Sequential Testing with Precedence Constraints

Duygu Ay duyguay@sabanciuniv.edu
Industrial Engineering Graduated

Cankut Coşkun cankutcoskun@sabanciuniv.edu
Computer Science, Sophomore

Project Supervisor: Tonguç Ünlüyurt
Industrial Engineering

Abstract

We consider the problem of minimum cost sequential testing (diagnosis) of a series (or parallel) system under precedence constraints. The model of the problem is a nonlinear integer program. We develop two different approach to solve the problem. Firstly, we develop and implement a Simulated Annealing algorithm for the problem. And, we compare the performance of the Simulated Annealing algorithm with optimal solutions. The simulated annealing algorithm is particularly effective as the problem size gets larger. Secondly, we develop a General Tree Type data search algorithm which finds and sorts all feasible permutations under precedence constraints in order to find the cost minimizing solution.

Keywords: Metaheuristic; Simulated Annealing; Tree Data Structure; Sequential Testing.

1 Introduction

In many practical situations, the problem occurs in testing a complex system through a series of its components. In many of these cases, testing a component is costly (it may cost money, take time, or incur pain on the patient, etc.), and therefore it is important to determine a best algorithm that minimizes the average cost of testing in the long run. More precisely, the sequential test problem requires that a system consisting of a series of components be correctly ordered as having the expected minimum cost.

Metaheuristics are strategies that guide the search process to provide a sufficiently good solution to an optimization problem, especially for big data. The sequential test problem requires a metaheuristic method for efficiency concern. According to previous researches, an ant colony algorithm is developed and implemented for the sequential testing problem of a series system under precedence constraints. This algorithm for special type of instances found optimal solutions in polynomial time. The ant colony algorithm is particularly effective as the problem size gets larger (Çatay, Özlük & Ünlüyurt, 2011).

In this particular study, we propose a simulated annealing algorithm which is one of the metaheuristic methods for the sequential testing problem of a series system under precedence constraints. We demonstrate the effectiveness of the proposed algorithm through average optimality gap calculations. Also, we propose a general tree type data search algorithm which enumerates all feasible permutations under precedence constraints.

2 Problem Definition

We consider a simple series system. Simple series system where the system functions if all the individual components function. We would like to find out if all components function or as having the expected minimum cost. The individual components can be in a working or failure state, and these components are independent of each other. We have two type parameters: First one is whether learning the state of individual components is costly (c_i), Second one is P_i, probability that component i functions. We continue testing components one by one until a failing component is found or all components are tested.
The strategy corresponds to a permutation of the components.
For a permutation π the expected cost is:

cπ(1)+ pπ(1) cπ(2)+ pπ(1) pπ(2) cπ(3)+ …..+ pπ(1) pπ(2) …pπ(n-1) cπ(n)

All permutations are not feasible due to physical/logical etc. reasons. These reasons can be described as precedence constraints. These constraints are described by an acyclic directed graph, where arc (i, j) means j can be tested only if i is tested before j. So, the problem is to find a feasible permutation with the minimum expected cost.

