7

Combinatorial Search and Heuristic Methods

We can solve many problems to optimality using exhaustive search techniques, although the time complexity can be enormous. For certain applications, it may pay to spend extra time to be certain of the optimal solution. A good example occurs in testing a circuit or a program on all possible inputs. You can prove the correctness of the device by trying all possible inputs and verifying that they give the correct answer. Verifying correctness is a property to be proud of. However, claiming that it works correctly on all the inputs you tried is worth much less.

Modern computers have clock rates of a few *gigahertz*, meaning billions of operations per second. Since doing something interesting takes a few hundred instructions, you can hope to search millions of items per second on contemporary machines.

It is important to realize how big (or how small) one million is. One million permutations means all arrangements of roughly 10 or 11 objects, but not more. One million subsets means all combinations of roughly 20 items, but not more. Solving significantly larger problems requires carefully pruning the search space to ensure we look at only the elements that really matter.

In this section, we introduce backtracking as a technique for listing all possible solutions for a combinatorial algorithm problem. We illustrate the power of clever pruning techniques to speed up real search applications. For problems that are too large to contemplate using brute-force combinatorial search, we introduce heuristic methods such as simulated annealing. Such heuristic methods are important weapons in any practical algorist's arsenal.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_7,

7.1 Backtracking

Backtracking is a systematic way to iterate through all the possible configurations of a search space. These configurations may represent all possible arrangements of objects (permutations) or all possible ways of building a collection of them (subsets). Other situations may demand enumerating all spanning trees of a graph, all paths between two vertices, or all possible ways to partition vertices into color classes.

What these problems have in common is that we must generate each one possible configuration exactly once. Avoiding both repetitions and missing configurations means that we must define a systematic generation order. We will model our combinatorial search solution as a vector $a = (a_1, a_2, ..., a_n)$, where each element a_i is selected from a finite ordered set S_i. Such a vector might represent an arrangement where a_i contains the ith element of the permutation. Or, the vector might represent a given subset S, where a_i is true if and only if the ith element of the universe is in S. The vector can even represent a sequence of moves in a game or a path in a graph, where a_i contains the ith event in the sequence.

At each step in the backtracking algorithm, we try to extend a given partial solution $a = (a_1, a_2, ..., a_k)$ by adding another element at the end. After extending it, we must test whether what we now have is a solution: if so, we should print it or count it. If not, we must check whether the partial solution is still potentially extendible to some complete solution.

Backtracking constructs a tree of partial solutions, where each vertex represents a partial solution. There is an edge from x to y if node y was created by advancing from x. This tree of partial solutions provides an alternative way to think about backtracking, for the process of constructing the solutions corresponds exactly to doing a depth-first traversal of the backtrack tree. Viewing backtracking as a depth-first search on an implicit graph yields a natural recursive implementation of the basic algorithm.

```
Backtrack-DFS(A, k)
    if A = (a_1, a_2, ..., a_k) is a solution, report it.
    else
        k = k + 1
        compute S_k
        while S_k ≠ ∅ do
            a_k = an element in S_k
            S_k = S_k − a_k
            Backtrack-DFS(A, k)
```

Although a breadth-first search could also be used to enumerate solutions, a depth-first search is greatly preferred because it uses much less space. The current state of a search is completely represented by the path from the root to the current search depth-first node. This requires space proportional to the *height* of the tree. In breadth-first search, the queue stores all the nodes at the current level, which

is proportional to the *width* of the search tree. For most interesting problems, the width of the tree grows exponentially in its height.

Implementation

The honest working `backtrack` code is given below:

```
bool finished = FALSE;            /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
     int c[MAXCANDIDATES];     /* candidates for next position */
     int ncandidates;          /* next position candidate count */
     int i;                    /* counter */

     if (is_a_solution(a,k,input))
             process_solution(a,k,input);
     else {
             k = k+1;
             construct_candidates(a,k,input,c,&ncandidates);
             for (i=0; i<ncandidates; i++) {
                     a[k] = c[i];
                     make_move(a,k,input);
                     backtrack(a,k,input);
                     unmake_move(a,k,input);
                     if (finished) return;   /* terminate early */
             }
     }
}
```

Backtracking ensures correctness by enumerating all possibilities. It ensures efficiency by never visiting a state more than once.

Study how recursion yields an elegant and easy implementation of the backtracking algorithm. Because a new candidates array `c` is allocated with each recursive procedure call, the subsets of not-yet-considered extension candidates at each position will not interfere with each other.

The application-specific parts of this algorithm consists of five subroutines:

- `is_a_solution(a,k,input)` – This Boolean function tests whether the first k elements of vector a from a complete solution for the given problem. The last argument, `input`, allows us to pass general information into the routine. We can use it to specify n—the size of a target solution. This makes sense when constructing permutations or subsets of n elements, but other data may be relevant when constructing variable-sized objects such as sequences of moves in a game.

- `construct_candidates(a,k,input,c,ncandidates)` – This routine fills an array c with the complete set of possible candidates for the kth position of a, given the contents of the first $k-1$ positions. The number of candidates returned in this array is denoted by `ncandidates`. Again, `input` may be used to pass auxiliary information.

- `process_solution(a,k,input)` – This routine prints, counts, or however processes a complete solution once it is constructed.

- `make_move(a,k,input)` *and* `unmake_move(a,k,input)` – These routines enable us to modify a data structure in response to the latest move, as well as clean up this data structure if we decide to take back the move. Such a data structure could be rebuilt from scratch from the solution vector a as needed, but this is inefficient when each move involves incremental changes that can easily be undone.

  These calls function as null stubs in all of this section's examples, but will be employed in the Sudoku program of Section 7.3 (page 239).

We include a global `finished` flag to allow for premature termination, which could be set in any application-specific routine.

To really understand how backtracking works, you must see how such objects as permutations and subsets can be constructed by defining the right state spaces. Examples of several state spaces are described in subsections below.

7.1.1 Constructing All Subsets

A critical issue when designing state spaces to represent combinatorial objects is how many objects need representing. How many subsets are there of an n-element set, say the integers $\{1, \ldots, n\}$? There are exactly two subsets for $n = 1$, namely $\{\}$ and $\{1\}$. There are four subsets for $n = 2$, and eight subsets for $n = 3$. Each new element doubles the number of possibilities, so there are 2^n subsets of n elements.

Each subset is described by elements that are in it. To construct all 2^n subsets, we set up an array/vector of n cells, where the value of a_i (true or false) signifies whether the ith item is in the given subset. In the scheme of our general backtrack algorithm, $S_k = (true, false)$ and a is a solution whenever $k = n$. We can now construct all subsets with simple implementations of `is_a_solution()`, `construct_candidates()`, and `process_solution()`.

```
is_a_solution(int a[], int k, int n)
{
        return (k == n);                /* is k == n? */
}
```

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
        c[0] = TRUE;
        c[1] = FALSE;
        *ncandidates = 2;
}


process_solution(int a[], int k)
{
        int i;                          /* counter */

        printf("{");
        for (i=1; i<=k; i++)
                if (a[i] == TRUE) printf(" %d",i);

        printf(" }\n");
}
```

Printing each out subset after constructing it proves to be the most complicated of the three routines!

Finally, we must instantiate the call to `backtrack` with the right arguments. Specifically, this means giving a pointer to the empty solution vector, setting $k = 0$ to denote that it is empty, and specifying the number of elements in the universal set:

```
generate_subsets(int n)
{
        int a[NMAX];                    /* solution vector */

        backtrack(a,0,n);
}
```

In what order will the subsets of $\{1, 2, 3\}$ be generated? It depends on the order of moves `construct_candidates`. Since *true* always appears before *false*, the subset of all trues is generated first, and the all-false empty set is generated last: {123}, {12}, {13}, {1}, {23}, {2}, {3}, {}

Trace through this example carefully to make sure you understand the backtracking procedure. The problem of generating subsets is more thoroughly discussed in Section 14.5 (page 452).

7.1.2 Constructing All Permutations

Counting permutations of $\{1, \ldots, n\}$ is a necessary prerequisite to generating them. There are n distinct choices for the value of the first element of a permutation. Once

we have fixed a_1, there are $n-1$ candidates remaining for the second position, since we can have any value except a_1 (repetitions are forbidden in permutation). Repeating this argument yields a total of $n! = \prod_{i=1}^{n} i$ distinct permutations.

This counting argument suggests a suitable representation. Set up an array/vector a of n cells. The set of candidates for the ith position will be the set of elements that have not appeared in the $(i-1)$ elements of the partial solution, corresponding to the first $i-1$ elements of the permutation.

In the scheme of the general backtrack algorithm, $S_k = \{1, \ldots, n\} - a$, and a is a solution whenever $k = n$:

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
        int i;                          /* counter */
        bool in_perm[NMAX];             /* who is in the permutation? */

        for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
        for (i=0; i<k; i++) in_perm[ a[i] ] = TRUE;

        *ncandidates = 0;
        for (i=1; i<=n; i++)
                if (in_perm[i] == FALSE) {
                        c[ *ncandidates] = i;
                        *ncandidates = *ncandidates + 1;
                }
}
```

Testing whether i is a candidate for the kth slot in the permutation can be done by iterating through all $k-1$ elements of a and verifying that none of them matched. However, we prefer to set up a bit-vector data structure (see Section 12.5 (page 385)) to maintain which elements are in the partial solution. This gives a constant-time legality check.

Completing the job requires specifying `process_solution` and `is_a_solution`, as well as setting the appropriate arguments to `backtrack`. All are essentially the same as for subsets:

```
process_solution(int a[], int k)
{
        int i;                          /* counter */

        for (i=1; i<=k; i++) printf(" %d",a[i]);

        printf("\n");
}
```

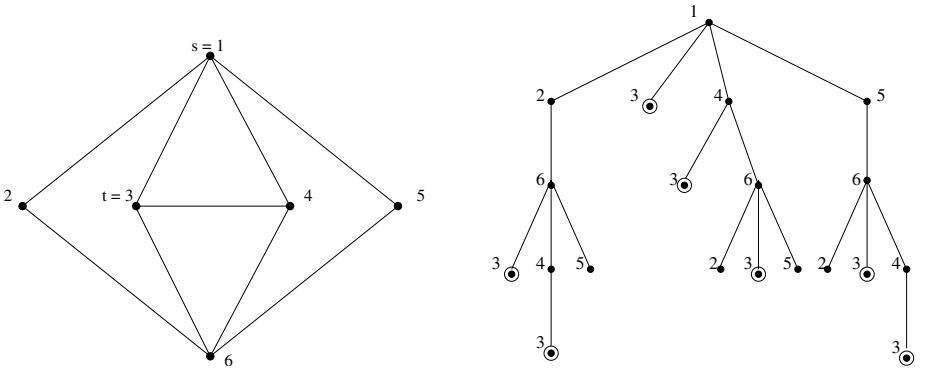

Figure 7.1: Search tree enumerating all simple s-t paths in the given graph (left).

```
is_a_solution(int a[], int k, int n)
{
        return (k == n);
}


generate_permutations(int n)
{
        int a[NMAX];                    /* solution vector */

        backtrack(a,0,n);
}
```

As a consequence of the candidate order, these routines generate permutations in *lexicographic*, or sorted order—i.e., 123, 132, 213, 231, 312, and 321. The problem of generating permutations is more thoroughly discussed in Section 14.4 (page 448).

