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");
}
```

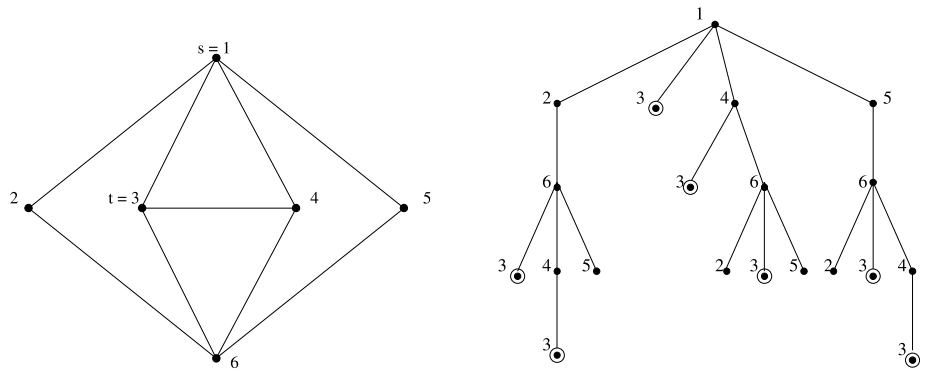

Figure 7.1: Search tree enumerating all simple s-t paths in the given graph (left).

```
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.