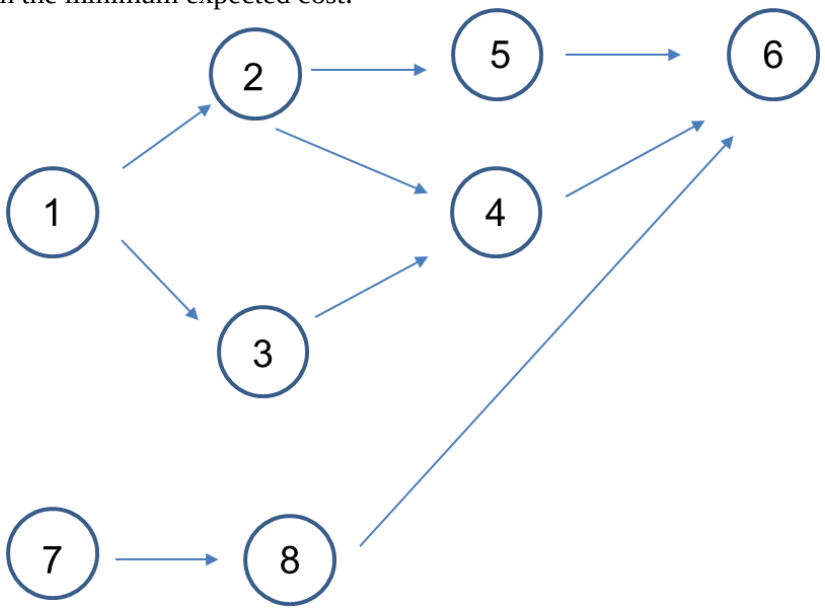


Figure 1: Acyclic Directed Graph

A possible solution:
1-3-2-5-7-4-8-6

Cost minimizing function:

$$\min_{\pi \in P} \sum_{i=1}^{n} c_{\pi(i)} \prod_{j=1}^{i-1} p_{\pi(j)}$$

$$
\begin{aligned}
&c_{\pi(1)} + \\
&p_{\pi(1)}\, c_{\pi(3)} + \\
&p_{\pi(1)}\, p_{\pi(3)}\, c_{\pi(2)} + \\
&p_{\pi(1)}\, p_{\pi(3)}\, p_{\pi(2)}\, c_{\pi(5)} + \\
&p_{\pi(1)}\, p_{\pi(3)}\, p_{\pi(2)}\, p_{\pi(5)}\, c_{\pi(7)}\,) \\
&+ p_{\pi(1)}\, p_{\pi(3)}\, p_{\pi(2)}\, p_{\pi(5)}\, p_{\pi(7)}\, c_{\pi(4)} \\
&+ p_{\pi(1)}\, p_{\pi(3)}\, p_{\pi(2)}\, p_{\pi(5)}\, p_{\pi(7)}\, p_{\pi(4)}\, c_{\pi(8)}\,) \\
&+ p_{\pi(1)}\, p_{\pi(3)}\, p_{\pi(2)}\, p_{\pi(5)}\, p_{\pi(7)}\, p_{\pi(4)}\, p_{\pi(8)}\, c_{\pi(6)}
\end{aligned}
$$

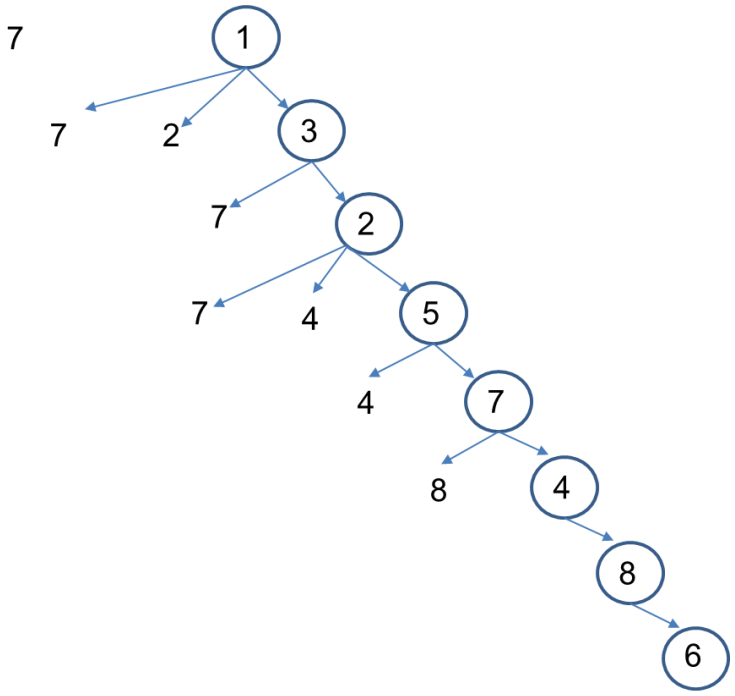


Figure 2: A possible path for the given graph in Figure 1

In this project, we aim to find the best permutation (the minimum cost) among all feasible permutations which satisfies the precedence constraints. We develop two different approach to solve this question: Our project consists of two part. Firstly, we develop a faster heuristic algorithm by using simulated annealing for efficiency concerns for the problem of sequential testing. Secondly, in order to find the cost minimizing permutation for the problem, enumeration of all possible solutions is developed by using a tree data structure.

