CS 6301 : Implementation of advanced DS&A

Implementation of DS&A - Project 3

Enumeration of permutations satisfying precedence constraints

Arjun Gopal
9/19/2014

Problem Statement

Write a program that visits all permutations of n distinct objects, numbered 1..n, that satisfy a given set of precedence constraints. A precedence constraint (a,b) means that we are interested in only those permutations in which a appears before b.

Input specification:
First line of input has 3 integers: n, c, and v.
The value of n is between 3 and 1000; c is the number of constraint pairs; v>0 means verbose output.
The next c lines of the input contain two integers each: a, b, representing the constraint (a,b).

Output specification:
If v=0 then the program outputs one line with two integers: the number of permutations of 1..n that satisfy the given precedence constraints, and the running time of your program in milliseconds (rounded to integer).
If v>0, then print the permutations, one per line, and the generate the output for v=0.

Sample input:
```
3 2 1
1 3
1 2
```

Sample output:
```
1 2 3
1 3 2
2 0
```

Introduction

"*Permutations with constraints on order of elements*" is a special case of permutation and is discussed in many research publications under different titles. But my researches on this narrowed down to "*Topological sorting*" problem and the same I have used to solve it. Before going with the Topological approach, I have tried a normal recursive permutation technique (n! running time) which is pretty much adaptive to the input. When compared the running time of both, I got 12 times running speed for the first approach which used the topological sorting technique.

Topological Sorting

Before going into the problem solution let's discuss a little about the Topological Sorting. "*In computer science, a topological sort (sometimes abbreviated topsort or toposort) or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering" (Reference : http://en.wikipedia.org/wiki/Topological_sorting)*

Let's simplify this with a real world problem. Assume a company has 10 employees and the manager should come up with a job scheduling plan. Out of this 10 employee, employee B's work can be started only when employee A completes his task. So, scheduling can happen in any order but A's time should always precede B's.

So our problem in terms of topological sorting is that, given n numbers, generate all the topological sorts which follows a constraint in order of some elements.

For example let n be 3

And constraints are 1,2 and 1,3

Then valid topological sorts are

1,2,3 and 1,3,2

Implementation Details

The implementation is based on the algorithm described **by Y. L. Varol and D. Rotem**, "*An Algorithm to generate all topological sorting arrangements*", published in April 1978. (Please see the reference).

I have added custom codes to generate an initial valid topological sort, which is the input to the algorithm. Both the algorithm and the custom codes are discussed in detail in this report.

Key Concepts

Assume we need to permute 4 numbers (1,2,3,4) with constraints (1,4) and (3,2). One of the valid permutations will be (1,4,3,2). Let me index this order from 0 to 3.

Index	0	1	2	3
	1	**4**	**3**	**2**

Now the constraints in terms of the above index will be (0,1) *[which refers to (1,4)]* and (2,3) *[which refers to (3,2)]*. Now the number of permutations on (0,1,2,3) with constraints (0,1) and (2,3) will be same as the number of permutations in the original problem. The thing which is to be noticed here is that for each new constraint (i,j) , i < j.

The algorithm uses this new constraints and work on the indexes to generate all the possible arrangements with the help of a matrix (incedenceMatrix) which is of the size n+1 X n+1. In the matrix for each constraint (i,j)

incedenceMatrix [i][j] = incedenceMatrix [j][i] = true. For all other locations except the last column, the value will be false.

The incidenceMatrix for the above sample problem will be as follows

0	1	0	0	1
1	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	0	0	1

Algorithm

```
PROCEDURE TOPSORT(VAR N: INTEGER; LOC: LIST1; P:LIST2; M:TABLE)

/* LOC: ARRAY[1..N+1] OF TYPE INTEGER */

/* P: ARRAY[1..N+1] OF TYPE INTEGER */

/* M : MATRIX OF SIZE N+1 by N+1 */

/* PROCEDURE WILL ACCESS ONLY FIRST N ELEMENTS OF LOC, N+1 ELEMENTS OF P AND FIRST N
ROWS OF M */

VAR I,K,K1,L : INTEGER

        OBJ_K, OBJ_K1 : OBJECT

BEGIN

        PROCESS
        I:=1

        WHILE I<N DO

         BEGIN K:=LOC[I]; K1=K+1; OBJ_K :=P[K]; OBJ_K1:=P[K1];

                IF M[I,OBJ_K1] THEN BEGIN FOR L:=K DOWNTO I+1

                        DO P[L]:=P[L-1];P[I]:=OBJ_K;

                                LOC[I]:=I;I=I+1;

                        END

                ELSE BEGIN P[K]:=OBJ_K1;

                        P[K1]=OBJ_K; LOC[I]:=K1;

                        I:=1; PROCESS

                        END

        END

END
```

Algorithm Explained

The above algorithm is the original algorithm explained by Varol and Rotem (*See Reference*). Let me explain the same in simple words.

