General Paradigms for Designing Algorithms and Programs

Outline

The Backtracking Method

General Design Paradigms

Introduction

- In the theory and practice of programming there exist many problems to be solved. From all these problems certain classes of problems can be distinguished. For problems that belong to the same class, the same general solution can be applied, of course with specific adjustments that depend on the concrete problem.
- In time, several general problem solving methods have been devised. Experienced programmers master these general methods and apply them almost automatically when they have the opportunity. We will present three such methods: Greedy, Divide and Conquer and Backtracking. A scrutinizing study of these methods is recommended, leading to the understanding of the context where they can be applied and to the mastering of the methods so that they can be applied in an automated manner.

Outline

The Backtracking Method

Backtracking

Significance of the Name

- *To backtrack* means "to go back along a path you have just followed". [Source: Cambridge Dictionaries Online at http://dictionary.cambridge.org]
- The basic principle is that of returning on the followed path. Backtracking algorithms exhaustively explore the solutions space on all possible paths. If at some point on the current exploration path it becomes obvious that there is no chance to reach a solution, than one step back is taken and another path is explored.
- The Backtracking method consists in repeatedly trying to find a solution, with the possibility of stepping back in case of failures.

Backtracking

Principles

- In general the solution to a Backtracking problem can be represented as an array $X = (x_0, x_1, \ldots, x_{n-1})$, $X \in S = S_0 \times S_1 \times \ldots \times S_{n-1}$, where the sets $S_0, S_1, \ldots, S_{n-1}$ are finite sets and $x_i \in S_i$.
- For each concrete problem certain relations between the elements of the X array must be respected. These relations are called *internal conditions*. The set S represents the *solutions space*. Those solutions that that satisfy the internal conditions are the *valid solutions*.
- We express the internal conditions through a logical function $Solution(x_0, x_1, \ldots, x_{n-1})$. An element $X = (x_0, x_1, \ldots, x_{n-1}) \in S$ is a valid solution if the function *Solution* applied to X returns *true*.

Backtracking

Principles

- The end purpose of a Backtracking algorithm can be either to find one solution, either to find all solutions. If only one solution is searched, it can be any solution, or an optimal solution according to some optimization criteria (minimization or maximization).
- A simple way of finding solutions is to generate all possible solutions and check which ones satisfy the internal conditions. But this method requires the longest execution time and is rarely applied in practice.

Backtracking

Principles

- An efficient Backtracking algorithm will avoid generating all possible solutions. To achieve this, the elements of array X will be assigned values one by one, in order. After assigning a value to x_k, a check is performed to see if the sequence $(x_0, \ldots, x_k)$ satisfies a set of *continuation conditions*. Only if these conditions are satisfied the next element x_{k+1} will be assigned a value.
- The continuation conditions are expressed through a function $Continue(x_0, \ldots, x_k)$.
- If at some point the continuation conditions are not satisfied, it means that there is no chance to find a solution that includes the elements $(x_0, \ldots, x_k)$. In this case we assign the next value from S_k to x_k. If all values in S_k are exhausted, then k is decreased with one and the same process is applied to x_{k-1}.

Backtracking

Pseudocode

$k \leftarrow 0$
while $k \geq 0$ **do**
  repeat
    $x_k \leftarrow$ next element chosen from S_k
  until $Continue(x_0, \ldots, x_k)$ or (no more elements to choose from S_k)
  if $Continue(x_0, \ldots, x_k)$ **then**
    if $k == n - 1$ **then**
      if $Solution(x_0, \ldots, x_{n-1})$ **then**
        print solution
      end if
    else
      $k \leftarrow k + 1$
    end if
  else
    $k \leftarrow k - 1$
  end if
end while

Backtracking

Characteristics

In conclusion:

- The name of the method comes from the steps that are reversed along search paths in case of failures.
- Continuation conditions are derived from internal conditions.
- Optimal selection of continuation condition can significantly reduce the computation effort and thus increase the speed of the program.
- This process of repeated trial and step back in case of failure (followed by picking a different continuation path and retrying) is naturally expressed in recursive manner.

