CBack: A Simple Tool for Backtrack Programming in C

KELD HELSGAUN
Department of Computer Science, Roskilde University,
DK-4000 Roskilde, Denmark
(email: keld@ruc.dk)

SUMMARY

Backtrack programming is such a powerful technique for problem solving that a number of languages, especially in the area of artificial intelligence, have built-in facilities for backtrack programming. This paper describes CBack, a simple, but general tool for backtrack programming in the programming language C. The use of the tool is illustrated through examples of a tutorial nature. In addition to the usual depth-first search strategy, CBack provides for the more general heuristic best-first search strategy. The implementation of CBack is described in detail. The source code, shown in its full length, is entirely written in ANSI C and highly portable across diverse computer architectures and C compilers.

KEYWORDS: Backtrack programming, Backtracking, C, Programming languages

INTRODUCTION

Backtrack programming is a well-known technique for solving combinatorial search problems.[1-4] The search is organized as a multi-stage decision process where, at each stage, a choice among a number of alternatives is to be made. Whenever it is found that the previous choices cannot possibly lead to a solution, the algorithm *backtracks*, that is to say, re-establishes its state exactly as it was at the most recent choice point and chooses the next untried alternative at this point. If all alternatives have been tried, the algorithm backtracks to the previous choice point.

Backtrack programming is often realized by recursion. A choice is made by calling a recursive procedure. A backtrack is made by returning from the procedure.[5] When a backtrack (return) is made, the programmer must take care that the program's variables are restored to their values at the time of choice (call). The programmer must ensure that sufficient information is saved in order to make this restoration possible.

Writing programs which explicitly handle their own backtracking can be difficult, tedious and error-prone. For this reason a number of high-level languages, especially the artificial intelligence languages, have been supplemented with special facilities for backtrack programming.[6] Such facilities have played an important part in the success of the logic programming language Prolog.[7] The advantage is that the programmer does not need to concern himself with the book-keeping tasks involved in backtracking, but may hand over these tasks to the underlying system and fully concentrate on solving the actual problem at hand. The utility and generality of the of the approach is also demonstrated by various efforts to extend traditional programming languages with facilities for backtrack programming, e.g. Fortran, Algol, Simula and Pascal.[8-11]

This paper describes a simple tool, CBack, for backtrack programming in the programming language C. The tool is a library consisting of a relatively small collection of program components, all written in the ANSI standard of C. CBack is highly portable and may easily be ported to most computer architectures and C compilers. The source code is included in this paper and should work under any C implementation that uses an ordinary runtime stack.

CBack is general, but at the same time quite simple to use. Depth-first is the default search strategy. If required, the user may easily obtain best-first search.

This paper demonstrates the use of CBack through examples of a tutorial nature and describes its implementation. In the Appendix is given a summary of the user facilities together with a short installation guide and the source code. The reader is assumed to have an elementary knowledge of C.[12]

The functions `Choice` and `Backtrack`

The kernel of CBack is the two functions `Choice` and `Backtrack.`

`Choice` is used when a choice is to be made among a number of alternatives. `Choice(N)`, where `N` is a positive integer denoting the number of alternatives, returns successive integer. `Choice` first returns the value 1, and the program continues. The values 2 to N are returned by `Choice` through subsequent calls of `Backtrack.`

A call of `Backtrack` causes the program to *backtrack*, that is to say, return to the most recent call of `Choice`, which has not yet returned all its values. The state of the program is re-established exactly as it was when `Choice` was called, and the next untried value is returned. All *automatic* variables of the program, i.e. local variables and register variables, will be re-established. The remaining variables, the *static* variables, are not touched. This property of static variables makes possible communication between calls of `Choice` and `Backtrack`.

On the other hand, the user may specify that static variables must be re-established too. This feature, along with other extensions, will be described later in this paper.

Calling `Choice` with a non-positive argument is equivalent to calling `Backtrack`.

During program execution several *unfinished* calls of `Choice` (calls which have not yet returned all their values) may coexist. `Backtrack` will always refer to the most recent unfinished call of `Choice`. For example, the program fragment

```
int i, j;
i = Choice(3);
j = Choice(2);
printf("i = %d, j = %d\n",i,j);
Backtrack();
```

produces the following output

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
i = 3, j = 2
```

If in calling `Backtrack,` no unfinished call of `Choice` exists, the program terminates. In order to let the user decide a course of action in this case, CBack offers the function pointer `Fiasco.` By assigning a function (without parameters) to `Fiasco`, this function will be called before the program terminates.

In the next section, the use of CBack is illustrated through a few simple examples.

ILLUSTRATIVE EXAMPLES

The 8-queens problem

A classical problem used for illustrating the backtrack programming technique is the 8-queens problem.[1-4, 8-10, 13-16] Here the task is to place eight queens on a chessboard so that no queen is under attack by another; that is, there is at most one queen at each row, column and diagonal. Figure 1 shows one of the solutions to the problem.

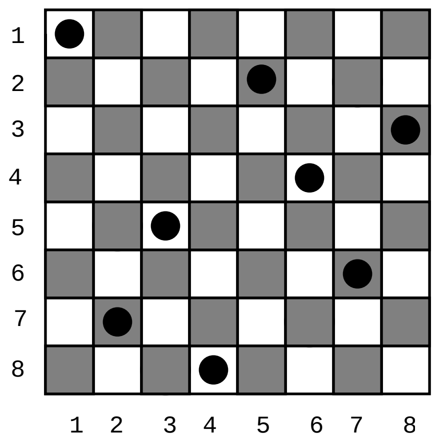


Figure 1. One solution of the 8-queens problem.

It is easy to see that in any solution there has to be one and only one queen per row and column on the board. If rows and columns are numbered from 1 to 8 (as in Figure 1), one may verify that either the sum or the difference of row and column numbers for all squares in a diagonal is constant. The sum, respectively the difference, uniquely determines a diagonal. These facts are exploited in the program Q8 in Figure 2 which solves the 8-queens problem using `Choice` and `Backtrack`.

For each row, `r`, one of the eight columns, `c`, is chosen in an attempt to place a queen on the square `(r,c)`. If a queen has already been placed in the same column or one of the two diagonals, the program backtracks and the next possible queen placement is examined. Otherwise, the queen is placed on the square by recording its row number in the three arrays `R`, `S` and `D`. These arrays represent queen placements in columns, “sum diagonals” and “difference diagonals”, respectively. For example, `R[c]` contains the row number of the queen placed in column `c`. If an array value is zero, no queen has yet been placed in the column or diagonal in question.