Once we are ready with the incidenceMatrix and number of items to be permuted, we can start the processing.

We initialize two arrays P and LOC and populate the 0 to N+1. Loop starts with a variable I , initialized to 0. The current location is taken from the LOC array as LOC[i]. Let's call it K. The element at (K+1)'th location will be taken from array P. Let's call this OBJ_K1. If location I,OBJ_K1 in M is false, we will swap P[K] and P[K+1] and visit a valid permutation. We reset I to 0 and continue. Else (We can't do swapping as we met a constraint at this point)we have to rotate the array by 1 to the left, set LOC[I]=I and increment I.

The loops go till I < N

Let's take an example

Let n be 4. The numbers to be permuted are {1,2,3,4} . Let's say the constraint is (3,2) and (3,4)

A valid permutation is 1,3,2,4

Index	0	1	2	3
	1	**3**	**2**	**4**

Table 1 : Index Mapping

So mapping to the indexes, new constraints are (1,2) (for 3,2) and (1,3) for (3,4)

We need to apply the algorithm to generate all the permutations of 0,1,2,3 with constraints 1,2 and 1,3

Matrix M would be

0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	0	1
0	0	0	0	1

Array P contains {0,1,2,3,4} // last element is not used

Array LOC contains {0,1,2,3,4} // last element is not used

visit the permutation P {0,1,2,3}

I is initialized to 0.

K = LOC[I] = 0;

K+1 = 1;

OBJ_K = P[K] = P[0] = 0;

OBJ_K1 = P[K1] = P[0] = 1;

M[I,OBJ_K1] = M[0,1] = 0. So we swap P[K] and P[K1] and visit the permutation {1,0,2,3,4}

LOC[I] = LOC[0] = K1=1.LOC becomes {1,1,2,3,4}; I is reset to 0.

----------Next Loop

K = LOC[I] = LOC[0] = 1;

K+1 = 2;

OBJ_K = P[K] = P[1] = 0;

OBJ_K1 = P[K1] = P[2] = 2;

M[I,OBJ_K1] = M[0,2] = 0. So we swap P[K] and P[K1] and visit the permutation {1,2,0,3,4}

LOC[I] = LOC[0] = K1=2.LOC becomes {2,1,2,3,4}; I is reset to 0.

----------Next Loop

K = LOC[I] = LOC[0] = 2;

K+1 = 3;

OBJ_K = P[K] = P[2] = 0;

OBJ_K1 = P[K1] = P[3] = 3;

M[I,OBJ_K1] = M[0,3] = 0. So we swap P[K] and P[K1] and visit the permutation {1,2,3,0,4}

LOC[I] = LOC[0] = K1=3.LOC becomes {3,1,2,3,4}; I is reset to 0.

----------Next Loop

K = LOC[I] = LOC[0] = 3;

K+1 = 4;

OBJ_K = P[K] = P[3] = 0;

OBJ_K1 = P[K1] = P[4] = 4;

M[I,OBJ_K1] = M[0,4] = 1. So we rotate the array to left by 1. And P becomes {0,1,2,3,4}. We will not visit the permutation. LOC[I]=LOC[0]=I=0 LOC becomes {0,1,2,3,4} we increment I by one and continue till I<N.

And So on.

In the above example, following are the permutations which we got in the initial 4 loops.

0,1,2,3,4

1,0,2,3,4

1,2,0,3,4

1,2,3,0,4

If we remove the last element and map it with table 1, then we get the following permutations

1,3,2,4

3,1,2,4

3,2,1,4

3,2,4,1

Which are valid permutations of the original problem.

Following is the algorithm followed to generate the initial topological sort

Reference : http://en.wikipedia.org/wiki/Topological_sorting#CITEREFKahn1962

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

Execution Statistics

These execution statistics are based on the 5 input files provided by professor

Sl No	Input	Execution Time (ms)
1	10.txt	0
2	20.txt	2
3	30.txt	10
4	40.txt	10
5	50.txt	18

Table 2 : Execution Statistics

Conclusion

The algorithm gives a very good performance for even bigger inputs. The project provided a good platform to understand about some new concepts like “Topological Sort” and also about some fast algorithms. The project took 2 days for implementation including the execution of test cases.

References

1. http://en.wikipedia.org/wiki/Topological_sorting
2. http://comjnl.oxfordjournals.org/content/24/1/83.full.pdf
3. http://en.wikipedia.org/wiki/Topological_sorting#CITEREFKahn1962