7.1.3 Constructing All Paths in a Graph

Enumerating all the simple s to t paths through a given graph is a more complicated problem than listing permutations or subsets. There is no explicit formula that counts the number of solutions as a function of the number of edges or vertices, because the number of paths depends upon the structure of the graph.

The starting point of any path from s to t is always s. Thus, s is the only candidate for the first position and $S_1 = \{s\}$. The possible candidates for the second position are the vertices v such that (s, v) is an edge of the graph, for the path wanders from vertex to vertex using edges to define the legal steps. In general,

S_{k+1} consists of the set of vertices adjacent to a_k that have not been used elsewhere in the partial solution A.

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
     int i;                    /* counters */
     bool in_sol[NMAX];        /* what's already in the solution? */
     edgenode *p;              /* temporary pointer */
     int last;                 /* last vertex on current path */

     for (i=1; i<NMAX; i++) in_sol[i] = FALSE;
     for (i=1; i<k; i++) in_sol[ a[i] ] = TRUE;

     if (k==1) {               /* always start from vertex 1 */
             c[0] = 1;
             *ncandidates = 1;
     }
     else    {
             *ncandidates = 0;
             last = a[k-1];
             p = g.edges[last];
             while (p != NULL) {
                     if (!in_sol[ p->y ]) {
                             c[*ncandidates] = p->y;
                             *ncandidates = *ncandidates + 1;
                     }
                     p = p->next;
             }
     }
}
```

We report a successful path whenever $a_k = t$.

```
is_a_solution(int a[], int k, int t)
{
        return (a[k] == t);
}

process_solution(int a[], int k)
{
        solution_count ++;        /* count all s to t paths */
}
```

The solution vector A must have room for all n vertices, although most paths are likely shorter than this. Figure 7.1 shows the search tree giving all paths from a particular vertex in an example graph.

7.2 Search Pruning

Backtracking ensures correctness by enumerating all possibilities. Enumerating all $n!$ permutations of n vertices of the graph and selecting the best one yields the correct algorithm to find the optimal traveling salesman tour. For each permutation, we could see whether all edges implied by the tour really exists in the graph G, and if so, add the weights of these edges together.

However, it would be wasteful to construct all the permutations first and then analyze them later. Suppose our search started from vertex v_1, and it happened that edge (v_1, v_2) was not in G. The next $(n-2)!$ permutations enumerated starting with (v_1, v_2) would be a complete waste of effort. Much better would be to prune the search after v_1, v_2 and continue next with v_1, v_3. By restricting the set of next elements to reflect only moves that are legal from the current partial configuration, we significantly reduce the search complexity.

Pruning is the technique of cutting off the search the instant we have established that a partial solution cannot be extended into a full solution. For traveling salesman, we seek the cheapest tour that visits all vertices. Suppose that in the course of our search we find a tour t whose cost is C_t. Later, we may have a partial solution a whose edge sum $C_A > C_t$. Is there any reason to continue exploring this node? No, because any tour with this prefix $a_1, \ldots, a_k$ will have cost greater than tour t, and hence is doomed to be nonoptimal. Cutting away such failed partial tours as soon as possible can have an enormous impact on running time.

Exploiting symmetry is another avenue for reducing combinatorial searches. Pruning away partial solutions identical to those previously considered requires recognizing underlying symmetries in the search space. For example, consider the state of our TSP search after we have tried all partial positions beginning with v_1. Does it pay to continue the search with partial solutions beginning with v_2? No. Any tour starting and ending at v_2 can be viewed as a rotation of one starting and ending at v_1, for these tours are cycles. There are thus only $(n-1)!$ distinct tours on n vertices, not $n!$. By restricting the first element of the tour to v_1, we save a factor of n in time without missing any interesting solutions. Detecting such symmetries can be subtle, but once identified they can usually be easily exploited.

> *Take-Home Lesson:* Combinatorial searches, when augmented with tree pruning techniques, can be used to find the optimal solution of small optimization problems. How small depends upon the specific problem, but typical size limits are somewhere between $15 \leq n \leq 50$ items.

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

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

Figure 7.2: Challenging Sudoku puzzle (l) with solution (r)

7.3 Sudoku

A Sudoku craze has swept the world. Many newspapers now publish daily Sudoku puzzles, and millions of books about Sudoku have been sold. British Airways sent a formal memo forbidding its cabin crews from doing Sudoku during takeoffs and landings. Indeed, I have noticed plenty of Sudoku going on in the back of my algorithms classes during lecture.

What is Sudoku? In its most common form, it consists of a 9×9 grid filled with blanks and the digits 1 to 9. The puzzle is completed when every row, column, and sector (3×3 subproblems corresponding to the nine sectors of a tic-tac-toe puzzle) contain the digits 1 through 9 with no deletions or repetition. Figure 7.2 presents a challenging Sudoku puzzle and its solution.

Backtracking lends itself nicely to the problem of solving Sudoku puzzles. We will use the puzzle here to better illustrate the algorithmic technique. Our state space will be the sequence of open squares, each of which must ultimately be filled in with a number. The candidates for open squares (i, j) are exactly the integers from 1 to 9 that have not yet appeared in row i, column j, or the 3×3 sector containing (i, j). We backtrack as soon as we are out of candidates for a square.

The solution vector `a` supported by `backtrack` only accepts a single integer per position. This is enough to store contents of a board square (1-9) but not the coordinates of the board square. Thus, we keep a separate array of `move` positions as part of our `board` data type provided below. The basic data structures we need to support our solution are:

```
#define DIMENSION 9                  /* 9*9 board */
#define NCELLS  DIMENSION*DIMENSION /* 81 cells in a 9*9 problem */

typedef struct {
    int x, y;                        /* x and y coordinates of point */
} point;
```

```
typedef struct {
    int m[DIMENSION+1][DIMENSION+1]; /* matrix of board contents */
    int freecount;                   /* how many open squares remain? */
    point move[NCELLS+1];            /* how did we fill the squares? */
} boardtype;
```

Constructing the candidates for the next solution position involves first picking the open square we want to fill next (`next_square`), and then identifying which numbers are candidates to fill that square (`possible_values`). These routines are basically bookkeeping, although the subtle details of how they work can have an enormous impact on performance.

```
construct_candidates(int a[], int k, boardtype *board, int c[],
                     int *ncandidates)
{
   int x,y;                     /* position of next move */
   int i;                       /* counter */
   bool possible[DIMENSION+1]; /* what is possible for the square */

   next_square(&x,&y,board);   /* which square should we fill next? */

   board->move[k].x = x;       /* store our choice of next position */
   board->move[k].y = y;

   *ncandidates = 0;

   if ((x<0) && (y<0)) return; /* error condition, no moves possible */

   possible_values(x,y,board,possible);
   for (i=0; i<=DIMENSION; i++)
           if (possible[i] == TRUE) {
                   c[*ncandidates] = i;
                   *ncandidates = *ncandidates + 1;
           }
}
```

We must update our `board` data structure to reflect the effect of filling a candidate value into a square, as well as remove these changes should we backtrack away from this position. These updates are handled by `make_move` and `unmake_move`, both of which are called directly from `backtrack`:

```
make_move(int a[], int k, boardtype *board)
{
        fill_square(board->move[k].x,board->move[k].y,a[k],board);
}


unmake_move(int a[], int k, boardtype *board)
{
        free_square(board->move[k].x,board->move[k].y,board);
}
```

One important job for these board update routines is maintaining how many free squares remain on the board. A solution is found when there are no more free squares remaining to be filled:

```
is_a_solution(int a[], int k, boardtype *board)
{
        if (board->freecount == 0)
                return (TRUE);
        else
                return(FALSE);
}
```

We print the configuration and turn off the backtrack search by setting off the global `finished` flag on finding a solution. This can be done without consequence because "official" Sudoku puzzles are only allowed to have one solution. There can be an enormous number of solutions to nonofficial Sudoku puzzles. Indeed, the empty puzzle (where no number is initially specified) can be filled in exactly 6,670,903,752,021,072,936,960 ways. We can ensure we don't see all of them by turning off the search:

```
process_solution(int a[], int k, boardtype *board)
{
        print_board(board);
        finished = TRUE;
}
```

This completes the program modulo details of identifying the next open square to fill (`next_square`) and identifying the candidates to fill that square (`possible_values`). Two reasonable ways to select the next square are:

- *Arbitrary Square Selection* – Pick the first open square we encounter, possibly picking the first, the last, or a random open square. All are equivalent in that there seems to be no reason to believe that one heuristic will perform any better than the other.

- *Most Constrained Square Selection* – Here, we check each of the open squares (i, j) to see how many number candidates remain for each—i.e. , have not already been used in either row i, column j, or the sector containing (i, j). We pick the square with the fewest number of candidates.

Although both possibilities work correctly, the second option is much, much better. Often there will be open squares with only one remaining candidate. For these, the choice is forced. We might as well make it now, especially since pinning this value down will help trim the possibilities for other open squares. Of course, we will spend more time selecting each candidate square, but if the puzzle is easy enough we may never have to backtrack at all.

If the most constrained square has two possibilities, we have a 1/2 probability of guessing right the first time, as opposed to a $(1/9)^{th}$ probability for a completely unconstrained square. Reducing our average number of choices from (say) 3 per square to 2 per square is an enormous win, since it multiplies for each position. If we have (say) 20 positions to fill, we must enumerate only $2^{20} = 1,048,576$ solutions. A branching factor of 3 at each of the 20 positions will result in over $3,000$ times as much work!

Our final decision concerns the `possible_values` we allow for each square. We have two possibilities:

- *Local Count* – Our backtrack search works correctly if the routine generating candidates for board position (i, j) (`possible_values`) does the obvious thing and allows all numbers 1 to 9 that have not appeared in the given row, column, or sector.

- *Look ahead* – But what if our current partial solution has some *other* open square where there are no candidates remaining under the local count criteria? There is no possible way to complete this partial solution into a full Sudoku grid. Thus there *really* are zero possible moves to consider for (i, j) because of what is happening elsewhere on the board!

  We will discover this obstruction eventually, when we pick this square for expansion, discover it has no moves, and then have to backtrack. But why wait, since all our efforts until then will be wasted? We are *much* better off backtracking immediately and moving on.[1]

Successful pruning requires looking ahead to see when a solution is doomed to go nowhere, and backing off as soon as possible.

Table 7.1 presents the number of calls to `is_a_solution` for all four backtracking variants on three Sudoku instances of varying complexity:

[1]This look-ahead condition might have naturally followed from the most-constrained square selection, had it been permitted to select squares with no moves. However, my implementation credited squares that already contained numbers as having no moves, thus limiting the next square choices to squares with at least one move.

Pruning Condition		Puzzle Complexity		
`next_square`	`possible_values`	Easy	Medium	Hard
arbitrary	local count	1,904,832	863,305	never finished
arbitrary	look ahead	127	142	12,507,212
most constrained	local count	48	84	1,243,838
most constrained	look ahead	48	65	10,374

Table 7.1: Sudoku run times (in number of steps) under different pruning strategies

- The *Easy* board was intended to be easy for a human player. Indeed, my program solved it without any backtracking steps when the most constrained square was selected as the next position.
- The *Medium* board stumped all the contestants at the finals of the World Sudoku Championship in March 2006. The decent search variants still required only a few backtrack steps to dispatch this problem.
- The *Hard* problem is the board displayed in Figure 7.2, which contains only 17 fixed numbers. This is the fewest specified known number of positions in any problem instance that has only one complete solution.

What is considered to be a "hard" problem instance depends upon the given heuristic. Certainly you know people who find math/theory harder than programming and others who think differently. Heuristic A may well think instance I_1 is easier than I_2, while heuristic B ranks them in the other order.

What can we learn from these experiments? Looking ahead to eliminate dead positions as soon as possible is the best way to prune a search. Without this operation, we never finished the hardest puzzle and took thousands of times longer than we should have on the easier ones.

Smart square selection had a similar impact, even though it nominally just rearranges the order in which we do the work. However, doing more constrained positions first is tantamount to reducing the outdegree of each node in the tree, and each additional position we fix adds constraints that help lower the degree for future selections.

It took the better part of an hour (48:44) to solve the puzzle in Figure 7.2 when I selected an arbitrary square for my next move. Sure, my program was faster in most instances, but Sudoku puzzles are designed to be solved by people using pencils in much less time than this. Making the next move in the most constricted square reduced search time by a factor of over 1,200. Each puzzle we tried can now be solved in seconds—the time it takes to reach for the pencil if you want to do it by hand.

This is the power of a pruning search. Even simple pruning strategies can suffice to reduce running time from impossible to instantaneous.

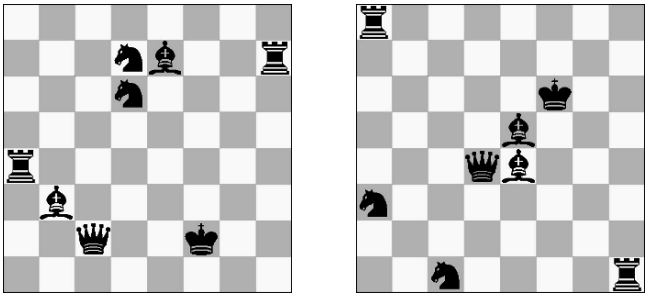

Figure 7.3: Configurations covering 63 but not 64 squares

7.4 War Story: Covering Chessboards

Every researcher dreams of solving a classical problem—one that has remained open and unsolved for over a century. There is something romantic about communicating across the generations, being part of the evolution of science, and helping to climb another rung up the ladder of human progress. There is also a pleasant sense of smugness that comes from figuring out how to do something that nobody could do before you.

There are several possible reasons why a problem might stay open for such a long period of time. Perhaps it is so difficult and profound that it requires a uniquely powerful intellect to solve. A second reason is technological—the ideas or techniques required to solve the problem may not have existed when it was first posed. A final possibility is that no one may have cared enough about the problem in the interim to seriously bother with it. Once, I helped solve a problem that had been open for over a hundred years. Decide for yourself which reason best explains why.

Chess is a game that has fascinated mankind for thousands of years. In addition, it has inspired many combinatorial problems of independent interest. The combinatorial explosion was first recognized with the legend that the inventor of chess demanded as payment one grain of rice for the first square of the board, and twice as much for the $(i+1)$st square than the ith square. The king was astonished to learn he had to cough up $\sum_{i=1}^{64} 2^i = 2^{65} - 1 = 36{,}893{,}488{,}147{,}419{,}103{,}231$ grains of rice. In beheading the inventor, the wise king first established pruning as a technique for dealing with the combinatorial explosion.

In 1849, Kling posed the question of whether all 64 squares on the board can be simultaneously threatened by an arrangement of the eight main pieces on the chess board—the king, queen, two knights, two rooks, and two oppositely colored bishops. Pieces do not threaten the square they sit on. Configurations that simultaneously threaten 63 squares, such as those in Figure 7.3, have been known for a long time, but whether this was the best possible remained an open problem. This problem

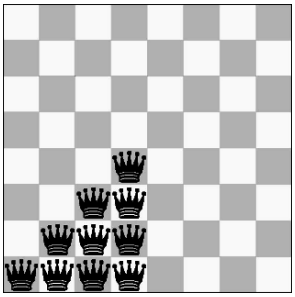

Figure 7.4: The ten unique positions for the queen, with respect to symmetry

seemed ripe for solution by exhaustive combinatorial searching, although whether it was solvable depended upon the size of the search space.

Consider the eight main pieces in chess (king, queen, two rooks, two bishops, and two knights). How many ways can they be positioned on a chessboard? The trivial bound is $64!/(64-8)! = 178{,}462{,}987{,}637{,}760 \approx 10^{15}$ positions. Anything much larger than about 10^9 positions would be unreasonable to search on a modest computer in a modest amount of time.

Getting the job done would require significant pruning. Our first idea was to remove symmetries. Accounting for orthogonal and diagonal symmetries left only ten distinct positions for the queen, shown in Figure 7.4.

Once the queen is placed, there are $64 \cdot 63/2 = 2{,}016$ distinct ways to position a pair of rooks or knights, 64 places to locate the king, and 32 spots for each of the white and black bishops. Such an exhaustive search would test $2{,}663{,}550{,}812{,}160 \approx 10^{13}$ distinct positions—still much too large to try.

We could use backtracking to construct all of the positions, but we had to find a way to prune the search space significantly. Pruning the search meant that we needed a quick way to prove that there was no way to complete a partially filled-in position to cover all 64 squares. Suppose we had already placed seven pieces on the board, and together they covered all but 10 squares of the board. Say the remaining piece was the king. Can there be a position to place the king so that all squares are threatened? The answer must be no, because the king can threaten at most eight squares according to the rules of chess. There can be no reason to test any king position. We might win big pruning such configurations.

This pruning strategy required carefully ordering the evaluation of the pieces. Each piece can threaten a certain maximum number of squares: the queen 27, the king/knight 8, the rook 14, and the bishop 13. We would want to insert the pieces in decreasing order of mobility. We can prune when the number of unthreatened squares exceeds the sum of the maximum coverage of the unplaced pieces. This sum is minimized by using the decreasing order of mobility.

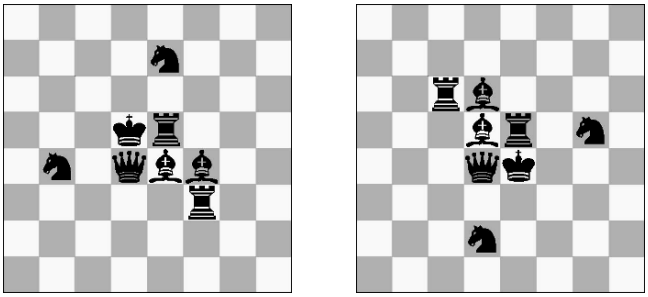

Figure 7.5: Weakly covering 64 squares

When we implemented a backtrack search using this pruning strategy, we found that it eliminated over 95% of the search space. After optimizing our move generation, our program could search over 1,000 positions per second. But this was still too slow, for $10^{12}/10^3 = 10^9$ seconds meant 1,000 days! Although we might further tweak the program to speed it up by an order of magnitude or so, what we really needed was to find a way to prune more nodes.

Effective pruning meant eliminating large numbers of positions at a single stroke. Our previous attempt was too weak. What if instead of placing up to eight pieces on the board simultaneously, we placed *more* than eight pieces. Obviously, the more pieces we placed simultaneously, the more likely they would threaten all 64 squares. But *if* they didn't cover, all subsets of eight distinct pieces from the set couldn't possibly threaten all squares. The potential existed to eliminate a vast number of positions by pruning a single node.

So in our final version, the nodes of our search tree corresponded to chessboards that could have any number of pieces, and more than one piece on a square. For a given board, we distinguished *strong* and *weak* attacks on a square. A strong attack corresponds to the usual notion of a threat in chess. A square is *weakly attacked* if the square is strongly attacked by some subset of the board—that is, a weak attack ignores any possible blocking effects of intervening pieces. All 64 squares can be weakly attacked with eight pieces, as shown in Figure 7.5.

Our algorithm consisted of two passes. The first pass listed boards where every square was weakly attacked. The second pass filtered the list by considering blocking pieces. A weak attack is much faster to compute (no blocking to worry about), and any strong attack set is always a subset of a weak attack set. The position could be pruned whenever there was a non-weakly threatened square.

This program was efficient enough to complete the search on a slow 1988-era IBM PC-RT in under one day. It did not find a single position covering all 64 squares with the bishops on opposite colored squares. However, our program showed that it is possible to cover the board with *seven* pieces provided a queen and a knight can occupy the same square, as shown in Figure 7.6.

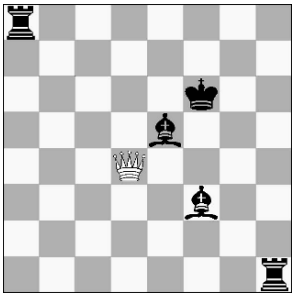

Figure 7.6: Seven pieces suffice when superimposing queen and knight (shown as a white queen)

Take-Home Lesson: Clever pruning can make short work of surprisingly hard combinatorial search problems. Proper pruning will have a greater impact on search time than any other factor.

7.5 Heuristic Search Methods

Heuristic methods provide an alternate way to approach difficult combinatorial optimization problems. Backtracking gave us a method to find the best of all possible solutions, as scored by a given objective function. However, any algorithm searching all configurations is doomed to be impossible on large instances.

In this section, we discuss such heuristic search methods. We devote the bulk of our attention to simulated annealing, which I find to be the most reliable method to apply in practice. Heuristic search algorithms have an air of voodoo about them, but how they work and why one method might work better than another follows logically enough if you think them through.

In particular, we will look at three different heuristic search methods: random sampling, gradient-descent search, and simulated annealing. The traveling salesman problem will be our ongoing example for comparing heuristics. All three methods have two common components:

- *Solution space representation* – This is a complete yet concise description of the set of possible solutions for the problem. For traveling salesman, the solution space consists of $(n-1)!$ elements—namely all possible circular permutations of the vertices. We need a data structure to represent each element of the solution space. For TSP, the candidate solutions can naturally be represented using an array S of $n-1$ vertices, where S_i defines the $(i+1)$st vertex on the tour starting from v_1.
- *Cost function* – Search methods need a *cost* or *evaluation* function to access the quality of each element of the solution space. Our search heuristic

identifies the element with the best possible score—either highest or lowest depending upon the nature of the problem. For TSP, the cost function for evaluating a given candidate solution S should just sum up the costs involved, namely the weight of all edges (S_i, S_{i+1}), where S_{n+1} denotes v_1.

7.5.1 Random Sampling

The simplest method to search in a solution space uses random sampling. It is also called the *Monte Carlo method*. We repeatedly construct random solutions and evaluate them, stopping as soon as we get a good enough solution, or (more likely) when we are tired of waiting. We report the best solution found over the course of our sampling.

True random sampling requires that we are able to select elements from the solution space *uniformly at random*. This means that each of the elements of the solution space must have an equal probability of being the next candidate selected. Such sampling can be a subtle problem. Algorithms for generating random permutations, subsets, partitions, and graphs are discussed in Sections 14.4-14.7.

