18/2 (2020), 107–120
DOI: 10.5485/TMCS.2020.0511
tmcs@science.unideb.hu
http://tmcs.math.unideb.hu

Teaching
Mathematics and
Computer Science

Task variations for backtrack

László Menyhárt and László Zsakó

Abstract. This article has been written for informatics teachers who want to issue backtrack based tasks on their lessons or as homework or on competitions. We present a few methods to generate a more complicated problem from a simpler task, which will be more complex, and its solution needs a good idea or trick. Starting from an example, we lead the reader through increasingly difficult task variations.

Key words and phrases: education, task, variation, backtrack.

MSC Subject Classification: 97P50.

Introduction

Our present article is for those informatics teachers, who wants to release tasks for their students on lessons, as a homework or on competitions with different difficulty and which are solvable with backtrack. Modifications can be needed if the same task wouldnt be solved with different ability groups, with different classes on the same year, or in every year or semester. On competitions tasks of different age groups can be variant.

A concrete example and its variations are analysed, and generally usable methods are determined from these.

Backtrack is very useful algorithm in a very wide range of problem solving. Its essence is to approach the task with a regular attempt. Sometimes this is the

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

best solution! However, most of the books on algorithms do not deal with backtrack. Some contain examples for backtrack like topics eight queens, tree traversal, games, evaluation of logical formulas (Shen, 2010) (Dasgupta, Papadimitriou, & Vazirani, 2006). This method is present in more university curriculum with topics eight queens, map colouring, solitaire, sudoku (Sullivan, 2012) (Harder, 2012) (Skiena, 2017) (Zelenski, 2008). On Etvs Lornd University Faculty of Informatics students learn this on course Artificial Intelligence (Lőrentey, Fekete, Fóthi, & Gregorics, 2001). None of them contain similar tasks for practice that we present in this article.

Backtrack is a general method that decreases the number of steps with using an idea (Wirth, 1976) instead of testing all possible cases (brute force). Proper data representation and skilful terms are needed.

Given `N` series with number of elements `M[1]`, `M[2]`, ..., `M[N]` respectively, but sometimes with same number of elements (`M`). Choose one item from each one so that the choices from each series affect others! This is a complicated search task, in which `N` elements must be chosen with a given attribute without looking at all the options.

A common feature of these tasks is that they result in a series. Every items of the result series are come from another series, but the items are related to each other (for example a queen cannot be placed where a former queen would attack her; one job cannot be given to two workers; once a bakery has run out of bread, it can no longer be ordered).

- Backtrack is usable when the search space can be imagined as a tree structure in which we search for a vertex from the root.
- The essence of the algorithm is that it takes a path from the starting point to a subdivision of the task, and if it turns out that it can no longer reach the destination, it goes back to an earlier decision point and chooses another path - a different subproblem.

First, we try to select an element from the first series, then from the next one, and we do this as long as possible. Denote `X[i]` the i^{th} item selected from the series! The value will be `0` if there was no choice yet. If there is no good item in the next series, a new choice should be found from the previous. At this backstep the previous choices must be deleted from which the stepping back is happened. The process will be finished when items were chosen from every series or after a lot of step back there is no possible item in the first series already, so there is no solution of the task.

```
Search(N,Exist,X):
  i:=1; X[]:=[0,...,0]
  While i≥1 and i≤N {there are more, but not ready}
    LookingForGoodElement(i,Exist,j)
    If Exist
      then X[i]:=j; i:=i+1  {forward}
      else X[i]:=0; i:=i−1  {backward}
  End of while
  Exist:=(i>N)
End of procedure.
```

A linear search starts in the i^{th} series: decision path `j` cannot be selected in step `i` if it is not good for the former selections or it is not good itself.

```
LookingForGoodElement(i,Exist,j):
  j:=X[i]+1
  While j≤M[i] and (isBad(i,j) or forbidden(j))
    j:=j+1
  End of while
  Exist:=(j≤M[i])
End of procedure.
```

It can be stated that each new choice may depend on all previous ones (formalized with `isBad` function), but not on later ones!

```
isBad(i,j):
  k:=1
  While k<i and allowed(i,j,k,X[k])
    k:=k+1
  End of while
  isBad:=(k<i)
End of function.
```

When all the possible solutions must be looked through, the easiest solution is a recursive function call for the backtrack and continue the search with a recursive function call.

