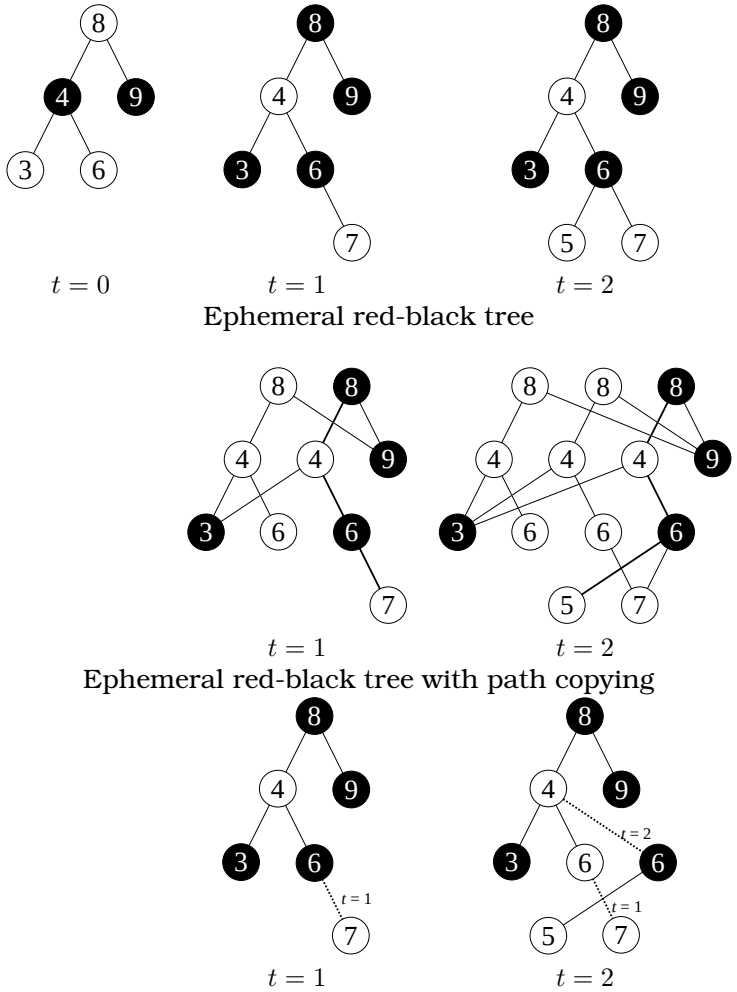


Figure 4.11: How to obtain a partially persistent red-black tree from an ephemeral one (top), containing indices 3,4,6,8,9 at t=0, with subsequent insertion of 7 and 5. Path copying (middle), with thick lines marking the copied part. Limited node copying (bottom) with dashed lines denoting the "extra" pointers with time stamp.

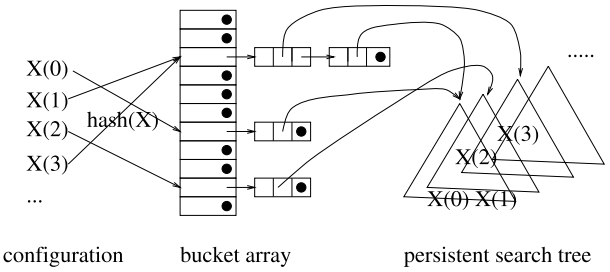


Figure 4.12: Open hashing scheme with persistent sets: a pointer to the appropriate root for configuration $X^{(t)}$ in the persistent search tree is stored in a linked list at a "bucket". Items on the list contain satellite data. The index of the bucket array is calculated from the configuration through a hashing function.

Chapter 5

Model-based search

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.
(Johann Von Neumann)

In the previous chapters we concentrated on how to solve optimization problems by applying different flavors of the local search paradigm. Nonetheless, the same problems can be thought of from a more global point of view, leading to a different family of useful optimization techniques.

5.1 Models of a problem

The main idea of model-based optimization is to create and maintain a *model* of the problem, whose aim is to provide some clues about the problem's solutions. If the problem is a function to be minimized, for instance, it is helpful to think of such model as a *simplified version* of the function itself; in more general settings, the model can be a probability distribution defining the estimated likelihood of finding a good quality solution at a certain point.

To solve a problem, we resort to the model in order to generate a candidate solution, then check it. The result of the check shall be used to refine the model, so that the future generation is biased towards better and better candidate solutions. Clearly, for a model to be useful it must provide as much information about the problem as possible, while being somehow "more tractable" (in a computational or analytical sense) than the problem itself. The initial model can be created through *a priori* knowledge or by uniformity assumptions.

Although memory-based technique can be used in both discrete and continuous domains, the latter case better supports our intuition. In Fig. 5.1 a function (continuous line) must be minimized. An initial model (the dashed line) provides a prior probability distribution for the minimum (in case of no prior knowledge, a uniform distribution can be assumed). Based on this estimate, some candidate minima are generated (points a through d), and the corresponding function values are computed. The model is updated (dotted line) to take into account the latest findings: the global minimum is more likely to occur around c and d, rather than a and b. Further model-guided generations and tests shall improve the distribution: eventually the region around the global minimum e shall be discovered and a high probability density shall be assigned to its surroundings. The same example also highlights a possible drawback of naïf applications of the technique: assigning a high probability to the neighborhood of c and d could lead to a negligible probability of selecting a point near e, so the global minimum would never be discovered. It looks like the emphasis is on *intensification* of the search. This is why, in practice, the models are corrected to ensure a significant probability of generating points also in unexplored regions.

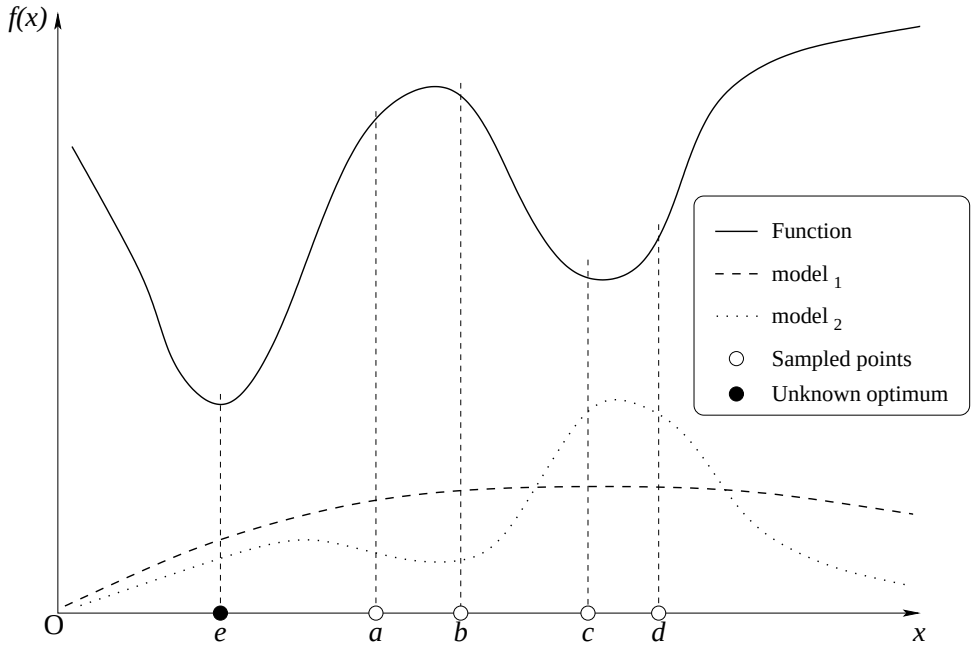


Figure 5.1: Model-based search: one generates sample points from model_1 and updates the generative model to increase the probability for point with low cost values (see model_2). In pathological cases, optimal point e runs the risk of becoming more and more difficult to generate.

An alternative possibility would be to avoid the model altogether and to keep in memory all or a fraction of the previously tested points (this hypothesis reminds of search algorithms based on a dynamic population of sample points, also known as Genetic or Evolutionary Algorithms). The informed reader will also notice a similarity between the machine learning subdivision between *instance-based* and *model-based* learning techniques, see for example [11].

Now that we are supported by intuition, let's proceed with the theoretical aspects. The discussion is inspired by the recent survey in [15]. The scheme of a model-based search approach is presented in Fig. 5.2. Represented entities are:

- a model used to generate sample solutions,
- the last samples generated,
- a memory containing previously accumulated knowledge about the problem (previous solutions and evaluations).

The process develops in an iterative way through a feedback loop were new candidates are generated by the model, and their evaluation —together with memory about past states— is used to improve the model itself in view of a new generation.

The design choices consist of defining a suitable generative model, and an appropriate learning rule to favor generation of superior models in the future steps. A lot of computational complexity can lurk within the second issue, which is in itself an optimization task. In particular one should avoid learning rules converging to local optima, as well as overtraining, which would hamper generalization.

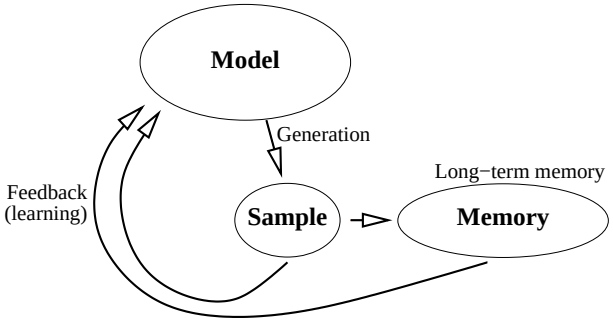


Figure 5.2: Model-based architecture: a generative model is updated after learning from the last generated samples and the previous long-term memory.

5.2 An example

Let's start from a simple model. The search space $\mathcal{X} = \{0,1\}^n$ is the set of all binary strings of length n, the generation model is defined by an n-tuple of parameters

$$\boldsymbol{p} = (p_1, \ldots, p_n) \in [0,1]^n,$$

where p_i is the probability of producing 1 as the i-th bit of the string and every bit is independently generated. The motivation is to "remove genetics from the standard genetic algorithm" [1]: instead of maintaining implicitly a statistic in a GA population, *statistics are maintained explicitly* in the vector (p_i).

The initial state of the model corresponds to indifference with respect to the bit values: $p_i = 0.5$, $i = 1, \ldots, n$. In the Population-Based Incremental Learning (PBIL) algorithm [1] the following steps are iterated:

1. Initialize $\boldsymbol{p}$;
2. **repeat**:
3. Generate a sample set S using the vector $\boldsymbol{p}$;
4. Extract a fixed number $\bar{S}$ of the best solutions from S;
5. **for each** sample $\boldsymbol{s} = (s_1, \ldots, s_n) \in \bar{S}$:
6. $\boldsymbol{p} \leftarrow (1-\rho)\boldsymbol{p} + \rho\boldsymbol{s}$,

where ρ is a learning rate parameter (regulating exploration versus exploitation). The moving vector $\boldsymbol{p}$ can be seen as representing a moving average of the best samples, a prototype vector placed in the middle of the cluster providing the best quality solutions. As a parallel with machine learning literature, the update rule is similar to that used in Learning Vector Quantization, see [7]. Variations include moving away from bad samples in addition to moving towards good ones. A schematic representation is shown in Fig. 5.3.

Although this simple technique shows results superior to GA on some benchmark tasks [1], the method has intrinsic weaknesses if the optimization landscape has a rich structure, for example more than a single cluster, or complex configurations of the optimal positions corresponding to *dependencies among the individual bits*.