Backtracking

Recursive Implementation - Pseudocode

- We assume that all sets S_k are stored as arrays and can be indexed in the form $S_k[i]$.

function $try(k)$
  for $i \in \{0, \ldots, |S_k|\}$ **do**
    choose element $S_k[i]$
    if element $S_k[i]$ is acceptable **then**
      add element $S_k[i]$ to solution
      if $k < n - 1$ **then**
        call $try(k + 1)$
      else
        print solution
      end if
      remove elements $S_k[i]$ from solution
    end if
  end for

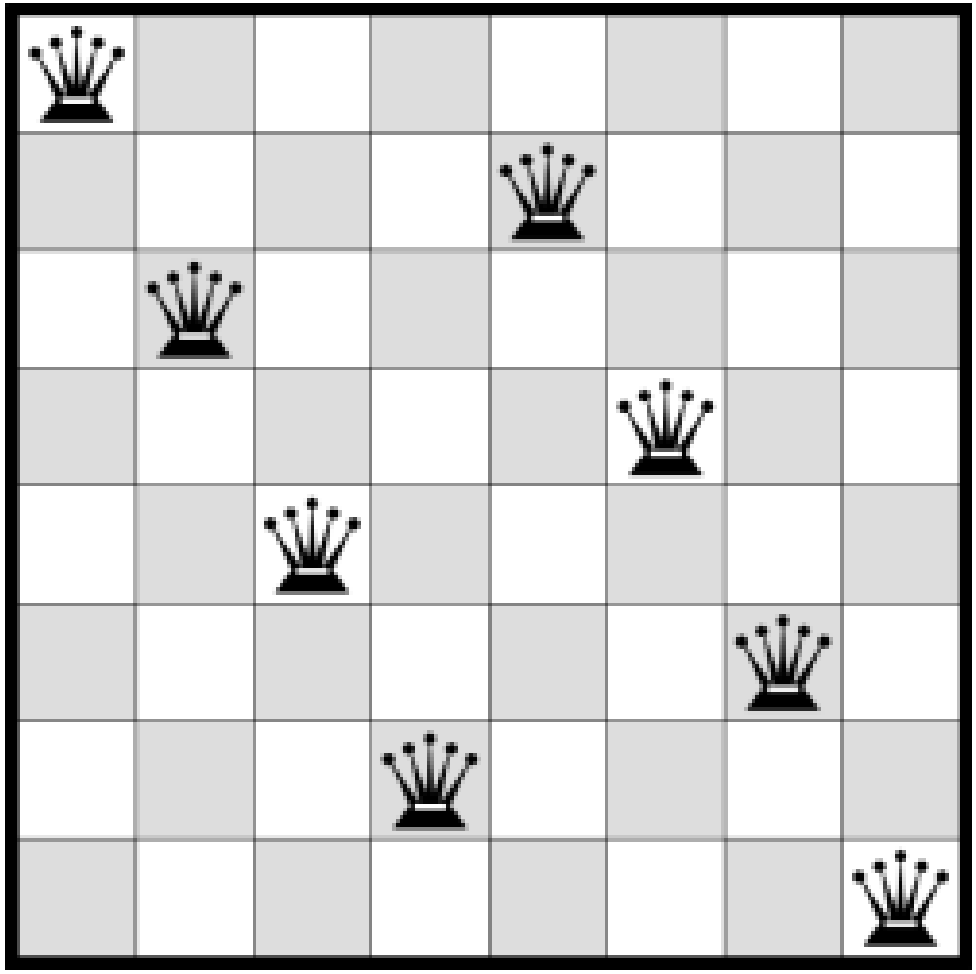
Backtracking

The Eight Queens Problem

- It is requested to place eight queens on a 8×8 chess board so that there are no two queens which attack each other. In the chess game the queen attacks on row, on column and on diagonals.
- This problem was investigated by Carl Friedrich Gauss in 1850, who did not solve it completely. Up to the present day there is no satisfying analytical solution to the problem. However, it can be solved by repeated trials, with a great deal of patience and accuracy (which are native qualities of a computer).
- The problem has 92 solutions, out of which, due to symmetry, only 12 are different.
- The problem can be easily extended to n queens to be placed on a $n \times n$ board.