```
#include "CBack.h"
main()
{
    int r, c;
    int R[9]={0}, S[15]={0}, D[15]={0};   /* clear board           */

    for (r = 1; r <= 8; r++) {            /* for each row, r       */
        c = Choice(8);                    /*   choose a column, c  */
        if (R[c] || S[r+c-2] || D[r-c+7]) /*   if (r,c) under attack */
            Backtrack();                  /*   then backtrack      */
        R[c] = S[r+c-2] = D[r-c+7] = r;   /*   else place queen    */
    }
    for (c = 1; c <= 8; c++)              /* print solution        */
        printf("(%d,%d)  ",R[c],c);
}
```

Figure 2. Program 8Q

An execution of the program produced the following output:

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

corresponding to the solution pictured in Figure 1.

The program finds only one solution of the problem. All solutions may be found by simply adding a `Backtrack`-call at the end of the main program. With this addition the program will not terminate until all choice possibilities are exhausted.

In this program all variables are automatic and will be re-established when the program backtracks. It easy to see, that this re-establishment is necessary for the correctness of the program. The variable `c` constitutes an exception; it does not need to be re-established, since, after backtracking, it is assigned a value as the result of `Choice`. This unnecessary re-establishment may be avoided just by moving the declaration of `c` outside `main`, or by declaring `c` static.

All static variables are untouched by the backtracking process. Sometimes, however, it is necessary for some static variables to be backtracked anyhow. This applies for example to external variables and variables allocated by the standard library functions `calloc`, `malloc` or `realloc`. Such variables can be made backtrackable by so-called *notification*. A call `Notify(V)`, where `V` is a static variable, specifies that `V` must be backtracked. Furthermore, CBack provides the functions `Ncalloc`, `Nmalloc` and `Nrealloc.` They have the same allocation effect as `calloc`, `malloc` and `realloc`, but in addition notify the allocated storage.

Figure 3 shows a program, NQ1, which makes use of these possibilities for the solution of the N-queens problem. The output from the program is the number of solutions. The value of `N` (the number of rows or columns) is read from the standard input. The variable `Count` represents the number of solutions found. Being external, `Count` is not backtracked. The program also demonstrates the use of the function pointer `Fiasco` for obtaining a desired program reaction when all choice possibilities are exhausted (here, output of the number of solutions).

```
#include "CBack.h"
int N, Count, r, c, *R, *S, *D;

void PrintCount()
{ printf("The %d-queens problem has %d solutions.\n",N,Count); }

main() {
    printf("Number of queens: "); scanf("%d",&N);
    Fiasco = PrintCount;
    Notify(r);
    R = (int*) Ncalloc(N+1,   sizeof(int));
    S = (int*) Ncalloc(2*N-1, sizeof(int));
    D = (int*) Ncalloc(2*N-1, sizeof(int));
    for (r = 1; r <= N; r++) {
        c = Choice(N);
        if (R[c] || S[r+c-2] || D[r-c+N-1])
            Backtrack();
        R[c] = S[r+c-2] = D[r-c+N-1] = r;
    }
    Count++;
    Backtrack();
}
```

Figure 3. Program NQ1.

An execution with N = 8 produced the following (correct) output:

```
The 8-queens problem has 92 solutions.
```

The efficiency of the actual implementation of CBack is such that usually the user-program's own computation time dominates the administration time involved in calling `Choice` and `Backtrack`. However, in the solution of problems so simple as the N-queens problem, the administration time dominates. In order to remedy this situation in such cases, the tool provides two functions: `NextChoice` and `Cut`.

`NextChoice` immediately returns the next untried value of the most recent `Choice`-call; but (unlike `Backtrack`) does not restore the program's state. If however, the most recent `Choice`-call is finished, calling `NextChoice` is equivalent to calling `Backtrack.`

For example, consider the following fragment of the last program (Figure 3):

```
c = Choice(N);
if (R[c] || S[r+c-2] || D[r-c+N-1])
    Backtrack();
```

Since the evaluation of the condition does not change the state of the program, it is unnecessary to restore the program state by calling `Backtrack`. Instead the function `NextChoice` can be used as follows:

```
c = Choice(N);
while (R[c] || S[r+c-2] || D[r-c+N-1])
    c = NextChoice();
```

The `Cut` function may be used when an immediate finish of the most recent `Choice`-call is wanted, even if there still might be untried alternatives. `Cut` deletes the most recent `Choice`-call and causes a backtrack to the previous Choice-call. For example, the last `Backtrack`-call in the N-queens program may be replaced by a call of `Cut`. There is only one possible position for the last queen, so it unnecessary `to` try the remaining alternative placements for this queen.

By making the efficiency improvements of the N-queens program as mentioned above, the execution time is typically reduced by more than a factor of 4 (for N = 8).

However, the best way to increase the efficiency of a backtrack program is to make sure that the program backtracks as soon as possible (i.e. prune the search tree as much as possible).

One of the techniques for achieving this is *forward checking.*[17] Whenever a choice is made, all future choices are checked; if at any time any future choice has no possibility of leading to a solution, a backtrack is made immediately. In the solution of the N-queens problem, for example, information may be stored about which squares are under attack by queens already placed on the board. Each time a new queen is placed on the board, this information is up-dated and, if there is no queen in a row, but all squares in this row are under attack, then it is possible to backtrack at once.

This technique is used in the program NQ2 in Figure 4. The program also exploits a tech-nique called *dynamic search rearrangement*[18]; whenever a new queen is to be placed, an at-tempt is made to place it in the row which has the lowest number of squares not under attack, i.e. in the row with the fewest choice alternatives. The program uses two arrays `Q` and `A` for this purpose. `A[r]` denotes the number of free squares in row r, and `Q[r][1:A[r]]` contains their column numbers.

```
#include "CBack.h"
#define N 8
int Count;

void PrintCount()
{ printf("The %d-queens problem has %d solutions.\n", N, Count); }

main()
{
    int r, c, rf, cf, i, j, A[N+1], Q[N+1][N+1];

    Fiasco = PrintCount;
    for (r = 1; r <= N; r++) {
        A[r] = N;
        for (c = 1; c <= N; c++)
            Q[r][c] = c;
    }
    A[0] = N+1;
    for (i = 1; i <= N; i++) {
        for (r = 0, rf = 1; rf <= N; rf++)  /* find best row, r     */
            if (A[rf] && A[rf] < A[r])
                r = rf;
        c = Q[r][Choice(A[r])];             /* choose c in r        */
        A[r] = 0;
        for (rf = 1; rf <= N; rf++) {
            for (j = 1; j <= A[rf]; ) {     /* check (r,c) against  */
                cf = Q[rf][j];              /* future (rf,cf)       */
                if (cf == c || r + c == rf + cf || r - c == rf - cf) {
                    if (A[rf] == 1)
                        Backtrack();
                    Q[rf][j] = Q[rf][A[rf]--]; /* exclude (rf,cf)   */
                }
                else
                    j++;
            }
        }
    }
    Count++;
    Backtrack();                            /* find next solution   */
}
```

Figure 4. Program NQ2.

Generation of permutations

Many combinatorial problems consist of determining a permutation which satisfies one or more given constraints.

For example, a solution of the N-queens problem may be expressed as a permutation P of integers from 1 to N, where P[i] denotes the column number of the queen placed on row number `i`. The condition of P being a solution is that it does not represent a board position where two queens are in the same diagonal.

CBack makes it easy to write an algorithm that generates all permutations of the elements of a set. Suppose `P` is an array of `N` integers.

```
int P[N+1];      /* P[0] is not used */
```

Then all permutations of `P[1:N]` may be determined by the following very compact algorithm. The idea is for each `i`< `N` to swap `P[i]`(= `k`) with `P[j]`, where `P[j]` is chosen systematically among `P[i]` and its subsequent elements (`i`<=`j`<=`N`).