```
random_sampling(tsp_instance *t, int nsamples, tsp_solution *bestsol)
{
        tsp_solution s;                 /* current tsp solution */
        double best_cost;               /* best cost so far */
        double cost_now;                /* current cost */
        int i;                          /* counter */

        initialize_solution(t->n,&s);
        best_cost = solution_cost(&s,t);
        copy_solution(&s,bestsol);

        for (i=1; i<=nsamples; i++) {
                random_solution(&s);
                cost_now = solution_cost(&s,t);
                if (cost_now < best_cost) {
                        best_cost = cost_now;
                        copy_solution(&s,bestsol);
                }
        }
}
```

When might random sampling do well?

- *When there are a high proportion of acceptable solutions* – Finding a piece of hay in a haystack is easy, since almost anything you grab is a straw. When solutions are plentiful, a random search should find one quickly.

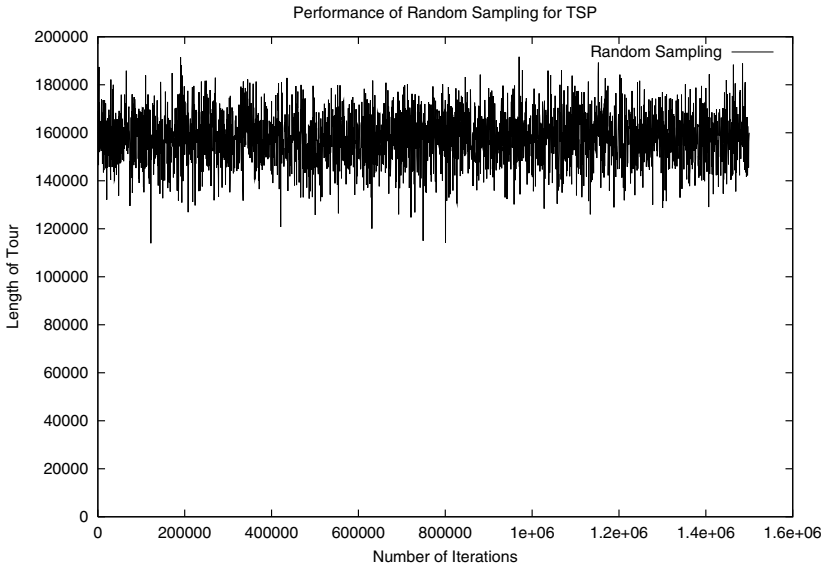

Figure 7.7: Search time/quality tradeoffs for TSP using random sampling.

Finding prime numbers is domain where a random search proves successful. Generating large random prime numbers for keys is an important aspect of cryptographic systems such as RSA. Roughly one out of every $\ln n$ integers are prime, so only a modest number of samples need to be taken to discover primes that are several hundred digits long.

- *When there is no coherence in the solution space* – Random sampling is the right thing to do when there is no sense of when we are getting *closer* to a solution. Suppose you wanted to find one of your friends who has a social security number that ends in 00. There is not much you can hope to do but tap an arbitrary fellow on their shoulder and ask. No cleverer method will be better than random sampling.

  Consider again the problem of hunting for a large prime number. Primes are scattered quite arbitrarily among the integers. Random sampling is as good as anything else.

How does random sampling do on TSP? Pretty lousy. The best solution I found after testing 1.5 million random permutations of a classic TSP instance (the capital cities of the 48 continental United States) was 101,712.8. This is more than three times the cost of the optimal tour! The solution space consists almost entirely

of mediocre to bad solutions, so quality grows very slowly with the amount of sampling / running time we invest. Figure 7.7 presents the arbitrary up-and-down movements of random sampling and generally poor quality solutions encountered on the journey, so you can get a sense of how the score varied over each iteration.

Most problems we encounter, like TSP, have relatively few good solutions but a highly coherent solution space. More powerful heuristic search algorithms are required to deal effectively with such problems.

Stop and Think: Picking the Pair

Problem: We need an efficient and unbiased way to generate random pairs of vertices to perform random vertex swaps. Propose an efficient algorithm to generate elements from the $\binom{n}{2}$ *unordered* pairs on $\{1, \ldots, n\}$ uniformly at random.

Solution: Uniformly generating random structures is a surprisingly subtle problem. Consider the following procedure to generate random unordered pairs:

```
i = random_int(1,n-1);
j = random_int(i+1,n);
```

It is clear that this indeed generates unordered pairs, since $i < j$. Further, it is clear that all $\binom{n}{2}$ unordered pairs can indeed be generated, assuming that `random_int` generates integers uniformly between its two arguments.

But are they uniform? The answer is no. What is the probability that pair $(1, 2)$ is generated? There is a $1/(n-1)$ chance of getting the 1, and then a $1/(n-1)$ chance of getting the 2, which yields $p(1, 2) = 1/(n-1)^2$. But what is the probability of getting $(n-1, n)$? Again, there is a $1/n$ chance of getting the first number, but now there is only one possible choice for the second candidate! This pair will occur n times more often than the first!

The problem is that fewer pairs start with big numbers than little numbers. We could solve this problem by calculating exactly how unordered pairs start with i (exactly $(n-i)$) and appropriately bias the probability. The second value could then be selected uniformly at random from $i+1$ to n.

But instead of working through the math, let's exploit the fact that randomly generating the n^2 *ordered* pairs uniformly is easy. Just pick two integers independently of each other. Ignoring the ordering (i.e. , permuting the ordered pair to unordered pair (x, y) so that $x < y$) gives us a $2/n^2$ probability of generating each unordered pair of distinct elements. If we happen to generate a pair (x, x), we discard it and try again. We will get unordered pairs uniformly at random in constant expected time using the following algorithm:

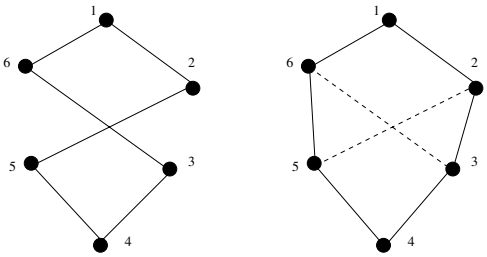

Figure 7.8: Improving a TSP tour by swapping vertices 2 and 6

```
do {
     i = random_int(1,n);
     j = random_int(1,n);
     if (i > j) swap(&i,&j);
} while (i==j);
```

■

7.5.2 Local Search

Now suppose you want to hire an algorithms expert as a consultant to solve your problem. You *could* dial a phone number at random, ask if they are an algorithms expert, and hang up the phone if they say no. After many repetitions you will probably find one, but it would probably be more efficient to ask the fellow on the phone for someone more likely to know an algorithms expert, and call *them* up instead.

A local search employs *local neighborhood* around every element in the solution space. Think of each element x in the solution space as a vertex, with a directed edge (x, y) to every candidate solution y that is a neighbor of x. Our search proceeds from x to the most promising candidate in x's neighborhood.

We certainly do *not* want to explicitly construct this neighborhood graph for any sizable solution space. Think about TSP, which will have $(n-1)!$ vertices in this graph. We are conducting a heuristic search precisely because we cannot hope to do this many operations in a reasonable amount of time.

Instead, we want a general transition mechanism that takes us to the next solution by slightly modifying the current one. Typical transition mechanisms include swapping a random pair of items or changing (inserting or deleting) a single item in the solution.

The most obvious transition mechanism for TSP would be to swap the current tour positions of a random pair of vertices S_i and S_j, as shown in Figure 7.8.

This changes up to eight edges on the tour, deleting the edges currently adjacent to both S_i and S_j, and adding their replacements. Ideally, the effect that these incremental changes have on measuring the quality of the solution can be computed incrementally, so cost function evaluation takes time proportional to the size of the change (typically constant) instead of linear to the size of the solution.

A local search heuristic starts from an arbitrary element of the solution space, and then scans the neighborhood looking for a favorable transition to take. For TSP, this would be *transition*, which lowers the cost of the tour. In a *hill-climbing* procedure, we try to find the top of a mountain (or alternately, the lowest point in a ditch) by starting at some arbitrary point and taking any step that leads in the direction we want to travel. We repeat until we have reached a point where all our neighbors lead us in the wrong direction. We are now *King of the Hill* (or *Dean of the Ditch*).

We are probably not *King of the Mountain*, however. Suppose you wake up in a ski lodge, eager to reach the top of the neighboring peak. Your first transition to gain altitude might be to go upstairs to the top of the building. And then you are trapped. To reach the top of the mountain, you must go downstairs and walk outside, but this violates the requirement that each step has to increase your score. Hill-climbing and closely related heuristics such as greedy search or gradient descent search are great at finding local optima quickly, but often fail to find the globally best solution.