Let `Cnt` the count of solutions and let `Y` the vector which contains the solutions! Instead of search we select the good initial solutions and continue the selection with recursion!

```
AllSolutions(i,N,Cnt,Y,X):
  If i>N then
    Cnt:=Cnt+1; Y[Cnt]:=X
  else
    For j=1 to N
      If not isBad(i,j) and not forbidden(j) then
```

```
        X[i]:=j; AllSolutions(i+1,N,Cnt,Y,X);
    End of For
  End of If
End of procedure.
```

Task-variations

The base task

A business is looking for workers for `N` different jobs. At applying all `M` applicant will share some certain information. The main task is to determine which job should be filled by whom.

Variation 1

A business is looking for workers for `N` different jobs. There are exactly `N` candidates and each applicant told us which jobs they are qualified. The boss of the business wants to recruit all the candidates and make all the jobs done.

Let value of `F[i,j]` is false, if the applicant `i` is not good at job `j` and it is true if the applicant is good at it.

```
Jobs(N,F,Exist,Y):
  i:=1; X[]:=[0,...,0]
  While i≥1 and i≤N {there are more, but not ready}
    LookingForGoodElement(i,F,Y,Exist,j)
    If Exist
      then X[i]:=j; i:=i+1  {forward}
      else X[i]:=0; i:=i−1  {backward}
  End of While
  Exist:=(i>N)
End of procedure.
```

It is clear, that the main procedure is the same, it should not be modified, the general schema must be copied only. Second level must be simplified, `N` went instead of `M[i]`, and the `forbidden(j)` function was changed to a matrix element reference.

```
LookingForGoodElement(i,F,Y,Exist,j):
  j:=X[i]+1
  While j≤N and (Occured(i,j) or not F[i,j])
    j:=j+1
  End of while
  Exist:=(j≤N)
End of procedure.
```

Choosing a job is not good on the third level if it was given to someone else.

```
Occured(i,j):
  k:=1
  While k<i and X[k]≠j
    k:=k+1
  End of while
  Occured:=(k<i)
End of Function.
```

Variation 2

This variation is the same as the first one with another representation. Let `D[i]` the number of jobs which can be done by an applicant `i`, `E[i,j]` is the serial number of the assumed j^{th} job by applicant `i`!

```
Jobs(N,D,E,Exist):
  i:=1; X[]:=[0,..,0]
  While i≥1 and i≤N {there are more, but not ready}
    LookingForGoodElement(i,D,E,Exist,j)
    If Exits
      then X[i]:=j; i:=i+1  {forward}
      else X[i]:=0; i:=i-1  {backward}
  End of while
  Exist:=(i>N)
  If Exist then For i=1 to N
                  X[i]:=E[i,X[i]]
                End of for
End of procedure.
```

It is clear, that the main procedure is the same, it should not be modified, the general schema must be copied only. The only modification is that the number of the job must be generated from the earlier solution because it was the number of the choice.

Second level must be simplified because now no need to check whether an applicant is good at a job or not.

```
LookingForGoodElement(i,D,E,Exist,j):
  j:=X[i]+1
  While j≤D[i] and isBad(i,j,E)
    j:=j+1
  End of while
  Exist:=(j≤D[i])
End of procedure.
```

The comparison will be a little more complicated because `j` and `X[k]` are not the serial number of the job, but it is serial number from the list of workers job.

```
isBad(i,j,E):
  k:=1
  While k<i and **E[k,X[k]]≠E[i,j]**
    k:=k+1
  End of while
  isBad:=(k<i)
End of function.
```

Variation 3

A business is looking for workers for `N` different jobs. There are exactly `N` candidates and each applicant told us which jobs they are qualified and how much salary would be asked. The boss of the business wants to recruit all the candidates and make all the jobs done, but up to an amount `S`.

Let value of `F[i,j]` is zero (`=0`) if the applicant `i` is not good at job `j`, and it is positive (`>0`) if he is good at this job and the value means the salary.

