Advances in Intelligent Systems Research, volume 147

International Conference on Network, Communication, Computer Engineering (NCCE 2018)

Algorithm Is the Soul of Programming

Chang Liu

Department of computer North China Electric Power University, China

Corresponding auther: 765648526@qq.com

Abstract. In this paper, the basic concept of the algorithm is introduced first. Then four basic design strategies commonly used in algorithm design are discussed. The divide-and-conquer strategy is the most commonly used strategy for designing effective algorithms. The dynamic programming algorithm is a method of filling the table from bottom to up. The greedy algorithm has a certain connection with the design concept of the dynamic programming algorithm, but its efficiency is higher, and it often leads to an optimal solution. The backtracking algorithm is a depth-first search method. In the discussion of each algorithm, this paper first introduce its basic ideas, and then explain how to solve problems under the guidance of basic ideas through examples.

Key words: Algorithm divide and conquer strategy dynamic programming algorithm; calculation order and result; backtracking algorithm

THE BASIC CONCEPT OF THE ALGORITHM

An algorithm is a method or a process to solve a problem. Strictly speaking, the algorithm is a finite sequence consisting of several instructions and satisfies the following four properties.

▪Input: There are zero or more externally supplied quantities as input to the algorithm.

▪Output: The algorithm produces at least one quantity as output.

▪Certainty: Each instruction that makes up the algorithm is clear and unambiguous.

▪Finite: The number of executions of each instruction in the algorithm is limited, and the time for executing each instruction is also limited.

There can be many ways to describe an algorithm, such as natural language, table, flowchart, high-level language, etc. However, for the algorithm, the language is not important, and the key is thinking.

DIVIDE AND CONQUER STRATEGY

The Basic Idea

A problem is decomposed into n smaller sub-problems that are independent of each other and the same as the original problem. Resolve these sub-problems recursively, and then combine the solutions of each sub-problem to get the solution to the original problem.

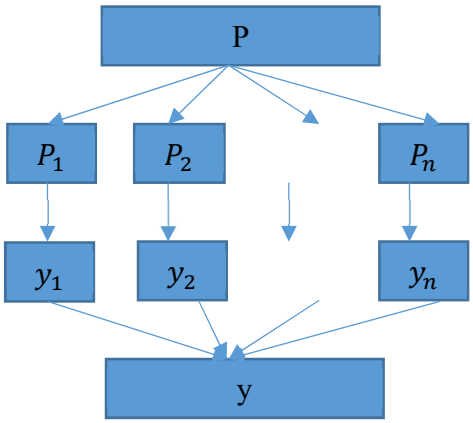

FIGURE 1. Divide and conquer strategy basic idea icon

Chessboard coverage problem

Problem Description

In a checkerboard composed of $2^k \times 2^k$ squares, if exactly one square is different from the other squares, the square is called a special square, and the checkerboard is said to be a special chessboard.

In the chessboard cover problem, all squares except a special square on a given special board are covered with 4 different L-shaped dominoes shown in figure. 2 , and any two dominoes cannot overlap[1].

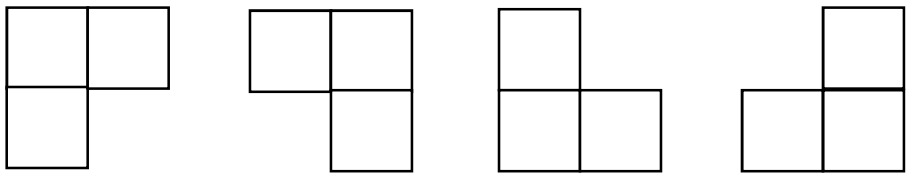

FIGURE 2. Different types of L-shaped dominoes

Example

Divide a chessboard with a size of $2^k \times 2^k$into four $2^{k-1} \times 2^{k-1}$ chessboards. The special square will be located in one of the 4 minor chessboards. In order to convert the remaining three non-specific checkerboards to a special board, an L-shaped domino overlays the meeting of the three chessboards, thereby transforming the original question into four smaller board coverage problems. Use this division recursively until the board is reduced to a 1×1 board.

