

Proseminar
Algorithms and data structures
(Prof.Dr.Brandenburg,SS2001)

Backtracking

C.JenniferNeumüller

26.07.01

Backtracking

The backtracking algorithm is a very simple method, which can easily be adapted and therefore is used for solving a wider range of problems. However, backtracking is not very efficient, so it is mainly used for n -complete problems, for which no better algorithm is known.

The basic idea is a trial and error method. Whenever the algorithm hits a dead end it backtracks one step and tries a different path.

The solution space of these problems can be represented as a tree structure, on which backtracking performs an "intelligent" depth-first-search, thereby searching the trees systematically for one or all answers to the problem.

The algorithm builds up the solution vector one component at a time, and checks at each step via bounding functions, if the current vector can in any way lead to a solution. If this is no longer possible, the algorithm will not continue with this subtree, but backtracks one step to the previous node and tries a different path from there. In this way it generates as few nodes as possible without missing a solution.

When using backtracking, the answer to the problem should be representable as a vector, which describes the path from the root to a leaf in the tree representing the solution space. Every element of the vector has to fulfil certain explicit constraints, i.e. is element of a given set. For example, every element of the vector has to be greater than zero. All vectors whose elements satisfy the explicit constraints form the solution space.

The implicit constraints of the problem determine, whether the vector found satisfies the criterion function and therefore is a solution or not. They state, how the elements of the vector must relate to each other.

Modifications of the criterion function are used as bounding functions, so that the algorithm generates as few nodes as possible without missing any of the solutions to the problem in order to minimize the running time.

The difficulty with good bounding functions is, that, since they are implemented at each step, they may take more time than is saved by pruning the tree.

So sometimes a decision has to be made, whether it is more efficient to use better bounding functions and generate less nodes, or to use less effective bounding functions that take less time to calculate, but then have to build up more and larger subtrees.

Then -queens problem

How can queens be placed on an $n \times n$ chessboard without any of them attacking?

The brute force method would be to search through all possible positions of n queens on $n \times n$ squares. Then the solution space would consist of $n!$ possibilities, i.e.:

$\frac{(n^2)!}{n!(n^2 - n)!}$ possibilities.

In order to reduce this solution space, one examines when two queens attack each other:

They attack when they are in the same row, column or diagonal. It can now be assumed without loss of generality, that the i^{th} queen is placed in the i^{th} row, thereby avoiding that two queens can be in the same row.

Each queen is assigned a different column, so the only thing left to check is whether two queens are in the same diagonal.

The rows are now numbered from the upper to the lower row and the columns from the left to the right. Hence the upper left corner has the coordinates $(1, 1)$ and the lower left corner $(n, 1)$.

With this numeration, two queens with the coordinates (i, j) and (k, l) are in the same descending diagonal if $i - j = k - l$

They are in the same ascending diagonal if $i + j = k + l$

The equation for descending diagonal can also be written as $i - k = j - l$,

and the equation for ascending diagonal as $i - k = l - j$

so in order to check whether two queens are in the same diagonal, no matter in which direction, it is sufficient to calculate, if $|i - k| = |j - l|$

The solution vector can be represented as a tuple of the numbers on n ton. The i^{th} number k in the tuple signifies that the i^{th} queen is placed in the k^{th} column, i.e. has the coordinates (i, k) .

At this point the solution space has been reduced to all permutations of the tuple of the numbers on n ton, so the size of the solution space is $n!$

Algorithm for n -queens problem:

```

private static int [] row = new int[8];
private static boolean [] column = new boolean[8];
private static boolean [] ascdiag = new boolean[15];
private static boolean [] descdiag = new boolean[15];

public static void place( int i ) {
    for( int j = 0; j < 8; j++ )
        if( column[j] && descdiag[i-j+7] && ascdiag[i+j] ) {
            row[i] = j;
            column[j] = false;          descdiag[i-j+7] = false;
            ascdiag[i+j] = false;
            if( i < 7 ) place( i + 1 ); else displayArray( row );
            column[j] = true;
            descdiag[i-j+7] = true;
            ascdiag[i+j] = true;
        }
    }
}

```

The algorithm places the i^{th} queen in the i^{th} row as far left as possible, without two queens attacking. If that doesn't yield a solution, the algorithm backtracks and places the queen in the next possible square to the right, where it cannot attack another queen. This provides a systematic search of the solutions space.

The algorithm uses arrays of booleans, in order to determine, if a column, an ascending or a descending diagonal is already occupied by a queen. When the 3rd column is for example already occupied, the array `column` would return false for `column[2]` (column[2] and not column[3]) because arrays start with the 0 position, not with the 1st).

The descending diagonals are calculated by $-j + 7$, because otherwise the descending diagonal would be numbered from -7 to 7 , now they are numbered from 0 to 14.

Subsetsum

The problem calls for finding all subsets of a given set of n positive weights w_i , whose elements sum up to a certain amount m .

The brute force approach would be to construct all subsets and then to compare the sum of each subset with the amount m .

The solutions space consists of 2^n possibilities, i.e. the power of the set of all subsets.

It is more convenient to represent the subsets as tuples of the length n containing only 1s and 0s. If the i^{th} element of the tuple is a one, this means that the i^{th} element of the set is also an element of the subset, if this is a zero, the i^{th} element of the set is not in the subset.

So the solution is a vector

$$x = (x_1, \dots, x_n), \quad x_i \in \{0, 1\} \forall i \in [1 : n]$$

The sum of each subset can now be calculated as:

$$\sum_{i=1}^n w_i x_i$$

The object is to generate as few nodes of the tree as possible in order to save time, so the algorithm should be able to recognize at an early stage whether the vector can in any way lead to a solution.

Since only positive weights are given, the vector being built cannot lead to a solution, if the sum of all elements that are already determined as part of the subset is greater than m .