3 A Faster Heuristic Algorithm by Using Simulated Annealing

Simulated annealing is a method for finding a good (not necessarily perfect) solution to an optimization problem. It accepts worse neighbors in order to avoid getting stuck in local optima; It can find the global optimum if run for a long enough amount of time. The problem of sequential testing with precedence constraints is a good example: we are looking to visit all components sequentially by looking precedence constraints in order to minimize the total expected cost. As the size of components gets larger, it becomes too computationally intensive to check every possible sequence. At that point, we need an algorithm.

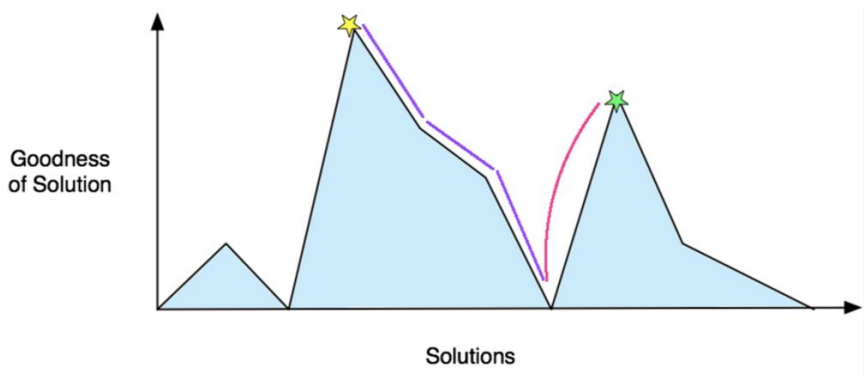


Figure 3: Simulated Annealing Algorithm Graph

For our project, we implement Simulated Annealing algorithm in R programming language. In the implementation, we have three function: Simulated annealing (simulated_annealing), Generate a neighbor sequence (genNeighSeq) and Calculating the cost of sequence (mincost). The key part is to generate a neighbor sequence. This function is recursive. It takes a feasible initial sequence, swaps two random consecutive points which have not a precedence relation and creates a new feasible sequence. We found good solutions close the global optimum at 100 iteration. The algorithm reads 450 txt files (our data) and writes the results in a excel file.

```
#create a neighbour sequence by swapping two consecutive points
genNeighSeq <- function(yy){
  j <- sample(1:(n-1), 1) #create a random number that will be swaped with neigh
  if(prematrix[yy[j], yy[j+1]] == 0){
    tmp2 <- yy[j]
    yy[j] <- yy[j+1]
    yy[j+1] <- tmp2
  } else{
    genNeighSeq(yy) #return function if does not find two consecutive points that has not precedence relation
  }
  return(yy)
}

sq <- seq(from =1, to= n, by=1) #Initial sequence
#objective function
mincost <- function(xx)
{
  r <- NULL
  pr = 1
  for (l in 2:n){
    pr <- pr*prob[xx[l-1]] #probility products
    r[l] <- cost[xx[l]]*pr #create a vector to store each product with cost
  }
  s <- cumsum(r[2:n]) #sum all products in the vector
  summ = cost[xx[1]] + s[n-1] #add the cost of first component

  return(summ)
}
```

```r
simulated_annealing <- function(func, s0, niter = 100, step = 0.8) {

  # Initialize
  ## s stands for state of sequence
  ## f stands for objective function value
  ## b stands for best
  ## c stands for current
  ## n stands for neighbor
  s_b <- s_c <- s_n <- s0
  f_b <- f_c <- f_n <- func(s_n)

  for (k in 1:niter) {
    Temp <- (1 - step)^k
    # consider a random neighbor sequence
    s_n <- genNeighSeq(s_c)
    f_n <- func(s_n)
    # update current state
    # f_n < f_c (p=1) vs f_c < f_n
    #evaluation of acceptance probability
    if (f_n < f_c || runif(1, 0, 1) < exp(-(f_n - f_c) / Temp)) {
      s_c <- s_n
      f_c <- f_n
    }
    # update best state
    if (f_n < f_b) {
      s_b <- s_n
      f_b <- f_n
    }
  }
  return(list(N= n, Type= substr(i, 2, stri_locate_last_fixed(i, 'A')),
              Instance= as.numeric(substr(i, stri_locate_last_fixed(i, 'A')+1, (nchar(i)-4))),
              Best_Value = f_b))
}
```