Assume that the gray board is a special board, and place the L-shaped dominoes in the order of upper left, upper right, lower right, and lower left. The same-color grid is an L-shaped domino, and the numbers in the squares represent the order of placing the L-shaped dominoes. When k=2,

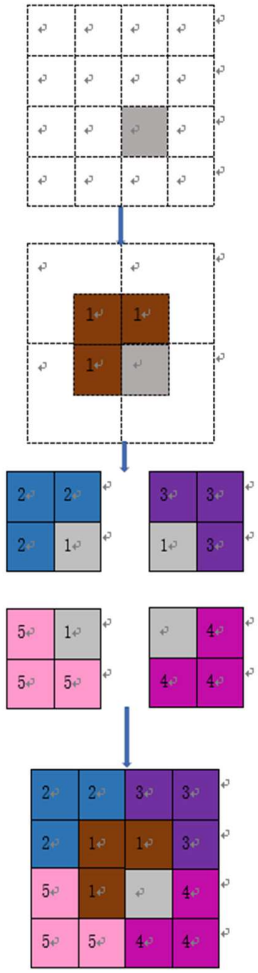

FIGURE 3. $2^2 \times 2^2$board corverage problem

Code

TABLE 1. Variable Description Table

Variable	Instructions
t	Numbers filled in squares
r	The row number of the square in the upper left corner of the board
c	The column number of the square in the upper left corner of the board
s	$s = 2^k$
dr	The row number of the special board
dc	The column number of the special board

```
int t=0;
void chess_board(int r,int c,int s,int dr,int dc)
{
if(s==1)
return;
int tt=t++;
s=s/2;
```

```
// To determine the location of (dr,dc)
// In the upper left 1/4 square
if(dr<r+s && dc<c+s)
{
g[r+s-1][c+s]=tt;
g[r+s][c+s]=tt;
g[r+s][c+s-1]=tt;
chess_board(r,c,s,dr,dc);// In the upper left 1/4 square
chess_board(r,c+s,s,r+s-1,c+s);// In the upper right 1/4 square
chess_board(r+s,c+s,s,r+s,c+s);// In the lower right 1/4 square
chess_board(r+s,c,s,r+s,c+s-1);// In the lower left 1/4 square
}
// In the upper right 1/4 square
if(dr<r+s && dc>=c+s)
{
g[r+s-1][c+s-1]=tt;
g[r+s][c+s-1]=tt;
g[r+s][c+s]=tt;
chess_board(r,c,s,r+s-1,c+s-1);// In the upper left 1/4 square
chess_board(r,c+s,s,dr,dc);// In the upper right 1/4 square
chess_board(r+s,c+s,s,r+s,c+s);// In the lower right 1/4 square
chess_board(r+s,c,s,r+s,c+s-1);// In the lower left 1/4 square
}
// In the lower right 1/4 square
if(dr>=r+s && dc>=c+s)
{
g[r+s-1][c+s-1]=tt;
g[r+s-1][c+s]=tt;
g[r+s][c+s-1]=tt;
chess_board(r,c,s,r+s-1,c+s-1);// In the upper left 1/4 square
chess_board(r,c+s,s,r+s-1,c+s);// In the upper right 1/4 square
chess_board(r+s,c+s,s,dr,dc);// In the lower right 1/4 square
chess_board(r+s,c,s,r+s,c+s-1);// In the lower left 1/4 square
}
// In the lower left 1/4 square
if(dr>=r+s && dc<c+s)
{
g[r+s-1][c+s-1]=tt;
g[r+s-1][c+s]=tt;
g[r+s][c+s]=tt;
chess_board(r,c,s,r+s-1,c+s-1);// In the upper left 1/4 square
chess_board(r,c+s,s,r+s-1,c+s);// In the upper right 1/4 square
chess_board(r+s,c+s,s,r+s,c+s);// In the lower right 1/4 square
chess_board(r+s,c,s,dr,dc);// In the lower left 1/4 square
}
}
```

DYNAMIC PROGRAMMING ALGORITHM

The basic idea

Dynamic programming is a bottom-up form-filling method of the system that is suitable for solving optimization problems. Usually can be designed according to the following 4 steps [1]:

i Find out the nature of the optimal solution and characterize its structure.

ii Recursively define the optimal value.
iii Calculate the optimal value from the bottom up.
iv According to the information obtained when calculating the optimal value, construct the optimal solution.

Multiplicative Product of Matrix

Problem Description

Given n matrices$\{A_1, A_2,\ldots, A_n\}$ where A_i and A_{i+1} are multiplicative, i=1, 2, ..., n-1. Considering the product $A_1\ A_2 \ldots\ A_n$ of these n matrices.