```
hill_climbing(tsp_instance *t, tsp_solution *s)
{
     double cost;                /* best cost so far */
     double delta;               /* swap cost */
     int i,j;                    /* counters */
     bool stuck;                 /* did I get a better solution? */
     double transition();

     initialize_solution(t->n,s);
     random_solution(s);
     cost = solution_cost(s,t);

     do {
             stuck = TRUE;
             for (i=1; i<t->n; i++)
                     for (j=i+1; j<=t->n; j++) {
                             delta = transition(s,t,i,j);
                             if (delta < 0) {
                                     stuck = FALSE;
                                     cost = cost + delta;
                             }
```

```
                        else
                                transition(s,t,j,i);
                    }
    } while (!stuck);
}
```

When does local search do well?

- *When there is great coherence in the solution space* – Hill climbing is at its best when the solution space is *convex*. In other words, it consists of exactly one hill. No matter where you start on the hill, there is always a direction to walk up until you are at the absolute global maximum.

  Many natural problems do have this property. We can think of a binary search as starting in the middle of a search space, where exactly one of the two possible directions we can walk will get us closer to the target key. The simplex algorithm for linear programming (see Section 13.6 (page 411)) is nothing more than hill-climbing over the right solution space, yet guarantees us the optimal solution to any linear programming problem.

- *Whenever the cost of incremental evaluation is much cheaper than global evaluation* – It costs $\Theta(n)$ to evaluate the cost of an arbitrary n-vertex candidate TSP solution, because we must total the cost of each edge in the circular permutation describing the tour. Once that is found, however, the cost of the tour after swapping a given pair of vertices can be determined in constant time.

  If we are given a very large value of n and a very small budget of how much time we can spend searching, we are better off using it to do several incremental evaluations than a few random samples, even if we are looking for a needle in a haystack.

The primary drawback of a local search is that soon there isn't anything left for us to do as we find the local optimum. Sure, if we have more time we could start from different random points, but in landscapes of many low hills we are unlikely to stumble on the optimum.

How does local search do on TSP? Much better than random sampling for a similar amount of time. With over a total of 1.5 million tour evaluations in our 48-city TSP instance, our best local search tour had a length of 40,121.2—only 19.6% more than the optimal tour of 33,523.7.

This is good, but not great. You would not be happy to learn you are paying 19.6% more taxes than you should. Figure 7.9 illustrates the trajectory of a local search: repeated streaks from random tours down to decent solutions of fairly similar quality. We need more powerful methods to get closer to the optimal solution.

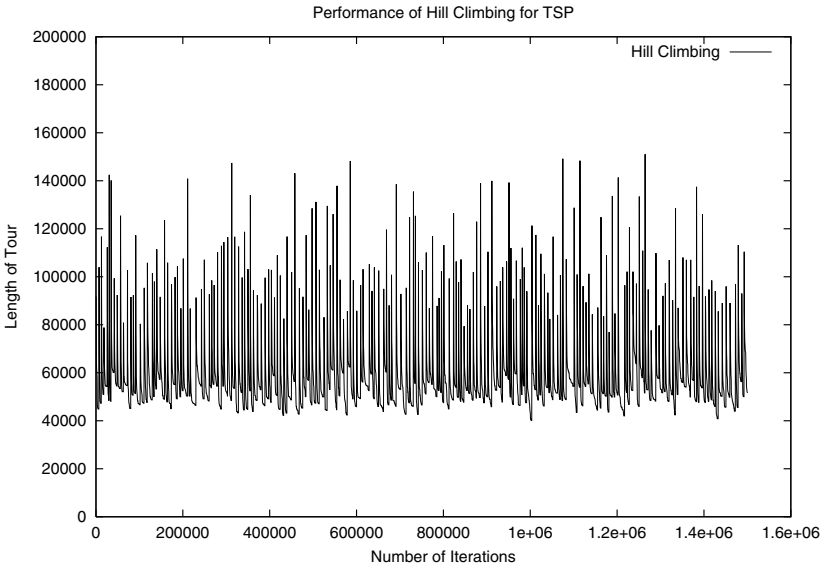

Figure 7.9: Search time/quality tradeoffs for TSP using hill climbing.

7.5.3 Simulated Annealing

Simulated annealing is a heuristic search procedure that allows occasional transitions leading to more expensive (and hence inferior) solutions. This may not sound like progress, but it helps keep our search from getting stuck in local optima. That poor fellow trapped on the top floor of a ski lodge would do better to break the glass and jump out the window if he really wanted to reach the top of the mountain.

The inspiration for simulated annealing comes from the physical process of cooling molten materials down to the solid state. In thermodynamic theory, the energy state of a system is described by the energy state of each particle constituting it. A particle's energy state jumps about randomly, with such transitions governed by the temperature of the system. In particular, the transition probability $P(e_i, e_j, T)$ from energy e_i to e_j at temperature T is given by

$$P(e_i, e_j, T) = e^{(e_i - e_j)/(k_B T)}$$

where k_B is a constant—called Boltzmann's constant.

What does this formula mean? Consider the value of the exponent under different conditions. The probability of moving from a high-energy state to a lower-energy state is very high. But, there is still a nonzero probability of accepting a

transition into a high-energy state, with such small jumps much more likely than big ones. The higher the temperature, the more likely energy jumps will occur.

```
Simulated-Annealing()
    Create initial solution S
    Initialize temperature t
    repeat
        for i = 1 to iteration-length do
            Generate a random transition from S to S_i
            If (C(S) ≥ C(S_i)) then S = S_i
            else if (e^((C(S)−C(S_i))/(k·t)) > random[0,1)) then S = S_i
        Reduce temperature t
    until (no change in C(S))
    Return S
```

What relevance does this have for combinatorial optimization? A physical system, as it cools, seeks to reach a minimum-energy state. Minimizing the total energy is a combinatorial optimization problem for any set of discrete particles. Through random transitions generated according to the given probability distribution, we can mimic the physics to solve arbitrary combinatorial optimization problems.

Take-Home Lesson: Forget about this molten metal business. Simulated annealing is effective because it spends much more of its time working on good elements of the solution space than on bad ones, and because it avoids getting trapped repeatedly in the same local optima.

As with a local search, the problem representation includes both a representation of the solution space and an easily computable cost function $C(s)$ measuring the quality of a given solution. The new component is the *cooling schedule*, whose parameters govern how likely we are to accept a bad transition as a function of time.

At the beginning of the search, we are eager to use randomness to explore the search space widely, so the probability of accepting a negative transition should be high. As the search progresses, we seek to limit transitions to local improvements and optimizations. The cooling schedule can be regulated by the following parameters:

- *Initial system temperature* – Typically $t_1 = 1$.
- *Temperature decrement function* – Typically $t_k = \alpha \cdot t_{k-1}$, where $0.8 \leq \alpha \leq 0.99$. This implies an exponential decay in the temperature, as opposed to a linear decay.
- *Number of iterations between temperature change* – Typically, 100 to 1,000 iterations might be permitted before lowering the temperature.

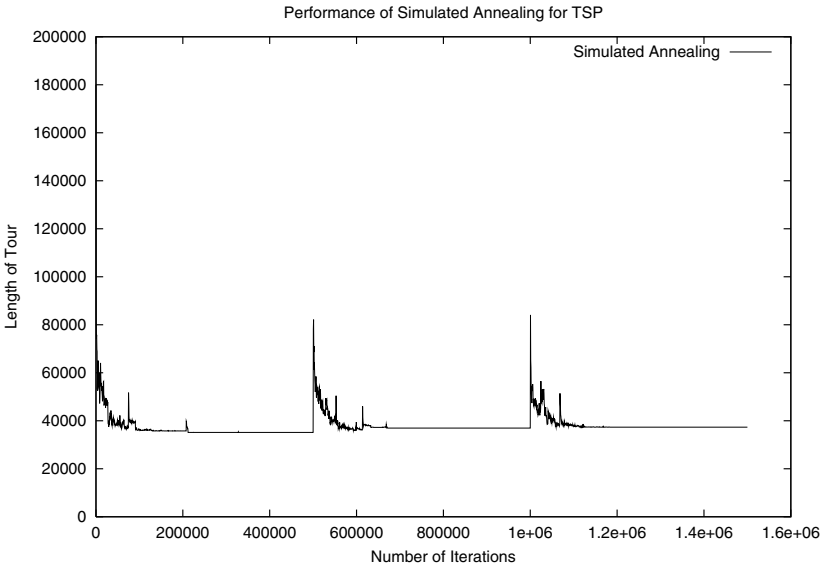

Figure 7.10: Search time/quality tradeoffs for TSP using simulated annealing

- *Acceptance criteria* – A typical criterion is to accept any transition from s_i to s_{i+1} when $C(s_{i+1}) < C(s_i)$, and also accept a negative transition whenever

  $$e^{-\frac{(C(s_i)-C(s_{i+1}))}{k \cdot t_i}} \geq r,$$

  where r is a random number $0 \leq r < 1$. The constant k normalizes this cost function so that almost all transitions are accepted at the starting temperature.

- *Stop criteria* – Typically, when the value of the current solution has not changed or improved within the last iteration or so, the search is terminated and the current solution reported.

Creating the proper cooling schedule is somewhat of a trial-and-error process of mucking with constants and seeing what happens. It probably pays to start from an existing implementation of simulated annealing, so check out my full implementation (at *http://www.algorist.com*) as well as others provided in Section 13.5 (page 407).

Compare the search/time execution profiles of our three heuristics. The cloud of points corresponding to random sampling is significantly worse than the solutions

encountered by the other heuristics. The scores of the short streaks corresponding to runs of hill-climbing solutions are clearly much better.

But best of all are the profiles of the three simulated annealing runs in Figure 7.10. All three runs lead to much better solutions than the best hill-climbing result. Further, it takes relatively few iterations to score most of the improvement, as shown by the three rapid plunges toward optimum we see with simulated annealing.

Using the same 1,500,000 iterations as the other methods, simulated annealing gave us a solution of cost 36,617.4—only 9.2% over the optimum. Even better solutions are available to those willing to wait a few minutes. Letting it run for 5,000,000 iterations got the score down to 34,254.9, or 2.2% over the optimum. There were no further improvements after I cranked it up to 10,000,000 iterations.

In expert hands, the best problem-specific heuristics for TSP can slightly outperform simulated annealing. But the simulated annealing solution works admirably. It is my heuristic method of choice.

Implementation

The implementation follows the pseudocode quite closely:

```
anneal(tsp_instance *t, tsp_solution *s)
{
  int i1, i2;                  /* pair of items to swap */
  int i,j;                     /* counters */
  double temperature;          /* the current system temp */
  double current_value;        /* value of current state */
  double start_value;          /* value at start of loop */
  double delta;                /* value after swap */
  double merit, flip;          /* hold swap accept conditions*/
  double exponent;             /* exponent for energy funct*/
  double random_float();
  double solution_cost(), transition();

  temperature = INITIAL_TEMPERATURE;

  initialize_solution(t->n,s);
  current_value = solution_cost(s,t);

  for (i=1; i<=COOLING_STEPS; i++) {
          temperature *= COOLING_FRACTION;

          start_value = current_value;

          for (j=1; j<=STEPS_PER_TEMP; j++) {
```

```
            /* pick indices of elements to swap */
            i1 = random_int(1,t->n);
            i2 = random_int(1,t->n);

            flip = random_float(0,1);

            delta = transition(s,t,i1,i2);
            exponent = (-delta/current_value)/(K*temperature);
            merit = pow(E,exponent);

            if (delta < 0)                              /*ACCEPT-WIN*/
                 current_value = current_value+delta;
            else { if (merit > flip)                    /*ACCEPT-LOSS*/
                 current_value = current_value+delta;
            else                                        /* REJECT */
                 transition(s,t,i1,i2);
            }
          }