```
for (k = P[i = 1]; i < N; P[j] = k, k = P[++i])
    P[i] = P[j = i-1 + Choice(N-i+1)];
```

When the loop ends, `P` contains a permutation of `P`'s original contents. The next permutation may be obtained by calling `Backtrack`.

This algorithm may be used in the construction of a general function for generating permutations. The function `Permute`, shown below, generates all permutations of `P[1:n]`. Its last parameter, `Check`, is a pointer to a function for handling partial permutations `P[1:i]`, where `1`<=`i`<=`n`. For example, `Check` may point to a function which calls `Backtrack` if a partial permutation does not satisfy some given constraints. The call `Check(P,n,i)` is intended to check the last element added, `P[i]`.

```
void Permute(int P[], int n, void (*Check) (int*, int, int))
{
    int i, j, k;

    for (k = P[i = 1]; i <= n; k = P[++i]) {
        P[i] = P[j = i-1 + Choice(n-i+1)];
        P[j] = k;
        if (Check)
            Check(P, n, i);
    }
}
```

For example, the N-queens problem may be solved by passing a pointer to the following function as an argument.

```
void CheckDiagonals(int P[], int n, int i)
{
    int j;

    for (j = 1; j < i; j++)
        if (i + P[i] == j + P[j] || i - P[i] == j - P[j])
            Backtrack();
}
```

A main program which prints the number of solutions for the N-queens problem using `Permute` and `CheckDiagonals` is shown in Figure 5.

```
main()
{
    int P[N+1], i;

    for (i = 1; i <= N; i++) /* initialize p                         */
        P[i] = i;
    if (Choice(2) == 1) {    /* Choice == 2: all solutions are found */
        Permute(P, N, CheckDiagonals);
        Count++;
        Backtrack();
    }
    printf("The %d-queen problem has %d solutions.\n", N, Count);
}
```

Figure 5. Program NQ3.

Just as in the previous versions, all solutions of the problem are determined. However, many of these solutions are *symmetric* (two solutions are said to be symmetric if one of them may be transformed into the other using one or more reflections around the board's two main diagonals and two central axes).

A simple but inefficient method for avoiding symmetric solutions is to record the solutions as they are found, but discard solutions which are symmetric with any previous solution.

It is not necessary to record solutions, if a precedence relation on the set of solutions can be defined. Each time a solution is found, it is compared with its 7 symmetric solutions and discarded if any of these precedes it. An example of such a relation is the lexicographic order of the permutations which represent solutions.

It is even better, if it is possible, to discover that completion of a partial solution eventually will lead to a solution which is symmetric with a previous or a future solution. Then the partial solution may be abandoned, thus avoiding much fruitless search effort.[3,13,19]

In the function `Check` below is shown how this may be achieved. Each partial solution, `P[1:i]`, is transformed into its 7 symmetric partial solutions. If any one of these constitutes a partial solution, `Q[1:i]`, which is lexicographically less than `P[1:i]`, then Backtrack is called.

```
void Check(int P[], int n, int i)
{
    int j, k, x, y, *Q;

    for (j = 1; j < i; j++)             /* check diagonals         */
        if (i + P[i] == j + P[j] || i - P[i] == j - P[j])
            Backtrack();
    Q = (int*) malloc((i+1)*sizeof(int));
    n++;
    for (k = 1; k <= 7; k++) {         /* for each transformation  */
        for (j = 1; j <= i; j++) {
            switch(k) {                /*    (j,P[j]) --> (x,y)    */
            case 1: x = j;       y = n - P[j]; break;
            case 2: x = n - j;    y = P[j];     break;
            case 3: x = n - j;    y = n - P[j]; break;
            case 4: x = P[j];     y = j;        break;
            case 5: x = P[j];     y = n - j;    break;
            case 6: x = n - P[j]; y = j;        break;
            case 7: x = n - P[j]; y = n - j;    break;
            }
            if (x > i)
                break;
            Q[x] = y;
        }
        if (j > i) {                   /* if Q[1:i] found          */
            for (j = 1; j < i && Q[j] == P[j]; j++)
                ;
            if (Q[j] < P[j]) {         /* if Q[1:i] < P[1:i]       */
                free(Q);
                Backtrack();
            }
        }
    }
    free(Q);
}
```

Execution times of some of the versions of the N-queens programs in this paper are tabulated in Table 1. Time is measured in cpu seconds on a Sun SPARCserver 10/30. The rightmost column of the table gives both the total number of solutions and the number of non-symmetric solutions.

N	NQ1	NQ1n	NQ2	NQ3	NQ3s	Solutions
5	0.02	0.02	0.00	0.00	0.02	10 / 2
6	0.03	0.02	0.00	0.00	0.02	4 / 1
7	0.17	0.05	0.02	0.02	0.05	40 / 8
8	0.67	0.20	0.07	0.13	0.22	92 / 12
9	3.42	1.00	0.13	2.53	0.57	352 / 46
10	15.65	3.77	0.72	2.53	4.63	724 / 92
11	81.40	17.50	3.17	32.27	22.93	2680 / 341
12	479.25	91.87	14.73	174.55	129.43	14200 / 1787

Table 1. Execution time (in seconds) for the N-queens program versions
NQ1: primitive algorithm
NQ1n: as NQ1; but using NextChoice and Cut
NQ2: forward checking and search rearrangement
NQ3: permutation generation
NQ3s: as NQ3, but with elimination of symmetric solutions

Syntax analysis

Syntax analysis is an area in which backtrack programming often may be used with advantage. Suppose the language S is defined by the following syntax rules:

```
<S> ::= b<S> | <T>c
<T> ::= a<T> | a<T>b | a
```

where a, b and c are the terminal symbols and S and T are the non-terminal symbols. S is the start symbol.

Using CBack it is easy to write a parser program that can decide whether a given text string is or is not a sentence in the language. The syntax rules may be expressed by two functions `S` and `T`:

```
void S()
{
    switch (Choice(2)) {
    case 1: test('b'); S(); break;
    case 2: T(); test('c'); break;
    }
}

void T()
{
    switch (Choice(3)) {
    case 1: test('a'); T(); test('a'); break;
    case 2: test('a'); T(); test('b'); break;
    case 3: test('a'); break;
    }
}
```

The function `test` is assumed to backtrack if the current character is not the expected one, but otherwise read the next character from input. If `Sym` denotes a pointer to the last character read, then `test` can be defined by the following macro:

```
#define test(C) if (*Sym++ != C) Backtrack()
```

A simple main program which reads a text string and decides whether it is a sentence in the language is shown below. It is assumed that the string is at most 80 characters in length and ends with a period.

```
char Input[81], *Sym;

void SyntaxError() { printf("Syntax error\n"); }

main()
{
    Fiasco = SyntaxError;
    Notify(Sym);              /* Sym must be backtracked      */
    fgets(Input,81,stdin);    /* read input                   */
    Sym = Input;
    S();                      /* analyse input                */
    test('.');                /* a period must end the string */
    printf("No errors\n");
}
```

String pattern-matching

An algorithm is desired which can decide if a given text string s matches another text string p.[16,20] The string p, the pattern, may contain occurrences of a “don’t care” symbol * which can match any substring. If, for example, p is the pattern ab*c, then p matches all strings starting with ab and ending with a c.

CBack makes it relatively simple to write a function which can decide if a string `s` matches a given pattern `p`. Such a function, `Match`, is shown below.