Let A_i be a $r_{i-1} \times r_i$ matrix, for example, A_1 is a $r_0 \times r_1$ matrix, and A_2 is a $r_1 \times r_2$ matrix.

Since the matrix multiplication satisfies the associative law, the sequential product of the computed matrix can have many different calculation orders. This order of calculation can be determined by brackets. If the calculation order of a matrix concatenated product is completely determined, that is, the concatenated product is completely bracketed, then the matrix concatenation products can be calculated by calling the standard algorithm of two matrix multiplications in this order. The fully bracketed matrix concatenation product can be recursively defined as:

i A single matrix is completely bracketed.

ii The matrix concatenated product A is completely bracketed, then A can be expressed as a product of two completely bracketed matrix concatenated products B and C and bracketed, ie, A=(BC).

Now consider the optimal parenthesizing method of the matrix concatenation products $A_1\ A_2 \ldots\ A_n$, that is, minimize the number of multiplication operations.

Resolution steps

The matrix concatenation product $A_i\ A_{i+1} \ldots\ A_j$ is simply referred to as A[i][j]. Assuming that the amount of multiplication is minimal, it is

$$(A_i \ldots A_k)(\ A_{k+1} \ldots A_j)$$

Using the binary pair <m,k>, m is the minimum multiplication workload, k indicates that the problem is broken after A_k.

$$A[i][j].m = \begin{cases} 0 & i = j \\ r[i-1] \times r[i] \times r[i+1] & j = i+1 \\ \min\left(A[i][k].m + A[k+1][j].m + r_{i-1} \times r_k \times r_j\right) & i \le k < j \end{cases}$$

With dynamic programming algorithms to solve problems, they can be calculated in a bottom-up manner based on their recurrence. During the calculation, save the solved sub-question answer. Each sub-problem is calculated only once, so that a large number of repeated calculations are avoided, and eventually the polynomial operation amount is obtained.

Suppose$r_0 = 3, r_1 = 5, r_2 = 2, r_3 = 6, r_4 = 6, r_5 = 4$

TABLE 2. Calculation order and result

i \ j	1	2	3	4	5
1	〈0, 1〉	〈30, 1〉	〈66, 2〉	〈138, 2〉	〈174, 2〉
2		〈0, 2〉	〈60, 2〉	〈132, 2〉	〈160, 2〉
3			〈0, 3〉	〈72, 3〉	〈120, 4〉
4				〈0, 4〉	〈144, 4〉
5					〈0, 5〉

First, start the calculation along the diagonal 1 of the table

r[1][1].m=0 r[1][1].k=1

r[2][2].m=0 r[2][2].k=2
r[3][3].m=0 r[3][3].k=3
r[4][4].m=0 r[4][4].k=4
r[5][5].m=0 r[5][5].k=5
Start the calculation along the diagonal 2 of the table
r[1][2].m=3×5×2=30 r[1][2].k=1
r[2][3].m=5×2×6=60 r[2][3].k=2
r[3][4].m=2×6×6=72 r[3][4].k=3
r[4][5].m=6×6×4=144 r[4][5].k=4
Start the calculation along the diagonal 3 of the table
i=1,j=3
When k=1,A[1][1].m+A[2][3].m+r_0×r_1×r_3=0+60+3×5×6=150
When k=2,A[1][2].m+A[3][3].m+r_0×r_2×r_3=30+0+3×2×6=66
r[1][3].m=66 r[1][3].k=2
i=2,j=4
When k=2,A[2][2].m+A[3][4].m+r_1×r_2×r_4=0+72+5×2×6=132
When k=3,A[2][3].m+A[4][4].m+r_1×r_3×r_4=60+0+5×6×6=240
r[2][4].m=132 r[2][4].k=2

