5. Escaping Local Optima

O! for a horse with wings!

Shakespeare, *Cymbeline*

We've discussed a few traditional problem-solving strategies. Some of them guarantee finding the global solution, others don't, but they all share a common pattern. Either they guarantee discovering the global solution, but are too expensive (i.e., too time consuming) for solving typical real-world problems, or else they have a tendency of "getting stuck" in local optima. Since there is almost no chance to speed up algorithms that guarantee finding the global solution, i.e., there is almost no chance of finding polynomial-time algorithms for most real problems (as they tend to be NP-hard), the other remaining option aims at designing algorithms that are capable of escaping local optima.

How can we do this? How can we design "a horse with wings"? We have already seen one possibility in chapter 2, where we discussed a procedure called an "iterated hill-climber." After reaching a local optimum, a new starting point is generated and the search is commenced all over again. Clearly, we can apply such a strategy with any other algorithm, but let's discuss some possibilities of escaping local optima within a single run of an algorithm.

Two main approaches that we'll discuss in this chapter are based on (1) an additional parameter (called *temperature*) that changes the probability of moving from one point of the search space to another, or (2) a memory, which forces the algorithm to explore new areas of the search space. These two approaches are called *simulated annealing* and *tabu search*, respectively. We discuss them in detail in the following two sections, but first let's provide some intuition connected with these methods and compare them to a simple local search routine.

We discussed the iterated hill-climber procedure in chapter 2. Let's now rewrite this procedure by omitting several lines of programming details and by listing only the most significant concepts. Also, let's restrict the algorithm to a single sequence of improvement iterations (i.e., $MAX = 1$, see figure 2.5). In such a case, a single run of a simple *local search* procedure can be described as follows:

```
procedure local search
begin
   $x$ = some initial starting point in $\mathcal{S}$
   while improve($x$) $\neq$ 'no' do
      $x$ = improve($x$)
   return($x$)
end
```

The subprocedure improve(x) returns a new point y from the neighborhood of x, i.e., $y \in N(x)$, if y is better than x, otherwise it returns a string "no," and in that case, x is a local optimum in $\mathcal{S}$.

In contrast, the procedure *simulated annealing* (discussed fully in section 5.1) can be described as follows:

```
procedure simulated annealing
begin
   $x$ = some initial starting point in $\mathcal{S}$
   while not termination-condition do
      $x$ = improve?($x$, $T$)
      update($T$)
   return($x$)
end
```

There are three important differences between simulated annealing and local search. First, there is a difference in how the procedures halt. Simulated annealing is executed until some external "termination condition" is satisfied as opposed to the requirement of local search to find an improvement. Second, the function "improve?(x, T)" doesn't have to return a better point from the neighborhood of x.[1] It just returns an *accepted* solution y from the neighborhood of x, where the acceptance is based on the current temperature T. Third, in simulated annealing, the parameter T is updated periodically, and the value of this parameter influences the outcome of the procedure "improve?" This feature doesn't appear in local search. Note also that the above sketch of simulated annealing is quite simplified so as to match the simplifications we made for local search. For example, we omitted the initialization process and the frequency of changing the temperature parameter T. Again, the main point here is to underline similarities and differences.

Tabu search (discussed fully in section 5.2) is almost identical to simulated annealing with respect to the structure of the algorithm. As in simulated annealing, the function "improve?(x, H)" returns an *accepted* solution y from the neighborhood of x, which need not be better than x, but the acceptance is based on the history of the search H. Everything else is the same (at least, from a high-level perspective). The procedure can be described as follows:

[1]This is why we added a question mark in the name of this procedure: "improve?".

```
procedure tabu search
begin
   $x$ = some initial starting point in $\mathcal{S}$
   while not termination-condition do
      $x$ = improve?($x$, $H$)
      update($H$)
   return($x$)
end
```

With these intuitions in mind and an understanding of the basic concepts, we're ready to discuss the details of these two interesting search methods.

5.1 Simulated annealing

Let's recall the detailed structure of the iterated hill-climber (chapter 2) because simulated annealing doesn't differ from it very much.

```
procedure iterated hill-climber
begin
   $t \leftarrow 0$
   initialize $best$
   repeat
      $local \leftarrow$ FALSE
      select a current point $\mathbf{v}_c$ at random
      evaluate $\mathbf{v}_c$
      repeat
         select all new points in the neighborhood of $\mathbf{v}_c$
         select the point $\mathbf{v}_n$ from the set of new points
            with the best value of evaluation function $eval$
         if $eval(\mathbf{v}_n)$ is better than $eval(\mathbf{v}_c)$
            then $\mathbf{v}_c \leftarrow \mathbf{v}_n$
            else $local \leftarrow$ TRUE
      until $local$
      $t \leftarrow t + 1$
      if $\mathbf{v}_c$ is better than $best$
         then $best \leftarrow \mathbf{v}_c$
   until $t = MAX$
end
```

Fig. 5.1. Structure of an iterated hill-climber

Note again that the inner loop of the procedure (figure 5.1) *always* returns a local optimum. This procedure only "escapes" local optima by starting a new search (outer loop) from a new (random) location. There are MAX attempts altogether. The best solution overall (*best*) is returned as the final outcome of the algorithm.

