SOLUTION OF DETERMINISTIC ASSEMBLY LINE BALANCING (ALB) PROBLEMS USING EVOLUTIONARY OPTIMIZATION ALGORITHMS

A report submitted in partial fulfillment of the requirements for the course

of

OPERATIONS RESEARCH

MEE437

by

PRASHANTH K R

09BME302

SCHOOL OF MECHANICAL AND BUILDING SCIENCES

Vellore – 632014, Tamil Nadu, India

www. vit.ac.in

APRIL 2012

CONTENTS

1. INTRODUCTION

1.1. Project Abstract

Assembly Line Balancing (ALB) is a task performed in order to level the workload across all processes in a cell or value stream to remove bottlenecks and excess capacity. In today's age of mass production, ALB plays a major role in achieving maximum efficiency in a supply chain. The productivity level of an assembly line generally depends on balancing performance. ALB is the problem of assigning tasks to successive workstations by satisfying some constraints and optimizing a performance measure. This performance measure is usually the minimization of the number of workstations utilized over the assembly line. Conventionally, these problems are tackled using heuristic procedures such as COMSOAL Algorithm or the Rank Positional Weight Algorithm depending on the nature of the balancing problem. In this project, I will try to implement a generalized method of solving ALBs using global search optimization algorithms. The project mainly focuses on solution of **Deterministic ALBs**.

I would be implementing a technique which is a modified version of a popular evolutionary optimization method called **Bacterial Foraging Optimization Algorithm** (BFOA/BFA).

Flow of the work goes as follows:

1. Select the parameter to be optimized (here, it is balancing efficiency)
2. Code a cost function from the input data the work allotment towards each machine as variables keeping in mind the precedence matrix
3. Implement Global Search Optimization methods as a code compatible to the cost function
4. Analyse the results obtained and compare with those from conventional techniques.

As a future prospect, we may take lessons from above and develop another computationally economical heuristic that outperforms the known methods of solving these problems. We can also undertake a comparative study that measures the computational time of the algorithm with respect to other techniques.

Data would be taken from **Dr. S. Narayanan's** work on developing a **New Efficient Heuristic** (NEHU) for solving ALBs. The study would be on the turbocharger manufacturing assembly line of **Turbo Energy Limited**, Pulivalam.

1.2. Assembly Line Balancing (ALB) Problem Solving

The most important aspect of an ALB is the **strict adherence to the precedence order** of the given tasks. While balancing, one has **no liberty to change the flow of work**, but can only streamline the same flow by **better segregation between successive workstations**.

In order to maintain the precedence order as described by the precedence matrix, a general approach is designed so that task assignment to workstations never violates the desired flow irrespective of the balancing efficiency achieved. This method, referred to as the **Generalized Algorithm** [Panneerselvam, et al, 1993] in standard text books ensures that no violation of precedence relationship occurs throughout the algorithm.

At every step of assignment this approach will generate lists of tasks that are eligible to be placed in the current workstation. A **global search algorithm will augment this process** by providing a heuristics for selection from the lists generated by the Generalized Algorithm.

The problem of ALB falls under the umbrella of **combinatorial problems** where placement of objects from one set to another influences the performance of the system.

In this problem, the main task is to maximize balancing efficiency by reducing idle time in each workstation.

2. LITERATURE REVIEW

1. *Prof. S. Narayanan's* doctoral work under the guidance of *Dr. Panneerselvam* titled **Development of New Efficient Heuristic for Deterministic Assembly Line Balancing Problem** is the key source of inspiration for this work. This exhaustive and highly detailed paper provides not only numerical data with solutions, but also a baseline for benchmarking new approaches to solving ALBs
   .
2. *Christian Becker & Armin Scholl* conducted a study on **A Survey on Problems and Methods in Generalized Assembly Line Balancing** where they compared various unconventional methods of solving straight and u-shaped ALB problems.

3. *Yakup Atasagun & Yakup Kara* actually implemented **Assembly Line Balancing Using Bacterial Foraging Optimization Algorithm**. The paper only gives the results of their findings across 64 solved ALB problems. It provides no code but some insight into how they implemented the algorithm.

4. *Emanuel Falkenauer* wrote a paper titled **Line Balancing in the Real World** where he talks about the gaps between the academic approach of solving ALBs and the problems industries face. The paper very lucidly describes this with a practical outlook in mind.

5. *S. G. Ponnambalam et al* from NIT Trichy implemented **A Multi-Objective Genetic Algorithm for Solving Assembly Line Balancing Problem**. The approach is compared with six widely used heuristic practices and the paper talks a lot about global search techniques in this field.

6. *Adil Baykasoglu and Türkay Dereli* published work on **Simple and U-Type Assembly Line Balancing by Using an Ant Colony Based Algorithm**. This paper proved that global search techniques can be more effective in solving such problems through a computational study. This paper served as one of the most useful resources.

7. *Nils Boysen et al* successfully attempted to answer questions on the key aspects of **Assembly line balancing: Which model to use when?** In this work, they gave a very easy to understand overview of various approaches to ALBs and pros and cons of each approach. As engineers, it is very essential to select the right approach.

8. *Sophie D. Lapierre et al* demonstrated **Balancing Assembly Lines with Tabu Search**, a global search combinatorial optimization technique. The performance of the algorithm is then evaluated on a large set of problems followed by a discussion the flexibility of the meta-heuristic and its ability to solve real industrial cases.