```
  int Match(char* s, char *p)
  {
      if (Choice(2) == 2)
          return 0;          /* no match, all alternatives exhausted */
      while (*p) {           /* while more characters in pattern p   */
          if (*p == '*') {   /* current character in pattern is * ?  */
              p++;           /* move to next character in pattern    */
              s += Choice(strlen(s)) - 1;     /* try to skip in s    */
          }
          else if (*p++ != *s++) /* if match, advance p and s        */
              Backtrack();       /* otherwise backtrack              */
       }
       if (*s)                   /* if no more characters in s       */
           Backtrack();          /* then backtrack                   */
       ClearChoices();           /* clear all pending Choice-calls   */
       return 1;                 /* and return 1 (a match was found) */
   }
```

This function exploits the user facility `ClearChoices` for deleting all unfinished `Choice`-calls. In this way the program is prevented from returning to the `Match`-function in an attempt to find an alternative match in the event that `Backtrack` is called.

IMPLEMENTATION

The central implementation problem is to find a suitable method for re-establishing the program state at a `Backtrack-call`. It must be possible to re-establish the state exactly as it was at the most recent unfinished `Choice`-call.

The following three basic methods are available:

(1) Reverse execution
(2) Recording relative state changes
(3) Copying of states

In the *reverse execution* method the `Choice`-state is re-established by running backwards all actions executed since the `Choice`-call and undoing their effects. For example, the effect of the assignment `a = b + 1` may be undone by executing `a = b - 1`. After that, the execution continues forwards with a new untried value for the `Choice`-call.[2,4]

In the second method, *recording relative state changes*, all state changes relative to a given state are recorded. This given state might, for example, be the state at the most recent unfinished `Choice`-call. When `Backtrack` is called, it is possible from these recordings to re-create the program state as it was at the `Choice`-call.[14,21] This method is commonly used in Prolog implementations where the relative state changes are recorded in the form of variable bindings.[22]

Adding backtrack primitives to a programming language by means of these two methods usually requires either compiler modifications, changes of the runtime system, or the construction of special preprocessor programs.

On the other hand, the third method, *copying of states*, is very simple to implement. At every `Choice`-call, a copy of the program's state is saved. When `Backtrack` is called, it is easy to re-create the program state from the saved copy. The copies are saved in a stack. A call of `Backtrack` re-establishes the state exclusively from the top element of the stack. When a `Choice`-call returns its last alternative, the corresponding copy is popped from the stack.

This method forms the basis of the implementation of CBack. As will be described in the following, it has been possible to achieve a high degree of portability and, at the same time, a reasonable efficiency regarding both time and space.

The first implementation problem to be discussed is the resumption of `Choice` at a `Backtrack` call. How is control to be transferred from `Backtrack` to `Choice`?

Here the standard C library functions `setjmp` and `longjmp` may contribute to a solution. These two functions jointly enable this kind of non-local control transfer.[23]

Calling `setjmp` causes sufficient information to be stored so that subsequent `longjmp`-calls can transfer the control back to the `setjmp`-call. The call `setjmp(env)` saves state information (the calling environment) in its array argument, `env`, for later use by `longjmp`. The call `longjmp(env,val)` restores the state saved by the most recent invocation of `setjmp` with the corresponding `env` argument. After `longjmp` is completed, program execution continues as if the corresponding invocation of `setjmp` had just returned the value specified by `val.`

At first sight these two functions seem to solve the problem: Choice calls setjmp, and Backtrack calls longjmp. It is necessary, however, to take an important restriction in the use of setjmp and longjmp into consideration: if the function containing the invocation of setjmp has terminated, the behaviour of longjmp is undefined. Thus, when a Choice-call has returned a value, thus leading the C system to believe that it has terminated, it is not possible for longjmp to return to the setjmp-call in Choice.

This problem may be solved, however, if it is possible to re-establish the program state in such a way that Choice still seems to be executing when longjmp is called. In this way, the longjmp-call in Backtrack will apparently be made by Choice. The longjmp-call is therefore legal and causes a jump to the setjmp-call in Choice, as intended. Suppose all information about non-terminated functions is available. Then a possible solution is to take a copy of this information before a setjmp-call, and copy it back before a longjmp-call. This method has been used in the present implementation.

Below is shown the implementation of the Choice-function. If the number of alternatives, N, is greater than one, then PushState is called to save information about the actual state of the program at the top of a stack. Information about the calling environment is saved by setjmp. When all choice possibilities are exhausted, the saved state is removed from the stack by a call of PopState.

```
unsigned long Choice(const long N)
{
    if (N <= 0)
        Backtrack();
    LastChoice = 1;
    if (N == 1)
        return 1;
    PushState();
    setjmp(TopState->Environment);  /* to be resumed by Backtrack */
    if (LastChoice == N)
        PopState();
    return LastChoice;
}
```

Each state element in the stack corresponds to an unfinished Choice-call and is represented by a structure of the type State.

```
typedef struct State {
    struct State *Previous;
    unsigned long LastChoice;
    jmp_buf Environment;
    char *StackTop, *StackBottom;
} State;
```

The stack is represented by a one-way list where each element's Previous refers to its predecessor. LastChoice is used to remember the last value returned by the Choice-call. Environment is the array used by setjmp for saving information about the calling environment. In continuation of the State structure, a copy of C's runtime stack is saved; the pointers StackBottom and StackTop are used for remembering the bounds of that area. The principle is illustrated in Figure 6.

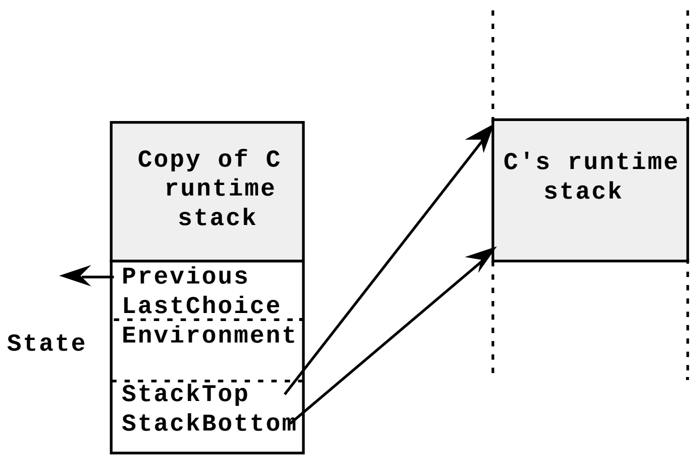


Figure 6. Principle for the use of `State`.

The state elements are dynamically allocated by `PushState` in C's runtime heap (using `malloc`), and are deallocated by PopState (using `free`).

Given the declarations

```
State *TopState = 0, *Previous;
unsigned long LastChoice;
void (*Fiasco)();
```

where `TopState` points to the topmost element of the stack of states, then the functions `Backtrack`, `PushState` and `PopState` may be sketched as follows.

