2. Basic Concepts

If we can really understand the problem,
the answer will come out of it,
because the answer is not separate
from the problem.

Krishnamurti, *The Penguin Krishnamurti Reader*

Three basic concepts are common to every algorithmic approach to problem solving. Regardless of the technique that you employ, you'll need to specify: (1) the representation, (2) the objective, and (3) the evaluation function. The representation encodes alternative candidate solutions for manipulation, the objective describes the purpose to be fulfilled, and the evaluation function returns a specific value that indicates the quality of any particular solution given the representation (or minimally, it allows for a comparison of the quality of two alternative solutions). Let's consider each of these aspects of problem solving in turn.

2.1 Representation

The three problems that we examined in the previous chapter provide a good basis for examining alternative representations. Starting with the satisfiability problem (SAT) where we have n variables that are logical bits, one obvious way to represent a candidate solution is a binary string of length n. Each element in the string corresponds to a variable in the problem. The size of the resulting search space under this representation is 2^n, as there are 2^n different binary strings of length n. Every point in this search space is a feasible solution.

For the n-city traveling salesman problem (TSP), we already considered one possible representation: a permutation of the natural numbers $1, ..., n$ where each number corresponds to a city to be visited in sequence. Under this representation, the search space $\mathcal{S}$ consists of all possible permutations, and there are $n!$ such orderings. However, if we're concerned with a symmetric TSP, where the cost of traveling from city i to j is the same in either direction, then we don't care if we proceed down the list of cities from right to left, or left to right. Either way, the tour is the same. This means that we can shrink the size of the search space by one-half. Furthermore, the circuit would be the same regardless of which city we started with, so this reduces the size of the search space by factor of n. Consequently, the real size of the search space reduces to $(n-1)!/2$, but we've seen that this is a huge number for even moderately large TSPs.

Z. Michalewicz et al., *How to Solve It: Modern Heuristics*
© Springer-Verlag Berlin Heidelberg 2004

For a nonlinear programming problem (NLP), the search space consists of all real numbers in n dimensions. We normally rely on floating-point representations to approximate the real numbers on a computer, using either single or double precision. We saw in the previous chapter that for six digits of precision on variables in a range from 0 to 10 there are 10^{7n} different possible values.

For each problem, the representation of a potential solution and its corresponding interpretation implies the search space and its size. This is an important point to recognize: The size of the search space is not determined by the problem; it's determined by your representation and the manner in which you handle this encoding.

Choosing the right search space is of paramount importance. If you don't start by selecting the correct domain to begin your search, you can either add numerous unviable or duplicate solutions to the list of possibilities, or you might actually preclude yourself from ever being able to find the right answer at all.

Here's a good example. There are six matches on a table and the task is to construct four equilateral triangles where the length of each side is equal to the length of a match. It's easy to construct two such triangles using five matches (see figure 2.1), but it's difficult to extend it further into four triangles, especially since only one match remains. Here, the formulation of the problem is misleading because it suggests a two-dimensional search space (i.e., the matches were placed on a table). But to find a solution requires moving into three dimensions (figure 2.2). If you start with the wrong search space, you'll never find the right answer.[1]

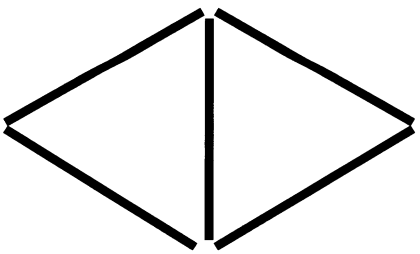

Fig. 2.1. Two triangles, five matches used, one match left

2.2 The objective

Once you've defined the search space, you have to decide what you're looking for. What's the objective of your problem? This is a mathematical statement of the task to be achieved. It's not a function, but rather an expression. For

[1]If you give this problem to another person, you can experiment by giving them some additional "hints" like: you can break the matches into smaller pieces. Such "hints" are *harmful* in the sense that (1) they don't contribute anything towards obtaining the solution, and (2) they expand the search space in a significant way! But don't try this on your friends.

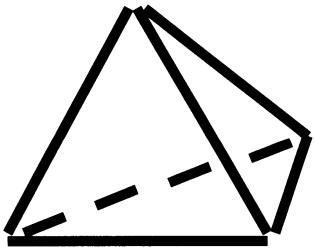

Fig. 2.2. Four triangles, six matches

example, consider the TSP. The objective is typically to minimize the total distance traversed by the salesman subject to the constraint of visiting each city once and only once and returning to the starting city. In mathematical terms, the objective is:

$$\min \sum dist(x, y).$$

For the SAT, the objective is to find the vector of bits such that the compound Boolean statement is satisfied (i.e., made TRUE). For the NLP, the objective is typically to minimize or maximize some nonlinear function of candidate solutions.

