Infrastructure for Full Enumeration of the Problem Implemented by n Steps

Yong Wei
School of Software, Shenzhen Institute of Information Technology, Shenzhen518172, Guangdong, China

Abstract
In this paper, based on the object-oriented programming, we propose a program infrastructure for solving all the solutions to the problems of which solution is composed of n steps’ selections. Firstly, we consider n steps as n objects, each object has a queue/stack to save all the possible options. From the first step to the final, each selection forms a solution, then return to the recent step of remaining selections, from it to the final, continue get the options to form solution. Until the completion of all the possible options for each step, so as to finish all solutions.

Key words: Permutation, Combination, Enumeration, Object-oriented Programming.

1. INTRODUCTION

Similar to combination generation algorithm, the permutation generating algorithm also had a lot of progresses with the development of computer technology. In 1977, Sedgewick Robert had compared to the previous ten algorithms(Robert Sedgewick, 1977). So far, the permutation generating algorithm has caused people’s attentions(Gao and Wang, 2003).

Enumeration algorithm is widely used. For example, the traveling salesman problem can be solved by using the enumeration method, so as to find out all the possible Hamilton circle in the network, and then choose the one of which right is the smallest(Wang, 2013).

Since 1970s, America S.EVEN et al. had demonstrated that TTP(timetable problem) was a NP complete(Shi, Zhang and Lin., 1986), many timetable scheduling systems emerged as the times required(Erben, Kepller, 1995; Caldeira and Rosa, 2007). Theory and practice showed that, namely “combinatorial explosion”, any information involved slightly changing would lead to great changes in timetable arrangements. Most of the TTP system softwares were aimed at the specific application environment, neglecting some restricted conditions, resulting in many TTP softwares were not ideal. Grouping combination is the problem that n objects are divided into k groups, and the grouping combination generator is an algorithm to enumerate all the groups. In fact TTP presences grouping combination properties(Wei., 2014), so we can implement the TTP by full groups’ enumeration. Compared with the genetic algorithm, it can avoid the problem of the convergence of the TTP.

Unfortunately, the enumeration of many problems was so large that general computer enumerating all the solutions was difficulty in a short time. But with the development of the computer system, especially the application of cloud computing, the method of the best solution from the full enumerations can be obtained(Zhang, Gu and Zheng, 2010; Luo, Jin, Song and Dong, 2011; Li and Zhen, 2011).

If a problem was completed by n steps and each step’s choosing formed a solution, this paper presented an infrastructure for the full enumeration algorithm based on object-oriented programming. Then we demonstrated the permutation and combination processes under the infrastructure.

2. INFRASTRUCTURE OF FULL ENUMERATION
The core idea of the full enumeration algorithm infrastructure for object-oriented programming is: If a problem is done by n steps s1, s2, ... sn, each step’s choice r2, r1,...fn constitutes a solution, assuming that each step has more possible options, we can use the queue q1, q2,...qn to save these options for the steps. All solutions of the problem can be enumerated by the following processes.

1) step by step from s\(i\rightarrow s_{i+1}\rightarrow...\rightarrow s_n\), select r1, r2,..., rn from the optional queue of the steps, got the first solution.
2) back to the nearest step si(1\(\leq i\leq n\)) of non-empty qi.
3) step by step from si\(\rightarrow s(i+1)\rightarrow...\rightarrow s_n\), gradually take out a choice to get the new solution r1, r2,... ri, r(i+1), ... , rn.
4) if step si of which queue qi is not empty, go to step2; otherwise ends the program.

Then this paper presents the infrastructure of the full enumeration algorithm with the idea of object-oriented programming.

The infrastructure of the whole enumeration is composed of two class, Step and Enum. The full enumeration method is implemented by the n steps, so these steps are described by Step.

class Step {
    no://step’s number
    r:// choice of the step
    q:// queue for the options
    boolean hasMore() {
    Object getNext() {

    }
}

Where "no" represents the number of steps, the "r" is the choice made by the step, and the "q" is a possible selections queue.

Step’s hasMore() method determines if there are more options, and getNext() method gets the next choice from "q".

Enum class implements numerating process. It mainly includes hasMore(), getNext(), adjustNextStep() and start() method. The first three methods need to be based on the specific sub-class’s implementation, start() method enumerates all solutions.

class Enum {
    boolean hasMore();//determine whether there are more solutions
    getNext(); //get next solution
    //Get next step’s optional queue based on the current step
    adjustNextStep(Step step); //begin to enumerate all solutions
    start(Step s[]){ //s is an array of steps
        Step step;//current step
        int i,j,k;
        push the first step into stack;
        outer loop: while the stack is not empty{
            step<--pop();//Get a recent step which q not empty
            i<--step.no;
            //Loop operation from ith step to the last step
            inner loop: k from ith to the last step{
                step<--s[k];
                if step.hasMore(){
                    step.r = step.getNxt();
                }
            }
        }
    }

    //adjust the next step’s options
    s[k+1].q = adjustNextStep(step);
}

if(step.hasMore()) //The step has more options, push in stack.
    stack.push(step);

inner loop ends, a solution obtained : r1, r2, … rn
}

Both Enum and Step contain hasMore() and getNext() methods, but they are different. The Step’s hasMore() determines whether the step has more options there, and the Enum’s hasMore() determines whether the end of all the solution.

The start() method is implemented by 2 loops. Inner loop, from the current step si to the final step sn, takes the step’s selecting rk(i<=k<=n); if the step’s qk is not empty, the step sk be pushed into the stack. After inner loop, r1, r2,…, rn form a new solution.