i=3,j=5
When k=3,A[3][3].m+A[4][5].m+r_2×r_3×r_5=0+144+2×6×4=192
When k=4,A[3][4].m+A[5][5].m+r_2×r_4×r_5=72+0+2×6×4=120
r[3][5].m=120 r[3][5].k=4
Start the calculation along the diagonal 4 of the table
i=1,j=4
When k=1,A[1][1].m+A[2][4].m+r_0×r_1×r_4=0+132+3×5×6=222
When k=2,A[1][2].m+A[3][4].m+r_0×r_2×r_4=30+72+3×2×6=138
When k=3,A[1][3].m+A[4][4].m+r_0×r_3×r_4=66+0+3×6×6=174
r[1][4].m=138 r[1][4].k=2
i=2,j=5
When k=2,A[2][2].m+A[3][5].m+r_1×r_2×r_5=0+120+5×2×4=160
When k=3,A[2][3].m+A[4][5].m+r_1×r_3×r_5=60+144+5×6×4=324
When k=4,A[2][4].m+A[5][5].m+r_1×r_4×r_5=132+0+5×6×4=252
r[2][5].m=160 r[2][5].k=2
Start the calculation along the diagonal 5 of the table
i=1,j=5
When k=1,A[1][1].m+A[2][5].m+r_0×r_1×r_5=0+160+3×5×4=220
When k=2,A[1][2].m+A[3][5].m+r_0×r_2×r_5=30+120+3×2×4=174
When k=3,A[1][3].m+A[4][5].m+r_0×r_3×r_5=66+144+3×6×4=282
When k=4,A[1][4].m+A[5][5].m+r_1×r_4×r_5=138+0+3×6×4=210
r[1][5].m=174 r[1][5].k=2
The best way to add parentheses is $(A_1A_2)((A_3A_4)A_5)$.

Code

TABLE 3. Variable Description Table

Variable	Instructions
n	Number of matrices
***r**	An array of matrix rows and columns
****m**	Array of optimal values
****k**	An array of best cut positions recorded

int matrix_chain(int *r, int n, int **m, int **k)

```
{
    for (int i = 0; i < n; i++)
    {
        m[i][i] = 0;
    }
    for (int r = 2; r <= n; r++)
    {
        for (int i = 0; i <= n - r; i++)
        {
            int j = i + r - 1;
            m[i][j] = m[i+1][j] + r[i-1] * r[i] * r[j];
      k[i][j]=j;
            for (int k = i; k <= j - 1; k++)
            {
                int tmp = m[i][k] + m[k + 1][j] + r[i-1] * r[k] * r[j];
                if (tmp < m[i][j])
                {
                    m[i][j] = tmp;
                    k[i][j] = k;
                }
            }
        }
    }
    return m[0][n - 1];
}
```

GREEDY ALGORITHM

The basic idea

Greedy algorithms always make the best choice at present. In other words, the greedy algorithm does not consider the overall optimality, and the choice it makes is only a partial optimal choice in a certain sense.

Knapsack problem

Problem Description

Give the number, weight, and value of the following n types of items; a backpack can load items with a weight of c at most, and find the best loading scheme to maximize the value of the backpack.

TABLE 4. Item Information Table

Number	**1**	**2**	**…**	**n**
Weight	W[1]	W[2]	…	W[n]
Value	V[1]	V[2]	…	V[n]

Problem-solving ideas

i Each item has a number, weight, value and loading ratio.

ii Calculate the unit value of each item and sort it later.

iii The backpack has a weight limit c, and packs as many items with the highest unit value as possible into the backpack. If this item is fully packed in a backpack and the total weight of the item in the backpack does not exceed c, then the item with the next highest value is selected and the backpack is loaded as much as possible until the backpack is full.

Code

TABLE 5. Variable Description Table

Variable	Variable
id	Item number
w	The weight of the item
v	The value of the item
x	Item loading rate $0 \leq x \leq 1$
n	The number of items
c	Item weight limit

```
struct TGoods
{
int id;
Float w,v;
Float x;
};
bool bGreat(const TGoods&x,const TGoods&y)
{
Return (x.v/x.w)>(y.v/y.w);
}
Vector<TGoods>gs;
Void Knapsack (int n,float c,TGoods gs)
{
Sort (gs.begin(),gs.end(),bGreat);
int i=0;
While(c>0&&i<n)
{
if(gs[i].w<c)
{
gs[i].x=1;
c=c-gs[i].w;
i++;
}
else {
gs[i].x=c/gs[i].w;
c=0;
Break;
}
}
}
```

BACKTRACKING ALGORITHM

The basic idea

The backtracking method first gives a fixed ordering to the rules, and then starts from the starting node (the root node) and searches the entire solution space in a depth-first manner. This starting node becomes the active node and it also becomes the current expansion node. At the current expansion node, the search moves in the depth direction to a new node. This new node becomes the new active node and becomes the current expansion node. If the current expansion node can no longer move in the depth direction, the current expansion node becomes a dead node. At this point, move back (backtrack) to the nearest live node and make this live node the current expansion node. The

backtracking method recursively searches through the solution space in this way until it finds that the desired solution or solution space has no live nodes [1].