2.3 The evaluation function

The objective, however, is not the same thing as the evaluation function. The latter is most typically a mapping from the space of possible candidate solutions under the chosen representation to a set of numbers (e.g., the reals), where each element from the space of possible solutions is assigned a numeric value that indicates its quality. The evaluation function allows you to compare the worth of alternative solutions to your problem as you've modeled it. Some evaluation functions allow for discerning a ranking of all possible solutions. This is called an *ordinal* evaluation function. Alternatively, other evaluation functions are *numeric* and tell you not only the order of the solutions in terms of their perceived quality, but also the degree of that quality.

For example, suppose we wanted to discover a good solution to a TSP. The objective is to minimize the sum of the distances between each of the cities along the route while satisfying the problem's constraints. One evaluation function might map each tour to its corresponding total distance. We could then compare alternative routes and not only say that one is better than another, but exactly how much better. On the other hand, it's sometimes computationally expensive to calculate the exact quality of a particular solution, and it might only be necessary to know approximately how good or bad a solution is, or simply whether or not it compares favorably with some other choice. In such

a case, the evaluation function might operate on two candidate solutions and merely return an indication as to which solution was favored.

In every real-world problem, you choose the evaluation function; it isn't given to you with the problem. How should you go about making this choice? One obvious criterion to satisfy is that when a solution meets the objective completely it also should have the best evaluation. The evaluation function shouldn't indicate that a solution that fails to meet the objective is better than one that meets the objective. But this is only a rudimentary beginning to designing the evaluation function.

Often, the objective suggests a particular evaluation function. For example, just above, we considered using the distance of a candidate solution to a TSP as the evaluation function. This corresponds to the objective of minimizing the total distance covered: the evaluation function is suggested directly from the objective. This often occurs in an NLP as well. For the problem of maximizing the function depicted in figure 1.2, the function itself can serve as an evaluation function with better solutions yielding larger values. But you can't always derive useful evaluation functions from the objective. For the SAT problem, where the objective is to satisfy the given compound Boolean statement (i.e., make it TRUE), every approximate solution is FALSE, and this doesn't give you any useful information on how to improve one candidate solution into another, or how to search for any better alternative. In these cases, we have to be more clever and adopt some surrogate evaluation function that will be appropriate for the task at hand, the representation we choose, and the operators that we employ to go from one solution to the next.

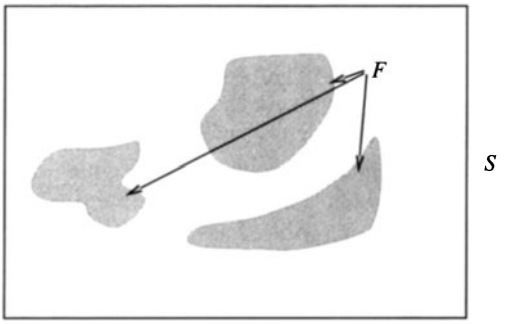

Fig. 2.3. A search space $\mathcal{S}$ and its feasible part $\mathcal{F}$. Note that the feasible regions may be disjoint.

When you design the evaluation function, you also have to consider that for many problems, the only solutions of interest will be in a subset of the search space $\mathcal{S}$. You're only interested in *feasible* solutions, i.e., solutions that satisfy problem-specific constraints. The NLP of figure 1.2 constitutes a perfect example: we're only interested in floating-point vectors such that the product of their component values isn't smaller than 0.75 and their total doesn't exceed

the product of 7.5 and the number of variables. In other words, the search space $\mathcal{S}$ for the NLP can be defined as the set of all floating-point vectors such that their coordinates fall between 0 and 10. Only a subset of these vectors satisfies the additional constraints (concerning the product and the sum of components). This subset constitutes the feasible part of the search space $\mathcal{F}$. In the remainder of the book, we will always distinguish between the search space $\mathcal{S}$ and the feasible search space $\mathcal{F} \subseteq \mathcal{S}$ (see figure 2.3). Note, however, that for the SAT and TSP, $\mathcal{S} = \mathcal{F}$; all of the points from the search space are feasible (presuming for the TSP that there is a path from every city to every other city).