Let's modify this procedure in the following way:

- Instead of checking *all* of the strings in the neighborhood of a current point $\mathbf{v}_c$ and selecting the best one, select only one point, $\mathbf{v}_n$, from this neighborhood.
- Accept this new point, i.e., $\mathbf{v}_c \leftarrow \mathbf{v}_n$, with some probability that depends on the relative merit of these two points, i.e., the difference between the values returned by the evaluation function for these two points.

By making such a simple modification, we obtain a new algorithm, the so-called *stochastic hill-climber* (figure 5.2):

```
procedure stochastic hill-climber
begin
  $t \leftarrow 0$
  select a current string $\mathbf{v}_c$ at random
  evaluate $\mathbf{v}_c$
  repeat
    select the string $\mathbf{v}_n$ from the neighborhood of $\mathbf{v}_c$
    select $\mathbf{v}_n$ with probability $\frac{1}{1+e^{\frac{eval(\mathbf{v}_c)-eval(\mathbf{v}_n)}{T}}}$
    $t \leftarrow t+1$
  until $t = MAX$
end
```

Fig. 5.2. Structure of a stochastic hill-climber

Let's discuss some features of this algorithm (note that the probabilistic formula for accepting a new solution is based on maximizing the evaluation function). First, the stochastic hill-climber has only one loop. We don't have to repeat its iterations starting from different random points. Second, the newly selected point is "accepted" with some probability p. This means that the rule of moving from the current point $\mathbf{v}_c$ to the new neighbor, $\mathbf{v}_n$, is probabilistic. It's possible for the new accepted point to be *worse* than the current point. Note, however, that the probability of acceptance depends on the difference in merit between these two competitors, i.e., $eval(\mathbf{v}_c) - eval(\mathbf{v}_n)$, and on the value of an additional parameter T. Also, T remains constant during the execution of the algorithm. Looking at the probability of acceptance:

$$p = \frac{1}{1+e^{\frac{eval(\mathbf{v}_c)-eval(\mathbf{v}_n)}{T}}},$$

what is the role of the parameter T? Assume, for example, that the evaluation for the current and next points are $eval(\mathbf{v}_c) = 107$ and $eval(\mathbf{v}_n) = 120$ (remember, the formula is for maximization problems only; if you are minimizing, you have to reverse the minuend and subtrahend of the subtraction operation). The difference in our example is $eval(\mathbf{v}_c) - eval(\mathbf{v}_n)$ or -13, meaning that the new point $\mathbf{v}_n$ is better than $\mathbf{v}_c$. What is the probability of accepting this new point based on different values of T? Table 5.1 details some individual cases.

Table 5.1. Probability p of acceptance as a function of T for the case where the new trial $\mathbf{v}_n$ is 13 points better than the current solution $\mathbf{v}_c$

T	$e^{\frac{-13}{T}}$	p
1	0.000002	1.00
5	0.0743	0.93
10	0.2725	0.78
20	0.52	0.66
50	0.77	0.56
10^{10}	0.9999...	0.5...

The conclusion is clear: the greater the value of T, the smaller the importance of the relative merit of the competing points $\mathbf{v}_c$ and $\mathbf{v}_n$! In particular, if T is huge (e.g., $T = 10^{10}$), the probability of acceptance approaches 0.5. The search becomes random. On the other hand, if T is very small (e.g., $T = 1$), the stochastic hill-climber reverts into an ordinary hill-climber! Thus, we have to find an appropriate value of the parameter T for a particular problem: not too low and not too high.

To gain some additional perspective, suppose that $T = 10$ for a given run. Let's also say that the current solution $\mathbf{v}_c$ evaluated to 107, i.e., $eval(\mathbf{v}_c) = 107$. Then, the probability of acceptance depends only on the value of the new string $\mathbf{v}_n$ as shown in table 5.2.

Table 5.2. Probability of acceptance as a function of $eval(\mathbf{v}_n)$ for $T = 10$ and $eval(\mathbf{v}_c) = 107$

$eval(\mathbf{v}_n)$	$eval(\mathbf{v}_c) - eval(\mathbf{v}_n)$	$e^{\frac{eval(\mathbf{v}_c)-eval(\mathbf{v}_n)}{10}}$	p
80	27	14.88	0.06
100	7	2.01	0.33
107	0	1.00	0.50
120	−13	0.27	0.78
150	−43	0.01	0.99

Now the picture is complete: if the new point has the same merit as the current point, i.e., $eval(\mathbf{v}_c) = eval(\mathbf{v}_n)$, the probability of acceptance is 0.5. That's reasonable. It doesn't matter which you choose because each are of equal quality. Furthermore, if the new point is better, the probability of acceptance is greater than 0.5. Moreover, the probability of acceptance grows together with the (negative) difference between these evaluations. In particular, if the new solution is *much* better than the current one (say, $eval(\mathbf{v}_n) = 150$), the probability of acceptance is close to 1.