          /* restore temperature if progress has been made */
          if ((current_value-start_value) < 0.0)
                 temperature = temperature/COOLING_FRACTION;
  }
}
```

7.5.4 Applications of Simulated Annealing

We provide several examples to demonstrate how these components can lead to elegant simulated annealing solutions for real combinatorial search problems.

Maximum Cut

The "maximum cut" problem seeks to partition the vertices of a weighted graph G into sets V_1 and V_2 to maximize the weight (or number) of edges with one vertex in each set. For graphs that specify an electronic circuit, the maximum cut in the graph defines the largest amount of data communication that can take place in the circuit simultaneously. As discussed in Section 16.6 (page 541), maximum cut is NP-complete.

How can we formulate maximum cut for simulated annealing? The solution space consists of all 2^{n-1} possible vertex partitions. We save a factor of two over all vertex subsets because vertex v_1 can be assumed to be fixed on the left side of the partition. The subset of vertices accompanying it can be represented using

a bit vector. The cost of a solution is the sum of the weights cut in the current configuration. A natural transition mechanism selects one vertex at random and moves it across the partition simply by flipping the corresponding bit in the bit vector. The change in the cost function will be the weight of its old neighbors minus the weight of its new neighbors. This can be computed in time proportional to the degree of the vertex.

This kind of simple, natural modeling is the right type of heuristic to seek in practice.

Independent Set

An "independent set" of a graph G is a subset of vertices S such that there is no edge with both endpoints in S. The maximum independent set of a graph is the largest such empty induced subgraph. Finding large independent sets arises in dispersion problems associated with facility location and coding theory, as discussed in Section 16.2 (page 528).

The natural state space for a simulated annealing solution would be all 2^n subsets of the vertices, represented as a bit vector. As with maximum cut, a simple transition mechanism would add or delete one vertex from S.

One natural cost function for subset S might be 0 if S contains an edge, and $|S|$ if it is indeed an independent set. This function ensures that we work towards an independent set at all times. However, this condition is strict enough that we are liable to move only in a narrow portion of the possible search space. More flexibility in the search space and quicker cost function computations can result from allowing nonempty graphs at the early stages of cooling. Better in practice would be a cost function like $C(S) = |S| - \lambda \cdot m_S/T$, where λ is a constant, T is the temperature, and m_S is the number of edges in the subgraph induced by S. The dependence of $C(S)$ on T ensures that the search will drive the edges out faster as the system cools.

Circuit Board Placement

In designing printed circuit boards, we are faced with the problem of positioning modules (typically, integrated circuits) on the board. Desired criteria in a layout may include (1) minimizing the area or aspect ratio of the board so that it properly fits within the allotted space, and (2) minimizing the total or longest wire length in connecting the components. Circuit board placement is representative of the kind of messy, multicriterion optimization problems for which simulated annealing is ideally suited.

Formally, we are given a collection of rectangular modules $r_1, \ldots, r_n$, each with associated dimensions $h_i \times l_i$. Further, for each pair of modules r_i, r_j, we are given the number of wires w_{ij} that must connect the two modules. We seek a placement of the rectangles that minimizes area and wire length, subject to the constraint that no two rectangles overlap each other.

The state space for this problem must describe the positions of each rectangle. To make this discrete, the rectangles can be restricted to lie on vertices of an integer grid. Reasonable transition mechanisms including moving one rectangle to a different location, or swapping the position of two rectangles. A natural cost function would be

$$C(S) = \lambda_{area}(S_{height} \cdot S_{width}) + \sum_{i=1}^{n}\sum_{j=1}^{n}(\lambda_{wire} \cdot w_{ij} \cdot d_{ij} + \lambda_{overlap}(r_i \cap r_j))$$

where λ_{area}, λ_{wire}, and $\lambda_{overlap}$ are constants governing the impact of these components on the cost function. Presumably, $\lambda_{overlap}$ should be an inverse function of temperature, so after gross placement it adjusts the rectangle positions to be disjointed.

Take-Home Lesson: Simulated annealing is a simple but effective technique for efficiently obtaining good but not optimal solutions to combinatorial search problems.

7.6 War Story: Only it is Not a Radio

"Think of it as a radio," he chuckled. "Only it is not a radio."

I'd been whisked by corporate jet to the research center of a large but very secretive company located somewhere east of California. They were so paranoid that I never got to see the object we were working on, but the people who brought me in did a great job of abstracting the problem.

The application concerned a manufacturing technique known as *selective assembly*. Eli Whitney started the Industrial Revolution through his system of *interchangeable parts*. He carefully specified the manufacturing tolerances on each part in his machine so that the parts were *interchangeable*, meaning that any legal cog-widget could be used to replace any other legal cog-widget. This greatly sped up the process of manufacturing, because the workers could just put parts together instead of having to stop to file down rough edges and the like. It made replacing broken parts a snap. This was a very good thing.

Unfortunately, it also resulted in large piles of cog-widgets that were slightly outside the manufacturing tolerance and thus had to be discarded. Another clever fellow then observed that maybe one of these defective cog-widgets could be used when all the *other* parts in the given assembly *exceeded* their required manufacturing tolerances. Good plus bad could well equal good enough. This is the idea of *selective assembly*.

"Each not-radio is made up of n different types of not-radio parts," he told me. For the ith part type (say the right flange gasket), we have a pile of s_i instances of this part type. Each part (flange gasket) comes with a measure of the degree to which it deviates from perfection. We need to match up the parts so as to create the greatest number of working not-radios as possible."

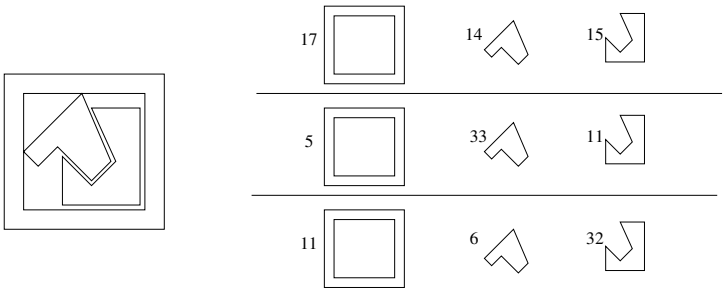

Figure 7.11: Part assignments for three not-radios, such that each had at most 50 points total defect

The situation is illustrated in Figure 7.11. Each not-radio consists of three parts, and the sum of the defects in any functional not-radio must total at most 50. By cleverly balancing the good and bad parts in each machine, we can use all the parts and make three working not-radios.

I thought about the problem. The simplest procedure would take the best part for each part type, make a not-radio out of them, and repeat until the not-radio didn't play (or do whatever a not-radio does). But this would create a small number of not-radios drastically varying in quality, whereas they wanted as many decent not-radios as possible.

The goal was to match up good parts and bad parts so the total amount of badness wasn't so bad. Indeed, the problem sounded related to *matching* in graphs (see Section 15.6 (page 498)). Suppose we built a graph where the vertices were the part instances, and add an edge for all two part instances that were within the total error tolerance. In graph matching, we seek the largest number of edges such that no vertex appears more than once in the matching. This is analogous to the largest number of two-part assemblies we can form from the given set of parts.

"I can solve your problem using matching," I announced, "Provided not-radios are each made of only two parts."

There was silence. Then they all started laughing at me. "*Everyone* knows not-radios have more than two parts," they said, shaking their heads.

That spelled the end of this algorithmic approach. Extending to more than two parts turned the problem into matching on hypergraphs,[2]—a problem which is NP-complete. Further, it might take exponential time in the number of part types just to build the graph, since we had to explicitly construct each possible hyperedge/assembly.

[2]A *hypergraph* is made up of edges that can contain more than two vertices each. They can be thought of as general collections of subsets of vertices/elements.

I went back to the drawing board. They wanted to put parts into assemblies so that no assembly would have more total defects than allowed. Described that way, it sounded like a packing problem. In the *bin packing* problem (see Section 17.9 (page 595)), we are given a collection of items of different sizes and asked to store them using the smallest possible number of bins, each of which has a fixed capacity of size k. Here, the assemblies represented the bins, each of which could absorb total defect $\leq k$. The items to pack represented the individual parts, whose size would reflect its quality of manufacture.

It wasn't pure bin packing, however, since parts came in different types. The application imposed constraints on the allowable contents of each bin. Creating the maximum number of not-radios meant that we sought a packing that maximized the number of bins which contained exactly one part for each of the m different parts types.

Bin packing is NP-complete, but is a natural candidate for a heuristic search approach. The solution space consists of assignments of parts to bins. We initially pick a random part of each type for each bin to give us a starting configuration for the search.

The local neighborhood operation involves moving parts around from one bin to another. We could move one part at a time, but more effective was *swapping* parts of a particular type between two randomly chosen bins. In such a swap, both bins remain complete not-radios, hopefully with better error tolerance than before. Thus, our swap operator required three random integers—one to select the appropriate parts type (from 1 to m) and two more to select the assembly bins involved (say from 1 to b).

The key decision was the cost function to use. They supplied the hard limit k on the total defect level for each *individual* assembly. But what was the best way to score a *set* of assemblies? We could just return the number of acceptable complete assemblies as our score—an integer from 1 to b. Although this was indeed what we wanted to optimize, it would not be sensitive to detect when we were making partial progress towards a solution. Suppose one of our swaps succeeded in bringing one of the nonfunctional assemblies much closer to the not-radio limit k. That would be a better starting point for further progress than the original, and should be favored.

My final cost function was as follows. I gave one point for every working assembly, and a significantly smaller total for each nonworking assembly based on how close it was to the threshold k. The score for a nonworking assembly decreased exponentially based on how much it was over k. Thus the optimizer would seek to maximize the number of working assemblies, and then try to drive another assembly close to the limit.

I implemented this algorithm, and then ran the search on the test case they provided. It was an instance taken directly from the factory floor. Not-radios turn out to contain $m = 8$ important parts types. Some parts types are more expensive than others, and so they have fewer available candidates to consider. The most

	suffix			
prefix	AA	AG	GA	GG
AA	$AAAA$	$AAAG$	$AAGA$	$AAGG$
AG	$AGAA$	$AGAG$	$AGGA$	$AGGG$
GA	$GAAG$	$GAAG$	$GAGA$	$GAGG$
GG	$GGAA$	$GGAG$	$GGGA$	$GGGG$

Figure 7.12: A prefix-suffix array of all purine 4-mers

constrained parts type had only eight representatives, so there could be at most eight possible assemblies from this given mix.

I watched as simulated annealing chugged and bubbled on the problem instance. The number of completed assemblies instantly climbed (1, 2, 3, 4) before progress started to slow a bit. Then came 5 and 6 in a hiccup, with a pause before assembly 7 came triumphantly together. But tried as it might, the program could not put together eight not-radios before I lost interest in watching.

I called and tried to admit defeat, but they wouldn't hear it. It turns out that the best the factory had managed after extensive efforts was only *six* working not-radios, so my result represented a significant improvement!

7.7 War Story: Annealing Arrays

The war story of Section 3.9 (page 94) reported how we used advanced data structures to simulate a new method for sequencing DNA. Our method, interactive sequencing by hybridization (SBH), required building arrays of specific oligonucleotides on demand.

A biochemist at Oxford University got interested in our technique, and moreover he had in his laboratory the equipment we needed to test it out. The Southern Array Maker, manufactured by Beckman Instruments, prepared discrete oligonucleotide sequences in 64 parallel rows across a polypropylene substrate. The device constructs arrays by appending single characters to each cell along specific rows and columns of arrays. Figure 7.12 shows how to construct an array of all $2^4 = 16$ purine (A or G) 4-mers by building the prefixes along rows and the suffixes along columns. This technology provided an ideal environment for testing the feasibility of interactive SBH in a laboratory, because with proper programming it gave a way to fabricate a wide variety of oligonucleotide arrays on demand.

However, we had to provide the proper programming. Fabricating complicated arrays required solving a difficult combinatorial problem. We were given as input a set of n strings (representing oligonucleotides) to fabricate in an $m \times m$ array (where $m = 64$ on the Southern apparatus). We had to produce a schedule of row and column commands to realize the set of strings S. We proved that the

problem of designing dense arrays was NP-complete, but that didn't really matter. My student Ricky Bradley and I had to solve it anyway.

"We are going to have to use a heuristic," I told him. "So how can we model this problem?"

"Well, each string can be partitioned into prefix and suffix pairs that realize it. For example, the string ACC can be realized in four different ways: prefix '' and suffix ACC, prefix A and suffix CC, prefix AC and suffix C, or prefix ACC and suffix ''. We seek the smallest set of prefixes and suffixes that together realize all the given strings," Ricky said.

"Good. This gives us a natural representation for simulated annealing. The state space will consist of all possible subsets of prefixes and suffixes. The natural transitions between states might include inserting or deleting strings from our subsets, or swapping a pair in or out."

"What's a good cost function?" he asked.

"Well, we need as small an array as possible that covers all the strings. How about taking the maximum of number of rows (prefixes) or columns (suffixes) used in our array, plus the number of strings from S that are not yet covered. Try it and let's see what happens."

Ricky went off and implemented a simulated annealing program along these lines. It printed out the state of the solution each time a transition was accepted and was fun to watch. The program quickly kicked out unnecessary prefixes and suffixes, and the array began shrinking rapidly in size. But after several hundred iterations, progress started to slow. A transition would knock out an unnecessary suffix, wait a while, then add a different suffix back again. After a few thousand iterations, no real improvement was happening.

"The program doesn't seem to recognize when it is making progress. The evaluation function only gives credit for minimizing the larger of the two dimensions. Why not add a term to give some credit to the other dimension."

Ricky changed the evaluation function, and we tried again. This time, the program did not hesitate to improve the shorter dimension. Indeed, our arrays started to be skinny rectangles instead of squares.

"OK. Let's add another term to the evaluation function to give it points for being roughly square."

Ricky tried again. Now the arrays were the right shape, and progress was in the right direction. But the progress was still slow.

"Too many of the insertion moves don't affect many strings. Maybe we should skew the random selections so that the important prefix/suffixes get picked more often."

Ricky tried again. Now it converged faster, but sometimes it still got stuck. We changed the cooling schedule. It did better, but was it doing well? Without a lower bound knowing how close we were to optimal, it couldn't really tell how good our solution was. We tweaked and tweaked until our program stopped improving.

Our final solution refined the initial array by applying the following random moves:

Figure 7.13: Compression of the HIV array by simulated annealing – after 0, 500, 1,000, and 5,750 iterations

---

- *swap* – swap a prefix/suffix on the array with one that isn't.
- *add* – add a random prefix/suffix to the array.
- *delete* – delete a random prefix/suffix from the array.
- *useful add* – add the prefix/suffix with the highest usefulness to the array.
- *useful delete* – delete the prefix/suffix with the lowest usefulness from the array.
- *string add* – randomly select a string not on the array, and add the most useful prefix and/or suffix to cover this string.

A standard cooling schedule was used, with an exponentially decreasing temperature (dependent upon the problem size) and a temperature-dependent Boltzmann criterion for accepting states that have higher costs. Our final cost function was defined as