The cost constantly rising with mandating any new employee, and we take advantage of it in the solution.

```
Jobs(N,Exist,Y):
  i:=1; X[]:=[0,...,0]
  While i≥1 and i≤N {there are more, but not ready}
    LookingForGoodElement(N,i,F,Y,Exist,j)
    If Exist **and Cost(N,i,j,F,Y)≤S**
      then X[i]:=j; i:=i+1  {forward}
      else X[i]:=0; i:=i−1  {backward}
  End of while
  Exist:=(i>N)
End of procedure.
```

The second level is almost the same as in the first variation, only the value of `F[i,j]` is not logical, but an integer number (the salary).

```
LookingForGoodElement(N,i,F,Y,Exist,j):
  j:=X[i]+1
  While j≤N and (isBad(i,j,Y) or **F[i,j]=0** )
```

```
    j:=j+1
  End of while
  Exist:=(j≤N)
End of procedure.
```

Calculating the cost is very simple, it can be done with a simple summary.

```
Cost(N,i,j,F,Y):
  s:=0
  For k=1 to i-1
    s:=s+F[i,X[k]]
  End of for
  s:=s+F[i,j]
  Cost:=s
End of function.
```

In this solution the `Cost` function is the **limitation**, so stepping back happened earlier because it can be detected that there cannot be good solution yet. So, the essence of the limitation is the earlier step back.

Variation 4

There are exactly `N` candidates to the `N` jobs and each applicant told us which jobs they are qualified and how much salary would be asked. The boss of the business wants to fill all the jobs but at the least cost to him.

His idea is this: first all possible job fill will be generated (if there is). It means that all applicant will have a serial number of a job with two conditions:

- we can choose a job for him, what he can do (`F(i,j)>0`);
- we can choose a job for him, what is not given to others (Value of `Occured(i,j,x)` function is `false`).

First parameter of `AllJob` procedure is the serial number of the actual applicant `i`, the second parameter is the number of the applicants (and so the jobs), it is `N`.

```
AllJob(i,N,F,Cnt,Y,X):
  If i>N
    then Cnt:=Cnt+1; Y[Cnt]:=X
    else For j=1 to N
           If not Occured(i,j,X) and F[i,j]>0
             then X[i]:=j; AllJob(i+1,N,F,Cnt,Y,X)
         End of for
  End of if
End of procedure.
```

```
Occured(i,j,X):
  k:=1
  While k<i and X[k]≠j
    k:=k+1
  End of while
  Occured:=(k<i)
End of function.
```

Now the task last part is to choose the most economical solution for the boss from the solutions gathered in vector Y. However, it may be revealed very soon that there may be too many elements in the vector Y. Here the new idea is that it is redundant to store all possible solutions, it is enough to store the most economical solution after every step.

The question is with which would the first solution be compared. Let be a new variable what contains the best cost till now. Initial value of this should be the possible maximum value at the beginning of the program. When we found a solution, that will be the better, so it will be changed to this new value.

```
BestJob(i,N,F,MaxCost,Y,X):
  If i>N then
    If Cost(N,F,X) <MaxCost
      then Y:=X; MaxCost:=Cost(N,F,X)
  else
    For j=1 to N
      If not Occured(i,j,X) and F[i,j]>0
        then X[i]:=j; BestJob(i+1,N,F,MaxCost,Y,X)
    End of for
  End of if
End of procedure.
```

```
Cost(N,F,X):
  s:=0
  For i=1 to N
    s:=s+F[i,X[i]]
  End of for
  Cost:=s
End of function.
```

It is possible of course, that there is no solution for this task, so there is no best solution. Now we can think another small refinement: if there is a solution, and we can see that the solution that is now being prepared will be not better, so it will be more expensive, we can stop the process.

Let the cost is a parameter of the procedure and the procedure will be continued if it does not reach the earlier maximum cost. That is why the cost must be calculated continuously instead of at the readiness of a solution.

The method must be changed, procedure `BestJob` will have a new parameter, it is the `cost` before the actual choices.

```
BestJob(i,cost ,N,F,MaxCost,Y,X):
  If i>N then
    If cost<MaxCost
      then Y:=X; MaxCost:=cost
    End of if
  else
    For j=1 to N
      If not Occured(i,j,X) and F[i,j]>0
          and cost+F[i,j]<MaxCost
      then
        X[i]:=j;
        BestJob(i+1,cost+F[i,j] ,N,F,MaxCost,Y,X)
      End of if
    End of for
  End of if
End of procedure.
```

Variation 5

A business is looking for workers for `N` different jobs. `M (M<N)` candidates arrived to the ad and each applicant told us which jobs they are qualified and how much salary would be asked. The boss of the business wants to hire all the applicants but at the least cost to him.