The main difference between the stochastic hill-climber and simulated annealing is that the latter changes the parameter T during the run. It starts with high values of T making the procedure more similar to a purely random search, and then gradually decreases the value of T. Towards the end of the run, the values of T are quite small so the final stages of simulated annealing merely resemble an ordinary hill-climber. In addition, we always accept new points if they are better than the current point. The procedure *simulated annealing* is given in figure 5.3 (again, we've assumed a maximization problem).

```
procedure simulated annealing
begin
  $t \leftarrow 0$
  initialize $T$
  select a current point $\mathbf{v}_c$ at random
  evaluate $\mathbf{v}_c$
  repeat
    repeat
      select a new point $\mathbf{v}_n$
        in the neighborhood of $\mathbf{v}_c$
      if $eval(\mathbf{v}_c) < eval(\mathbf{v}_n)$
        then $\mathbf{v}_c \leftarrow \mathbf{v}_n$
        else if $random[0,1) < e^{\frac{eval(\mathbf{v}_n)-eval(\mathbf{v}_c)}{T}}$
          then $\mathbf{v}_c \leftarrow \mathbf{v}_n$
    until (termination-condition)
    $T \leftarrow g(T,t)$
    $t \leftarrow t+1$
  until (halting-criterion)
end
```

Fig. 5.3. Structure of simulated annealing

Simulated annealing — also known as Monte Carlo annealing, statistical cooling, probabilistic hill-climbing, stochastic relaxation, and probabilistic exchange algorithm — is based on an analogy taken from thermodynamics. To grow a crystal, you start by heating a row of materials to a molten state. You then reduce the temperature of this *crystal melt* until the crystal structure is *frozen in*. Bad things happen if the cooling is done too quickly. In particular, some irregularities are locked into the crystal structure and the trapped energy level is much higher than in a perfectly structured crystal.[2] The analogy between the physical system and an optimization problem should be evident. The basic "equivalent" concepts are listed in table 5.3.

[2]A similar problem occurs in metallurgy when heating and cooling metals.

Table 5.3. Analogies between a physical system and an optimization problem

Physical System	Optimization Problem
state	feasible solution
energy	evaluation function
ground state	optimal solution
rapid quenching	local search
temperature	control parameter T
careful annealing	simulated annealing

As with any search algorithm, simulated annealing requires the answers for the following problem-specific questions (see chapter 2):

- What is a solution?
- What are the neighbors of a solution?
- What is the cost of a solution?
- How do we determine the initial solution?

These answers yield the structure of the search space together with the definition of a neighborhood, the evaluation function, and the initial starting point. Note, however, that simulated annealing also requires answers for additional questions:

- How do we determine the initial "temperature" T?
- How do we determine the cooling ratio $g(T, t)$?
- How do we determine the termination condition?
- How do we determine the halting criterion?

The temperature T must be initialized before executing the procedure. Should we start with $T = 100$, $T = 1000$, or something else? How should we choose the termination condition after which the temperature is decreased and the annealing procedure reiterates? Should we execute some number of iterations or should we instead use some other criterion? Then, how much or by what factor should the temperature be decreased? By one percent or less? And finally, when should the algorithm halt, i.e., what is the "frozen" temperature?

Most implementations of simulated annealing follow a simple sequence of steps:

```
STEP 1: $T \leftarrow T_{max}$
        select $\mathbf{v}_c$ at random

STEP 2: pick a point $\mathbf{v}_n$ from the neighborhood of $\mathbf{v}_c$
        if $eval(\mathbf{v}_n)$ is better than $eval(\mathbf{v}_c)$
            then select it ($\mathbf{v}_c \leftarrow \mathbf{v}_n$)
            else select it with probability $e^{\frac{-\Delta eval}{T}}$
        repeat this step $k_T$ times

STEP 3: set $T \leftarrow rT$
        if $T \geq T_{min}$
            then goto STEP 2
            else goto STEP 1
```

Here we have to set the values of the parameters T_{max}, k_T, r, and T_{min}, which correspond to the initial temperature, the number of iterations, the cooling ratio, and the frozen temperature, respectively. Let's examine some possibilities by applying the simulated annealing technique to the SAT, TSP, and NLP, respectively.

Spears [444] applied simulated annealing (SA) on hard satisfiability problems. The procedure SA-SAT described in [444] is displayed in figure 5.4. Its control structure is similar to that of the GSAT (chapter 3). The outermost loop variable called "tries" of SA-SAT corresponds to the variable i of GSAT (see figure 3.5 from chapter 3). These variables keep track of the number of independent attempts to solve the problem. T is set to T_{max} at the beginning of each attempt in SA-SAT (by setting the variable j to zero) and a new random truth assignment is made. The inner repeat loop tries different assignments by probabilistically flipping each of the Boolean variables. The probability of a flip depends on the improvement δ of the flip and the current temperature. As usual, if the improvement is negative, the flip is quite unlikely to be accepted and vice versa: positive improvements make accepting flips likely.

There's a major difference between GSAT and SA-SAT, and this is the essential difference between any local search technique and simulated annealing: GSAT can make a backward move (i.e., a decrease in the number of satisfied clauses) if other moves are not available. At the same time, GSAT cannot make two backward moves in a row, as one backward move implies the existence of the next improvement move! On the other hand, SA-SAT can make an arbitrary sequence of backward moves, thus it can escape local optima!