5.3 Dependent probabilities

Estimates of probability densities for optimization considering possible dependencies in the form of pairwise conditional probabilities are studied in [3], which also suggest a clear theoretical framework. Their MIMIC technique (Mutual-Information-Maximizing Input Clustering) aims at estimating a probability density for points with value below a given threshold (remember that the function is to be *minimized*). The method aims at modeling the distribution p^θ,

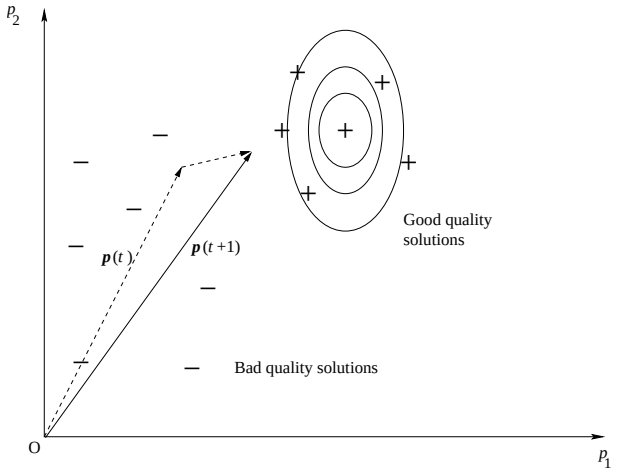


Figure 5.3: PBIL: the "prototype" vector **p** gradually shifts towards good quality solutions (qualitative example in two dimensions).

uniform over inputs X with $f(X) \leq \theta$, zero elsewhere:

$$p^{\theta}(x) = \begin{cases} \dfrac{1}{\text{measure}(X)} & \text{if } f(x) \leq \theta \\ 0 & \text{otherwise.} \end{cases}$$

If this distribution is known and the best value $\hat{\theta} = \min_X f(X)$ is also known, a single sample from $p^{\hat{\theta}}$ would be sufficient to identify the optimizer.

Before proceeding with the method, a proper threshold θ is to be selected. In the absence of a priori knowledge, we can aim at tuning θ so that it permits a non-trivial estimation of p^{θ} (if θ is too low no sample point will make it, if it is too large all points are in). The choice in [3] is to *adapt* θ so that it is equal to a fixed N-th percentile of the data (so that a specific fraction of the sample points is below the threshold). If the model is appropriate, one aims at a *threshold-descent* behavior: the threshold will be high at the beginning, then the areas leading to lower and lower costs will be identified leading to a progressively lower threshold (a similar technique will be encountered for racing in Chapter 8). The algorithm therefore proceeds as follows:

1. Generate a random initial population uniformly in the input space;
2. $\theta_0 \leftarrow$ mean f value on this population; $t \leftarrow 0$;
3. **repeat**:
4. Update the density estimator model p^{θ_t};
5. Generate more samples from the model p^{θ_t};
6. $\theta_{t+1} \leftarrow N$-th percentile of the data;
7. retain only the points less than θ_{t+1}; $t \leftarrow t + 1$;

What is left is the choice of a (parametric) model so that it can be used and updated within acceptable CPU times and without requiring an excessive number of sample points. In [3] one first approximates the true distribution $p(X)$ with a distribution $\hat{p}(X)$ chosen from a reduced set of possibilities. The closest $\hat{p}(X)$ is defined as the one minimizing the Kullback-Leibler divergence $D(p\|\hat{p})$, a widely used measurement of the agreement between distributions. Finally, because identifying the optimal $\hat{p}(X)$ requires excessive computational resources, one resorts to a greedy construction of the distribution $\hat{p}(X)$.

Now that the path is clear, let's see the details. The joint probability distribution

$$p(X) = p(X_1|X_2, \ldots, X_n)p(X_2|X_3, \ldots, X_n) \ldots p(X_{n-1}|X_n)p(X_n)$$

is approximated starting from pairwise conditional probabilities $p(X_i|X_j)$. In detail, one considers a class of probability distributions $\hat{p}_\pi(X)$ where each member is defined by a permutation π of the indices between 1 and n.

$$\hat{p}_\pi(X) = p(X_{\pi_1}|X_{\pi_2})p(X_{\pi_2}|X_{\pi_3})\cdots p(X_{\pi_{n-1}}|X_{\pi_n})p(X_{\pi_n})$$

One aims at picking the the permutation π such that $\hat{p}_\pi(X)$ has minimum divergence with the actual distribution. The Kullback-Leibler divergence $D(p\|\hat{p})$ to minimize is:

$$\begin{aligned} D(p\|\hat{p}_\pi) &= \int_{\mathcal{X}} p(X)\big(\log p(X) - \log \hat{p}_\pi(X)\big)\,\mathrm{d}X \\ &= E_p[\log p] - E_p[\log \hat{p}_\pi] \\ &= -h(p) + \underbrace{h(X_{i_1}|X_{i_2}) + h(X_{i_2}|X_{i_3}) + \cdots + h(X_{i_{n-1}}|X_{i_n}) + h(X_{i_n})}, \end{aligned} \tag{5.1}$$

where the properties of the logarithm and the definition of entropy h were used. Actually, the first term in the summation does not depend on π so that the cost function to be minimized is give by the rest of the summation (the terms within the underbrace). Because searching among all $n!$ permutations is too demanding one adopts a greedy algorithm to fix indices sequentially:

1. $i_n \leftarrow \arg\min_j \hat{h}(X_j)$;
2. **for** $k = n-1, n-2, \ldots, 1$:
3. $\quad i_k \leftarrow \underset{j \notin \{i_{k+1},\ldots,i_n\}}{\arg\min} \hat{h}(X_j|X_{i_{k+1}})$.

where $\hat{h}$ is the estimated empirical entropy. After the distribution is chosen one generates samples as follows:

1. Randomly pick a value for X_{i_n} based on the estimated $\hat{p}(X_{i_n})$;
2. **for** $k = n-1, n-2, \ldots, 1$:
3. $\quad$ pick a value for X_{i_k} based on the estimated $\hat{p}(X_{i_k}|X_{i_{k+1}})$.

While an approximated probability distribution is found in an heuristic greedy manner above, an alternate standard technique is that of *stochastic gradient ascent*. For simplicity, assume that f is positive (otherwise make it positive by applying a suitable transformation). Here one starts from an initial value for the parameters $\boldsymbol{\theta}$ of the probabilistic construction model and does steepest descent where the exact gradient is estimated by sampling [12]. In detail, let's assume that the probabilistic sample generation model is determined by a vector of parameters $\boldsymbol{\theta} \in \Theta$, and that the related probability distribution over generated solutions $p_{\boldsymbol{\theta}}$ is differentiable. The combinatorial optimization problem is now substituted with a continuous optimization problem: determine the model parameters leading to the highest expected value of f:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} E_{p_{\boldsymbol{\theta}}} f(X)$$

Gradient ascent consists of starting from an initial $\boldsymbol{\theta}^0$ and, at each step, calculating the gradient and updating $\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \epsilon_t \nabla E(\boldsymbol{\theta}^t)$, where ϵ_t is a suitably small step-size.

Next, because

$$\begin{aligned} \nabla E(\boldsymbol{\theta}) &= \nabla \sum_X f(X) P_{\boldsymbol{\theta}}(X) = \sum_X f(X) \nabla P_{\boldsymbol{\theta}}(X) \\ &= \sum_X f(X) P_{\boldsymbol{\theta}} \frac{\nabla P_{\boldsymbol{\theta}}(X)}{P_{\boldsymbol{\theta}}} = \sum_X f(X) P_{\boldsymbol{\theta}} \nabla \ln P_{\boldsymbol{\theta}}, \end{aligned} \tag{5.2}$$

The gradient of the expectation can be substituted with an empirical mean from samples $\boldsymbol{s}$ extracted from the distribution S_t, obtaining the following update rule:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \epsilon_t \sum_{s \in S_t} f(X) \nabla \ln P_{\boldsymbol{\theta}^t} \tag{5.3}$$

We leave to the reader the gradient calculation, because it is dependent on the specific model.

5.4 The cross-entropy model

The *cross-entropy* method of [14] [2] builds upon the above ideas and upon [13] which considers an adaptive algorithm for estimating probabilities of rare events in stochastic systems. One starts from a distribution p_0 which is progressively updated to increase the probability of generating good solutions. If we multiply at each iteration the starting distribution by the function evaluation $\hat{p}(X) \propto p_t(X)f(X)$ we obtain higher probability values for higher function values, it looks like we are on the correct road: when n goes to infinity $p_n(X) \propto p_t(X)(f(X))^n$ the probability tends to be different from zero only at the global optima solutions! Unfortunately there is an obstacle: if p_t is a distribution which can be obtained by fixing parameters $\boldsymbol{\theta}$ in our model, there is no guarantee that also $\hat{p}$ is going to be a member of this parametric family. Here our cross-entropy measure (CE) comes to the rescue, we will *project* $\hat{p}$ to the closest distribution is the parametric family, where closeness is measured by the Kullback-Leibler divergence:

$$D(\hat{p}\|p) = \sum_X \hat{p} \ln \frac{\hat{p}(X)}{p(X)} \tag{5.4}$$

or, taking into account that $\sum_X \hat{p} \ln \hat{p}$ is constant when minimizing over p, by the *cross-entropy*:

$$h(\hat{p}|p) = -\sum_X \hat{p} \ln p(X) \tag{5.5}$$

Because our $\hat{p}$ is proportional to $p_t \cdot f$, the final distribution update is given by solving the maximization problem:

$$p_{t+1} = \arg\max_{\boldsymbol{\theta}} \sum_X p_t(X)f(X) \ln p(X). \tag{5.6}$$

What holds for the original f also holds for monotonic transformations of f, which may also depend on memory (so that different functions can be used at different steps). An example is an indicator function $I(f < \theta_t)$ as used in the MIMIC technique, or a Boltzmann function $f_T = \exp -f/T$ where the temperature T can be adapted (large T values tend to smooth f differences leading to a less aggressive search).

Similarly to what was done for stochastic gradient descent, a fast sample approximation can substitute the summation over the entire solution space:

$$p_{t+1} = \arg\max_{\boldsymbol{\theta}} \sum_{s \in S_t} f(X) \ln p(X). \tag{5.7}$$

We already anticipate the final part of the story: the above maximization problem usually cannot be solved exactly but again one can resort to stochastic gradient ascent, for example starting from the previous p_t solution (and again remembering the pitfalls of local minima). If instead of a complete ascent one follows only one step along the (estimated) gradient on actually recovers the stochastic gradient ascent rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon_t \sum_{s \in S_t} f(s) \nabla \ln p_{\boldsymbol{\theta}_t}(s) \tag{5.8}$$

The Cross-Entropy method therefore appears as a generalization of the SGA technique, allowing for time-dependent quality functions.

The theoretical framework above is surely of interest, unfortunately many issues have to be specified before obtaining an effective algorithm for a specific problem. In particular the model has to be sufficiently simple to allow fast computation but also sufficiently flexible to accommodate the structural properties of specific landscapes. In addition local minima are haunting along the way of gradient ascent in the solution of the embedded optimization tasks implicit in the technique. An intrinsic danger is related on the "intensification" (exploitation) flavor of the approach, one tends to search where previous searches have been successful.

Unfortunately, a new gold mine will never be found if the first iterations concentrate the exploration too much, see also Fig. 5.1. Contrary to gold mining, the reward in optimization is not on revisiting old good solutions but on rapidly exploiting a promising region and then moving on to explore uncharted territory.

In spite of the practical challenges, some approaches based on analogies with ant colonies behavior [4] are based on similar principles: solution construction happens via a model which is influenced by the quality of previously generated solutions, although the most successful ACO applications are in fact combinations of the basic technique with advanced memory-based features, local search, and problem specific heuristic information about the a priori desirability of the different solution components.