The solution is very similar to the previous one: it is ready here, when all `M` applicants got a job instead of number `N`:

```
BestJob(i,cost,N,M,F,MaxCost,Y,X):
  If i>M then
    If cost<MaxCost then
      Y:=X; MaxCost:=cost
    End of if
  else
    For j=1 to N
      If not Occured(i,j,X) and F[i,j]>0 and cost+F[i,j]<MaxCost
      then X[i]:=j; BestJob(i+1,cost+F[i,j],N,M,F,MaxCost,Y,X)
    End of for
  End of if
End of procedure.
```

Variation 6

A business is looking for workers for `N` different jobs. There are `M` (`M>N`) applicants to the ads, and each applicant told us which jobs they are qualified and how much salary would be asked. The boss of the business wants to fill all the jobs but at the least cost to him.

Idea of the solution: do not search a job for the applicants, but we can search applicant for the jobs! Thus, the solution will almost same as the earlier one, only indexes should be interchanged.

```
BestJob(i,cost,N,M,F,MaxCost,Y,X):
  If i>N then
    If cost<MaxCost then
      Y:=X; MaxCost:=cost
    End of if
  else
    For j=1 to **M**
      If not Occured(j,i,X) and F[j,i]>0 and cost+F[j,i]<MaxCost then
        X[j]:=I; BestJob(i+1,cost+F[i,j],N,M,F,MaxCost,Y,X)
      End of for
  End of if
End of procedure.
```

Variation 7

A business is looking for workers for `N` different jobs. There are exactly `N` applicants and each applicant told us which jobs they are qualified and how much salary would be asked.

The boss of the business wants to fill all the jobs but at the least cost to him. Maybe it is not possible, but we are interested in a solution when the most job is filled.

Idea of the solution: Let a new fictive job as an element `N+1` which is known everybody! Let the cost of this be greater than every other cost in the table! More people can choose this job `N+1`. Demonstrable that the most economical solution will contain the minimal fictive jobs so the most jobs will be filled.

```
BestJob(i,cost,N,F,maxValue,MaxCost,Y,X):
  If i>N then
    If cost+**maxValue** <MaxCost
      then Y:=X; MaxCost:=cost+**maxValue**
    End of If
  else
```

```
    For j=1 to N
      If not Occured(i,j,X) and F[i,j]>0 and cost+F[i,j]<MaxCost then
        X[i]:=j;
        BestJob(i+1,cost+F[i,j],N,F,maxValue,MaxCost,Y,X)
      End of if
    End of for
  End of if
End of procedure.
```

Change the basic task

We changed the number of the jobs and number of the applicants in the presented task. We modified the known information whether we know the salary or not. We changed the number of the required solution, so we need all or the best one. Once we had limitation for data.

Generally, we can say that the size (`N` and `M`) of the multidimension data of backtrack and their relationship to each other can serve to create variations.

There are newer opportunities to create other variations with changing the types, meaning of the given data or with adding other additional, clarifying or new information or with carrying info effect a new limitation.

The problem can be modified with changing the base task thus we get a large number of task variation packages.

We can formulate the next combinatorial issues independently from the task where all solutions are required. Every time the task is to generate the X vector corresponding to the question. So, at these tasks the investigation of backtrack algorithm is the important, how X_j and X_i are interdependent. For the sake of simplicity X vector should contain integer numbers between `1` and `N`!

All permutations without repetition

Prepare all possible sequences of `N` different data! The task is to generate all `X` vectors where $X_j X_i$ in case of `j` < `i`.

All permutations with repetition

Produce all possible sequences of `N` different data, where the i^{th} data occurs L_i times! The task is to generate all `X` vectors where for all `i` indexes

$$\sum_{\substack{j=1 \\ X_i=X_j}}^{i} 1 \le L_i$$

All variations without repetition

Select `K` different data from N different data where the choices with different order are different results! The task is to generate all X vector with K elements where $X_j X_i$ in case of `j` $<$ `i`.