Backtracking

The Eight Queens Problem



Backtracking

The Eight Queens Problem - Solution

- On any row or column of the board there will be exactly one queen. The board is visited line by line ($k \in \{0, \ldots, 7\}$), and on each line column by column ($i \in \{0, \ldots, 7\}$), and the queens are placed in those squares which are not attacked by any of the previously placed queens. Backtracking will be used for visiting the board.
- Since on every line of the board there needs to be exactly one queen, we can encode a solution in an array $C = (c_0, \ldots, c_7)$, where c_k stored the column where the queen from line k is placed ($c_k \in S_k = \{0, \ldots, 7\}$).
- The space of possible solutions is the set $S = S_0 \times S_1 \times S_2 \times S_3 \times S_4 \times S_5 \times S_6 \times S_7$.
- The internal conditions result from the rules of chess and consist in the fact that two queens cannot be placed in the same column or on the same diagonal.

Backtracking

The Eight Queens Problem - Internal Conditions

- The *Solution* function must check that there are not two queens which are placed in the same column or attack each other over a diagonal.
- The check is simple. First we must check if in the array $(c_0, \ldots, c_7)$ there are duplicate elements. If there are duplicate elements it means we have queens placed in the same column.
- Then we need to check that $\forall i, k \in \{0, \ldots, 7\}$ we have $|i - k| \neq |c_i - c_k|$. This is the condition for two queens not attacking each other over a diagonal.
- Similar checks need to be performed in the *Continue* function.

Backtracking

The Eight Queens Problem - Implementation

```
/*
 * The size of the chess board.
 */
#define N 8

/*
 * Special value used for marking a row which does not
 * contain any queen.
 */
#define INVALID -1

int main(void)
{
  /*
   * c[k] stores the column where the queen in row k is
   * placed.
   */
  int c[N];
```

Backtracking

The Eight Queens Problem - Implementation

```
int k, i, can_continue, attacks;

/*
 * We keep track of the number of solutions found.
 */
int count = 0;

/*
 * Initially we don't have any queen placed on any row.
 */
for (i = 0; i < N; i++)
  c[i] = INVALID;

/*
 * k shows the current row where we are trying to place
 * a queen.
 */
k = 0;
```

Backtracking

The Eight Queens Problem - Implementation

```
while (k >= 0) {
  /*
   * Choose values from the set S[k] and for each value
   * evaluate the "Continue" function (implemented
   * here, not in separate function).
   */
  do {
    /*
     * If element c[k] is INVALID, it means we have not
     * tried to place a queen on this row k. We start
     * with the first column.
     */
    if (c[k] == INVALID)
      c[k] = 0;
    /*
     * Otherwise we continue from the last column where
     * we tried to place the queen.
     */
    else
      c[k]++;
```

Backtracking

The Eight Queens Problem - Implementation

```
/*
 * If we are still on the board, we start
 * evaluating "Continue".
 */
if (c[k] < N) {
  /*
   * We check if the newly placed queen on this row
   * k attacks any queen placed on rows smaller
   * than k.
   */
  attacks = 0;
  for (i = 0; !attacks && (i < k); i++) {
    /*
     * Check if queens i and k attack each other on
     * column.
     */
    if (c[i] == c[k])
      attacks = 1;
```

Backtracking

The Eight Queens Problem - Implementation

```
        /*
         * Check if queens i and k attack each other on
         * diagonals.
         */
        else if (abs(i - k) == abs(c[i] - c[k]))
          attacks = 1;
      }

      /*
       * "Continue" return true if the new queen does
       * not attack any previously placed queen.
       */
      can_continue = !attacks;
    }

    /*
     * Off the board there is no chance for a solution.
     */
    else
      can_continue = 0;
```

Backtracking

The Eight Queens Problem - Implementation

```
}
while (!can_continue && (c[k] < N));
/*
 * We will stop the loop above either when we find an
 * element for which "Continue" returns true, either
 * when there is nothing left to choose from S[k].
 */


/*
 * If "Continue" returned true, we consider the queen
 * placed and we move on.
 */
if (can_continue) {
```