Figure 4: The Implementation of Simulated Annealing Algorithm

After the implementation, we change simulated annealing parameters that are temperature and iteration number. We compare the results. Then, we take the optimality gap with optimal solutions for each 450 data and take the average optimality gap for each parameter in R. Then, we write results in a excel file. We realized that as iteration number increases, the results are closer to optimal solutions, but there is no significant change with temperature factor. Tables at the bottom shows the average optimality gap. Iteration numbers are 200 vs 100 from top to bottom.

N	Average of gap1_T1	Average of gap2_T5000	Average of gap3_T10000
12	0.046862061	0.041887763	0.043707631
20	0.04906011	0.051389186	0.049505382
50	0.03284961	0.042362008	0.03583971
100	0.081736001	0.078531407	0.076992747
200	0.041722509	0.058489951	0.053930083
Grand Total	0.050446058	0.054532063	0.051995111

N	Average of gap1_T1	Average of gap2_T5000	Average of gap3_T10000
12	0.053078477	0.052592884	0.052489446
20	0.058871529	0.056236679	0.060872264
50	0.067879709	0.06362445	0.068714526
100	0.106441551	0.100481755	0.109619567
200	0.058871338	0.070088152	0.070498499
Grand Total	0.069028521	0.068604784	0.07243886

Figure 5: Tables of Average Optimality gap with Simulated Annealing Parameters for each Data size

4 Enumeration of All Possible Solutions Using a Tree

We develop an enumeration algorithm which finds and sorts all feasible permutations under precedence constraints in order to find the cost minimizing solution. Since not all permutations are feasible under precedence constraints the first main objective of our algorithm is the enumeration of all feasible permutations.

We build a General Tree Type data search algorithm. Each tree starts with a root which does not have any precedence constraint and add all possible members of the next generation. Thus, the second level of the tree is created. Each second-generation child adds its own children and create the third level. It goes on like this until the N_{th} generation. The N_{th} generation is called as leaves of the tree. Each leaf is a possible solution which follows a unique path.

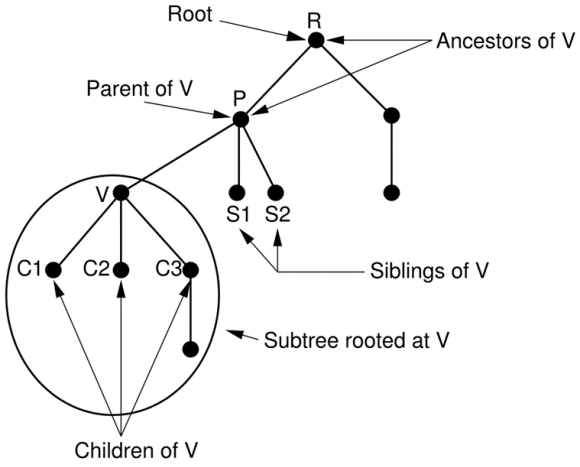


Figure 6: Tree Structure

Components	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	0	0	0	1	1	0	0	0
3	0	0	0	1	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0

Figure 7: The Adjacency Matrix of the Graph in the Figure 1

When a column of the matrix consists of all zeroes, that column is the root of a tree. After chosen root we extract the column and the row belongs to that component. Simply we are extracting Minor ($M_{i\times i}$) of that component. All zero columns of minor will give us children for next generation. Until the minor becomes 1 x 1 matrix algorithm keep on extracting and adding new generations. When it reaches to end genetic info of leaves keeps a unique solution.

Components		2	3	4	5	6	7	8
2		0	0	1	1	0	0	0
3		0	0	1	0	0	0	0
4		0	0	0	0	1	0	0
5		0	0	0	0	1	0	0
6		0	0	0	0	0	0	0
7		0	0	0	0	0	0	1
8		0	0	0	0	0	0	0

Figure 8: Step 1 of the Constructed Algorithm

All zeroes columns are 2, 3, 7. Therefore, there are 3 children of the root 1. There can be sequences start with 1-2, 1-3, 1-7. At this stage the leftmost child is 1-2. The algorithm keeps on searching until the first leaf by looking the leftmost child. When the algorithm reaches the leftmost leaf, then it starts to search for a right sibling until there is no right sibling left. It moves parent node and same process repeats for the possible children 3, 5, 7.