The Greedy Randomized Adaptive Search (GRASP) framework [5] is based on repetitions of randomized greedy constructions and local search starting from the just constructed solution. Different starting points for local search are obtained by either stochastically breaking ties during the selection of the next solution component or by *relaxing the greediness*, i.e., putting at each greedy iteration a fraction of the most promising solution components in a restricted candidate list, from which the winner is randomly extracted. The bigger the candidate list, the larger the exploration characteristics of the construction. In spite of the name, there is actually no adaptation to previous runs in GRASP, but the opportunity arises for a set of self-tuning possibilities. For example the size of the candidate list can be self-tuned, or statistics about relationships between final quality and individual components (or more complex structural properties learned from the previous constructions) can influence the choice of the next solution component to add, going towards the model-base search context. Preliminary investigations about Reactive-GRASP are presented in [10, 6]. The purpose is to adapt the size of the candidate list by considering information gathered about the quality of solutions generated with different values. In detail, a parameter α is defined as the fraction of elements that are considered in the restricted candidate list; a set of possible values $\{\alpha_1, \ldots, \alpha_n\}$ is considered. A value of α is chosen at each construction with probability p_i. The adaptation acts on the probabilities: at the beginning they are uniform ($p_i = 1/n$), while after a number of GRASP constructions the average solution value a_i obtained when using α_i is computed and the probabilities are updated so that they become proportional to these average values (of course scaled so that they sum up to one). The power of reactive-GRASP is derived both from the additional diversification implicit when considering different values for α and on the reactive adaptation of the probabilities.

Estimation of Distributions (EDA) [8] algorithms have been proposed in the framework of evolutionary computation for modeling promising solutions in a probabilistic manner, and use it to produce the next generation. A survey in [9] considers population-based probabilistic search algorithms based on modeling promising solutions by estimating their probability distribution and using the model to guide the exploration of the search space".

Bibliography

[1] S. Baluja and R. Caruana, *Removing the genetics from the standard genetic algorithm*, Tech. report, School of Computer Science, Carnegie Mellon University, 1995, CMU-CS-95-141.

[2] P. de Boer, D. Kroese, S. Mannor, and R. Rubinstein, *A tutorial on the cross-entropy method*, Annals of Operations Research **134** (2005), 19–67.

[3] Jeremy S. de Bonet, Charles L. Isbell Jr., and Paul Viola, *MIMIC: Finding optima by estimating probability densities*, Advances in Neural Information Processing Systems (Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, eds.), vol. 9, The MIT Press, 1997, p. 424.

[4] Marco Dorigo and Christian Blum, *Ant colony optimization theory: a survey*, Theor. Comput. Sci. **344** (2005), no. 2-3, 243–278.

[5] T.A. Feo and M.G.C. Resende, *Greedy randomized adaptive search procedures*, Journal of Global Optimization **6** (1995), 109–133.

[6] Fernando C. Gomes, Panos Pardalos, Carlos S. Oliveira, and Mauricio G. C. Resende, *Reactive grasp with path relinking for channel assignment in mobile phone networks*, DIALM '01: Proceedings of the 5th international workshop on Discrete algorithms and methods for mobile computing and communications (New York, NY, USA), ACM Press, 2001, pp. 60–67.

[7] 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.

[8] H. Mühlenbein and G. Paa, *From recombination of genes to the estimation of distributions i. binary parameters*, Parallel Problem Solving from NaturePPSN IV (A. Eiben, T. Bäck, M. Shoenauer, and H. Schwefel, eds.), 1996, p. 178187.

[9] M. Pelikan, D.E. Goldberg, and F. Lobo, *A survey of optimization by building and using probabilistic models*, Computational Optimization and Applications **21** (2002), no. 1, 5–20.

[10] M. Prais and C. C. Ribeiro, *Reactive grasp: An application to a matrix decomposition problem in tdma traffic assignment*, INFORMS JOURNAL ON COMPUTING **12** (2000), no. 3, 164–176.

[11] J. R. Quinlan, *Combining instance-based and model-based learning*, Proceedings of the Tenth International Conference on Machine Learning (Amherst, Massachusetts), Morgan Kaufmann, 1993, pp. 236–243.

[12] H. Robbins and S. Monro, *A stochastic approximation method*, Ann. Math. Stat. **22** (1951), 400–407.

[13] R. Rubinstein, *Optimization of computer simulation models with rare events*, European Journal of Operations Research **99** (1997), 89–112.

[14] _____, *The cross-entropy method for combinatorial and continuous optimization*, METHODOLOGY AND COMPUTING IN APPLIED PROBABILITY **1** (1999), no. 2, 127–190.

[15] M. Zlochin, M. Birattari, N. Meuleau, and M. Dorigo, *Model-based search for combinatorial optimization*, pp. 373–395.

Chapter 6

Reacting on the objective function

How to Do a Proper Push-Up. Push-ups aren't just for buff army trainees; they are great upper body, low-cost exercise. Here's the proper way to do them anywhere. (ad. in the web)

This chapter considers reactive modification of the objective function in order to support appropriate diversification of the search process. Contrary to to the model-based search techniques, here the focus is not on modeling a solution-generation process to intensify the search in promising regions, 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. 6.1, 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.

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 problem is of particular interest. Different variations of local search with randomness techniques have been proposed for SAT and MAX-SAT starting from the late eighties, for some examples see [11], [27], and the updated review of [14]. These techniques were in part motivated by previous applications of "min-conflicts" heuristics in the area of Artificial Intelligence, see for example [10] and [18].

The influential algorithm GSAT [27] 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. Different "noise" strategies to escape from attraction basins are added to GSAT in [25, 26]. In particular, the GSAT-with-walk algorithm.

The algorithm is briefly summarized in Fig. 6.2. 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, 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.

In local search for SAT [27] methods (GSAT) single bits are flipped repeatedly in order to increase the number of satisfied clauses. After a local minimum is reached a *plateau* search phase, when bits are flipped without changing the number of satisfied clauses, is often a tax to pay before reaching attraction basins with a lower number of unsatisfied clauses. Instead of executing an approximate random-walk trajectory along the plateau it would be better to "give some tilt" to the plateau surface to fasten the identification of other attraction basins in the neighborhood. If all clauses are not born to be equal, getting different values in the neighborhood is facilitated, and soon after the first proposals, method based on clause

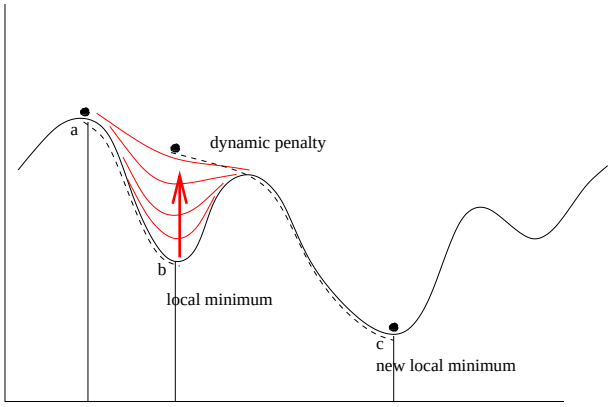


Figure 6.1: Transformation of the objective function to gently push the solution out of a given local minimum.

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

Figure 6.2: The "GSAT-with-walk" algorithm. RANDOM generates random numbers in the range $[0, 1]$

weighting appeared [19, 28].

Clause–weighting has been proposed in [24] 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.

A weight w_i is associated to each clause, and the objective 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 [28] suggest 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 one which remain unsatisfied after a try of local search descent followed by plateau search, and their weight is increased so that future runs will give them more priority when picking a move. More algorithm based on the same weighting principle are proposed in [7, 8], where clause weights are updated after each flip: the reaction from the unsatisfied clauses is now immediate as one does not wait until the end of a try (weighted GSAT or WGSAT). If weights are only increased, after some time their size becomes large and their relative magnitude will reflect 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 [8], either *reducing* the clause weight when a clause is satisfied, or storing the weights increments which

took place recently, which is obtained by a weight decay scheme (each weight is reduced by a factor p 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 (see Chapter 4). 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:

$$w_i \leftarrow p\, w_i + \delta$$

where one introduces a decay rate p and a "learning rate" δ. A faster decay (lower p 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."

A different opportunity for self-adaptation is identified in [12], which proposes and **adaptive noise** mechanism for WalkSAT. In WalkSAT [25], 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 [17] 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. Parameters with a diversifying effect similar to the noise in WalkSAT are present in many techniques (for example, in prohibition-based techniques, see Chapter 4). [17] considers this generalized notion of a "noise" parameter and suggests tuning the proper noise value by testing different settings through a preliminary series of short runs on a specific instance. Furthermore, an heuristic statistics to track which is closely related to the algorithm performance is suggested as 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%. A 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 ([21] uses Brent's method [6]).

An adaptive noise scheme is also proposed in [12], where the noise setting p is dynamically adjusted based on search progress. 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 [21].

A more recent proposal of a dynamic local search (DSL) for SAT is in [30]. The authors start from the Exponentiated Sub-Gradient (ESG) algorithm [22], 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 $\bar{w}$: $w \leftarrow w\ \rho + (1-\rho)\ \bar{w}$. They also introduce a *reactive version* of SAPS (RSAPS) that adaptively tunes one of the algorithm's important parameters. Following a scheme similar to that of [12], higher noise levels are determined in a reactive manner if and only if there is evidence of search stagnation, otherwise they are gradually reduced.

While we concentrated on the SAT problem above, a similar approach has been proposed with the term of Guided Local Search (GLS) [31, 33]. GLS aims at enabling intelligent search schemes to 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) [35] and from tabu search [9], simulated annealing [15],

and tunneling [16]. 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 [32], GENET's mechanism for escaping resembles *reinforcement learning* [3]: 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 [16] 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 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 [16, 2, 20]. Citing from a seminal paper [16], 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* [33].

A complete GLS scheme [33] 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)

neighborhood activation scheme depending on the current state.

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

$$h(X) = f(X) + \lambda \sum_i p_i \, I_i(X) \tag{6.1}$$

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:

$$I_i(X') \, \frac{c_i}{1 + p_i} \tag{6.2}$$

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 both implementation details which speedup each step but do not impact the dynamics (which do not change the search trajectory), for example an incremental evaluation of the h function, and 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. Let's note that 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 terminated 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, "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 the "don't look bits" method in [5]. 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 insert an edge incident to node i.

The parameter λ 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 [33].

While the motivations of GLS are clear, the interaction between the different ingredients causes a somewhat complicated dynamics. Let's note in passing that different units of measure for the cost in (6.1) 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 "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 [34]. One of the strategies avoids visiting points that are close to points visited before, a generalization of the STRICT-TS strategy, see Chapter 4. A recent algorithm with an *adaptive clause weight redistribution* is presented in [13], 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".

6.1 Eliminating plateaus by looking inside the problem structure

In the above presentation we considered modifications of the objective functions in order to modify the trajectory dynamics to escape from already-visited attractors. We now consider modifications which have a different purpose: that of eliminating *plateaus*. A *plateau* is a situation where one has a local minimum, but some neighbors have *the same* f value. By moving on a *plateau* one keeps a good starting point at a low f value, with the usual hope

to eventually reach an improving move. But large plateaus are always "embarrassing": one is stuck at a flat desert looking for water, no sun to give a direction. If one is not careful a lot of time can be spent looking around, maybe retracing the previous steps, with the thirst growing harder and harder. One would like some hints about a promising direction to take, maybe some mold caused by humidity so that water will get closer and closer by following the mold. Coming back from mirages to ...algorithms, one aims at breaking the ties among seemingly equivalent solutions (when considering only the f values). In particular, it may be the case that, while f is constant, some internal changes in the solution structure will eventually favor the discovery of an improving move. For example, in the MAX-SAT problem, even if the number of satisfied clauses remains the same, the amount of *redundancy* in their satisfaction (the number of different literals which make a clause satisfied) may pave the way to eventually flipping a variable which is redundant to satisfy some already-satisfied clauses in order to satisfy a new one. Aiming at a redundant satisfaction eliminates the embarrassment in selecting among seemingly similar situations and favors an improvement after a smaller number of steps than those required by a random-walk on the plateau.

6.1.1 Non-oblivious local search for SAT

In the design of efficient approximation algorithms for MAX–SAT an approach of interest is based on the use of *non-oblivious functions* independently introduced in [1] and in [29]. Let us consider the classical local search algorithm LS for MAX–SAT, here redefined as *oblivious* local search (LS-OB). 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 [4], the actual average results obtained on benchmark instances.