The remaining parameters of SA-SAT have the same meaning as in other implementations of simulated annealing. The parameter r represents a decay rate for the temperature, i.e., the rate at which the temperature drops from T_{max} to T_{min}. The drop is caused by incrementing j, as

$$T = T_{max} \cdot e^{-j \cdot r}.$$

Spears [444] used

```
procedure SA-SAT
begin
   tries ← 0
   repeat
      v ← random truth assignment
      $j \leftarrow 0$
      repeat
         if v satisfies the clauses then return v
         $T = T_{max} \cdot e^{-j \cdot r}$
         for $k = 1$ to the number of variables do
         begin
            compute the increase (decrease) $\delta$ in the
               number of clauses made true if $v_k$ was flipped
            flip variable $v_k$ with probability $(1 + e^{-\frac{\delta}{T}})^{-1}$
            v ← new assignment if the flip is made
         end
         $j \leftarrow j + 1$
      until $T < T_{min}$
      tries ← tries +1
   until tries = MAX-TRIES
end
```

Fig. 5.4. Structure of the SA-SAT algorithm

$$T_{max} = 0.30 \text{ and } T_{min} = 0.01$$

for the maximum and minimum temperatures, respectively. The decay rate r depended on the number of variables in the problem and the number of the "try" (r was the inverse of the product of the number of variables and the number of the try). Spears commented on the choice of these parameters:

> Clearly these choices in parameters will entail certain tradeoffs. For a given setting of MAX-TRIES, reducing T_{min} and/or increasing T_{max} will allow more tries to be made per independent attempt, thus decreasing the number of times that 'tries' can be incremented before the MAX-TRIES cutoff is reached. A similar situation occurs if we decrease or increase the decay rate. Thus, by increasing the temperature range (or decreasing the decay rate) we reduce the number of independent attempts, but search more thoroughly during each attempt. The situation is reversed if one decreases the temperature range (or increases the decay rate). Unfortunately it is not at all clear whether it is generally better to make more independent attempts, or to search more thoroughly during each attempt.

The experimental comparison between GSAT and SA-SAT on hard satisfiability problems indicated that SA-SAT appeared to satisfy at least as many formulas as GSAT, with less work [444]. Spears also presented experimental evidence that the relative advantage of SA-SAT came from its backward moves, which helped escape local optima.

It's interesting to note that the TSP was one of the very first problems to which simulated annealing was applied! Moreover, many new variants of simulated annealing have also been tested on the TSP. A standard simulated annealing algorithm for the TSP is essentially the same as the one displayed in figure 5.3, where $\mathbf{v}_c$ is a tour and $eval(\mathbf{v}_c)$ is the length of the tour $\mathbf{v}_c$. The differences between implementations of simulated annealing are in (1) the methods of generating the initial solution, (2) the definition of a neighborhood of a given tour, (3) the selection of a neighbor, (4) the methods for decreasing temperature, (5) the termination condition, (6) the halting condition, and (7) the existence of a postprocessing phase.

All the above decisions are important and far from straightforward. They are also not independent. For example, the number of steps at each temperature should be proportional to the neighborhood size. For each of these facets there are many possibilities. We might start from a random solution, or instead take an output from a local search algorithm as the initial tour. We can define neighborhoods of various sizes and include a postprocessing phase where a local search algorithm climbs a local peak (as simulated annealing doesn't guarantee that this will happen for all schedules for reducing T over time).

For more details on various possibilities of applying simulated annealing to the TSP, with an excellent discussion on techniques for accelerating the algorithm and other improvements, see [242].

Simulated annealing can also easily be applied to the NLP. Since we are now concerned with continuous variables, the neighborhood is often defined on the basis of a Gaussian distribution (for each variable), where the mean is kept at the current point and the standard deviation is set to one-sixth of the length of the variable's domain (so that the length from the midpoint of the domain to each boundary equals three standard deviations). Thus, if the current point is

$$\mathbf{x} = (x_1, \dots, x_n),$$

where $l_i \le x_i \le u_i$ for $i = 1, \dots, n$, then the neighbor $\mathbf{x}'$ of $\mathbf{x}$ is selected as

$$x_i' \leftarrow x_i + N(0, \sigma_i),$$

where $\sigma_i = (u_i - l_i)/6$ and $N(0, \sigma_i)$ is an independent random Gaussian number with mean of zero and standard deviation σ_i.

At this stage the change in the function value can be calculated:

$$\Delta eval = eval(\mathbf{x}) - eval(\mathbf{x}').$$

If $\Delta eval > 0$ (assume we face a minimization problem), the new point $\mathbf{x}'$ is accepted as a new solution; otherwise, it is accepted with probability

$$e^{\Delta eval / T}.$$

The parameter T again indicates the current temperature.