2.4 Defining a search problem

We can now define a search problem.[2] Given a search space $\mathcal{S}$ together with its feasible part $\mathcal{F} \subseteq \mathcal{S}$, find $x \in \mathcal{F}$ such that

$$eval(x) \leq eval(y),$$

for all $y \in \mathcal{F}$. Note that here we're using an evaluation function for which solutions that return smaller values are considered better (i.e., the problem is one of minimization), but for *any* problem, we could just as easily use an evaluation function for which larger values are favored, turning the search problem into one of maximization. There is nothing inherent in the original problem or an approximate model that demands that the evaluation function be set up for minimization or maximization. Indeed, the objective itself doesn't appear anywhere in the search problem defined above. This problem statement could just as easily be used to describe a TSP, a SAT, or an NLP. The search itself doesn't know what problem you are solving! All it "knows" is the information that you provide in the evaluation function, the representation that you use, and the manner in which you sample possible solutions. If your evaluation function doesn't correspond with the objective, you'll be searching for the right answer to the wrong problem!

The point x that satisfies the above condition is called a *global* solution. Finding such a global solution to a problem might be very difficult. In fact, that's true for all three of the example problems discussed so far (i.e., the SAT, TSP, and NLP), but sometimes finding the best solution is easier when we can concentrate on a relatively small subset of the total (feasible or perhaps also infeasible) search space. Fewer alternatives can sometimes make your life easier. This is a fundamental observation that underlies many search techniques.

[2]The terms "search problem" and "optimization problem" are considered synonymous. The search for the best feasible solution is the optimization problem.

2.5 Neighborhoods and local optima

If we concentrate on a region of the search space that's "near" some particular point in that space, we can describe this as looking at the *neighborhood* of that point. Graphically, consider some abstract search space $\mathcal{S}$ (figure 2.4) together with a single point $x \in \mathcal{S}$. The intuition is that a neighborhood $N(x)$ of x is a set of all points of the search space $\mathcal{S}$ that are close in some measurable sense to the given point x.

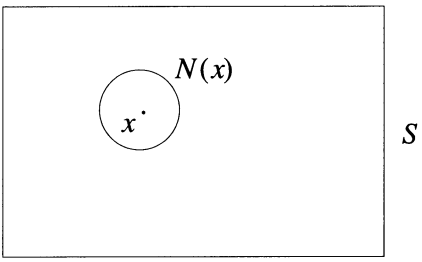

Fig. 2.4. A search space $\mathcal{S}$, a potential solution x, and its neighborhood $N(x)$ denoted by the interior of the circle around x

We can define the nearness between points in the search space in many different ways. Two important possibilities include:

- We can define a distance function *dist* on the search space $\mathcal{S}$:

  $$dist : \mathcal{S} \times \mathcal{S} \longrightarrow \mathbb{R},$$

  and then define the neighborhood $N(x)$ as

  $$N(x) = \{y \in \mathcal{S} \ : \ dist(x, y) \leq \epsilon\},$$

  for some $\epsilon \geq 0$. Another way to say this is that if y satisfies the condition for $N(x)$ above then it is in the ϵ-neighborhood of x. Note that when we're dealing with search spaces defined over continuous variables (e.g., the NLP) a natural choice for defining a neighborhood is the Euclidean distance:

  $$dist(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

  This is also the case for the SAT problem where the distance between two binary strings can be defined as a Hamming distance, i.e., the number of bit positions with different truth assignments. Be wary, though, that just because something is a "natural" choice doesn't necessarily mean it's the best choice in light of other considerations. Natural choices do, however, often provide a good starting position.

- We can define a mapping m on the search space $\mathcal{S}$ as

  $$m : \mathcal{S} \longrightarrow 2^{\mathcal{S}},$$

  and this mapping defines a neighborhood for any point $x \in \mathcal{S}$. For example, we can define a *2-swap* mapping for the TSP that generates a new set of potential solutions from any potential solution x. Every solution that can be generated by swapping two cities in a chosen tour can be said to be in the neighborhood of that tour under the 2-swap operator. In particular, a solution x (a permutation of $n = 20$ cities):

  $$15 - 3 - 11 - 19 - 17 - 2 - ... - 6,$$

  has $\frac{n(n-1)}{2}$ neighbors. These include:

  $15 - 17 - 11 - 19 - 3 - 2 - ... - 6$ (swapping cities at the second and fifth locations),
  $2 - 3 - 11 - 19 - 17 - 15 - ... - 6$ (swapping cities at the first and sixth locations),
  $15 - 3 - 6 - 19 - 17 - 2 - ... - 11$ (swapping cities at the third and last locations),
  etc.