Genetic info: 1-2
Possible children: 3, 5, 7

Components	1	2	3	4	5	6	7	8
1								
2								
3			0	1	0	0	0	0
4			0	0	0	1	0	0
5			0	0	0	1	0	0
6			0	0	0	0	0	0
7			0	0	0	0	0	1
8			0	0	0	0	0	0

Figure 9: Step 2 of the Constructed Algorithm

Genetic info: 1-2-3
Possible children: 4, 5, 7

Components	1	2	3	4	5	6	7	8
1								
2								
3								
4				0	0	1	0	0
5				0	0	1	0	0
6				0	0	0	0	0
7				0	0	0	0	1
8				0	0	0	0	0

Figure 10: Step 3 of the Constructed Algorithm

Genetic info: 1-2-3-4
Possible children: 5, 7

Components	1	2	3	4	5	6	7	8
1								
2								
3								
4								
5					0	1	0	0
6					0	0	0	0
7					0	0	0	1
8					0	0	0	0

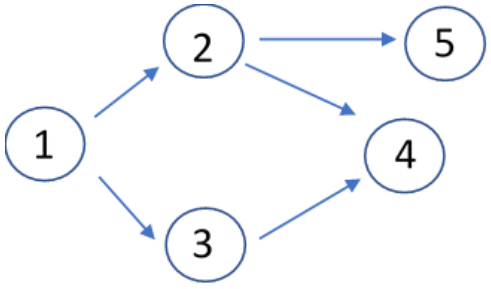
Figure 11: Step 4 of the Constructed Algorithm

Genetic info: 1-2-3-4-5
Possible children: 6, 7

Components	1	2	3	4	5	6	7	8
1								
2								
3								
4								
5								
6						0	0	0
7						0	0	1
8						0	0	0

Figure 12: Step 5 of the Constructed Algorithm

Building an example with 5 components:



	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	1
3	0	0	0	1	0
4	0	0	0	0	0
5	0	0	0	0	0