In the area of numerical optimization, the issues of generating the initial solution, defining the neighborhood of a given point, and selecting particular neighbors are straightforward. The usual procedure employs a random start and Gaussian distributions for neighborhoods. But implementations differ in the methods for decreasing temperature, the termination condition, the halting condition, and the existence of a postprocessing phase (e.g., where we might include a gradient-based method that would locate the local optimum quickly). Note that continuous domains also provide for an additional flexibility: the size of the neighborhood can decrease together with the temperature. If parameters σ_i decrease over time, the search concentrates around the current point resulting in better fine tuning.

5.2 Tabu search

The main idea behind tabu search is very simple. A "memory" forces the search to explore new areas of the search space. We can memorize some solutions that have been examined recently and these become tabu (forbidden) points to be avoided in making decisions about selecting the next solution. Note that tabu search is basically deterministic (as opposed to simulated annealing), but it's possible to add some probabilistic elements to it [186].

The best way to explain the basic concepts of tabu search is by means of an example. We'll start with tabu search applied to the SAT. Later we'll provide a general discussion on some interesting aspects of this method and we'll conclude by providing an additional example of this technique regarding the TSP.

Suppose we're solving the SAT problem with $n = 8$ variables. Thus, for a given logical formula F we're searching for a truth assignment for all eight variables such that the whole formula F evaluates to TRUE. Assume further that we have generated an initial assignment for $\mathbf{x} = (x_1, \ldots, x_8)$; namely,

$$\mathbf{x} = (0, 1, 1, 1, 0, 0, 0, 1).$$

As usual, we need some evaluation function that provides feedback for the search. For example, we might calculate a weighted sum of a number of satisfied clauses, where the weights depend on the number of variables in the clause. In this case, the evaluation function should be maximized (i.e., we're trying to satisfy all of the clauses), and let's assume that the above random assignment provides the value of 27. Next, we have to examine the neighborhood of $\mathbf{x}$, which consists of eight other solutions, each of which can be obtained by flipping a single bit in the vector $\mathbf{x}$. We evaluate them and select the best one. At this stage of the search, this is the same as in a hill-climbing procedure.

Suppose that flipping the third variable generates the best evaluation (say, 31), so this new vector yields the current best solution. Now the time has come to introduce the new facet of tabu search: a memory. In order to keep a record of our actions (moves), we'll need some memory structures for bookkeeping.

We'll remember the index of a variable that was flipped, as well as the "time" when this flip was made, so that we can differentiate between older and more recent flips. In the case of the SAT problem, we need to keep a time stamp for each position of the solution vector: the value of the time stamp will provide information on the recency of the flip at this particular position. Thus, a vector M will serve as our memory. This vector is initialized to 0 and then at any stage of the search, the entry

$$M(i) = j \text{ (when } j \neq 0)$$

might be interpreted as "j is the most recent iteration when the i-th bit was flipped" (of course, $j = 0$ implies that the i-th bit has never been flipped). It might be useful to change this interpretation so we can model an additional aspect of memory: After some period of time (i.e., after some number of iterations), the information stored in memory is erased. Assuming that any piece of information can stay in a memory for at most, say, five iterations, a new interpretation of an entry

$$M(i) = j \text{ (when } j \neq 0)$$

could be: "the i-th bit was flipped $5-j$ iterations ago." Under this interpretation, the contents of the memory structure M after one iteration in our example is given in figure 5.5. Recall that the flip on the third position gave the best result. Note that the value "5" can be interpreted as "for the next five iterations, the third bit position is not available (i.e., tabu)."

0	0	5	0	0	0	0	0

Fig. 5.5. The contents of the memory after iteration 1

It might be interesting to point out that the main difference between these two equivalent interpretations is simply a matter of implementation. The latter approach interprets the values as the number of iterations for which a given position is not available for any flips. This interpretation requires that *all* nonzero entries of the memory are decreased by one at every iteration to facilitate the process of forgetting after five iterations. On the other hand, the former interpretation simply stores the iteration number of the most recent flip at a particular position, so it requires a current iteration counter t which is compared with the memory values. In particular, if $t - M(i) > 5$, i.e., the flip on the i-th position occurred earlier than five iterations ago (if at all), it should be forgotten. This interpretation, therefore, only requires updating a *single* entry in the memory per iteration, and increasing the iteration counter. Later on we'll assume the latter interpretation in an example, but most *implementations* of tabu search use the former interpretation for efficiency.

Let's say that after four additional iterations of selecting the best neighbor — which isn't necessarily better than the current point, and this is why we can escape from local optima — and making an appropriate flip, the memory has the contents as shown in figure 5.6.

3	0	1	5	0	4	2	0

Fig. 5.6. The contents of the memory after five iterations

The numbers present in the memory (figure 5.6) provide the following information:

> Bits 2, 5, and 8 are available to be flipped any time. Bit 1 is not available for the next three iterations, bit 3 isn't available but only for the next iteration, bit 4 (which was just flipped) is not available for the next five iterations, and so on.

In other words, the most recent flip (iteration 5) took place at position 4 (i.e., $M(4) = 5$: bit 4 was flipped $5-5 = 0$ iterations ago). Then, the other previously flipped bits are:

bit 6 (at iteration 4), (as $M(6) = 4$),
bit 1 (at iteration 3), (as $M(1) = 3$),
bit 7 (at iteration 2), (as $M(7) = 2$) and, of course,
bit 3 (at iteration 1).