```
void Backtrack(void)
{
    if (!TopState) {if (Fiasco) Fiasco(); exit(0);}
    LastChoice = ++TopState->LastChoice;
    Restore C's runtime stack from TopState;
    longjmp(TopState->Environment, 1);        /* resume Choice */
}

void PushState(void)
{
    Previous = TopState;
    TopState = (State*) malloc(sizeof(State) + StackSize);
    TopState->Previous = Previous;
    TopState->LastChoice = LastChoice;
    Store a copy of C's runtime stack in TopState;
}

void PopState(void)
{
    Previous = TopState->Previous;
    free(TopState);
    TopState = Previous;
}
```

Here remains only to be specified how the contents of C's runtime stack are saved and restored by `PushState` and `Backtrack`, respectively. Two global pointers, `StackBottom` and `StackTop`, are used for this purpose. They refer to the first and last character, respectively, in that part of C’s runtime stack which is to be saved. `StackBottom` is usually determined at the program start, whereas `StackTop` varies during program execution.

The present implementation is independent of the orientation of C’s runtime stack. But if it is assumed here, for the sake of simplicity, that the stack grows from low addresses towards high addresses (i.e. `StackBottom <= StackTop`), then `PushState` may be programmed as follows.

```
#define StackSize (StackTop - StackBottom + 1)

 void PushState(void)
 {
     char Dummy;

     StackTop = &Dummy;
     Previous = TopState;
     TopState = (State*) malloc(sizeof(State) + StackSize);
     TopState->Previous = Previous;
     TopState->LastChoice = LastChoice;
     TopState->StackBottom = StackBottom;
     TopState->StackTop = StackTop;
     memcpy((char*) TopState + sizeof(State), StackBottom, StackSize);
 }
```

The address of the character `Dummy` is used to determine the top address of C’s run-time stack. Whether it actually is the current top address is not important, as long as it is greater than the top address before the call of `PushState`. The copying of the stack contents is done by using the standard C library function `memcpy`.

The function `Backtrack`, shown below, determines the current stack top in the same way. As long as the current top address is lower than the stack top address for the program state to be re-established, `Backtrack` is called recursively. This prevents the restored stack contents from being destroyed by the subsequent call of `longjmp`.

```
void Backtrack(void)
 {
     char Dummy;

     if (!TopState) {if (Fiasco) Fiasco(); exit(0);}
     if (TopState->StackTop > &Dummy)
         Backtrack();
     LastChoice = ++TopState->LastChoice;
     StackBottom = TopState->StackBottom;
     StackTop = TopState->StackTop;
     memcpy(StackBottom, (char*) TopState + sizeof(State), StackSize);
     longjmp(TopState->Environment, 1);
 }
```

The present implementation of the tool also provides facilities for “notification” of variables. Notified variables have to be saved when `Choice` is called, and restored when `Backtrack` is called. The implementation of this feature is quite simple. Notifications are recorded in a list of structures of the type `Notification`:

```
typedef struct Notification {
    void *Base;
    size_t Size;
    struct Notification *Next;
} Notification;
```

where `Base` denotes the start address of the notified variable, `Size` its size measured in number of characters, and `Next` points to the next notification in the list.

Let `FirstNotification` point to the first notification of the list. Then the contents of notified variables can be saved by `PushState` as follows:

```
B = (char*) TopState + sizeof(State);
for (N = FirstNotification; N; B += N->Size, N = N->Next)
    memcpy(B, N->Base, N->Size);
```

and restored by `Backtrack`:

```
B = (char*) TopState + sizeof(State);
for (N = FirstNotification; N; B += N->Size, N = N->Next)
    memcpy(N-Base, B, N->Size);
```

The last implementation problem to be mentioned has to do with the determination of `StackBottom`, the address of the first character in that part of C's runtime stack which is saved when `Choice` is called.

In some C systems the stack bottom is always placed at the same storage address, independently of the program to be executed. In such systems `StackBottom` may be initialized with this address. This initialization can be done once and for all when CBack is installed.

In order to make the software independent of any assumptions about the stack bottom, the user is offered the macro `Backtracking` with the following definition:

```
#define Backtracking(S) {char Dummy; StackBottom = &Dummy; S;}
```

`Backtracking(S)`, where `S` is a sentence, determines the value of StackBottom before `S` is executed. Suppose, for example, that all automatic variables in a program are to be backtracked. In this case, the function `main` is given a new name, for example `Problem`, and `Backtracking` is called as shown below.

```
main() Backtracking(Problem())
```

GENERALIZED BACTRACKING

So far in this description of CBack, the basic search method has been *depth-first* search. At each `Choice`-call the program completes the exploration of an alternative before the next alternative is tried.

One danger of this search method is that the program may waste a lot of time in making extensive investigations in directions which turn out to be blind alleys. Therefore there is a need for a means of directing the search away from such futile explorations. A means of achieving this is *best-first* search. Here the search process focuses on those alternatives which seem to be the most promising in order to solve the problem. That alternative which currently seems to be the most promising, is examined first.

The tool may easily be extended with facilities for best-first search. To each `Choice`-call the user may attach a heuristic value, `Merit,` which expresses how promising the current state seems to be. The greater the `Merit`-value is, the closer the solution seems to be.

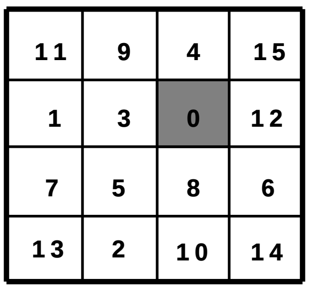
The `Merit`-value is given before the call of `Choice`, for example

```
Merit = value;
i = Choice(N);
```

`Choice` and `Backtrack` work, as described earlier; but with the change that program execution no longer proceeds with the *last* unfinished `Choice`-call, but with the *best* one, i.e. that unfinished `Choice`-call which currently has the greatest `Merit`-value attached. In case of more than one `Choice`-call having the same greatest `Merit`-value, the program continues with the most recent one of these calls.

This extension is simple to implement. Instead of saving copies of the program state in a stack, as described in the previous section, they are saved in a priority queue. Since this may be done with a few lines of code (in `Choice`), and the extension in no way affects the usual backtrack programs, the generalized backtrack facilities have been included in the actual implementation.

In the following a simple example is given of the application of the tool for best-first search, namely the so-called 15-puzzle. Fifteen numbered pieces have been placed on a quadratic board, for example as shown below.



One of the squares is empty (the hatched square containing a zero). In each move it is permitted to move a piece to the empty square from one of the neighbouring squares in the same row or column. The problem is to find a sequence of legal moves that leads from the initial board position to a goal position, for example to

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

A program which solves this problem is sketched in Figure 7. The program uses a structure type, `Position`, for representing positions which arise during the search. The positions with their piece placements, `Piece`, are saved in a tree; the pointer `Dad` refers to the previous position. The saved positions are used for printing the intervening positions between the start position and the goal position if a solution is found. And, they are used to prevent the same position from being analysed more than once. In this program, however, the search for such possible duplicate positions is confined to the ancestors of the current position.

Initialisation of the start position and the goal position might be made as follows