Figure 13: Acyclic Directed Graph and Adjacency Matrix of an Example Sequence

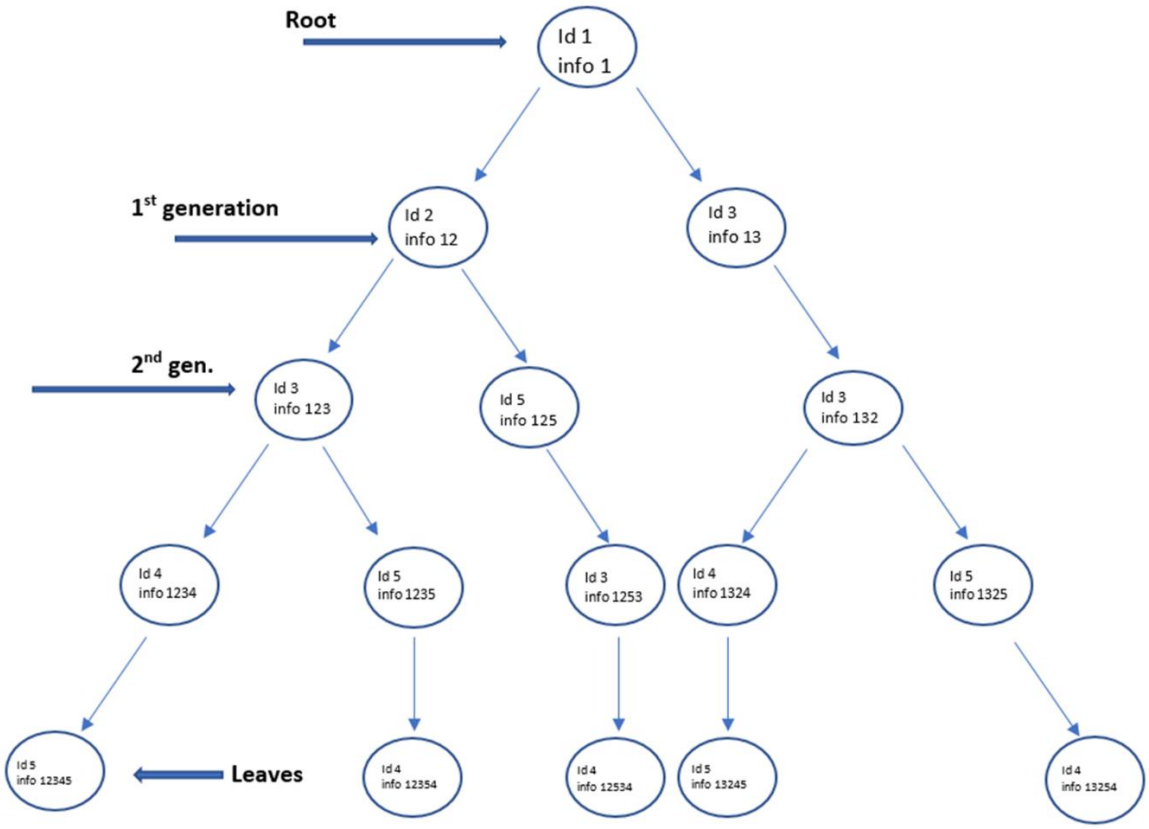


Figure 14: Example Results of the Constructed Algorithm

Implementation is done by using programming language C++. Simple series system data is given as an adjacency matrix of a directed graph which also includes probability and the cost of testing. To represent a component a data aggregator is implemented. GTreeNode struct includes following data:

1- The **id** of the component,

2- Genetic **info** from previous generations that keeps the path passed through.

3- An unsigned integer variable keeps the **generation** number of the current node.

4- A node pointer points to **parent** node to move previous generation.

5- Second pointer points to the **leftmost child** to create a new generation.

6- The third one points to the **right sibling**.

7- A matrix data to find children of the next generation. **Matrix** is derived from the original adjacency matrix according to chosen component. Matrix is implemented in the integer type vector of vectors.

```
class GTree
{

public:

    GTree(string);//Default constructor
    GTree(const GTree &);//Copy constructor
    ~GTree();//Destructor
    void addLeftMostChild(string);
    void addRightSibling(string);
    void moveParent();
    GTreeNode* FindLeaf();
    void deleteNode();
    unsigned int num_Children () const;
    unsigned int Size() const;
    bool isLeaf( GTreeNode* ) const;
    bool isLastChild(GTreeNode* ) const;
    bool isRoot();
    void Init();

private:

    GTreeNode* root;
    unsigned int size;
    GTreeNode* current;
    //friend class GTreeIterator ;
};
#ifndef _GTREE
#define _GTREE

#include <string>
#include <vector>

using namespace std;


struct GTreeNode
{
    string id;
    string info;
    unsigned int generation;
    GTreeNode* parent;
    GTreeNode* rightSibling;
    GTreeNode* leftMostChild;
    vector<vector<int>> Matrix;

    GTreeNode::GTreeNode () { ... }

    GTreeNode::GTreeNode //Constructor with parameters
    (string i, string s, unsigned int g, GTreeNode*p = NULL, GTreeNode*rS = NULL, GTreeNode* lmc = NULL) { ... }

    GTreeNode::GTreeNode(string i) { ... }

};
```

Figure 15: The Implementation of Enumeration Algorithm

5 Conclusion and Future Work

We realized that as iteration number increases, the results are closer to optimal solutions, but there is no significant change with temperature factor for simulated annealing heuristic algorithm. Simulated annealing is a metaheuristic method. Future work can be on metaheuristic. Better results can be found with other metaheuristic methods.

Implementation of the enumeration algorithm has not finished yet. Total cost of each leaf will be calculated and stored in a container data type. After enumeration succeeds Search for the minimum algorithm will find the feasible solution with minimum cost.

References

Çatay, B., Özlük, Ö., Ünlüyurt, T. (2011). TestAnt: An ant colony system approach to sequential testing under precedence constraints. *Expert Systems with Applications*, 38 (12), 14945-14951. doi: 10.1016/j.eswa.2011.05.053.