By consequence, the current solution is $\mathbf{x} = (1,1,0,0,0,1,1,1)$ and let's say that it evaluates out at 33. Now, let's examine the neighborhood of $\mathbf{x}$ carefully. It consists of eight solutions,

$\mathbf{x}_1 = (0,1,0,0,0,1,1,1)$,
$\mathbf{x}_2 = (1,0,0,0,0,1,1,1)$,
$\mathbf{x}_3 = (1,1,1,0,0,1,1,1)$,
$\mathbf{x}_4 = (1,1,0,1,0,1,1,1)$,
$\mathbf{x}_5 = (1,1,0,0,1,1,1,1)$,
$\mathbf{x}_6 = (1,1,0,0,0,0,1,1)$,
$\mathbf{x}_7 = (1,1,0,0,0,1,0,1)$, and
$\mathbf{x}_8 = (1,1,0,0,0,1,1,0)$,

which correspond to flips on bits 1 to 8, respectively. After evaluating each we know their respective merits, but tabu search utilizes the memory to force the search to explore new areas of the search space. The memorized flips that have been made recently are tabu (forbidden) for selecting the next solution. Thus at the next iteration (iteration 6), it's impossible to flip bits 1, 3, 4, 6, and 7, since all of these bits were tried "recently." These forbidden (tabu) solutions (i.e., the solutions that are obtained from the forbidden flips) are not considered, so the next solution is selected from a set of $\mathbf{x}_2$, $\mathbf{x}_5$, and $\mathbf{x}_8$.

Suppose that the best evaluation of these three possibilities is given by $\mathbf{x}_5$ which is 32. Note that this value represents a decrease in the evaluation between the current best solution and the new candidate. After this iteration the contents of the memory change as follows: All nonzero values are decreased by one to reflect the fact that all of the recorded flips took place one generation earlier. In particular, the value $M(3) = 1$ is changed to $M(3) = 0$; i.e., after five generations the fact that the third bit was flipped is removed from the memory. Also, since the selected new current solution resulted from flipping the fifth bit, the value of $M(5)$ is changed from zero to five (for the next five iterations, this position is tabu). Thus, after the sixth iteration the contents of memory now appear as shown in figure 5.7.

2	0	0	4	5	3	1	0

Fig. 5.7. The contents of the memory after six iterations

Further iterations are similarly executed. At any stage, there is a current solution being processed which implies a neighborhood, and from this neighborhood, tabu solutions are eliminated from possible exploration.

Upon reflection, such a policy might be too restrictive. It might happen that one of the tabu neighbors, say $\mathbf{x}_6$, provides an excellent evaluation score. One that's much better than the score of any solution considered previously. Perhaps we should make the search more flexible. If we find an outstanding solution, we might forget about the principles![3] In order to make the search more flexible, tabu search considers solutions from the *whole* neighborhood, evaluates them all, and under "normal" circumstances selects a non-tabu solution as the next current solution, whether or not this non-tabu solution has a better evaluation score than the current solution. But in circumstances that aren't "normal," i.e., an outstanding tabu solution is found in the neighborhood, such a superior solution is taken as the next point. This override of the tabu classification occurs when a so-called *aspiration criterion* is met.

Of course there are also other possibilities for increasing the flexibility of the search. For example, we could change the previous deterministic selection procedure into a probabilistic method where better solutions have an increased chance of being selected. In addition, we could change the memory horizon during the search: sometimes it might be worthwhile to remember "more," and at other times to remember "less" (e.g., when the algorithm hill-climbs a promising area of the search space). We might also connect this memory horizon to the size of the problem (e.g., remembering the last $\sqrt{n}$ moves, where n provides a size measure of the instance of the problem).

Another option is even more interesting! The memory structure discussed so far can be labeled as *recency-based* memory, as it only records some ac-

[3]This is quite typical in various situations in ordinary life. When seeing a great opportunity it's easy to forget about some principles!

tions of the last few iterations. This structure might be extended by a so-called *frequency-based* memory, which operates over a much longer horizon. For example (referring back to the SAT problem considered earlier), a vector H may serve as a long-term memory. This vector is initialized to 0 and at any stage of the search the entry

$$H(i) = j$$

is interpreted as "during the last h iterations of the algorithm, the i-th bit was flipped j times." Usually, the value of the horizon h is quite large, at least in comparison with the horizon of recency-based memory. Thus after, say 100 iterations with $h = 50$, the long-term memory H might have the values displayed in figure 5.8.

5	7	11	3	9	8	1	6

Fig. 5.8. The contents of the frequency-based memory after 100 iterations (horizon $h = 50$)

These frequency counts show the distribution of moves throughout the last 50 iterations. How can we use this information? The principles of tabu search indicate that this type of memory might be useful to *diversify* the search. For example, the frequency-based memory provides information concerning which flips have been under-represented (i.e., less frequent) or not represented at all, and we can diversify the search by exploring these possibilities.