N queen problem

Problem Description

Place n queens that are not attacked on the n × n grid. According to chess rules, a queen can attack a chess piece that is in the same row or on the same column or on the same diagonal line. The n queen problem is equivalent to placing n queens on the n × n grid, and any 2 queens are not placed on the same row or on the same column or on the same diagonal line.

Example

Use g[n] [n] to represent the square on the board. The sequence of rules is g[1] [1], g[1] [2],...,g[1][n],g[2][1],g[2][2],...,g[2][n],g[n][1]...g[n][n]. The n-tuple x [1: n] represents the solution of the n-queen problem. Where x[i] =k indicates that queen i is placed in the kth column of the i-th row of the board.

When n=4,

i g[1][1] is the starting node, x[1]=1; searching for a new node in depth direction, according to the rules, any 2 queens are not placed on the same row or on the same column or on the same diagonal, so , x[2]=3, g[2][3] becomes the current expansion node.

ii Continuing to search for a new node in the depth direction, it is found that it could no longer move in the depth direction, so g [2][3] became a dead node. Move back to the nearest live node, g [1] [1].

iii Search for a new node in depth direction, so x [2]=4, g[2][4] becomes the current expansion node.

iv Continue searching for a new node in depth direction, so x [3]=2, g[3][2] becomes the current expansion node.

v Continuing to search for a new node in the depth direction, it is found that it could no longer move in the depth direction, so g [3][2] became a dead node. Move back to the nearest live node, g [2] [4].

vi When searching for a new node in the depth direction, it is found that it can no longer move in the depth direction, so g[2][4] becomes a dead node. Move back to the nearest live node, g [1] [1].

vii When searching for a new node in the depth direction, it is found that it can no longer be moved in the depth direction, so g [1][1] becomes a dead node.

viii At this point, g[1][2] is the current expansion node, x[1]=2; continue to search for a new node in depth direction, x[2]=4, g[2][4] becomes the current expansion node.

ix Continue to search for a new node in depth direction, x [3]=1, g[3][1] becomes the current expansion node.

x Continue to search for a new node in depth direction, x [4]=3, g[4][3] becomes the current expansion node.

At this time, find a solution and, for the same reason,could find out all solutions to this problem.

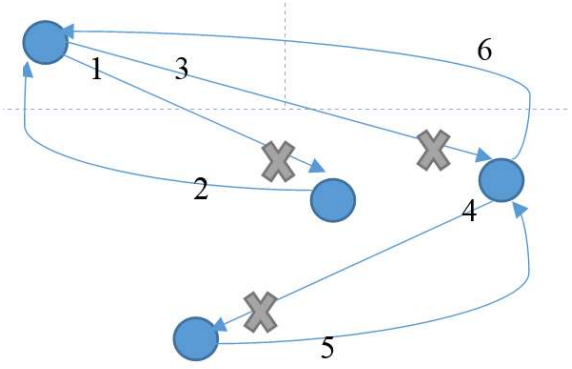

FIGURE 4. Problem-solving diagram

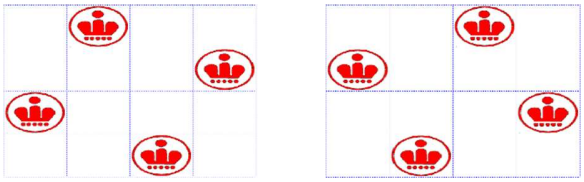

FIGURE 5. Two solutions to the 4-queen's problem

Code

```
bool bPlace(int k)
{
	for(int j=1;j<=k-1;j++)
	{
		if(x[j]==x[k]||abs(x[k]-x[j])==(k-j))
		{
			return false;
		}
	}
	return true;
}
void BackTrack(int t)
{
	if(t<=n)
	{
		for(int i=1;i<=n;i++)
		{
			x[t]=i;
			if(bPlace(t))
			{
				BackTrack(t+1);
			}
		}
	}
}
```

REFERENCES

1. XiaoDongWang, Computer Algorithm Design and Analysis [M],PUBLISHING HOUSE OF ELECTRCNICS INDUSTRY,2014:20-21,44,115