Let us introduce the notation and mention a theoretical result for MAX–2–SAT. Given an assignment X, let S_i denote the set of clauses in the given task in which exactly i literals are true and let $w(S_i)$ denote the cardinality of S_i. In addition, a d-neighborhood of a given truth assignment is defined as the set of all assignment where the value of at most d variables is changed. The theoretically-derived non–oblivious function for MAX–2–SAT is:

$$f_{NOB}(X) = \frac{3}{2}w(S_1) + 2w(S_2)$$

Theorems 7-8 of [29] state that the performance ratio for any oblivious local search algorithm with a d-neighborhood for MAX–2–SAT is $2/3$ for any $d = o(n)$, while non-oblivious local search with an 1-neighborhood achieves a performance ratio $3/4$. Therefore LS-NOB improves considerably the performance ratio even if the search is restricted to a much smaller neighborhood. In general, 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:

$$f_{NOB}(X) = \sum_{i=1}^{k} c_i w(S_i)$$

and the above given performance ratio is obtained if the quantities $\Delta_i = c_{i+1} - c_i$ satisfy:

$$\Delta_i = \frac{1}{(k-i+1)\begin{pmatrix} k \\ i-1 \end{pmatrix}} \left[\sum_{j=0}^{k-i} \begin{pmatrix} k \\ j \end{pmatrix} \right]$$

Because the positive factors c_i that multiply $w(S_i)$ 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. 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.

Let's note that a similar proposal to define an objective function that considers "how strongly clauses are satisfied" has been proposed later in [23], coupled with a multiplicative re-weighting of unsatisfied clauses ("smoothed descent and flood"). According to the authors, "additive updates do not work very well because clauses develop large weight differences over time, and this causes the update mechanism to lose its ability to rapidly adapt the weight profile to new regions of the search space". Again, the possibility to react rapidly to local characteristics is deemed of particular importance.

Bibliography

[1] P. Alimonti, *New local search approximation techniques for maximum generalized satisfiability problems*, Proc. Second Italian Conf. on Algorithms and Complexity, 1994, pp. 40–53.

[2] J. Bahren, V. Protopopescu, and D. Reister, *Trust: a deterministic algorithm for global optimization*, Science **276** (1997), 10941097.

[3] 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** (1983), 834–846.

[4] R. Battiti and M. Protasi, *Solving max-sat with non-oblivious functions and history-based heuristics*, Tech. report, Dipartimento di Matematica, Unversita' di Trento, Via Sommarive, 14 - 38050 Povo (Trento) Italia, March 1996.

[5] J.L. Bentley, *Experiments on traveling salesman heuristics*, Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (1990), 91–99.

[6] R. P. Brent, *Algorithms for minimization without derivatives*, Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1973.

[7] J. Frank, *Weighting for godot: Learning heuristics for GSAT*, PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, vol. 13, JOHN WILEY & SONS LTD, USA, 1996, pp. 338–343.

[8] _____, *Learning short-term weights for GSAT*, Proc. INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, vol. 15, LAWRENCE ERLBAUM ASSOCIATES LTD, USA, 1997, pp. 384–391.

[9] F. Glover, *Tabu search - part i*, ORSA Journal on Computing **1** (1989), no. 3, 190–260.

[10] J. Gu, *Parallel algorithms and architectures for very fast ai search*, Ph.D. thesis, University of Utah, 1989.

[11] Jun Gu, *Efficient local search for very large-scale satisfiability problem*, ACM SIGART Bulletin **3** (1992), no. 1, 8–12.

[12] H.H. Hoos, *An adaptive noise mechanism for WalkSAT*, PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, vol. 18, AAAI Press; MIT Press, 1999, pp. 655–660.

[13] A. Ishtaiwi, J. R. Thornton, Sattar A. Anbulagan, and D. N. Pham, *Adaptive clause weight redistribution*, Proceedings of the 12th International Conference on the Principles and Practice of Constraint Programming, CP-2006, Nantes, France, 2006, pp. 229–243.

[14] Gu J. and Du B., *A multispace search algorithm (invited paper)*, DIMACS Monograph on Global Minimization of Nonconvex Energy Functions, to appear.

[15] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), 671–680.

[16] A. Levy and A. Montalvo, *The tunneling algorithm for the global minimization of functions*, SIAM Journal on Scientific and Statistical Computing **6** (1985), 15–29.

[17] D. McAllester, B. Selman, and H. Kautz, *Evidence for invariants in local search*, PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, no. 14, JOHN WILEY & SONS LTD, USA, 1997, pp. 321–326.

[18] 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** (1992), no. 1-3, 161–205.

[19] P. Morris, *The breakout method for escaping from local minima*, PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, no. 11, JOHN WILEY & SONS LTD, USA, 1993, p. 40.

[20] E.M. Oblow, *Pt: a stochastic tunneling algorithm for global optimization*, Journal of Global Optimization **20** (2001), no. 2, 191–208.

[21] D.J. Patterson and H. Kautz, *Auto-walksat: A self-tuning implementation of walk-sat*, Electronic Notes in Discrete Mathematics (ENDM), 2001.

[22] D. Schuurmans, F. Southey, and R.C. Holte, *The exponentiated subgradient algorithm for heuristic boolean programming*, Proc. INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, vol. 17, LAWRENCE ERLBAUM ASSOCIATES LTD, USA, 2001, pp. 334–341.

[23] Dale Schuurmans and Finnegan Southey, *Local search characteristics of incomplete sat procedures*, Artif. Intell. **132** (2001), no. 2, 121–150.

[24] B. Selman and H.A. Kautz, *An empirical study of greedy local search for satisfiability testing*, Proceedings of the eleventh national Conference on Artificial Intelligence (AAAI-93) (Washington, D. C.), 1993, to appear.

[25] B. Selman, H.A. Kautz, and B. Cohen, *Noise strategies for improving local search*, PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, vol. 12, JOHN WILEY & SONS LTD, USA, 1994.

[26] _____, *Local search strategies for satisfiability testing*, Proceedings of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability (M. Trick and D. S. Johson, eds.), DIMACS Series on Discrete Mathematics and Theoretical Computer Science, no. 26, 1996, pp. 521–531.

[27] B. Selman, H. Levesque, and D. Mitchell, *A new method for solving hard satisfiability problems*, Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92) (San Jose, Ca), July 1992, pp. 440–446.

[28] Bart Selman and Henry Kautz, *Domain-independent extensions to GSAT: Solving large structured satisfiability problems*, Proceedings of IJCAI-93, 1993, pp. 290–295.

[29] S.Khanna, R.Motwani, M.Sudan, and U.Vazirani, *On syntactic versus computational views of approximability*, Proc. 35th Ann. IEEE Symp. on Foundations of Computer Science, 1994, pp. 819–836.

[30] F. Hutter D.A.D. Tompkins and H.H. Hoos, *Scaling and probabilistic smoothing: Efficient dynamic local search for sat*, Proc. Principles and Practice of Constraint Programming - CP 2002 : 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, LNCS, vol. 2470, Springer Verlag, 2002, pp. 233–248.

[31] C. Voudouris and E. Tsang, *Partial constraint satisfaction problems and guided local search*, Proceedings of 2nd Int. Conf. on Practical Application of Constraint Technology (PACT 96), London, April 1996, pp. 337–356.

[32] Chris Voudouris and Edward Tsang, *The tunneling algorithm for partial CSPs and combinatorial optimization problems*, Tech. Report CSM-213, 1994.

[33] Christos Voudouris and Edward Tsang, *Guided local search and its application to the traveling salesman problem*, European Journal of Operational Research **113** (1999), 469–499.

[34] B.W. Wah and Z. Wu, *Penalty Formulations and Trap-Avoidance Strategies for Solving Hard Satisfiability Problems*, Journal of Computer Science and Technology **20** (2005), no. 1, 3–17.

[35] C.J. Wang and E.P.K. Tsang, *Solving constraint satisfaction problems using neural networks*, Proc. Second International Conference on Artificial Neural Networks, 1991, pp. 295–299.

Chapter 7

Algorithm portfolios and restart strategies

Union gives strength.
(Aesop)

7.1 Introduction: portfolios and restarts

Let us consider *Las Vegas* algorithms, which always terminate with a correct solution to a problem with a stochastic distribution of solution times. Let us assume that we are interested both in the expected value of the solution time and in its standard deviation. 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 solution times and standard deviations: one is based on restarting an algorithm if it does not terminate within a given time, the other one is based on combining more runs in a time-sharing interleaving manner: the portfolio approach.

The *algorithm portfolio* method, first proposed in [8], follows 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. Using an algorithm portfolio consists of running 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. 7.1.

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 termination time can be of 1 second or of 1000 seconds, with the same probability. If I run a single process the expected termination time is approximately of 500 seconds. If I run more copies, the probability that at least one of them is lucky (i.e., 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 time of the restart strategy will be much less than 500 seconds.

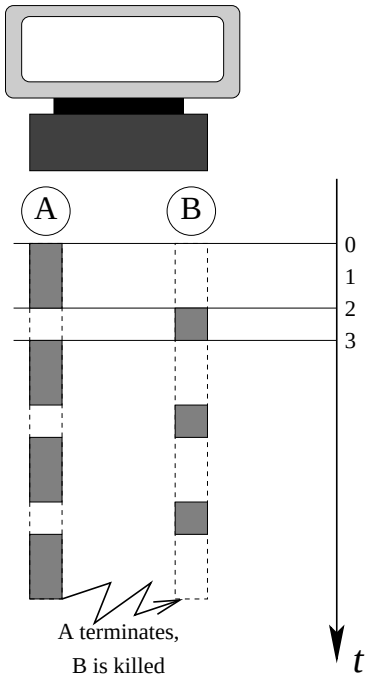


Figure 7.1: A sequential portfolio strategy.

As another example, when surfing the web the response time to deliver a page can vary a lot. Again, it is intuitive that clicking again on the same link after the patience is finished can save the user from an "endless" waiting time.

7.2 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 $a_2 = 1 - \alpha_1$. The completion time of the two-process portfolio system is therefore described by the random variable

$$T = \min\{\alpha_1^{-1} T_1, \alpha_2^{-1} T_2\}. \tag{7.1}$$

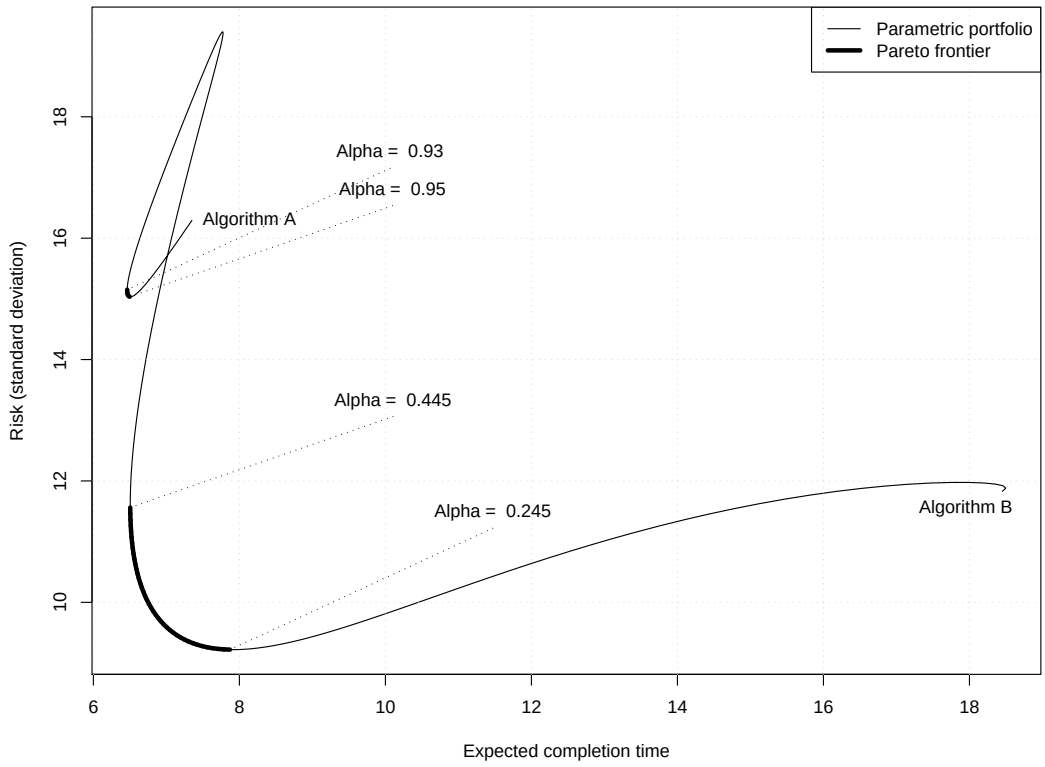


Figure 7.2: Expected time versus standard deviation (risk) plot. The efficient frontier contains the set of non-dominated configurations (a.k.a. Pareto-optimal or extremal points).