```
for (r = 1; r <= 4; r++) {
    for (c = 1; c <= 4; c++) {
        scanf("%d",&P[r][c]);
        if (!P[r][c]) {X = r; Y = c;}
        G[r][c] = (r != 4 || c != 4 ? (r-1)*4 + c : 0);
    }
}
```

```
#include "CBack.h"
#define P CurrentPosition.Piece
#define G GoalPosition.Piece

typedef struct Position {
    int Piece[5][5];       /* Piece[r][c]: number of piece on (r,c) */
    struct Position *Dad;  /* points to previous position           */
} Position;

Position GoalPosition;

main()
{
    Position CurrentPosition = {0}, *s;
    int r, c, X, Y;                     /* (X,Y): the empty square  */

    Initialize CurrentPosition, GoalPosition and (X,Y);

    while (memcmp(P, G, sizeof(P))) {   /* while solution not found */
        Merit = merit value;            /* compute Merit            */
        switch(Choice(4)) {             /* choose a move            */
        case 1: r = X;     c = Y + 1; break;
        case 2: r = X + 1; c = Y;     break;
        case 3: r = X;     c = Y - 1; break;
        case 4: r = X - 1; c = Y;     break;
        }
        if (r < 1 || r > 4 || c < 1 || c > 4) /* outside board ?    */
            Backtrack();
        P[X][Y] = P[r][c];              /* make move                */
        P[X = r][Y = c] = 0;
        for (s = CurrentPosition.Dad; s; s = s->Dad)
            if (!memcmp(P, s->Piece, sizeof(P)))  /* duplicate?     */
                Backtrack();
        s = (Position*) malloc(sizeof(Position));
        *s = CurrentPosition;           /* copy CurrentPosition     */
        CurrentPosition.Dad = s;
    }
    Print Solution;
}
```

Figure 7. Program for solving the 15-puzzle.

It is difficult to specify a heuristic value, `Merit`, which gives a good estimate of how close a given position is to the goal position. Below is shown a computational method which, in practice, has given reasonable execution times. The sum of squared Manhattan distances between the current and desired positions of the pieces is computed, and 100 is added for each pair of neighbouring pieces which have been exchanged in relation to their desired placements. This computed value is used with an opposite sign as the `Merit`-value.

```
Merit = 0;
for (r = 1; r <= 4; r++) {
    for (c = 1; c <= 4; c++) {
        if (P[r][c] && P[r][c] != G[r][c]) { /* if P[r][c] misplaced */
            d = abs(1+(P[r][c]-1)/4 - r) +   /* d = Manhattan        */
                abs(1+(P[r][c]-1)%4 - c);    /*     distance to goal */
            Merit -= d*d;
            if (G[r][c]) {
                if (r < 4 && P[r][c] == G[r+1][c]
                          && P[r+1][c] == G[r][c])
                    Merit -= 100;                  /* exchanged pair */
                if (c < 4 && P[r][c] == G[r][c+1]
                          && P[r][c+1] == G[r][c])
                    Merit -= 100;                  /* exchanged pair */
            }
        }
    }
}
```

Execution of the program with this heuristic value resulted in a solution of 111 moves. On a Sun SPARCserver 10/30 the execution time was only 0.22 cpu seconds. The computation of `Merit` may be changed to implicate the number of moves. By replacing the first statement, `Merit = 0`, by `Merit = -Moves`, where `Moves` denotes the number of moves, a much shorter solution was obtained, namely 63 moves. The execution time was measured as 0.33 seconds.

EVALUATION

Ease of use

CBack appears to be very easy to use. Having knowledge of only two primitives, `Choice` and `Backtrack`, the user may solve many problems of a combinatorial nature in a simple manner. A contributory reason for this simplicity is a clear and natural distinction between variables to be backtracked and variables which are not backtracked. The default is that all automatic variables (including variables in the runtime stack) are backtracked. The remaining variables are usually not backtracked, but may be made backtrackable by means of "notification".

The standard search method is depth-first search. Best-first search may be achieved just by assigning the variable `Merit` a value before calling `Choice`. This extension of the possibilities of backtracking has no influence whatsoever on programs that use depth-first search.

In the case of an error in the use of the tool, the program is terminated with a brief explanatory error message. A call of `Backtrack` when there is no unfinished `Choice`-call to return to, also causes program termination. The function pointer `Fiasco` may conveniently be used in this case to cause a desired reaction.

Extensibility

The present version of CBack contains few, but general facilities. An extension with extra facilities may be made, if required. In many cases the extension can be made by simple addition. Suppose for example that there is a need of a function, `Select`, which corresponds to `Choice` but returns integer values from any interval [`a;b`]. Select can easily be implemented by the following macro definition:

```
#define Select(a, b) ((a) - 1 + Choice((b) - (a) + 1))
```

Another example is a function, `Pick`, for the systematic choice of the elements of a one-way list. If each element is a structure of type `Element` having a pointer, `Next`, to the next element in the list, a possible, but somewhat inefficient implementation of `Pick` could be