All combinations without repetition

Select K different data from N different data where the choices with different order are different results! Since there is only one item between the choices the same elements with different order, so the task is to generate all X vector with K elements where $X_j < X_i$ in case of `j` $<$ `i`.

All combinations with repetition

Select K not necessarily different data from N different data where the choices with different order are different results! Since there is only one item between the choices the same elements with different order, so the task is to generate all X vector with K elements where $X_j X_i$ in case of `j` $<$ `i`.

All partitions for the sum of K non-negative elements

The task is to generate K-digit numbers where the sum of the digits is exactly N. In this case isBad function must cause step back when for an index i the

$$\sum_{j=1}^{i} X_i > N$$

All partitions for the sum of K positive elements

The task is to generate numbers up to N digits where the sum of the digits is exactly N and none of the digits is 0. In this case isBad function must cause step back when for the index i the

$$\sum_{j=1}^{i} X_i > N$$

Summary

The mathematical foundations of the method were dealt with by kos Fthi and his colleagues (Harangozó, Nyékyné Gaizler, Fóthi, & Konczné Nagy, 1995), but a different approach must take in public education. It should be based on algorithm thinking instead of mathematics.

The purpose of this article is to evaluate and demonstrate the feasibility of creating backtrack tasks of varying complexity. Through an example, we illustrate the variations that can be made to a basic task.

We generally determined methods from the edifications of listed examples, with which former modifications could come. Even more the problem solvers get a new problem with the modification of base task or with changing the topic.

We hope that the patterns and methods presented here help our readers to do their later work easier when setting up backtrack tasks with varying complexity. Examples can be found on GitHub (Menyhárt, 2021).

Acknowledgements

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

References

Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2006). *Algorithms.* New York: McGraw-Hill.

Harangozó, É., Nyékyné Gaizler, J., Fóthi, Á., & Konczné Nagy, M. (1995). Demonstration of a problem-solving method. In *Acta cybernetica 12* (p. 71-82).

Harder, D. W. (2012). *Backtracking.* University of Waterloo, Ontario, Canada. Retrieved from `https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Backtracking/`

Lőrentey, K., Fekete, I., Fóthi, Á., & Gregorics, T. (2001). On the wide variety of backtracking algorithms. In E. Kovács & Z. Winkler (Eds.), *Proceedings of the 5th international conference on applied in-formatics: Education and other fields of applied informatics, computer graphics, computer statistics and modeling* (p. 165-174). Eger: Molnr s Trsa 2001 Kft.

Menyhárt, L. (2021). *Examples for this article.* Retrieved from `https://github.com/laszlogmenyhart/task-variations-for-backtrack`

Shen, A. (2010). *Algorithms and programming: Problems and solutions.* New York: Springer. doi: 10.1007/978-1-4419-1748-5

Skiena, S. (2017). *Analysis of algorithms: Backtracing.* Stony Brook University New York. Retrieved from `https://www3.cs.stonybrook.edu/~skiena/373/newlectures/lecture15.pdf`

Sullivan, D. G. (2012). *Recursion and recursive backtracking.* Harvard Extension School. Retrieved from `https://sites.fas.harvard.edu/~cscie119/lectures/recursion.pdf`

Wirth, N. (1976). *Algorithms + data structures = programs.* New Jersey: Prentice-Hall.

Zelenski, J. (2008). *Exhaustive recursion and backtracking.* Harvard Extension School. Retrieved from `https://see.stanford.edu/materials/icspacs106b/h19-recbacktrackexamples.pdf`

LÁSZLÓ MENYHÁRT
ELTE EÖTVÖS LORÁND UNIVERSITY, BUDAPEST,
HUNGARY 3IN RESEARCH GROUP, MARTONVÁSÁR,
HUNGARY

E-mail: `menyhart@inf.elte.hu`

LÁSZLÓ ZSAKÓ
ELTE EÖTVÖS LORÁND UNIVERSITY, BUDAPEST,
FACULTY OF INFORMATICS
HUNGARY

E-mail: `zsako@caesar.elte.hu`

(Received July, 2020)