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 E(T) and risk $\sqrt{\mathrm{Var}(T)}$ becomes available. Fig. 7.2 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 suffers from a large standard deviation (because the distribution is bimodal or heavy-tailed), while algorithm $\mathcal{B}$ has a higher expected completion time, 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

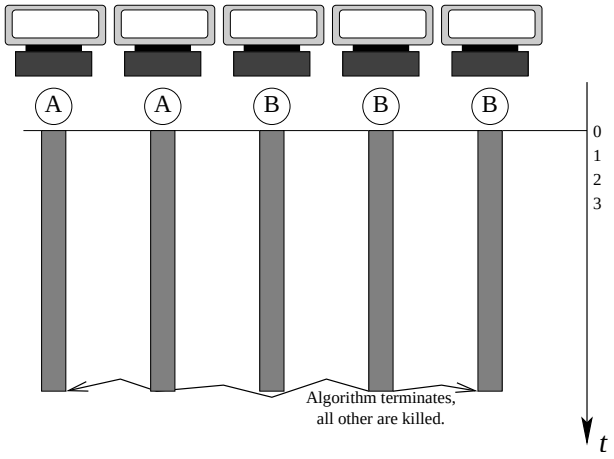


Figure 7.3: A portfolio strategy on a parallel machine.

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 are obviously discardable, there are no free meals and a higher return comes with a higher risk.

7.2.1 Parallel processing

Let us consider a different context [6] 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. 7.3. 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:

$$p(t) = \sum_{i=1}^{N} \binom{N}{i} p_1(t)^i S_1(t)^{N-i} \tag{7.2}$$

The portfolio survival function $S(t)$ is easier to compute on the basis of the survival function of the single process $S_1(t)$:

$$S(t) = S_1(t)^N \tag{7.3}$$

When two algorithms are considered, the probability computation has to be modified to take into account 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).

$$p(t) = \sum_{\substack{0 \le i_1 \le n_1 \\ 0 \le i_2 \le n_2 \\ i_1 + i_2 \ge 1}} \binom{n_1}{i_1} p_1(t)^{i_1} S_1(t)^{n_1 - i_1} \binom{n_2}{i_2} p_2(t)^{i_2} S_2(t)^{n_2 - i_2}. \quad (7.4)$$

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 termination times. Portfolios can be effective to "cure" the typical 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 [5]. 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 [8, 6] show that in some cases only 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 [6] 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.

7.3 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.

If the distributions $p_i(t)$ are unknown, or if they are only partially known, one has to resort to reactive strategies, 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 a first value of f_1 and then refine the estimate of $p_i(t)$ when more information is received and use it to define subsequent values of f_1. A preliminary suggestion of dynamic online strategies is present in [8].

A "life-long learning" approach for dynamic algorithm portfolios is considered in [2]. 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 [1], is fully in the reactive search spirit. In the inter-problem AOTA framework, see Fig. 7.4, 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 solution time of the whole set of instances. The model used to predict the termination time $p_i(t)$ 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 with a heuristic function which is decreasing for longer estimated termination times τ_i.

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 7.4: The inter-problem AOTA framework.

7.4 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 [13]:

$$E[T] < E[T - \tau | T > \tau]. \tag{7.5}$$

Whether restart is beneficial or not depends of course on the distribution of solution times. As a trivial example, if the distribution is exponential, restarting the algorithm does not modify the expected solution time.

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 optimal solution will be visited, an enormous number of iterations can be spent in the attraction basin around the local minimizer before escaping. Restart is a method of choice to escape 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

$$S_\tau(t) = \Pr(T_\tau > t) = (1 - P_\tau)^{\lfloor t/\tau \rfloor} \Pr(T > t \bmod \tau). \tag{7.6}$$

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 [10]. In this case, the expected value of the total run-time T_τ , i.e., the sum of run-times of the successful run, and all previous unsuccessful runs is equal to:

$$E(T_\tau) = \frac{\tau - \int_0^\tau F(t)\,\mathrm{d}t}{F(\tau)} \tag{7.7}$$

where $F(\tau)$ is the cumulative distribution function of the run-time 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:

$$\int_0^\tau tp(t)\,\mathrm{d}t + \tau(1 - F(\tau))$$

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:

$$\int_0^\tau tF'(t)\,\mathrm{d}t.$$

The result follows from the fact that:

$$\frac{\mathrm{d}}{\mathrm{d}t}(tF(t)) = tF'(t) + F(t)$$

and therefore:

$$\int_0^\tau tF'(t)\,\mathrm{d}t + \int_0^\tau F(t)\,\mathrm{d}t = \tau F(\tau)$$

giving (7.7).

In the discrete case:

$$E(T_\tau) = \frac{\tau - \sum_{t<\tau} F(t)}{F(\tau)} \tag{7.8}$$

If the distribution is known, an optimal cutoff time can be determined by minimizing (7.7). If the distribution is not known, a universal non-uniform strategy, with cutoff sequence: $(1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, \ldots)$ achieves a performance within a logarithmic factor of the expected run-time of the optimal policy, see [10] for details.

Calculating the run-time 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 [11]. 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 $\mathcal{T} = (t_1, t_2, \ldots, t_k)$ given θ:

$$\mathcal{L}(\mathcal{T}|\theta) = \prod_{i=1}^{k} \mathcal{L}(t_i|\theta) = \prod_{i=1}^{k} g(t_i|\theta) \tag{7.9}$$

With censoring, some experimental runs will exceed the cutoff time t_c. In these cases the corresponding multiplicative term in (7.9) is substituted with

$$\mathcal{L}_c(t_c|\theta) = \int_{t_c}^{\infty} g(\tau|\theta)\,\mathrm{d}\tau = 1 - G(t_c|\theta) \tag{7.10}$$

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 run-time distribution, ii) determine the "best" parameters of the model by maximizing the likelihood, where some terms are substituted with the censored likelihood of (7.10), iii) use the estimated run-time distribution to determine the optimal restart time. Some examples of parametric models are considered in [3].

7.5 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 during a run or during a sequence of runs on related instances become available. Real-time observations about the characteristics of a specific instance and the state of the solver *during a run* permit better results.

In [7, 9] it is shown how to use features capturing the state of a solver during the initial phase of the run to **predict the length of a run**, so that the prediction can be used by dynamic restart policies. Bayesian models to predict the run time 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 in a supervised manner. 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 [7] 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. In spite of the crudeness of the model (for example no observations during the steps after O are used, only a bound and not the exact expected number of steps is minimized) 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), see [7] for details.

The assumption of independence among runs is relaxed in [12]. 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 7.4. The work in [12] considers the context where one among several RTD 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 evidence obtained during the run about the fact that a SAT instance is or is not satisfiable).

7.6 Summary

Bibliography

[1] M. Gagliolo and J. Schmidhuber, *A neural network model for inter-problem adaptive online time allocation*, Proceedings Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005, 15th Int. Conf. (Warsaw) (W. Duch et al., ed.), vol. 2, Springer, Berlin, 2005, pp. 7–12.

[2] ———, *Dynamic algorithm portfolios*, Proceedings AI and MATH '06, Ninth International Symposium on Artificial Intelligence and Mathematics (Fort Lauderdale, Florida), Jan 2006.

[3] ———, *Impact of censored sampling on the performance of restart strategies*, CP 2006 - Twelfth International Conference on Principles and Practice of Constraint Programming - Nantes, France, Springer, Berlin, Sep 2006, pp. 167–181.

[4] ———, *Learning restart strategies*, IJCAI 2007 - Twentieth International Joint Conference on Artificial Intelligence, January 6-12, Hyderabad, India, 2007, in press.

[5] Carla Gomes, Bart Selman, Nuno Crato, and Henry Kautz, *Heavy-tailed phenomena in satisfiability and constraint satisfaction problems*, J. of Automated Reasoning **24** (2000), no. (1/2), 67–100.

[6] Carla P. Gomes and Bart Selman, *Algorithm portfolios*, Artif. Intell. **126** (2001), no. 1-2, 43–62.

[7] E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and D. M. Chickering, *A bayesian approach to tackling hard computational problems*, Seventeenth Conference on Uncertainty in Artificial Intelligence (Seattle,USA), Aug 2001, pp. 235–244.

[8] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg, *An economics approach to hard computational problems*, Science **275** (1997), 51–54.

[9] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman, *Dynamic restart policies*, Eighteenth national conference on Artificial intelligence (Menlo Park, CA, USA), American Association for Artificial Intelligence, 2002, pp. 674–681.

[10] M. Luby, A. Sinclair, and D. Zuckerman, *Optimal speedup of las vegas algorithms*, Information Processing Letters **47** (1993), no. 4, 173180.

[11] W. Nelson, *Applied life data analysis*, Jon Wiley, New York, 1982.

[12] Y. Ruan, E. Horvitz, and H. Kautz, *Restart policies with dependence among runs: A dynamic programming approach*, (2002).

[13] A. van Moorsel and K. Wolter, *Analysis and algorithms for restart*, 2004.

Chapter 8

Racing

The smart player goes with the winners. . .

8.1 Introduction

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 learning loop while the portfolio or restart scheme run. 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 we get more and more information about the relative performance and we can update periodically our bets on the winning horses, which are assigned a growing fraction of the available future computing cycles, see Fig. 8.1.

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 CPU cycles to maximize 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 is known one would immediately pull only the better 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 better 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 [7] 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 [4, 3]. An asymptotically optimal algorithm is presented in [10], in the assumption of a generalized extreme value (GEV) payoff distribution for each arm. Our explanation follows closely [9], which presents a simple distribution-free approach.

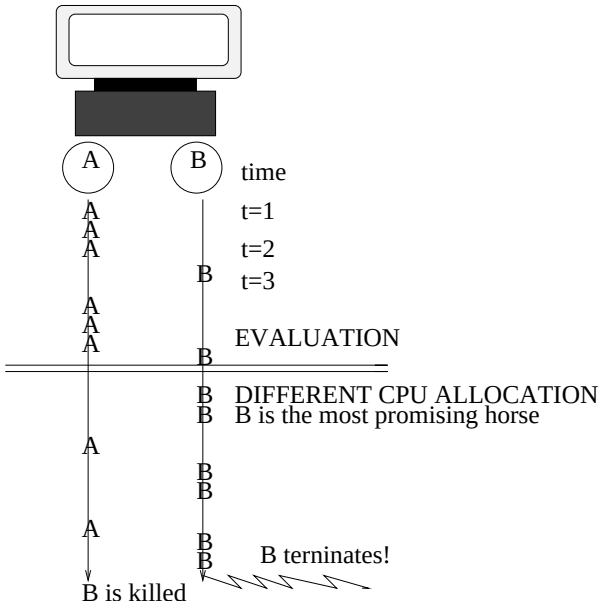


Figure 8.1: A racing strategy. The different horses (algorithms) are evaluated periodically to reallocate the CPU time shares.