$$cost = 2 \times max + min + \frac{(max - min)^2}{4} + 4(str_{total} - str_{in})$$

where max is the size of the maximum chip dimension, min is the size of the minimum chip dimension, $str_{total} = |S|$, and str_{in} is the number of strings from S currently on the chip.

How well did we do? Figure 7.13 shows the convergence of a custom array consisting of the 5,716 unique 7-mers of the HIV virus. Figure 7.13 shows snapshots of the state of the chip at four points during the annealing process (0, 500, 1,000, and the final chip at 5,750 iterations). Black pixels represent the first occurrence of an HIV 7-mer. The final chip size here is 130×132—quite an improvement over

the initial size of 192×192. It took about fifteen minutes' worth of computation to complete the optimization, which was perfectly acceptable for the application.

But how well did we do? Since simulated annealing is only a heuristic, we really don't know how close to optimal our solution is. I think we did pretty well, but can't really be sure. Simulated annealing is a good way to handle complex optimization problems. However, to get the best results, expect to spend more time tweaking and refining your program than you did in writing it in the first place. This is dirty work, but sometimes you have to do it.

7.8 Other Heuristic Search Methods

Several heuristic search methods have been proposed to search for good solutions for combinatorial optimization problems. Like simulated annealing, many techniques relies on analogies to real-world physical processes. Popular methods include *genetic algorithms*, *neural networks*, and *ant colony optimization*.

The intuition behind these methods is highly appealing, but skeptics decry them as voodoo optimization techniques that rely more on nice analogies to nature than demonstrated computational results on problems that have been studied using other methods.

The question isn't whether you can get decent answers for many problems given enough effort using these techniques. Clearly you can. The real question is whether they lead to *better* solutions with *less implementation complexity* than the other methods we have discussed.

In general, I don't believe that they do. But in the spirit of free inquiry, I introduce genetic algorithms, which is the most popular of these methods. See the chapter notes for more detailed readings.

Genetic Algorithms

Genetic algorithms draw their inspiration from evolution and natural selection. Through the process of natural selection, organisms adapt to optimize their chances for survival in a given environment. Random mutations occur in an organism's genetic description, which then get passed on to its children. Should a mutation prove helpful, these children are more likely to survive and reproduce. Should it be harmful, these children won't, and so the bad trait will die with them.

Genetic algorithms maintain a "population" of solution candidates for the given problem. Elements are drawn at random from this population and allowed to "reproduce" by combining aspects of the two-parent solutions. The probability that an element is chosen to reproduce is based on its "fitness,"—essentially the cost of the solution it represents. Unfit elements die from the population, to be replaced by a successful-solution offspring.

The idea behind genetic algorithms is extremely appealing. However, they don't seem to work as well on practical combinatorial optimization problems as simulated

annealing does. There are two primary reasons for this. First, it is quite unnatural to model applications in terms of genetic operators like mutation and crossover on bit strings. The pseudobiology adds another level of complexity between you and your problem. Second, genetic algorithms take a very long time on nontrivial problems. The crossover and mutation operations typically make no use of problem-specific structure, so most transitions lead to inferior solutions, and convergence is slow. Indeed, the analogy with evolution—where significant progress require millions of years—can be quite appropriate.

We will not discuss genetic algorithms further, to discourage you from considering them for your applications. However, pointers to implementations of genetic algorithms are provided in Section 13.5 (page 407) if you really insist on playing with them.

> *Take-Home Lesson:* I have *never* encountered any problem where genetic algorithms seemed to me the right way to attack it. Further, I have *never* seen any computational results reported using genetic algorithms that have favorably impressed me. Stick to simulated annealing for your heuristic search voodoo needs.

7.9 Parallel Algorithms

Two heads are better than one, and more generally, n heads are better than $n-1$. Parallel processing is becoming more important with the advent of cluster computing and multicore processors. It seems like the easy way out of hard problems. Indeed, sometimes, for some problems, parallel algorithms are the most effective solution. High-resolution, real-time graphics applications must render thirty frames per second for realistic animation. Assigning each frame to a distinct processor, or dividing each image into regions assigned to different processors, might be the only way to get the job done in time. Large systems of linear equations for scientific applications are routinely solved in parallel.

However, there are several pitfalls associated with parallel algorithms that you should be aware of:

- *There is often a small upper bound on the potential win* – Suppose that you have access to twenty processors that can be devoted exclusively to your job. Potentially, these could be used to speed up the fastest sequential program by up to a factor of twenty. That is nice, but greater performance gains maybe possible by finding a better sequential algorithm. Your time spent parallelizing a code might well be better spent enhancing the sequential version. Performance-tuning tools such as profilers are better developed for sequential machines than for parallel models.

- *Speedup means nothing* – Suppose my parallel program runs 20 times faster on a 20-processor machine than it does on one processor. That's great, isn't

it? If you always get linear speedup and have an arbitrary number of processors, you will eventually beat any sequential algorithm. However, a carefully designed sequential algorithm can often beat an easily-parallelized code running on a typical parallel machine. The one-processor parallel version of your code is likely to be a crummy sequential algorithm, so measuring speedup typically provides an unfair test of the benefits of parallelism.

The classic example of this occurs in the minimax game-tree search algorithm used in computer chess programs. A brute-force tree search is embarrassingly easy to parallelize: just put each subtree on a different processor. However, a lot of work gets wasted because the same positions get considered on different machines. Moving from a brute-force search to the more clever alpha-beta pruning algorithm can easily save 99.99% of the work, thus dwarfing any benefits of a parallel brute-force search. Alpha-beta can be parallelized, but not easily, and the speedups grow surprisingly slowly as a function of the number of processors you have.

- *Parallel algorithms are tough to debug* – Unless your problem can be decomposed into several independent jobs, the different processors must communicate with each other to end up with the correct final result. Unfortunately, the nondeterministic nature of this communication makes parallel programs notoriously difficult to debug. Perhaps the best example is *Deep Blue*—the world-champion chess computer. Although it eventually beat Kasparov, over the years it lost several games in embarrassing fashion due to bugs, mostly associated with its extensive parallelism.

I recommend considering parallel processing only after attempts at solving a problem sequentially prove too slow. Even then, I would restrict attention to algorithms that parallelize the problem by partitioning the input into distinct tasks where no communication is needed between the processors, except to collect the final results. Such large-grain, naive parallelism can be simple enough to be both implementable and debuggable, because it really reduces to producing a good sequential implementation. There can be pitfalls even in this approach, however, as shown in the war story below.

7.10 War Story: Going Nowhere Fast

In Section 2.8 (page 51), I related our efforts to build a fast program to test Waring's conjecture for pyramidal numbers. At that point, my code was fast enough that it could complete the job in a few weeks running in the background of a desktop workstation. This option did not appeal to my supercomputing colleague, however.

"Why don't we do it in parallel?" he suggested. "After all, you have an outer loop doing the same calculation on each integer from 1 to 1,000,000,000. I can split this range of numbers into different intervals and run each range on a different processor. Watch, it will be easy."

He set to work trying to do our computations on an Intel IPSC-860 hypercube using 32 nodes with 16 megabytes of memory per node—very big iron for the time. However, instead of getting answers, I was treated to a regular stream of e-mail about system reliability over the next few weeks:

- "Our code is running fine, except one processor died last night. I will rerun."
- "This time the machine was rebooted by accident, so our long-standing job was killed."
- "We have another problem. The policy on using our machine is that nobody can command the entire machine for more than thirteen hours, under any condition."

Still, eventually, he rose to the challenge. Waiting until the machine was stable, he locked out 16 processors (half the computer), divided the integers from 1 to 1,000,000,000 into 16 equal-sized intervals, and ran each interval on its own processor. He spent the next day fending off angry users who couldn't get their work done because of our rogue job. The instant the first processor completed analyzing the numbers from 1 to 62,500,000, he announced to all the people yelling at him that the rest of the processors would soon follow.

But they didn't. He failed to realize that the time to test each integer increased as the numbers got larger. After all, it would take longer to test whether 1,000,000,000 could be expressed as the sum of three pyramidal numbers than it would for 100. Thus, at slower and slower intervals each new processor would announce its completion. Because of the architecture of the hypercube, he couldn't return any of the processors until our entire job was completed. Eventually, half the machine and most of its users were held hostage by one, final interval.

What conclusions can be drawn from this? If you are going to parallelize a problem, be sure to balance the load carefully among the processors. Proper load balancing, using either back-of-the-envelope calculations or the partition algorithm we will develop in Section 8.5 (page 294), would have significantly reduced the time we needed the machine, and his exposure to the wrath of his colleagues.

Chapter Notes

The treatment of backtracking here is partially based on my book *Programming Challenges* [SR03]. In particular, the `backtrack` routine presented here is a generalization of the version in Chapter 8 of [SR03]. Look there for my solution to the famous *eight queens problem*, which seeks all chessboard configurations of eight mutually nonattacking queens on an 8×8 board.

The original paper on simulated annealing [KGV83] included an application to VLSI module placement problems. The applications from Section 7.5.4 (page 258) are based on material from [AK89].

The heuristic TSP solutions presented here employ vertex-swap as the local neighborhood operation. In fact, edge-swap is a more powerful operation. Each edge-swap changes two edges in the tour at most, as opposed to at most four edges with a vertex-swap. This improves the possibility of a local improvement. However, more sophisticated data structures are necessary to efficiently maintain the order of the resulting tour [FJMO93].

The different heuristic search techniques are ably presented in Aarts and Lenstra [AL97], which I strongly recommend for those interested in learning more about heuristic searches. Their coverage includes *tabu search*, a variant of simulated annealing that uses extra data structures to avoid transitions to recently visited states. Ant colony optimization is discussed in [DT04]. See [MF00] for a more favorable view of genetic algorithms and the like.

More details on our combinatorial search for optimal chessboard-covering positions appear in our paper [RHS89]. Our work using simulated annealing to compress DNA arrays was reported in [BS97]. See Pugh [Pug86] and Coullard et al. [CGJ98] for more on selective assembly. Our parallel computations on pyramidal numbers were reported in [DY94].

7.11 Exercises

Backtracking

7-1. *[3]* A *derangement* is a permutation p of $\{1, \ldots, n\}$ such that no item is in its proper position, i.e. $p_i \neq i$ for all $1 \leq i \leq n$. Write an efficient backtracking program with pruning that constructs all the derangements of n items.

7-2. *[4] Multisets* are allowed to have repeated elements. A multiset of n items may thus have fewer than $n!$ distinct permutations. For example, $\{1, 1, 2, 2\}$ has only six different permutations: $\{1, 1, 2, 2\}$, $\{1, 2, 1, 2\}$, $\{1, 2, 2, 1\}$, $\{2, 1, 1, 2\}$, $\{2, 1, 2, 1\}$, and $\{2, 2, 1, 1\}$. Design and implement an efficient algorithm for constructing all permutations of a multiset.

7-3. *[5]* Design and implement an algorithm for testing whether two graphs are isomorphic to each other. The graph isomorphism problem is discussed in Section 16.9 (page 550). With proper pruning, graphs on hundreds of vertices can be tested reliably.

7-4. *[5]* Anagrams are rearrangements of the letters of a word or phrase into a different word or phrase. Sometimes the results are quite striking. For example, "MANY VOTED BUSH RETIRED" is an anagram of "TUESDAY NOVEMBER THIRD," which correctly predicted the result of the 1992 U.S. presidential election. Design and implement an algorithm for finding anagrams using combinatorial search and a dictionary.

7-5. *[8]* Design and implement an algorithm for solving the subgraph isomorphism problem. Given graphs G and H, does there exist a subgraph H' of H such that G is isomorphic to H'? How does your program perform on such special cases of subgraph isomorphism as Hamiltonian cycle, clique, independent set, and graph isomorphism?

7-6. *[8]* In the turnpike reconstruction problem, you are given $n(n-1)/2$ distances in sorted order. The problem is to find the positions of n points on the line that give rise to these distances. For example, the distances $\{1, 2, 3, 4, 5, 6\}$ can be determined by placing the second point 1 unit from the first, the third point 3 from the second, and the fourth point 2 from the third. Design and implement an efficient algorithm to report all solutions to the turnpike reconstruction problem. Exploit additive constraints when possible to minimize the search. With proper pruning, problems with hundreds of points can be solved reliably.

Combinatorial Optimization

For each of the problems below, either (1) implement a combinatorial search program to solve it optimally for small instance, and/or (2) design and implement a simulated annealing heuristic to get reasonable solutions. How well does your program perform in practice?

7-7. *[5]* Design and implement an algorithm for solving the bandwidth minimization problem discussed in Section 13.2 (page 398).

7-8. *[5]* Design and implement an algorithm for solving the maximum satisfiability problem discussed in Section 14.10 (page 472).

7-9. *[5]* Design and implement an algorithm for solving the maximum clique problem discussed in Section 16.1 (page 525).

7-10. *[5]* Design and implement an algorithm for solving the minimum vertex coloring problem discussed in Section 16.7 (page 544).

7-11. *[5]* Design and implement an algorithm for solving the minimum edge coloring problem discussed in Section 16.8 (page 548).

7-12. *[5]* Design and implement an algorithm for solving the minimum feedback vertex set problem discussed in Section 16.11 (page 559).

7-13. *[5]* Design and implement an algorithm for solving the set cover problem discussed in Section 18.1 (page 621).

Interview Problems

7-14. *[4]* Write a function to find all permutations of the letters in a particular string.

7-15. *[4]* Implement an efficient algorithm for listing all k-element subsets of n items.

7-16. *[5]* An anagram is a rearrangement of the letters in a given string into a sequence of dictionary words, like *Steven Skiena* into *Vainest Knees*. Propose an algorithm to construct all the anagrams of a given string.

7-17. *[5]* Telephone keypads have letters on each numerical key. Write a program that generates all possible words resulting from translating a given digit sequence (e.g., 145345) into letters.

7-18. *[7]* You start with an empty room and a group of n people waiting outside. At each step, you may either admit one person into the room, or let one out. Can you arrange a sequence of 2^n steps, so that every possible combination of people is achieved exactly once?

7-19. *[4]* Use a random number generator (rng04) that generates numbers from $\{0, 1, 2, 3, 4\}$ with equal probability to write a random number generator that generates numbers from 0 to 7 (rng07) with equal probability. What are expected number of calls to rng04 per call of rng07?

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

7-1. "Little Bishops" – Programming Challenges 110801, UVA Judge 861.

7-2. "15-Puzzle Problem" – Programming Challenges 110802, UVA Judge 10181.

7-3. "Tug of War" – Programming Challenges 110805, UVA Judge 10032.

7-4. "Color Hash" – Programming Challenges 110807, UVA Judge 704.