If the stack is not empty, outer loop pops the last step of which "q" is not empty from the stack.
3. APPLICATION OF THE FULL ENUMERATION INFRASTRUCTURE

3.1 Permutation Generation

The following demonstrated how to use this infrastructure to implement the full permutation with Permutation class to inherit Enum:

```java
class Permutation extends Enum{
    Step s[];//Step array, keep all step objects
    adjustNextStep(step){ }//implementation of adjustNextStep() method
    boolean hasMore();//implementation of method
    Object getNext();//implementation of method
}
```

The programming created an object of Permutation, initialized the s[], and called start(s) to complete the full permutation generation algorithm.

The permutation of 4 objects 1, 2, 3, 4, could be implemented by 4 steps s1, s2, s3, s4 through each step took an option. got the first solution by the order of s1 -> s2 -> s3 -> s4:

1. **s1.q** = 1, 2, 3, 4  
   s1.r = 1
   push s1
   **s2.q** <= get 2, 3, 4 through calling adjustNextStep(s1)
   s2.r = 2
   push s2
   **s3.q** <= get 3, 4 through adjustNextStep(s2)
   s3.r = 3
   push s3
   **s4.q** <= get 4 through adjustNextStep(s3)
   s4.r = 4
   get solution 1: 1, 2, 3, 4

   pop s3
   s3.r = 4
   s4.q <= get 3 through calling adjustNextStep(s3)
   get solution 2: 1, 2, 4, 3

   pop s2
   s2.r = 3
   s3.q <= get 2, 4 through calling adjustNextStep(s2)
   s3.r = 2
   push s3
   **s4.q** <= get 4 through adjustNextStep(s3)
   s4.r = 4
   get solution 3: 1, 3, 2, 4

   pop s3
   s3.r = 4
   s4.q <= get 2 through calling adjustNextStep(s3)
   get solution 4: 1, 3, 4, 2

   get last solution: 4, 3, 2, 1

   The adjustNextStep(step) method calculated the next step which may have selections based on the current step. In the permutation generation algorithm, each step took an option, and the rest of the i-th step’s options was the optional queue of the (i+1)-th step. So the process of adjustNextStep (step) method was as follows:

   ```java
   adjustNextStep(step){
       i <= step.no;
       return: remove r1, r2,... ri from all selecting objects
   }
   ```

3.2. Combined Generation

The generation of combined C(4,3) could be seen as processes of three steps s1, s2 and s3, their selections r1, r2, r3 constituted a combination of a generation. The start() method demonstrated the generation process, see below:

   got the first solution by order s1 -> s2 -> s3.
s1.q = 1, 2, 3, 4
s1.r = 1
push s1
s2.q <- got 2, 3, 4 through calling adjustNextStep(s1)
s2.r = 2
push s2
s3.q <- got 3, 4 through calling adjustNextStep(s2)
s3.r = 3
push s3
got solution 1: 1, 2, 3

pop s3
s3.r = 4
s4.q <- got 3 through calling adjustNextStep(s3)
got solution 2: 1, 2, 4

pop s2
s2.r = 3
s3.q <- got 2, 4 through calling adjustNextStep(s2)
s3.r = 2
push s3
got solution 3: 1, 3, 2

pop s3
s3.r = 4
got solution 4: 1, 3, 4

As a result of the combination generation was independent of the order of selection in each step, the solution 1 and 3 was the same solution. After the inner loop in the start() method, we added a limited condition ” r1<r2<r3 ” to remove repeated solutions.

4. ALGORITHM ANALYSIS

4.1 Space Complexity

The permutation generation of n objects requires n steps s1, s2, ..., sn. s1.q requires n spaces. After one object is selected, the s2.q needs n-1 spaces, …, the last step has only one option. So:

\[ n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2} \]

Thus, the space complexity of permutation generation is \( S(n^2) \).

The combination generation of \( C(n, k) \) is implemented by \( k(k<=n) \) steps s1, s2, ..., sk. The number of space for each step is: n, n-1, ..., (n-k+1), the total space required:

\[ n + (n-1) + \ldots + (n-k+1) = (n + n - k + 1)((n - (n-k +1) +1 )/2 \]

\[ = (2n - k + 1)k/2 \]

So space complexity of the combination generation of \( C(n, k) \) is \( S(2nk-k^2) \).

4.2 Time Complexity

In accordance with the start() method of the infrastructure, it gets a solution after each end of the inner loop. The full enumerations of n objects’ permutation are n!. So the time complexity is \( O(n!) \).

Similar to the permutation generation, the time complexity of the problem of generating the \( C(n, k) \) is \( O(n!/k!(n-k)!) \).

5. CONCLUSIONS

Since the 60's of last century, there had appeared many algorithms to describe the permutation and combination generation with pure mathematical methods, the data structure should be set up according to the rules of computer language. For the problem of n steps’ selections constituting a solution, this paper put forward the infrastructure based on the idea of object-oriented programming to solve it. Permutation and combination generation problem can be considered as the problem of the multiple steps’ selections, and the process of generating algorithm was demonstrated in this paper. We can use this infrastructure to solve the problem of the n queens, the topological sort and the generation of tree, etc.

The infrastructure is universal. The time complexity of the enumeration process is generally related to the number of the enumerations, so the efficiency is high. The finite steps dictate the tolerability of the space complexity. Contrast to recursion, the infrastructure will not increase the problem’s expansion caused by the expansion of the scale.

In summary, the program infrastructure can not only implement a lot of enumeration algorithms, but also be efficient and practical.
Acknowledgements

This work was supported by Shenzhen Institute of Information Technology scientific research raising project (Special topic of school enterprise cooperation)”open mobile educational management system”(lg2014010).

REFERENCES