8.2 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, see Fig. 8.2 and Fig. 8.3. The upper bound is derived from Chernoff's inequality and is as follows:

$$U(\mu, n) = \begin{cases} \mu + \frac{\alpha + \sqrt{2n\mu\alpha + \alpha^2}}{n} & \text{if } n > 0 \\ \infty & \text{otherwise} \end{cases} \tag{8.1}$$

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 $X = \sum_{i=1} nX_i$ 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:

$$P\left[\frac{X}{n} < (1-\beta)\mu\right] < e^{-\frac{n\mu\beta^2}{2}} \tag{8.2}$$

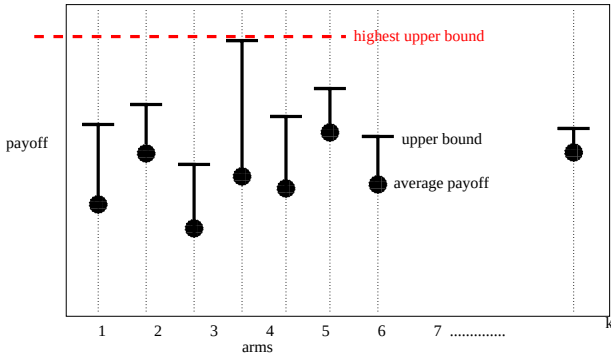


Figure 8.2: Racing with interval estimation. At each iteration an estimate of the expected payoff of each arm as well as its "error bar" are available.

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 8.3: The CHERNOFF-INTERVAL-ESTIMATION routine.

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. 8.3, 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:

$$(1 - \delta)2\sqrt{3\mu^* n(k-1)\alpha} + \delta\mu^* n \tag{8.3}$$

A similar algorithm based on Chernoff-Hoeffding's inequality has been presented in a previous work [1]. 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 [1] for more details and experimental results.

8.3 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 [9] 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 by 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

1. **function Threshold_Ascent**(s, n, δ)
2. Thres $\leftarrow 0$
3. **forall** $i \in \{1, 2, ..., k\}$
4. ⌈ **forall** R values
5. ⌊ Initialize $n_{i,R} \leftarrow 0$
6. **repeat** n times:
7. ⌈ **while** (number of payoffs received above threshold $\geq s$)
8. $Thres \leftarrow Thres + \Delta$ (raise threshold)
9. $\hat{i} \leftarrow \arg\max_i U(\bar{\nu}_i, n_i)$
10. pull arm $\hat{i}$, receive payoff R
11. ⌊ $n_{i,R} \leftarrow n_{i,R} + 1$

Figure 8.4: The THRESHOLD-ASCENT routine.

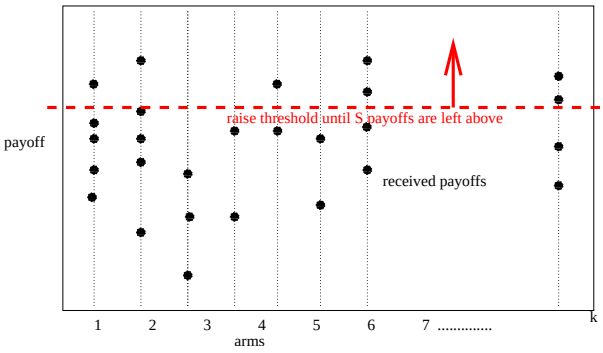


Figure 8.5: Threshold ascent: the threshold is progressively raised until a selected number of experimented payoffs is left.

heuristic solution presented in [9] reactively learns the appropriate threshold while the racing scheme runs, see Fig. 8.5 and Fig. 8.4.

The threshold starts from zero (remember that all values are bounded in $[0, 1]$), and it is progressively raised so that an appropriate number of less than s received rewards about the threshold is reached. 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 with 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, which is updated in the algorithm. 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 performances. 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.

8.4 Racing for off-line configuration of meta-heuristics

The context here is that of selecting in an off-line manner the best configuration of parameters θ for an heuristic solving repetitive problems [2]. 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 in the above example over different instances and different runs, the ideal solution of the configuration problem is:

$$\theta^* = \arg\min_{\theta} \mathcal{C}(\theta) \tag{8.4}$$

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):

$$\mathcal{C}(\theta) = E_{I,C}\left[c(\theta, i)\right] = \int_I \int_C c(\theta, i) dP_C(c|\theta, i) dP_I(i) \tag{8.5}$$

Because the probability distributions are not known at the beginning, ideally one could use a brute force approach, considering a very large number of instances and runs, tending to infinity, to calculate the expected value for each of the finite configurations. 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.

First, the above integral in (8.5) is estimated in a Monte-Carlo fashion by considering a set of instances. Second, 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 statistically sound criterion to determine that a candidate configuration θ_j is significantly worse than the the current best configuration available, given the current state of the experimentation.

The situation is illustrated in Fig. 8.6, 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 checks to see whether its optimistic value (top error bar) can beat the pessimistic real value of the best performer, see Fig. 8.6.

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 [8], 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:

$$Prob(\|E_{true} - E_{est}\| > \epsilon) < 2e^{\frac{-n\epsilon^2}{B^2}} \tag{8.6}$$

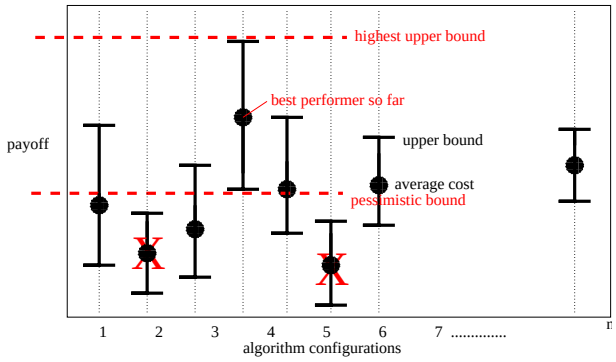


Figure 8.6: Racing for off-line optimal configuration of meta-heuristics. At each iteration an estimate of the expected performance with error bars is available. Error bars are reduced when more tests are executed, exact value depends also on confidence parameter δ. In the figure, configurations 2 and 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).

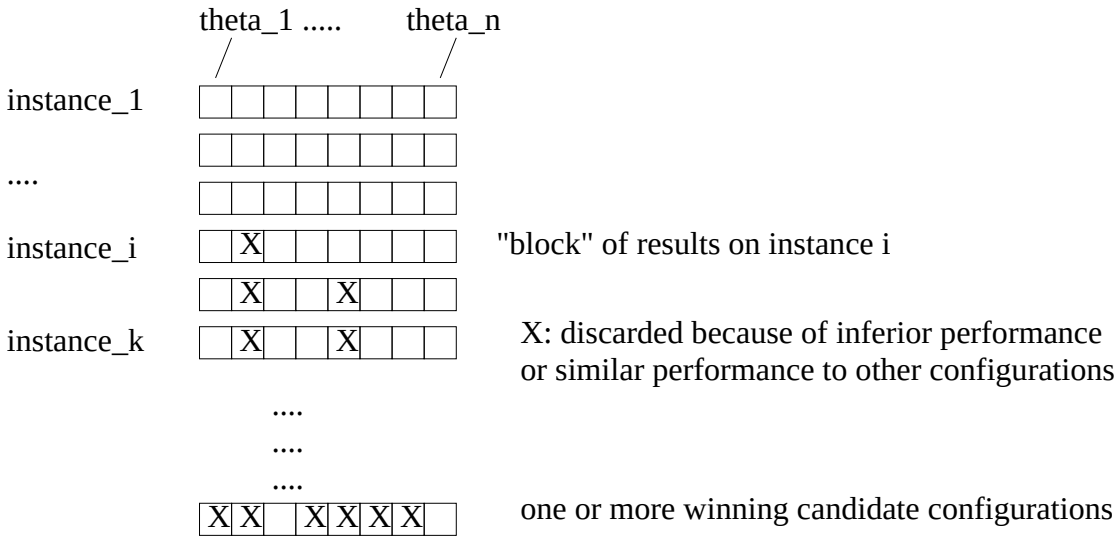


Figure 8.7: Racing for off-line optimal configuration of meta-heuristics. The most promising candidate algorithm configurations are identified asap so that these can be evaluated with a more precise estimate (more test instances). Each block corresponds to results of the various configurations on the same instance.

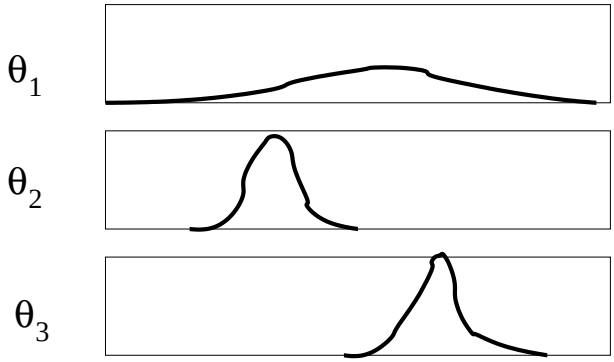


Figure 8.8: Bayesian elimination of inferior models, from the posterior distribution of costs of the different models one can eliminate the models which are inferior in a statistically significant manner, for example model θ_3 in the figure, in favor of model θ_2, while the situation is still undecided for model θ_1.

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 (8.6) (we want the probability of a large error to be less than δ), one easily solves for the error bar $\epsilon(n, \delta)$:

$$\epsilon(n, \delta) = \sqrt{\frac{B^2 \log(2/\delta)}{2n}} \tag{8.7}$$

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 [8].

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, this is the second method proposed in [8]. One candidate model is eliminated if the probability that a second model has a better expected performance is above the usual confidence threshold:

$$Prob(E^j_{true} > E^{j'}_{true} \| e_j(1), ..., e_j(n), e_{j'}(1), ..., e_{j'}(n)) > 1 - \delta \tag{8.8}$$

Additional methods for shrinking the intervals, as well as suggestions for using a statistical method known as blocking are explained in [8]. Model selection in continuous space is considered in [6].

In [2] the focus is explicitly on meta-heuristics configuration. Blocking through ranking is used in their F-RACE algorithm (based on the Friedman test), in addition to an aggregate test over all candidates performed before considering pairwise comparisons. Each block, see fig. 8.7 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 [5] considers the statistics T:

$$T = \frac{(n-1) \sum_{j=1}^{n} \left(R_j - \frac{k(n+1)}{2} \right)^2}{\sum_{l=1}^{k} \sum_{j=1}^{n} {R_{lj}}^2 - \frac{kn(n+1)^2}{4}} \tag{8.9}$$

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 considered different if:

$$\frac{\|R_j - R_h\|}{\sqrt{\frac{2k(1-\frac{T}{k(n-1)})\left(\sum_{l=1}^{k}\sum_{j=1}^{n} {R_{ij}}^2 - \frac{kn(n+1)^2}{4}\right)}{(k-1)(n-1)}}} > t_{1-\delta/2} \tag{8.10}$$

where $t_{1-\delta/2}$ is the $(1-\delta/2)$ quantile of the Student's t distribution.

Bibliography

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer, *Finite-time analysis of the multiarmed bandit problem*, Machine Learning **47** (2002), no. 2/3, 235–256.

[2] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, *A racing algorithm for configuring metaheuristics*, Proceedings of the Genetic and Evolutionary Computation Conference (San Francisco, CA, USA) (W.B. Langdon *et al.*, ed.), Morgan Kaufmann Publishers, 2002, Also available as: AIDA-2002-01 Technical Report of Intellektik, Technische Universität Darmstadt, Darmstadt, Germany, pp. 11–18.