Backtracking

The Eight Queens Problem - Implementation

```
    /*
     * If we placed all queens, then we have a solution
     * to the problem. There is no need to evaluate the
     * "Solution" function, because the "Continue"
     * functions guarantee that the solution is
     * correct.
     */
    if (k == N - 1) {
      for (i = 0; i < N; i++)
        printf("%d ", c[i]);
      printf("\n");
      count++;
    }

    /*
     * If we still have queens to place, we move ahead
     * to the next queen.
     */
    else
      k++;
}
```

Backtracking

The Eight Queens Problem - Implementation

```
        /*
         * If "Continue" returned false, it means we could
         * not find another way of placing the queen on row
         * k. This we remove the queen from the row (set c[k]
         * to INVALID) and we step back and continue with the
         * previous queen (k--).
         */
        else {
          c[k] = INVALID;
          k--;
        }
      }

      printf("%d solutions.\n", count);
      return 0;
}
```

Backtracking

The Eight Queens Problem - Recursive Implementation

```
void queens(int k)
{
  int i;
  for (i = 0; i < N; i++) {
    c[k] = i;                    /* Place queen on row k in column
                                  * i. */
    if (can_continue(k)) {       /* If it does not attack
                                  * other queens. */
      if (k < N - 1)
        queens(k + 1);           /* If there are queens left to
                                  * place, move on to the next. */
      else
        print_solution();        /* If all queens were
                                  * placed, print
                                  * solution. */
    }
    c[k] = INVALID;              /* Remove queen from row k. */
  }
}
```

Backtracking

The Knight's Tour Problem

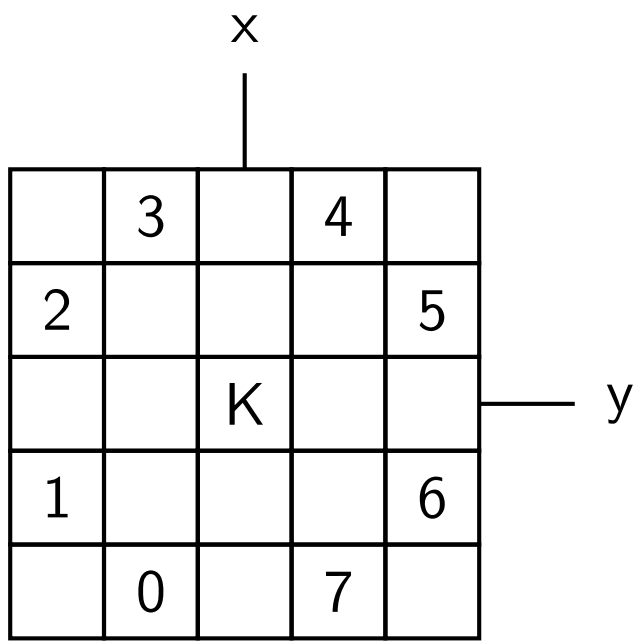
- Given a chess board of size $n \times n$, find a path that takes a knight from an initial position (i, j) and moves him exactly once through all squares on the board.
- This problem can be solved with a Backtracking algorithm. The fundamental operation of the algorithm is to detect at each step if there is a possibility to make another move, and if there is no such possibility to step back to the previous move.
- If at some step there is no possibility to make further moves, the last steps are reverted until another move becomes possible.
- For a board of size $n \times n$ we will store the path in an array c of length $n \times n$. Every element of this array will be an array of two elements that will store the line and the column of one position of the knight: **int** $c[N*N][2]$;

Backtracking

The Knight's Tour Problem

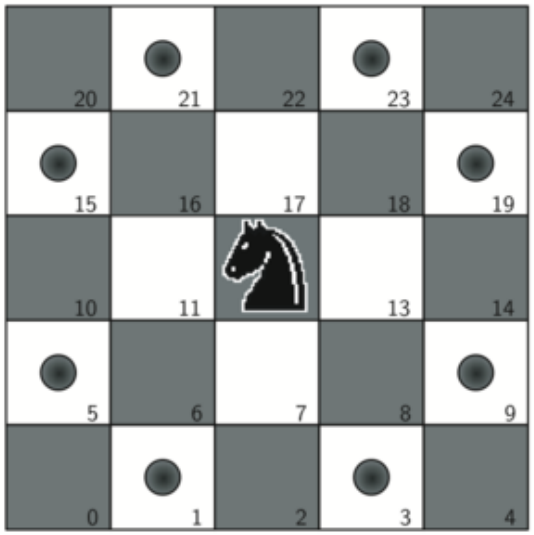
- The new coordinates of a move are computed starting from the current coordinates, by adding fixed values of ± 1 and ± 2, according to the eight possibilities of moving a knight.