```
Element *Pick(Element *L)
{
    return (!L->Next || Choice(2) == 1 ? L : Pick(L->Next));
}
```

Other extensions may require a certain knowledge of the implemented code. If, for example, statistics about the stack of program states are desired, it is necessary to know the internal data structure (`State`). However, the clear code makes it practicable to make such extensions, if required.

Efficiency

The efficiency of CBack is reasonably good. The execution time bottleneck arises when saving and restoring the program state (i.e. call of memcpy). If the program state is changed relatively little between calls of `Choice` and subsequent calls of `Backtrack`, then an ordinary recursive function which explicitly saves and restores the values of the variables may be somewhat faster. This is the case for several of the examples used in this paper.

The power of the tool is particularly apparent when implementing algorithms in which backtracking is an extensive and complicated matter. For example, in algorithms using dynamic data structures. What is also important is that the tool greatly simplifies the programming process. The user may concentrate on solving the actual problem at hand and focus on achieving the best possible pruning of the search tree. Moreover, it is a great advantage that the depth-first search strategy can easily be replaced by best-first search (as opposed to an algorithm based on recursion).

In connection with best-first search it should be noted that the priority queue has been implemented as an ordered one-way list. An efficiency improvement might be obtained by using a heap instead.

Another aspect is the demand on storage space. At each call of `Choice` a copy of the current program state is saved in a list. The copy remains in that list until the `Choice`-call in question has returned all its values. How much space each copy takes up depends on the user-program and the C compiler. Using the GNU C compiler on Sun SPARCserver 10/30, the fixed part (the `State` part) of each copy occupies 60 bytes. The remaining part (runtime stack and notification part) varies of course from program to program. For the program NQ1 (Figure 3) with N = 8 this part occupies 388 bytes. Thus the total storage requirement for each copy is 448 bytes. The requirement may be lowered to 332 bytes by replacing the `int`-type specification in the program with a `char`-type specification.

The storage requirement may be a problem in certain applications; but usually it is of decisive significance only when best-first search is used.

Portability

In the implementation of CBack, great emphasis has been put on its portability. In spite of its interference with C's runtime stack, the software may be ported without any changes to most C compilers. CBack has been installed and tested successfully with several C compilers, for instance Sun C, GNU C, Metrowerks C, MPW C, THINK C, VAX ULTRIX and Borland C. The code should work under any C implementation that uses an ordinary runtime stack (as opposed to a linked list of frames). A short guide in the Appendix describes how CBack is installed.

REFERENCES

1. S. W. Golomb and L. D. Baumert,
Backtrack programming.
Journal ACM **12**(4), 516-524 (1965).

2. R. W. Floyd,
Nondeterministic algorithms.
Journal ACM **14**(4), 636-644 (1967).

3. J. R. Bitner and E. M. Reingold,
Backtrack Programming Techniques.
Commun. ACM **18**(11), 651-656 (1975).

4. J. Cohen,
Non-deterministic algorithms.
Computing Surveys **11**(2), 79-94 (1979).

5. N. Wirth,
Algorithms and Data Structures.
Prentice Hall, 1986.

6. O. G. Bobrow and R. Raphael,
New programming languages for artificial intelligence research.
Computing Surveys **6**(3), 155-174 (1974).

7. I. Bratko,
Prolog programming for artificial intelligence.
2nd edn, Addison-Wesley,1990.

8. J. Cohen and E. Carton,
Non-deterministic Fortran.
Computer Journal **17**(1), 44-51 (1974).

9. P. Johansen,
Non-deterministic programming.
BIT **7**, 289-304 (1967).

10. K. Helsgaun,
Backtrack Programming with SIMULA.
Computer Journal **27**(2), 151-158 (1984).

11. G. Lindstrom,
Backtracking in a generalized control setting.
ACM Trans. Prog. Lang. Sys. **1**(1) 8-26 (1979).

12 . B. W. Kernighan and D. M. Ritchie,
The C Programming Language.
2nd edn, Prentice Hall,1988.

13. J. P. Fillmore and S. G. Williamson,
On Backtracking: A Combinatorial Description of the Algorithm.
SIAM J. Comput. **3**(1), 41-55 (1974).

14. J. A. Self,
Embedding non-determinism.
Software-Practice and Experience **5**, 221-227 (1975).

15. D. R. Hanson,
A Procedure Mechanism for Backtrack Programming.
ACM Proc. annual conf., 401-405 (1976).

16. C. Montangero, G. Pacini and F. Turini,
Two-level Control Structure for Nondeterministic Programming.
Comm. ACM **20**(10), 725-730 (1977).

17. R. M. Haralick and G. L. Elliot,
Increasing tree search efficiency for constraint satisfaction problems.
Artificial Intelligence **14**, 263-313 (1980).

18. P. W. Purdom, Jr.,
Search rearrangement backtracking and polynomial average time.
Artificial Intelligence **21**(1-2), 117-133 (1983).

19. C. A. Brown, L. Finkelstein and P. W. Purdom, Jr.,
Backtrack Searching in the Presence of Symmetry.
Lecture Notes in Computer Science **357**, 99-110 (1989).

20. R. E. Griswold and D. R. Hanson,
Language facilities for programmable backtracking.
SIGPLAN Not. **12**(8), 94-99 (1977).

21. C. Prenner, J. Spitzen and B. Wegbreit,
An implementation of backtracking for programming languages.
Proc. 27th Nat. Conf. ACM, 763-771 (1972).

22. J. A. Campbell (ed.),
Implementations of Prolog.
Halsted Press, 1985.

23. P. J. Plauger,
The Standard C Library.
Prentice Hall,1992.

APPENDIX

Summary of the user facilities

`unsigned long Choice(const long N)`
generates successive integer values from 1 to `N`. Calling `Choice` returns the value 1. The values 2 to `N` are returned through subsequent calls of `Backtrack`. Calling `Choice` with a non-positive argument is equivalent to calling `Backtrack`. If more than one `Choice`-call simultaneously may return a value, then the call having the greatest value of `Merit` will return first (in case of equality, the most recent call). After such a return, all automatic and notified variables have the same values as at the time of the initiation of the Choice-call in question.

`void Backtrack(void)`
causes the program execution to continue at the unfinished call of `Choice` having the greatest `Merit`-value (in case of equality, the most recent call). If in calling `Backtrack`, no unfinished call of `Choice` exists, then `Fiasco` is called after which the program terminates.

`void (*Fiasco)(void)`
is a pointer to a parameter free function. In case `Backtrack` is called in a situation where all `Choice`-calls are finished, the program terminates. If `Fiasco` points to a function, it is called immediately before the termination. `Fiasco` has the value zero at program start, but may be assigned a value by the user.

`long Merit`
Each call of `Choice` may be given a heuristic value by assigning the variable `Merit` a value before the call. When `Choice` and `Backtrack` are called, the program continues with the `Choice`-call having the greatest `Merit`-value (in case of equality, the most recent one). `Merit` is backtracked. Its value is zero at program start.

`unsigned long NextChoice(void)`
immediately returns the next value for the most recent `Choice`-call; but (unlike `Backtrack`) does not restore the program's state. The return value is excluded as a possible future return value for the `Choice`-call. If however, the most recent `Choice`-call is finished, calling `NextChoice` is equivalent to calling `Backtrack`.

`void Cut(void)`
deletes the most recent `Choice`-call and causes a `Backtrack` to the `Choice`-call having the greatest `Merit`-value.

`void ClearChoices(void)`
deletes all unfinished `Choice`-calls.

`void *NotifyStorage(void *Base, size_t Size)`
"notifies" a storage area, where `Base` is its start address and `Size` denotes its size (measured in characters). This causes the area to have its contents re-established each time `Backtrack` is called. By the resumption of `Choice` the area is restored to its contents at the time of call. Notification is only allowed when there are no unfinished `Choice`-calls.

In order to make it easy for the user to make notifications the macros `Notify`, `Ncalloc`, `Nmalloc` and `Nrealloc` are offered.

`void *Notify(V)`
is equivalent to the call `NotifyStorage(&V,sizeof(V)).`

void *Ncalloc(size_t N, size_t Size)
void *Nmalloc(size_t Size)
void *Nrealloc(void *P, size_t Size)
correspond to the standard C library functions `calloc`, `malloc` and `realloc`; but with the addition that the allocated storage is notified.

void RemoveNotification(void *Base)
deletes a possible notification of the storage pointed to by `Base`. Notifications may be deleted only if there are no unfinished `Choice`-calls.

void Nfree(void *P)
corresponds to the standard C library function `free`; but with the addition that a possible notification of the storage area is cleared.

void ClearNotifications(void)
deletes all notifications.

void ClearAll(void)
deletes all notifications as well as all unfinished `Choice`-calls.

void Backtracking(S)
For some C systems it is necessary to use the macro `Backtracking` in order to achieve correct execution. `Backtracking(S)`, where `S` is a statement, ensures that `S` is executed with the correct effect of the backtrack facilities. Generally the call of `Backtracking` is the only statement in `main`, while `S` is merely a function call which causes the rest of the program to be executed.

Installation guide

The software has been written in the ANSI standard for C. If the C compiler concerned does not fulfil this standard, some simple changes are necessary:

- Replace all occurrences of `void`* with `char`*.

- Rewrite all function declarations to the old form (i.e. without prototypes).

- If `<stddef.h>` is not available, then insert the following line in the beginning of `CBack.h`:

```
typedef unsigned long int size_t;
```

- If `<stdlib.h>` is not available, then use `<malloc.h>` instead.

- If the function `labs` is not available, then use `abs` instead (in the macro definition of `StackSize` in the beginning of `CBack.c`).

The code may now be compiled and tested with some simple examples (e.g. program 8Q in Figure 2). It will, however, be necessary to use the macro `Backtracking`. In the program the function name `main` is changed, for example to `Problem,` and the following line is appended to the program:

```
main() Backtracking(Problem())
```

If C’s runtime stack always has its bottom at the same address, then the `Backtracking` macro can be made superfluous (in most applications). After having executed the following small program

```
main() Backtracking(printf("%lx\n",StackBottom))
```

the address written by the program is inserted as the initial value for the variable `StackBottom` (in the beginning of `CBack.c)`.

Some C systems do not always keep the runtime stack up-to-date but keep some of the variable values in registers. This is for example the case for C systems on Sun machines. If this is the case, the macro `Synchronize` must be used; the comment characters are simply removed from the macro definition (in the beginning of `CBack.c`).

Source code of CBack.h

```
#ifndef Backtracking
#define Backtracking(S) {char Dummy; StackBottom = &Dummy; S;}
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <setjmp.h>

#define Notify(V) NotifyStorage(&V, sizeof(V))
#define Nmalloc(Size) NotifyStorage(malloc(Size), Size)
#define Ncalloc(N, Size) NotifyStorage(calloc(N,Size), (N)*(Size))
#define Nrealloc(P, Size)\
        (RemoveNotification(P),\
         NotifyStorage(realloc(P, Size), Size))
#define Nfree(P) (RemoveNotification(P), free(P))
#define ClearAll() (ClearChoices(), ClearNotifications())

unsigned long Choice(const long N);
void Backtrack(void);
unsigned long NextChoice(void);
void Cut(void);
void ClearChoices(void);

void *NotifyStorage(void *Base, size_t Size);
void RemoveNotification(void *Base);
void ClearNotifications(void);

extern void (*Fiasco)();
extern long Merit;
extern char *StackBottom;
#endif
```

Source code of CBack.c

```
#include "CBack.h"
char *StackBottom = (char*) 0xeffff347;
long Merit;
void (*Fiasco)(void);

#define StackSize labs(StackBottom - StackTop)
#define Synchronize /* {jmp_buf E; if (!setjmp(E)) longjmp(E,1);} */

typedef struct State {
    struct State *Previous;
    unsigned long LastChoice, Alternatives;
    long Merit;
    char *StackBottom, *StackTop;
    jmp_buf Environment;
} State;

typedef struct Notification {
    void *Base;
    size_t Size;
    struct Notification *Next;
} Notification;

static State *TopState = 0, *Previous, *S;
static unsigned long LastChoice = 0, Alternatives = 0;
static char *StackTop;
static Notification *FirstNotification = 0;
static size_t NotifiedSpace = 0;

static void Error(char *Msg)
{
    fprintf(stderr,"Error: %s\n",Msg);
    exit(0);
}

static void PopState(void)
{
    Previous = TopState->Previous;
    free(TopState);
    TopState = Previous;
}
```

```
static void PushState(void)
{
    char *B;
    Notification *N;

    StackTop = (char*) &N;
    Previous = TopState;
    TopState = (State*) malloc(sizeof(State)+NotifiedSpace+StackSize);
    if (!TopState)
        Error("No more space available for Choice");
    TopState->Previous = Previous;
    TopState->LastChoice = LastChoice;
    TopState->Alternatives = Alternatives;
    TopState->Merit = Merit;
    TopState->StackBottom = StackBottom;
    TopState->StackTop = StackTop;
    B = (char*) TopState + sizeof(State);
    for (N = FirstNotification; N; B += N->Size, N = N->Next)
        memcpy(B, N->Base, N->Size);
    Synchronize;
    memcpy(B,StackBottom < StackTop ? StackBottom : StackTop, StackSize);
}

unsigned long Choice(const long N)
{
    if (N <= 0)
        Backtrack();
    LastChoice = 1;
    Alternatives = N;
    if (N == 1 && (!TopState || TopState->Merit <= Merit))
        return 1;
    PushState();
    if (!setjmp(TopState->Environment)) {
        if (Previous && Previous->Merit > Merit) {
           for (S = Previous; S->Previous; S = S->Previous)
                if (S->Previous->Merit <= Merit)
                    break;
            TopState->Previous = S->Previous;
            S->Previous = TopState;
            TopState->LastChoice = 0;
            TopState = Previous;
            Backtrack();
        }
    }
    if (LastChoice == Alternatives)
        PopState();
    return LastChoice;
}
```

```
void Backtrack(void)
{
    char *B;
    Notification *N;

    if (!TopState) {
        if (Fiasco)
            Fiasco();
        exit(0);
    }
    StackTop = (char*) &N;
    if ((StackBottom < StackTop) == (StackTop < TopState->StackTop))
        Backtrack();
    LastChoice = ++TopState->LastChoice;
    Alternatives = TopState->Alternatives;
    Merit = TopState->Merit;
    StackBottom = TopState->StackBottom;
    StackTop = TopState->StackTop;
    B = (char*) TopState + sizeof(State);
    for (N = FirstNotification; N; B += N->Size, N = N->Next)
        memcpy(N->Base, B, N->Size);
    Synchronize;
    memcpy(StackBottom < StackTop ? StackBottom : StackTop,B, StackSize);
    longjmp(TopState->Environment, 1);
}

unsigned long NextChoice(void)
{
    if (++LastChoice > Alternatives)
        Backtrack();
    if (LastChoice == Alternatives)
        PopState();
    else
        TopState->LastChoice = LastChoice;
    return LastChoice;
}

void Cut(void)
{
    if (LastChoice < Alternatives)
        PopState();
    Backtrack();
}

void *NotifyStorage(void *Base, size_t Size)
{
    Notification *N;

    if (TopState)
        Error("Notification (unfinished Choice-calls)");
    for (N = FirstNotification; N; N = N->Next)
        if (N->Base == Base)
            return 0;
    N = (Notification*) malloc(sizeof(Notification));
    if (!N)
        Error("No more space for notification");
    N->Base = Base;
    N->Size = Size;
    NotifiedSpace += Size;
    N->Next = FirstNotification;
    FirstNotification = N;
    return Base;
}
```

```
void RemoveNotification(void *Base)
{
    Notification *N, *Prev = 0;

    if (TopState)
        Error("RemoveNotification (unfinished Choice-calls)");
    for (N = FirstNotification; N; Prev = N, N = N->Next) {
        if (N->Base == Base) {
            NotifiedSpace -= N->Size;
            if (!Prev)
                FirstNotification = N->Next;
            else
                Prev->Next = N->Next;
            free(N);
        }
    }
}

void ClearChoices(void)
{
    while (TopState)
        PopState();
    LastChoice = Alternatives = 0;
}

void ClearNotifications(void)
{
   while (FirstNotification)
       RemoveNotification(FirstNotification->Base);
}
```