  In contrast, for the SAT problem we could define a *1-flip* mapping that generates a set of potential solutions from any other potential solution x. These are all the solutions that can be reached by flipping a single bit in a particular binary vector, and these would be in the neighborhood of x under the 1-flip operator. For example, a solution x (a binary string of, say, $n = 20$ variables):

  01101010001000011111

  has n neighbors. These include:

  11101010001000011111 (flipping the first bit),
  00101010001000011111 (flipping the second bit),
  01001010001000011111 (flipping the third bit),
  etc.

  In this case we could also define the neighborhood in terms of a distance function: the neighborhood contains all of the strings with a Hamming distance from x that is less than or equal to one.

With the notion of a "neighborhood" defined, we can now discuss the concept of a *local* optimum. A potential solution $x \in \mathcal{F}$ is a local optimum with respect to the neighborhood N, if and only if

$$eval(x) \leq eval(y),$$

for all $y \in N(x)$ (again assuming a minimization criterion). It's often relatively easy to check whether or not a given solution is a local optimum when the size of the neighborhood is very small, or when we know something about the derivative of the evaluation function (if it exists).

Many search methods are based on the statistics of the neighborhood around a given point; that is, the sequence of points that these techniques generate while searching for the best possible solution relies on *local* information at each step along the way. These techniques are designed to locate solutions within a neighborhood of the current point that have better corresponding evaluations. Appropriately, they are known as "neighborhood" or "local" search strategies [476]. For example, suppose you're facing the problem of maximizing the function $f(x) = -x^2$ and you've chosen to use $f(x)$ itself as an evaluation function. Better solutions generate higher values of $f(x)$. Say your current best solution x is 2.0, which has a worth of -4.0. You might define an ϵ-neighborhood with an interval of 0.1 on either side of 2.0 and then search within that range. If you sampled a new point in [1.9, 2.1] and found that it had a better evaluation than 2.0, you'd replace 2.0 with this better solution and continue on from there; otherwise, you'd discard the new solution and take another sample from [1.9, 2.1].

Naturally, most evaluation functions in real-world problems don't look like a quadratic bowl such as $f(x) = -x^2$. The situation is almost always more difficult than this. The evaluation function defines a response surface that is much like a topography of hills and valleys (e.g., as shown in figure 1.2), and the problem of finding the best solution is similar to searching for a peak on a mountain range while walking in a dense fog. You can only sample new points in your immediate vicinity and you can only make local decisions about where to walk next. If you always walk uphill, you'll eventually reach a peak, but this might not be the highest peak in the mountain range. It might be just a "local optimum." You might have to walk downhill for some period of time in order to find a position such that a series of local decisions within successive neighborhoods leads to the global peak.

Local search methods (chapter 3) present an interesting trade-off between the size of the neighborhood $N(x)$ and the efficiency of the search. If the size of the neighborhood is relatively small then the algorithm may be able to search the entire neighborhood quickly. Only a few potential solutions may have to be evaluated before a decision is made on which new solution should be considered next. However, such a small range of visibility increases the chances of becoming trapped in a local optimum! This suggests using large neighborhoods: a larger range of visibility could help in making better decisions. In particular, if the visibility were unrestricted (i.e., the size of the neighborhood were the same as the size of the whole search space), then eventually we'd find the best series of steps to take. The number of evaluations, however, might become enormous, so we might not be able to complete the required calculations within the lifetime of the universe (as noted in chapter 1). When using local search methods, the

appropriate size of the neighborhood can't be determined arbitrarily. It has to fit the task at hand.

2.6 Hill-climbing methods

Let's examine a basic *hill-climbing* procedure and its connection with the concept of a neighborhood. Hill-climbing methods, just like all local search methods (chapter 3), use an iterative improvement technique.[3] The technique is applied to a single point — the current point — in the search space. During each iteration, a new point is selected from the neighborhood of the current point. If that new point provides a better value in light of the evaluation function, the new point becomes the current point. Otherwise, some other neighbor is selected and tested against the current point. The method terminates if no further improvement is possible, or we run out of time or patience.

It's clear that such hill-climbing methods can only provide locally optimum values, and these values depend on the selection of the starting point. Moreover, there's no general procedure for bounding the relative error with respect to the global optimum because it remains unknown. Given the problem of converging on only locally optimal solutions, we often have to start hill-climbing methods from a large variety of different starting points. The hope is that at least some of these initial locations will have a path that leads to the global optimum. We might choose the initial points at random, or on some grid or regular pattern, or even in the light of other information that's available, perhaps as a result of some prior search (e.g., based on some effort someone else made to solve the same problem).