[3] Vincent Cicirello and Stephen Smith, *The max k-armed bandit: A new model for exploration applied to search heuristic selection*, 20th National Conference on Artificial Intelligence (AAAI-05), July 2005, Best Paper Award.

[4] Vincent A. Cicirello and Stephen F. Smith, *Principles and practice of constraint programming cp 2004*, Lecture Notes in Computer Science, vol. 3258, ch. Heuristic Selection for Stochastic Search Optimization: Modeling Solution Quality by Extreme Value Theory, pp. 197–211, Springer Berlin / Heidelberg, 2004.

[5] W. J. Conover, *Practical nonparametric statistics*, John Wiley & Sons, December 1999.

[6] Artur Dubrawski and Jeff Schneider, *Memory based stochastic optimization for validation and tuning of function approximators*, Conference on AI and Statistics, 1997.

[7] Philip W. L. Fong, *A quantitative study of hypothesis selection*, International Conference on Machine Learning, 1995, pp. 226–234.

[8] Oden Maron and Andrew W. Moore, *The racing algorithm: Model selection for lazy learners*, Artificial Intelligence Review **11** (1997), no. 1-5, 193–225.

[9] M. J. Streeter and S.F. Smith, *A simple distribution-free approach to the max k-armed bandit problem*, Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP 2006), 2006.

[10] Matthew J. Streeter and Stephen F. Smith, *An asymptotically optimal algorithm for the max k-armed bandit problem*, AAAI, 2006.

Chapter 9

Metrics, landscapes and features

Measure what is measurable, and make measurable what is not so.
(Galileo Galilei)

In the scientific challenge of using learning strategies in the area of heuristics, finding appropriate metrics and appropriate features is an indispensable building block.

Let's consider some challenging questions:

- How can I **predict the future evolution of an heuristic**? E.g., the running time to completion, the probability of finding a solution within given time bounds, etc.
- How can I predict which is **the most effective heuristic for a given problem**, or for a specific instance?
- How can I predict that a problem is **intrinsically more difficult** for a given search technique?

While a complete review of the literature in this area is beyond the scope of this book, in the next sections we will briefly mention some interesting research issues related to: evaluating input parameters (features), see Section 9.1, measuring individual algorithm components and selecting them based on a diversification and bias metric, see Section 9.2 and Section 9.3, measuring problem difficulty, see Section 9.4.

9.1 Selecting features with mutual information

As in all scientific challenges the development of models with predicting power has to start from appropriate measurements, statistics, *input features*. The literature for selecting features is very rich in the area of pattern recognition, neural networks, and machine learning. Before starting to learn a parametric or non-parametric model from the examples one must be sure that the input data (input features) have sufficient information to predict the outputs. This qualitative criterion can be made precise in a statistical way with the notion of *mutual information* (MI for short).

An output distribution is characterized by an *uncertainty* which can be measures from the probability distribution of the outputs. The theoretically sound way to measure the uncertainty is with the *entropy*, see below for the precise definition. Now, after one knows a specific input value x , the uncertainty in the output can decrease, depending on the form of the conditional distribution $p(x|y)$. The amount by which the uncertainty in the output decreases after the input is known is termed *mutual information*.

If the mutual information between a feature and the output is zero, knowledge of the input does not reduce the uncertainty in the output. In other words, the selected feature

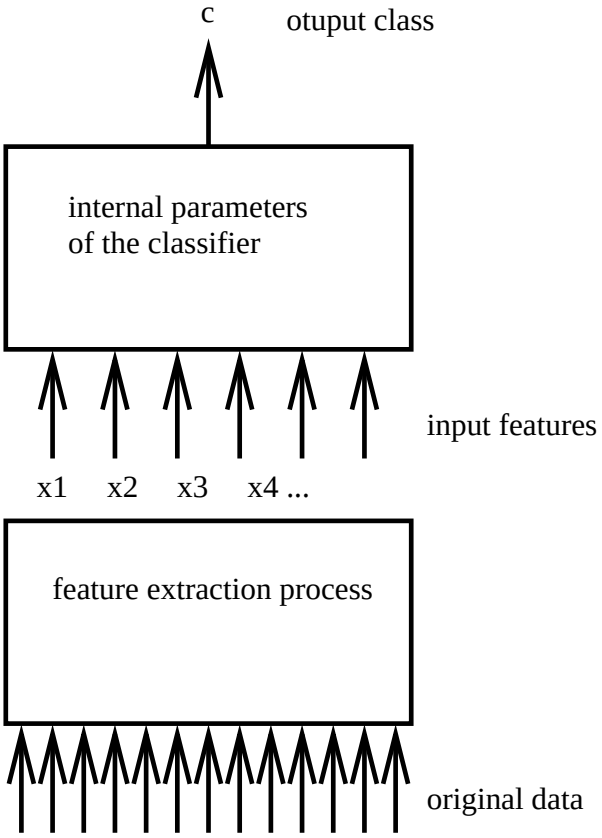


Figure 9.1: A classifier mapping input features extracted from the data to an output class.

cannot be used (in isolation) to predict the output - no matter how sophisticated our model. The MI measure between a vector of input features and the output (the desired prediction) is therefore very relevant to identify promising (informative) features. Its use in feature selection is pioneered in [1].

Let's now come to some definitions for the case of a classification task where the output variable c identifies one among N_c classes and the input variable $\mathbf{x}$ has a finite set of possible values, see Fig. 9.1. For example, one may think about predicting whether a run of an algorithm on an instance will converge (class 1) or not (class 0) within the next minute. Among the possible features extracted from the data one would like to obtain a highly-informative set, so that the classification problem starts from sufficient information, and only the actual construction of the classifier is left. One may ask at this point: why not using the raw data instead of features? For sure there is not better way to use all possible information. True, but the *curse of dimensionality* holds here: if the dimension of the input is too large, the learning task becomes unmanageable. Think for example about the difficulty of estimating probability distributions from samples in very large-dimensional spaces. Heuristically, one aims at a small subset of features, possibly close to the smallest possible, which contains sufficient information to predict the output.

If $P(c), c = 1, ..., N_c$, are the probabilities of the different output values, the initial uncertainty in the output class is measured by the entropy:

$$H(C) = -\sum_{c=1}^{N_c} P(c) \ln P(c) \tag{9.1}$$

The average uncertainty after knowing the feature vector $\mathbf{x}$ with n components is the *con-*

ditional entropy:

$$H(C|\mathbf{x}) = -\sum_{i=1}^{n} P(\mathbf{x}) \left(\sum_{c=1}^{N_c} P(c|\mathbf{x}) \ln P(c|\mathbf{x}) \right) \tag{9.2}$$

where $P(c|\mathbf{x})$ is the *conditional* probability of class c given input $\mathbf{x}$. The conditional entropy is always less-than or equal-to the initial entropy. It is equal if and only if the input features and the output class are statistically independent: the joint probability $P(c, \mathbf{x})$ is equal to $P(c)\ P(\mathbf{x})$. The amount by which the uncertainty decreases is by definition the **mutual information** $I(X; C)$ between variables $\mathbf{x}$ and c:

$$I(X; C) = I(C; X) = H(C) - H(C|X) \tag{9.3}$$

An equivalent expression which makes the symmetry between X and C evident is:

$$I(X; C) = \sum_{c,x} P(x, c) \ln \frac{P(c, f)}{P(c)\ P(f)} \tag{9.4}$$

Although very powerful theoretically, estimating the MI for a high-dimensional feature vector starting from labeled samples is not a trivial task. An heuristic method which uses only the MI between individual features and the output is presented in [1], using Fraser's algorithm in [9].

Let's note that the MI measure is different from the widely-used correlation measure. A feature can be informative even if not linearly correlated with the output.

The widely used measure of *linear relationship* is the Pearson product-moment **correlation coefficient**, which is obtained by dividing the covariance of the two variables by the product of their standard deviations. In detail, the correlation $\rho_{X,Y}$ between random variables X and Y with expected values μ_X and μ_Y and standard deviations σ_X and σ_Y is defined as:

$$\rho_{X,Y} = \frac{\operatorname{cov}(X, Y)}{\sigma_X \sigma_Y} = \frac{E((X - \mu_X)(Y - \mu_Y))}{\sigma_X \sigma_Y}, \tag{9.5}$$

where E is the expected value of the variable and cov is the covariance. After simple transformations one obtains the equivalent formula:

$$\rho_{X,Y} = \frac{E(XY) - E(X)E(Y)}{\sqrt{E(X^2) - E^2(X)}\ \sqrt{E(Y^2) - E^2(Y)}} \tag{9.6}$$

The correlation value is between 1 and -1. Correlation close to 1 means increasing linear relationship (an increase of X relative to the mean is usually accompanied by an increase of Y), close to -1 means a decreasing linear relationship. The closer the coefficient is to zero, the weaker the correlation between the variables. For example the plot of (X, Y) points looks like an isotropic cloud around the expected values, without an evident direction.

As mentioned before, statistically independent variables have zero correlation, but zero correlation does not imply that the variables are independent. The correlation coefficient detects only *linear* dependencies between two variables: it may well be that one variable has full information and actually determines the value of the second, as in the case that $y = f(X)$, while still having zero correlation. To make it short: trust correlation only if you have reasons to suspect *linear* relationships, use mutual information to estimate arbitrary dependencies!

9.2 Measuring local search components

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 some promising results will be finally 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", it can determine a winner in a fairer horse-race, but still 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 is capable of predicting 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. To make some examples, the work in [2] 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.

9.3 Selecting components based on diversification and bias

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 in [2] 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 metrics (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 relation of partial order (Sec. 9.3.2).
- The conjecture is validated by a competitive analysis of the components on a benchmark suite [2].

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:

$$X^{(t+1)} = \mu_{r(t)} X^{(t)} \quad \text{where} \quad r(t) = \textsc{Random}\ \{1, n\}$$

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:

$$H(X^{(t)}, X^{(0)}) = \sum_{i=1}^{n} x_i^{(t)}$$

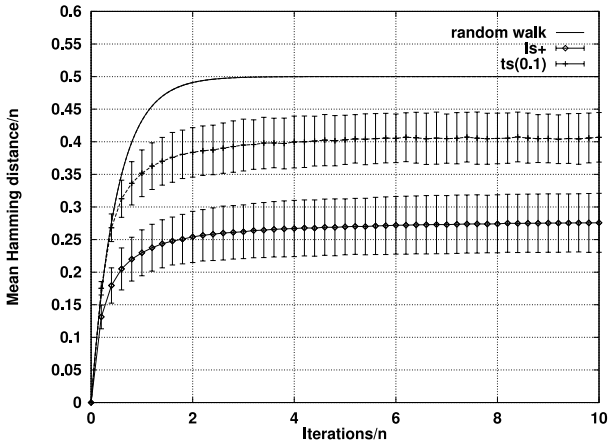


Figure 9.2: Average Hamming distance reached by Random Walk, LS$^+$ and TS(0.1) from the first local optimum of LS, with standard deviation (MAX-3-SAT). Random walk evolution is also reported for reference.

and therefore the expected value of the Hamming distance at time t, defined as $\widehat{H}^{(t)} = \widehat{H}(X^{(t)}, X^{(0)})$, is:

$$\widehat{H}^{(t)} = \sum_{i=1}^{n} \widehat{x_i}^{(t)} = n\, \widehat{x}^{(t)} \tag{9.7}$$

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:

$$\widehat{x}^{(t+1)} = \widehat{x}^{(t)}\,(1-p) \;+\; (1-\widehat{x}^{(t)})\,p = \widehat{x}^{(t)} \;+\; p\,(1-2\widehat{x}^{(t)}) \tag{9.8}$$

It is straightforward to derive the following theorem:

Theorem 1 *If $n > 2$ (and therefore $0 < p < \frac{1}{2}$) the difference equation 9.8 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:*

$$\widehat{x}^{(t)} \;=\; \frac{1-(1-2p)^t}{2} \tag{9.9}$$

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:

$$\widehat{H}^{(t)} \approx t \tag{9.10}$$

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 the 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. 9.2 shows the average **Hamming distance as a function of the additional iterations after reaching the LS optimum**, see [2] for experimental details.