The use of long-term memory in tabu search is usually restricted to some special circumstances. For example, we might encounter a situation where all non-tabu moves lead to a worse solution. This is a special situation indeed: all legal directions lead to inferior solutions! Thus, to make a meaningful decision about which direction to explore next, it might be worthwhile to refer to the contents of the long-term memory.

There are many possibilities here for incorporating this information into the decision-making process. The most typical approach makes the most frequent moves less attractive. Usually the value of the evaluation score is decreased by some penalty measure that depends on the frequency, and the final score implies the winner.

To illustrate this point by an example, assume that the value of the current solution $\mathbf{x}$ for our SAT problem is 35. All non-tabu flips, say on bits 2, 3, and 7, provide values of 30, 33, and 31, respectively, and none of the tabu moves provides a value greater than 37 (the highest value found so far), so we can't apply the aspiration criterion. In such circumstances we refer to the frequency-based memory (see figure 5.8). Let's assume that the evaluation formula for a new solution $\mathbf{x}'$ used in such circumstances is

$$eval(\mathbf{x}') - penalty(\mathbf{x}'),$$

where *eval* returns the value of the original evaluation function (i.e., 30, 33, and 31, respectively, for solutions created by flipping the 2nd, 3rd, and 7th bit), and

$$penalty(\mathbf{x}') = 0.7 \cdot H(i),$$

where 0.7 serves as a coefficient and $H(i)$ is the value taken from the long-term memory H:

7 — for a solution created by flipping the 2nd bit,
11 — for a solution created by flipping the 3rd bit, and
1 — for a solution created by flipping the 7th bit.

The new scores for the three possible solutions are then

$$30 - 0.7 \cdot 7 = 25.1,\ 33 - 0.7 \cdot 11 = 25.3, \text{ and } 31 - 0.7 \cdot 1 = 30.3,$$

and so the third solution (i.e., flipping the 7th bit) is the one we select.

The above option of including frequency values in a penalty measure for evaluating solutions diversifies the search. Of course, many other options might be considered in connection with tabu search. For example, if we have to select a tabu move, we might use an additional rule (so-called *aspiration by default*) to select a move that is the "oldest" of all those considered. It might also be a good idea to memorize not only the set of recent moves, but also whether or not these moves generated any improvement. This information can be incorporated into search decisions (so-called *aspiration by search direction*). Further, it's worthwhile to introduce the concept of *influence*, which measures the degree of change of the new solution either in terms of the distance between the old and new solution, or perhaps the change in the solution's feasibility if we're dealing with a constrained problem. The intuition connected with influence is that a particular move has a larger influence if a corresponding "larger" step was made from the old solution to the new. This information can be also incorporated into the search (so-called *aspiration by influence*). For more details on the many available options in tabu search plus some practical hints for implementation see [186].

We conclude this section by providing an additional example of possible structures used in tabu search while approaching the TSP. For this problem, we can consider moves that swap two cities in a particular solution.[4] The following solution (for an eight-city TSP),

$$(2, 4, 7, 5, 1, 8, 3, 6),$$

has 28 neighbors, since there are 28 different pairs of cities, i.e., $\binom{8}{2} = \frac{7 \cdot 8}{2} = 28$, that we can swap. Thus, for a recency-based memory we can use the structure given in figure 5.9, where the swap of cities i and j is recorded in the i-th row and the j-th column (for $i < j$). Note that we interpret i and j as *cities*, and not as their positions in the solution vector, but this might be another possibility

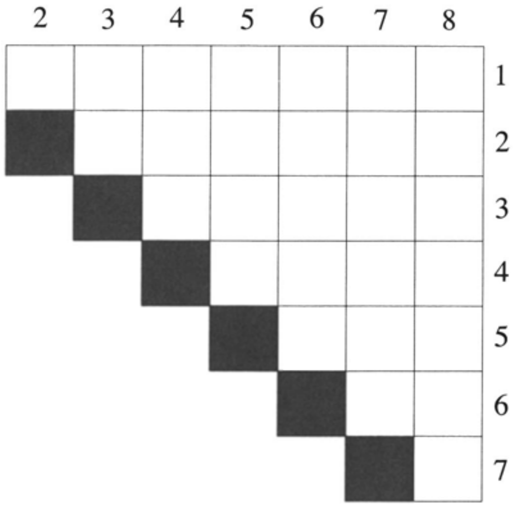

Fig. 5.9. The structure of the recency-based memory for the TSP

to consider. Note that the same structure can also be used for frequency-based memory.

For clarity, we'll maintain the number of remaining iterations for which a given swap stays on the tabu list (recency-based memory) as with the previous SAT problem, while the frequency-based memory will indicate the totals of all swaps that occurred within some horizon h. Assume both memories were initialized to zero and 500 iterations of the search have been completed. The current status of the search then might be as follows. The current solution is

(7, 3, 5, 6, 1, 2, 4, 8)

with the total length of the tour being 173. The best solution encountered during these 500 iterations yields a value of 171. The status of the recency-based and frequency-based memories are displayed in figures 5.10 and 5.11, respectively.

As with the SAT problem, it's easy to interpret the numbers in these memories. The value $M(2, 6) = 5$ indicates that the most recent swap was made for cities 2 and 6, i.e., the previous current solution was

(7, 3, 5, 2, 1, 6, 4, 8).