There are a few versions of hill-climbing algorithms. They differ mainly in the way a new solution is selected for comparison with the current solution. One version of a simple iterated hill-climbing algorithm is given in figure 2.5 (steepest ascent hill-climbing). Initially, all possible neighbors of the current solution are considered, and the one $\mathbf{v}_n$ that returns the best value $eval(\mathbf{v}_n)$ is selected to compete with the current string $\mathbf{v}_c$. If $eval(\mathbf{v}_c)$ is worse than $eval(\mathbf{v}_n)$, then the new string $\mathbf{v}_n$ becomes the current string. Otherwise, no local improvement is possible: the algorithm has reached a local or global optimum ($local$ = TRUE). In such a case, the next iteration ($t \leftarrow t + 1$) of the algorithm is executed with a new current string selected at random.

The success or failure of a single iteration (i.e., one complete climb) of the above hill-climbing algorithm is determined completely by the initial point. For problems with many local optima, particularly those where these optima have large *basins of attraction,* it's often very difficult to locate a globally optimal solution.

[3]The term *hill-climbing* implies a maximization problem, but the equivalent *descent* method is easily envisioned for minimization problems. For convenience, the term hill-climbing will be used here to describe both methods without any implied loss of generality.

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

Fig. 2.5. A simple iterated hill-climber

Hill-climbing algorithms have several weaknesses:

- They usually terminate at solutions that are only locally optimal.
- There is no information as to the amount by which the discovered local optimum deviates from the global optimum, or perhaps even other local optima.
- The optimum that's obtained depends on the initial configuration.
- In general, it is *not* possible to provide an upper bound for the computation time.

On the other hand, there is one alluring advantage for hill-climbing techniques: they're very easy to apply! All that's needed is the representation, the evaluation function, and a measure that defines the neighborhood around a given solution.

Effective search techniques provide a mechanism for balancing two apparently conflicting objectives: *exploiting* the best solutions found so far and at the same time *exploring* the search space.[4] Hill-climbing techniques exploit the best available solution for possible improvement but neglect exploring a large

[4]This balance between exploration and exploitation was noted as early as the 1950s by the famous statistician G.E.P. Box [54].

portion of the search space $\mathcal{S}$. In contrast, a random search — where points are sampled from $\mathcal{S}$ with equal probabilities — explores the search space thoroughly but foregoes exploiting promising regions of the space. Each search space is different and even identical spaces can appear very different under different representations and evaluation functions. So there's no way to choose a single search method that can serve well in every case. In fact, this can be proved mathematically [499, 156].

Getting stuck in local optima is a serious problem. It's one of the main deficiencies that plague industrial applications of numerical optimization. Almost every solution to real-world problems in factory scheduling, demand forecasting, land management, and so forth, is at best only locally optimal.

What can we do? How can we design a search algorithm that has a chance to escape local optima, to balance exploration and exploitation, and to make the search independent from the initial configuration? There are a few possibilities, and we'll discuss some of them in chapter 5, but keep in mind that the proper choices are always problem dependent. One option, as we discussed earlier, is to execute the chosen search algorithm for a large number of initial configurations. Moreover, it's often possible to use the results of previous trials to improve the choice of the initial configuration for the next trial. It might also be worthwhile to introduce a more complex means for generating new solutions, or enlarge the neighborhood size. It's also possible to modify the criteria for accepting transitions to new points that correspond with a *negative* change in the evaluation function. That is, we might want to accept a worse solution from the local neighborhood in the hope that it will eventually lead to something better.

But before we move to more sophisticated search methods, we need to review the basic, classic problem-solving techniques, such as dynamic programming, the A^* algorithm, and a few others. Before we do that, you get another chance to display your prowess by solving an interesting problem in geometry.

2.7 Can you sink this trick shot?

Now that you've seen a number of different search strategies and have been introduced to the issues of choosing a representation, the objective, and an evaluation function, let's see if you can solve this challenging puzzle. Remember, you might have to rely on some information that we presented in an earlier section — or maybe not! We're not telling!

> A single ball rests calmly on a square-shaped billiard table. The ball is hit and it moves on the table for an infinite amount of time, obeying an elementary law of physics: The angles α and β (figure 2.6) are always equal. Under what circumstances will the movement of the ball be cyclic?

Please put the book aside and think about it.

One way we might approach the problem is to restrict our attention to the surface of the billiard table and try to reason "within" this square. It's very