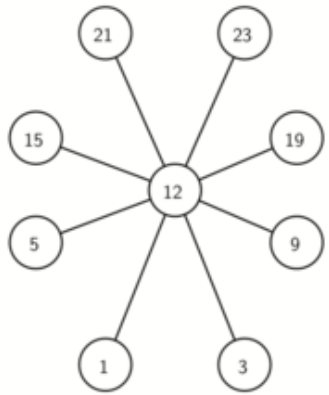
```
int dx[8] = { -1, -2, -2, -1, 1, 2, 2, 1 };
int dy[8] = { -2, -1, 1, 2, 2, 1, -1, -2 };
```



Backtracking

The Knight's Tour Problem





Backtracking

The Knight's Tour Problem

38	35	62	25	60	23	10	7
63	26	37	34	11	8	59	22
36	39	28	61	24	57	6	9
27	64	33	40	5	12	21	58
50	29	4	13	48	41	56	19
1	14	49	32	53	20	47	44
30	51	16	3	42	45	18	55
15	2	31	52	17	54	43	46

One possible solution for covering the Knight Tour

Backtracking

The Knight's Tour Problem - Implementation

```
int c[N * N][2];          /* The path of the knight on the
                           * board. An array of pairs of
                           * coordinates (line,column). */

int count = 0;            /* The number of paths found. */

void back(int step)
{
  int i, j, can_continue;

  /*
   * If the entire board is covered, print the path.
   */
  if (step == N * N) {
    for (i = 0; i < step; i++)
      printf("(%d,%d) ", c[i][0], c[i][1]);
    printf("\n");
    count++;
  }
```

Backtracking

The Knight's Tour Problem - Implementation

```
/*
 * If there are still squares to cover, try to make
 * another move.
 */
else {
  /*
   * Try all eight possible moves of the knight.
   */
  for (i = 0; i < 8; i++) {
    c[step][0] = c[step - 1][0] + dy[i];
    c[step][1] = c[step - 1][1] + dx[i];
```

Backtracking

The Knight's Tour Problem - Implementation

```
/*
 * Check if the move is valid. First condition is
 * to be on the board.
 */
if ((c[step][0] >= 0) && (c[step][0] < N) &&
    (c[step][1] >= 0) && (c[step][1] < N)) {
  /*
   * The second condition is to not have been in
   * this square before.
   */
  can_continue = 1;
  for (j = 0; can_continue && (j < step); j++) {
    if ((c[j][0] == c[step][0]) &&
        (c[j][1] == c[step][1]))
      can_continue = 0;
  }
```

Backtracking

The Knight's Tour Problem - Implementation

```
            /*
             * If the move is valid, remember it and continue
             * * to a next move.
             */
            if (can_continue)
              back(step + 1);
          }
        }
      }
    }

    int main(void)
    {
      int i, j;
```

Backtracking

The Knight's Tour Problem - Implementation

```
    /*
     * Start from each square on the board and find paths
     * from it.
     */
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++) {
        /*
         * The first move will be in square (i,j).
         */
        c[0][0] = i;
        c[0][1] = j;
        /*
         * We invoke the recursive function to find the
         * next moves.
         */
        back(1);
      }

    printf("%d paths.\n", count);
    return 0;
}
```

Further Reading

Algorithms, Robert Sedgewick, Addison - Wesley P. C., 1988.

- Chapter 39.

Algorithms in C, Robert Sedgewick, Addison - Wesley P. C., 1998.

- Chapters 5 and 44.

Data Structures and Algorithms, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Adisson - Wesley P.C., 1985.

- Sections 10.1, 10.2 and 10.4.

Introduction to Algorithms, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, Clifford Stein, MIT Press, Mc Graw - Hill B. C. 2002.

- Chapters 16 and 23. Section 35.2.