Backtracking

Backtracking is a systematic method to iterate through all the possible configurations of a search space. It is a general algorithm/technique which must be customized for each individual application.

In the general case, we will model our 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.

At each step in the backtracking algorithm, we start from a given partial solution, say, $a = (a_1, a_2, ..., a_k)$, and try to extend it by adding another element at the end. After extending it, we must test whether what we have so far is a solution – if so, we should print it, count it, or do what we want with it. If not, we must then check whether the partial solution is still potentially extendible to some complete solution. If so, recur and continue. If not, we delete the last element from a and try another possibility for that position, if one exists.

Implementation

The honest working code is given below. We include a global `finished` flag to allow for premature termination, which could be set in any application-specific routine.

```
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];
                        backtrack(a,k,input);
                        if (finished) return;   /* terminate early */
                }
        }
}
```

Application-Specific Routines

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

- **is_a_solution(a,k,input)** – This Boolean function tests whether the first k elements of vector a are a complete solution for the given problem. The last argument, **input**, allows us to pass general information into the routine.

- **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**.

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

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

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.

Constructing All Subsets

We can construct the 2^n subsets of n items by iterating through all possible 2^n length-n vectors of *true* or *false*, letting the ith element denote whether item i is or is not in the subset.

Using the notation of the general backtrack algorithm, $S_k = (true, false)$, and a is a solution whenever $k \geq n$.

```
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");
}
```

Finally, we must instantiate the call to `backtrack` with the right arguments.

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

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

Constructing All Permutations

To avoid repeating permutation elements, we must ensure that the ith element of the permutation is distinct from all the elements before it.

To use the notation 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;
                }
}
```

Completing the job of generating permutations 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");
}
```

```
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);
}
```

The Eight-Queens Problem

The eight queens problem is a classical puzzle of positioning eight queens on an 8×8 chessboard such that no two queens threaten each other.

Implementing a `backtrack` search requires us to think carefully about the most concise, efficient way to represent our solutions as a vector. What is a reasonable representation for an n-queens solution, and how big must it be?

To make a backtracking program efficient enough to solve interesting problems, we must prune the search space by terminating every search path the instant it becomes clear it cannot lead to a solution.

ince no two queens can occupy the same column, we know that the n columns of a complete solution must form a permutation of n. By avoiding repetitive elements, we reduce our search space to just $8! = 40{,}320$ – clearly short work for any reasonably fast machine.

The critical routine is the candidate constructor. We repeatedly check whether the kth square on the given row is threatened by any previously positioned queen. If so, we move on, but if not we include it as a possible candidate:

Implmementation

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
      int i,j;                        /* counters */
      bool legal_move;                /* might the move be legal? */

      *ncandidates = 0;
      for (i=1; i<=n; i++) {
          legal_move = TRUE;
          for (j=1; j<k; j++) {
                  if (abs((k)-j) == abs(i-a[j]))  /* diagonal threat */
                          legal_move = FALSE;
                  if (i == a[j])                  /* column threat */
                          legal_move = FALSE;
          }
          if (legal_move == TRUE) {
                  c[*ncandidates] = i;
                  *ncandidates = *ncandidates + 1;
          }
      }
}
```

The remaining routines are simple, particularly since we are only interested in counting the solutions, not displaying them:

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

        solution_count ++;
}

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

Finding the Queens

The main program is as follows:

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

        solution_count = 0;
        backtrack(a,0,n);
        printf("n=%d  solution_count=%d\n",n,solution_count);
}
```

and yields the following answers:

```
n=1  solution_count=1
n=2  solution_count=0
n=3  solution_count=0
n=4  solution_count=2
n=5  solution_count=10
n=6  solution_count=4
n=7  solution_count=40
n=8  solution_count=92
n=9  solution_count=352
n=10  solution_count=724
n=11  solution_count=2680
n=12  solution_count=14200
n=13  solution_count=73712
n=14  solution_count=365596
```

Assigned Problems

110801 (Little Bishops) – How many ways can we put down k mutally non-attacking bishops on an $n \times n$ board? Can we count the bishops without explicitly constructing every configuration?

110802 (15-Puzzle Problem) – Can you find a minimum- or near-minimum length path to solve the 15-puzzle? How do we prune quickly, and how do we eliminate duplicates?

110806 (Garden of Eden) – Given a cellular automata state t and a transition rule, does there exist a previous state s such that s goes to t?

110807 (Colour Hash) – Does there exist a sequence of moves to reorder the pieces of this puzzle? What is the right representation of the puzzle?