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

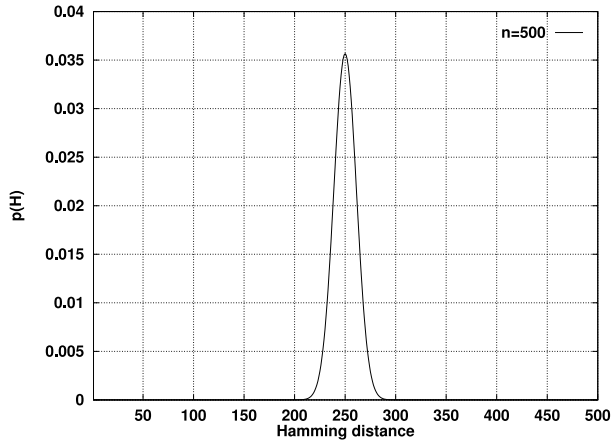


Figure 9.3: Probability of different Hamming distances for n = 500.

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

$$\left(\begin{array}{c} n \\ H \end{array} \right) \times \frac{1}{2^n} \tag{9.11}$$

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 [11]:

$$Pr[H \leq (1-\theta)pn] \leq e^{-\theta^2 np/2} \tag{9.12}$$

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. 9.3.

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 $n/2$.

9.3.1 The diversification-bias compromise (D-B plots)

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 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 $4n$

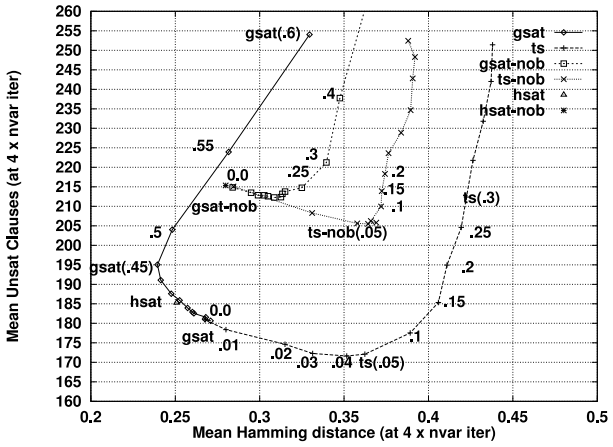


Figure 9.4: Diversification-bias plane. Mean number of unsatisfied clauses after $4\ n$ iterations versus mean Hamming distance. MAX-3-SAT tasks. Each run starts from GSAT local optimum, see [2].

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. 9.4. 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 Sec. 6.1.1. 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. 9.4. By considering the parametric curve for GSAT-with-walk(p) (label "gsat" in Fig. 9.4) 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 is that there are two sources of "randomness" in the GSAT-with-walk algorithm (see Fig. 6.2), 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 the results are collected in Fig. 9.5, where the average number of ties (number of moves achieving the largest Δf) is plotted as a function of p, with statistical error bars. The hypothesis is confirmed. 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. 9.4 is Fixed-TS(T_f), and the effect of a simple **aspiration criterion** [10], and a **tie-breaking rule** for it is studied in [2].

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.

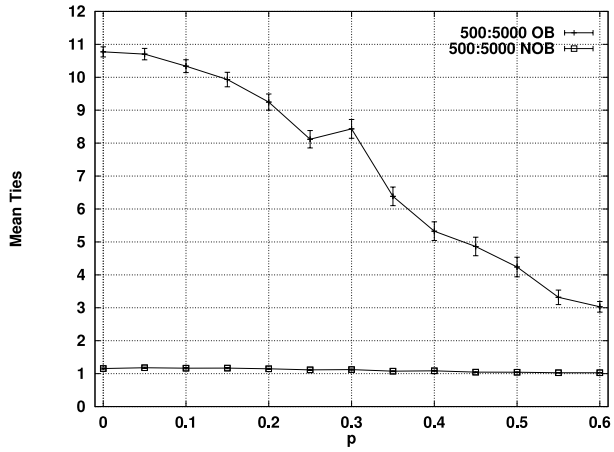


Figure 9.5: Mean number of ties as a function of the probability p in GSAT-with-walk(p), for the oblivious (OB) and non-oblivious (NOB) f. Values at $4\,n$ iterations.

9.3.2 A conjecture: better algorithms are Pareto-optimal in D-B plots

Aa conjecture about the relevance of the diversification-bias metric is proposed in [2]. A relation of partial order, denoted by the symbol $\geq$ and called "domination", in 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 $\widehat{H_{n\,A}} \geq \widehat{H_{n\,B}}$ *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 [2], with fully satisfactory results.

A definition of three metrics is used in [18] [22] 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.

Once the motivation for a *speed of coverage* measure is intuitively clear, the detailed definition and implementation is somewhat challenging. In [18] 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 it is not so fast, actually it can be NP-hard for problems with binary strings, and one has to resort to approximations [18]. 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 [22].

9.4 How to measure problem difficulty

While before we concentrated on understanding *why* an algorithm is better performing, here we consider the issue of understanding **why a problem is more difficult to solve** for a stochastic local search method. One aims at 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* leading to a more difficult exploration.

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 the function value f. Statistical mechanics has been very successful in the past at relating local and global behaviors of systems [12], 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 numbers are large, the variance in the behavior is very small so that most members of the ensemble will behave in a similar way. As an example, 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, the probability to observe 51% of the molecules in one container is very close to zero if the containers are filled with air at normal pressure: 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 (the molecule count is left as an exercise to the reader).

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 progressively 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. For example, an extensive review of models applied to constraint satisfaction problems, in particular the graph coloring problem, is present in [12]. The SAT problem, in particular the 3-SAT, has been the playground for many investigations, see for example [6], [7], [17], [20].

Phase-transitions have been identified as a mechanism to study and explain problem difficulty. A phase transition in a physical system 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 [8, 4]: as a function of the average node degree some macroscopic property like connectivity change in a very rapid manner. [14] 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. "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."

Constraint satisfaction and SAT phase transitions have been widely analyzed, for a few references see [5] [16] [19] [15] [21] [17]. A clear introduction to phase transitions and the search problem is present in [13]. A surprising result is that **hard problem instances are concentrated near the same parameter values for a wide variety of common search heuristics**, on average. This location also corresponds to a **transition between solvable and unsolvable instances**. For example, a complete backtracking algorithm on the solution tree and local search show very long computing times for SAT problems in the same transition region when more clauses are added to the instances.

For backtracking, this is due to a competition between two factors: i) number of solutions and ii) 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 with no solution. The **critically constrained** instances in between are the hardest ones.

For local search one has to be careful. The method is not complete and one must limit the experimentation to solvable instances. One may naively 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. Intuition helps, for a growing body of experimental research see for example [6] for results on CSP and SAT, [7] for experimental 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 very 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 for NP-hard problems (let's remember that the computational complexity classes are defined through a *worst-case* analysis), see for example [19] for generating hard Satisfiability problems.

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.

The work [6] demonstrates that the logarithm of the number of optimal solutions accounts for a large portion of the variability in local search cost for the SAT problem. The papers [17] and [20] study the distribution of SAT solutions and demonstrate that the size of the *backbone* (the set of Boolean variables that have the same value in all optimal solutions) is positively correlated to the solution cost. The contribution [23] considers the Job Shop Scheduling problem (JSP) and demonstrates experimentally that the mean distance between random local minima and the nearest optimal solution is highly correlated with the cost of solving the problem to optimality (a simple version of tabu search is used in the tests).

Big-Valley models [3] (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 *auto-correlation of the time-series of f values produced by a random walk*. The autocorrelation function of a random process (ACF) describes the correlation between the process at different points in time. Let X^t be the search configuration at time t. If X^t has mean μ and variance σ^2 then the ACF is

$$R(t,s) = \frac{E[(X_t - \mu)(X_s - \mu)]}{\sigma^2} \tag{9.13}$$

The *correlation length* is a measure derived from the ACF of the range over which fluctuations of f in one region of space are correlated with those in another region.

Unfortunately, for many problems the correlation length is a function of problem size and it does not explain the variance in computational cost among instances of the same size [23].

Bibliography

[1] R. Battiti, *Using the mutual information for selecting features in supervised neural net learning*, IEEE Transactions on Neural Networks **5** (1994), no. 4, 537–550.

[2] R. Battiti and M. Protasi, *Reactive search, a history-sensitive heuristic for MAX-SAT*, ACM Journal of Experimental Algorithmics **2** (1997), no. ARTICLE 2, http://www.jea.acm.org/.

[3] KD Boese, AB Kahng, and S. Muddu, *On the big valley and adaptive multi-start for discrete global optimizations*, Operation Research Letters **16** (1994), no. 2.

[4] Bollobás, *Random Graphs*, Cambridge University Press, 2001.

[5] P. Cheeseman, B. Kanefsky, and W.M. Taylor, *Where the really hard problems are*, Proceedings of the 12th IJCAI (1991), 331–337.

[6] David A. Clark, Jeremy Frank, Ian P. Gent, Ewan MacIntyre, Neven Tomov, and Toby Walsh, *Local search and the number of solutions*, Principles and Practice of Constraint Programming, 1996, pp. 119–133.

[7] James M. Crawford and Larry D. Auton, *Experimental results on the crossover point in random 3-sat*, Artif. Intell. **81** (1996), no. 1-2, 31–57.

[8] P. Erdos and A. Renyi, *On random graphs*, Publ. Math. Debrecen **6** (1959), 290–297.

[9] Andrew M. Fraser and Harry L. Swinney, *Independent coordinates for strange attractors from mutual information*, Phys. Rev. A **33** (1986), no. 2, 1134–1140.

[10] F. Glover, *Tabu search - part i*, ORSA Journal on Computing **1** (1989), no. 3, 190–260.

[11] T. Hagerup and C. Rueb, *A guided tour of chernoff bounds*, Information Processing Letters **33** (1989/90), 305–308.

[12] Tad Hogg, *Applications of statistical mechanics to combinatorial search problems*, vol. 2, pp. 357–406, World Scientific, Singapore, 1995.

[13] Tad Hogg, Bernardo A. Huberman, and Colin P. Williams, *Phase transitions and the search problem*, Artif. Intell. **81** (1996), no. 1-2, 1–15.

[14] B.A. Huberman and T. Hogg, *Phase transitions in artificial intelligence systems*, Artificial Intelligence **33** (1987), no. 2, 155–171.

[15] Scott Kirkpatrick and Bart Selman, *Critical behavior in the satisfiability of random boolean expressions*, Science **264** (1994), 1297–1301.

[16] D. Mitchell, B. Selman, and H. Levesque, *Hard and easy distributions of SAT problems*, Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92) (San Jose, Ca), July 1992, pp. 459–465.

[17] Andrew J. Parkes, *Clustering at the phase transition*, AAAI/IAAI, 1997, pp. 340–345.

[18] Dale Schuurmans and Finnegan Southey, *Local search characteristics of incomplete sat procedures*, Artif. Intell. **132** (2001), no. 2, 121–150.

[19] Bart Selman, David G. Mitchell, and Hector J. Levesque, *Generating hard satisfiability problems*, Artif. Intell. **81** (1996), no. 1-2, 17–29.

[20] Josh Singer, Ian Gent, and Alan Smaill, *Backbone Fragility and the Local Search Cost Peak*, Journal of Artificial Intelligence Research **12** (2000), 235–270.

[21] B.M. Smith, *Phase transition and the mushy region in constraint satisfaction problems*, Proceedings of the 11th European Conference on Artificial Intelligence (1994), 100–104.

[22] Finnegan Southey, *Theory and applications of satisfiability testing*, ch. Constraint Metrics for Local Search, pp. 269–281, Springer Verlag, 2005.

[23] 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** (2003), no. 2, 189–217.