In case the current sum plus the sum of all elements left is smaller than m , the current vector cannot solve this problem either.

So the bounding functions for sets of positive weights are:

$$\sum_{i=1}^k w_i x_i \leq m \quad \text{and} \quad \sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m$$

If this set of weights is also sorted, the first bounding function can be strengthened.

Whenever the current sum plus the next element is now greater than m , there is also no use in continuing with this vector.

One can now use

$$\sum_{i=1}^k w_i x_i + w_{k+1} \leq m \quad \text{and} \quad \sum_{i=1}^k w_i x_i + \sum_{i=k+1}^n w_i \geq m$$

as bounding functions.

This algorithm for subset sum is implemented for sorted lists of positive weights only, the solution is printed as a tuple of ones and zeros. It uses the previous two criteria as bounding functions:

```
public static final int SIZE = 6;
public static final int LIMIT = 30;
public static int k = 0;
public static int s = 0;
public static int r = 0;
public static int[] tuple = new int[SIZE];
public static int[] input = {5, 10, 12, 13, 15, 18};
public static void SumOfSub (int s, int k, int r) {
    tuple[k] = 1;
    if( s + input[k] == LIMIT ) {
        output();
    } else {
        if((s+input[k]+input[k+1] <= LIMIT)&&(s+r >=m))
            SumOfSub( s + input[k], k + 1, r - input[k] );
        if(( s+r-input[k] >= LIMIT)&&(s+input[k+1] <= LIMIT )) {
            tuple[k] = 0;
            SumOfSub (s, k + 1, r - input[k]);
        }
    }
}
```

}
 }
 }

In the beginning r is calculated as the sum of all elements of the set, s is zero.

The algorithm changes r and s at each step, such that r and s are always:

$$r = \sum_{i=k}^n w_i x_i \quad , \quad s = \sum_{i=0}^{k-1} w_i x_i$$

Subset sum can also be implemented for lists containing negative numbers:

In case that the list is sorted, the same bounding functions as for sorted lists with only positive numbers can be used.

If the list is not sorted, these bounding functions may be used:

$$\sum_{i=1}^k w_i x_i + \sum_{j=k+1}^n w_j q_j \leq m, \quad q_j = 1 \quad \text{if} \quad w_j < 0 \quad \text{and}$$

$$\sum_{i=1}^k w_i x_i + \sum_{j=k+1}^n w_j p_j \geq m, \quad p_j = 1 \quad \text{if} \quad w_j \geq 0$$

The knapsack problem is a modification of subset sum, where each weight w_j of these items is also assigned a value v_j . The problem is now to find the subset, whose sum is smaller or equal to m , and whose sum of the values attached to the elements is maximal.

A convenient way to find a solution for this problem is to calculate for each element a value/weight ratio. Then the solution contains the elements with the highest ratios, whose added weights are smaller or equal to m .

Complexity

The complexity of a problem solved through backtracking depends on 4 factors:

The number of x_i , that fulfill the explicit constraints. The greater the number is, the more x_j have to be tested in the next step

The time it takes to generate the next x_j . With some problems that may be very complicated. For our examples, this takes only a linear amount of time.

The running time of the bounding functions is very important, since the bounding functions are called at each step of the backtracking algorithm. Sometimes weaker bounding functions may be preferable because they take less running time.

The most important factor is the number of x_i that satisfy the bounding functions. This determines, how many subtrees are generated at each step and therefore how fast the tree grows.

Therefore the worst case running time is either $O(p(n)^k)$ or $O(q(n)n!)$, $p(n)$ and $q(n)$ being polynomials, that depend on the running time of the bounding functions.

It is $O(p(n)^k)$, if the power of the solution space is k , $O(q(n)n!)$, if the power is $n!$

Because the power of the set of all possible solutions of the queens problem is $n!$ and the bounding function takes a linear amount of time to calculate, the running time of the **n queens problem** is **$O(n!)$**

Subsets generates at each step maximal 2^{n-1} new subtrees, and the running time of the bounding function is linear, so the running time is **$O(2^n)$** .

Other examples for backtracking

Travelling salesman: the search for a path through an undirected graph, such that each node is only visited once, i.e. a hamiltonian circle

Three-dimensional matching: an extension of the “stable marriage” problem (two-dimensional matching) into another dimension. Two dimensional matching can be solved by a faster algorithm than backtracking, but by adding another dimension, the problem becomes NP -complete.

Solution of boolean expressions: when does a boolean expression yield true?

Partitioning of a set of weights such that the sum of the partitions is equal.

For 2 partitions:
$$\sum_{i=1}^k w_i = \sum_{j=k+1}^n w_j$$

Literature:

E. Horowitz, S. Sahni, S. Rajasekaran: Computer Algorithms, Computer Science Press 1998

Websites:

<http://www.ibrtse.com/delphi/backtracking.html>

<http://www.laurentianum.waf-online.de/laurentianum/backtr.htm>

<http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/vanbeek96a-html/node6.html>

http://students.ceid.upatras.gr/~papagel/project/kef5_8.htm

<http://www.siteexperts.com/tips/functions/ts20/page1.asp>

<http://www.educeth.ethz.ch/informatik/vortraege/backtracking/demos.html>

<http://www.mathematik.uni-ulm.de/sai/ss99/prog/skript/backtracking.html>

<http://www.cs.sunysb.edu/~algorithm/lectures-good/node11.html>

<http://www.animal.ahrgr.de/Anims/singleAnim.php3?lang=de&anim=58>

<http://www.cse.ucsc.edu/classes/cmps012b/Spring97/Lecture07/tsld006.htm>

<http://www.grimmels.de/Mhonzen/Rucksackproblem-web/backtracking1.htm>