Therefore, swapping cities 2 and 6 is *tabu* for the next five iterations. Similarly, swaps of cities 1 and 4, 3 and 7, 4 and 5, and 5 and 8, are also on the tabu list. Out of these, the swap between cities 1 and 4 is the oldest (i.e., it happened five iterations ago) and this swap will be removed from the tabu list after the

[4]We do not make any claim that this decision is the best. For example, it's often more meaningful to swap edges of a tour.

2	3	4	5	6	7	8	
0	0	1	0	0	0	0	1
■	0	0	0	5	0	0	2
	■	0	0	0	4	0	3
		■	3	0	0	0	4
			■	0	0	2	5
				■	0	0	6
					■	0	7

Fig. 5.10. The contents of the recency-based memory M for the TSP after 500 iterations. The horizon is five iterations.

2	3	4	5	6	7	8	
0	2	3	3	0	1	1	1
■	2	1	3	1	1	0	2
	■	2	3	3	4	0	3
		■	1	1	2	1	4
			■	4	2	1	5
				■	3	1	6
					■	6	7

Fig. 5.11. The contents of the frequency-based memory F for the TSP after 500 iterations. The horizon is 50 iterations.

next iteration. Note that only 5 swaps (out of 28 possible swaps) are forbidden (tabu).

The frequency-based memory provides some additional statistics of the search. It seems that swapping cities 7 and 8 was the most frequent (it happened

6 times in the last 50 swaps), and there were pairs of cities (like 3 and 8) that weren't swapped within the last 50 iterations.

The neighborhood of a tour was defined by a swap operation between two cities in the tour. This neighborhood is not the best choice either for tabu search or for simulated annealing. Many researchers have selected larger neighborhoods. For example Knox [263] considered neighborhoods based on a *2-interchange* move (see figure 3.6) that's defined by deleting two nonadjacent edges from the current tour and adding two other edges to obtain a new feasible tour. The outline of a particular implementation of tabu search reported in [263] is given in figure 5.12.

```
procedure tabu search
begin
   tries ← 0
   repeat
      generate a tour
      count ← 0
      repeat
         identify a set T of 2-interchange moves
         select the best admissible move from T
         make appropriate 2-interchange
         update tabu list and other variables
         if the new tour is the best-so-far for a given 'tries'
         then update local best tour information
         count ← count +1
      until count = ITER
      tries ← tries +1
      if the current tour is the best-so-far (for all 'tries')
      then update global best tour information
   until tries = MAX-TRIES
end
```

Fig. 5.12. Particular implementation of tabu search for the TSP

Knox [263] made a tour tabu if *both* added edges of the interchange were on the tabu list. The tabu list was updated by placing the added edges on the list (deleted edges were ignored). What's more, the tabu list was of fixed size. Whenever it became full, the oldest element in the list was replaced by the new deleted edge. At initialization, the list was empty and all of the elements of the aspiration list were set to large values. Note that the algorithm examines *all* neighbors, that is, all of the *2-interchange* tours.

Knox [263] indicated that the best results were achieved when

- The length of the tabu list was $3n$ (where n is the number of cities of the problem).

- A candidate tour could override the tabu status if both edges passed an aspiration test, which compared the length of the tour with aspiration values for both added edges. If the length of the tour was better (i.e., smaller) than *both* aspiration values, the test was passed.
- The values present on the aspiration list were the tour costs prior to the interchange.
- The number of searches (MAX-TRIES) and the number of interchanges (ITER) depended on the size of the problem. For problems of 100 cities or less, MAX-TRIES was 4, and ITER was set to $0.0003 \cdot n^4$.

Of course, there are many other possibilities for implementing tabu lists, aspiration criteria, generating initial tours, etc. The above example illustrates only one possible implementation of tabu search for the TSP. You might generate initial tours at random, or by some other means to ensure diversity. The maximum number of iterations might be some function of the improvement made so far. No doubt you can imagine any number of other possibilities.

5.3 Summary

Simulated annealing and tabu search were both designed for the purpose of escaping local optima. However, they differ in the methods used for achieving this goal. Tabu search usually makes uphill moves only when it is stuck in local optima, whereas simulated annealing can make uphill moves at any time. Additionally, simulated annealing is a stochastic algorithm, whereas tabu search is deterministic.

Compared to the classic algorithms offered in chapter 4, we can see that both simulated annealing and tabu search work on complete solutions (as did the techniques in chapter 3). You can halt these algorithms at any iteration and you'll have a solution at hand, but you might notice a subtle difference between these methods and the classic techniques. Simulated annealing and tabu search have more parameters to worry about, such as temperature, rate of reduction, a memory, and so forth. Whereas the classic methods were designed just to "turn the crank" to get the answer, now you have to start really thinking, not just whether or not the algorithm makes sense for your problem, but how to choose the parameters of the algorithm so that it performs optimally. This is a pervasive issue that accompanies the vast majority of algorithms that can escape local optima. The more sophisticated the method, the more you have to use your judgment as to how it should be utilized.