9. *Matthias Amen's* work titled **Heuristic Methods for Cost-Oriented Assembly Line Balancing: A Comparison on Solution Quality and Computing Time** talks about the various compromises a developer has to make when he creates his own heuristic to solve any kind of an optimization problem. The paper also demonstrates cases where the author achieves significantly better results than conventional priority based rules.

10. *Armin Scholl & Robert Klein* performed a numerical experiment in their work, **Balancing Assembly Lines Effectively: A Computational Comparison** where an extensive study was undertaken in search for the most effective branch and bound procedures for solving simple ALB problems.

3. METHODOLOGY

3.1. Problem Statement

The objective here is to balance an assembly line of a Turbocharger Manufacturing Facility which consists of the following tasks with the given duration as well as immediately preceding tasks:

Node	Job Description	Time	Preceding Tasks
1	Sub-Assembly of Back Plate	32	None
2	Sub-Assembly of Central Housing	105	1
3	Compressor Wheel Gap Testing	40	2
4	Turbine Wheel Gap Testing	40	2
5	Heat Shield Gap Testing	40	2
6	Sleeve Height Testing	40	2
7	Socket Head Bolt Tightening & Torque Setting	45	3, 4, 5, 6
8	Parts Assembly 1	90	7
9	Parts Assembly 2	30	8
10	Parts Assembly 3	30	9
11	Parts Assembly 4	30	10
12	Bolt Tightening & Clamp Tightening	100	11
13	Torque Application	70	12
14	Angle Setting by Bevel	42	13
15	Radial Play Testing	35	14
16	Axial Play Testing	42	14
17	Compressor Housing & Segment Bolting	70	11
18	Torque Application	80	17
19	Angle Setting	46	18
20	Name Plate Punching	120	1
21	Riveting Name Plate with Compressor Housing	30	20
22	Final Viewing	30	15,16,19,21

The flow chart of the same can be seen in the appendix. The following table would be formatted into an Excel Spreadsheet and will be used as input in MATLAB.

The company operates one shift of 8 hours a day. The targeted production volume of the turbocharger is 160 units per shift. As per these details, the **cycle time is 180 seconds**.

3.2. Global Search Techniques

These are a special class of optimization algorithms that **search the entire feasible set** of possible solutions to provide the most optimal set of parameters for the system. Since it is not computationally possible to search the entire feasible set by repeatedly plugging in every possible value into the objective function (in this case, say balancing efficiency) and then selecting the parameters that give the best value of the objective function, global search methods are usually also a **set a heuristics that give a direction to search** for optimal solutions.
In other words, these methods **narrow the search area for possible solutions**, thereby reducing computational loads compared to **brute-force methods**. These are very efficient techniques of solving multivariable as well as combinatorial problems as they provide avenues for testing the right set of combinations that lead to the optimal solution.

3.3. Evolutionary Optimization Algorithms

Many Global Search Techniques are inspired from physical and biological phenomena and are therefore referred to as **Evolutionary Optimization Algorithms**.
Some examples of such algorithms are:

- **Genetic Algorithm**, inspired from Darwin's theory of Natural Selection (Evolution of Species) that successfully explains the level of sophistication different living creatures have achieved depending on the natural constraints on the species.
- **Particle Swarm Optimization**, inspired from the searching/hunting techniques of swarm creatures such as honeybees, locusts or hornets. They work by constantly communicating locations of high food availability to the rest of the swarm and maintaining a memory of previous locations for the same.
- **Ant Colony Optimization** is inspired from the way social terrestrial insects such as ants and termites search for food locations. The worker

ants leave behind a trail of pheromones which indicates to the next batch of workers locations of high food availability. Greater workforce is allocated to such locations.

- **Simulated Annealing**, inspired from the physical process of annealing, where random motion of particles inside a mass of near molten metal tries to achieve a state of minimum energy through slow cooling.

3.4. Bacterial Foraging Algorithm

This is the global search evolutionary algorithm that would be utilized in this project. It has been chosen due to its simplicity and ease in implementation as well as the efficiency it offers. The algorithm is inspired from the method in which bacteria and other unicellular organisms search for nutrition and reproduce inside a liquid matrix.

These creatures reproduce through binary fission where the bacterium splits itself into two independent organisms. In order to be in a position to do the same, it must gain adequate nutrition. Here, nutrition refers to the extent of optimality achieved by the algorithm.

Bacteria usually die and fail to reproduce where the availability of nutrition is deficient. The algorithm translates this phenomenon to lessening the number of search particles in regions where solutions of low optimality are found.

One reproduced, the bacteria scatter in random directions from the place where the parent got its last nutrition, i.e. the place where birth took place. The algorithm, tries to search the region around a place of a more optimal solution by increasing the number of particles (by reproducing for) searching in hope of achieving greater optimality in the vicinity.

Steps:

1. Generate random vectors for possible solutions.
2. Compute Fitness
3. Kill unfit (bottom 50%) and reproduce fit (top 50%)
4. Tumble and Run (add a small random vector to the new population)
5. Go back to step 2 till stopping condition reached.

3.5. Proposed Hybrid Algorithm

The proposed algorithm takes lessons from the Generalized Algorithm for maintaining precedence and the bacterial foraging algorithm as the governing heuristic for selection of tasks once lists for assignable tasks (List A) is generated for each vacancy in the workstation.

The flow of the algorithm is as follows:

1. Input the flowchart of the process through a precedence matrix.
2. Input the duration of each task and cycle time.
3. Calculate minimum no. of workstations required based on cycle time, and total duration.
4. Using BFA, generate weights for each task for each workstation.
5. From the Generalized Algorithm, generate lists for candidate tasks for each workstation sequentially.
6. Using the weights generated using the BFA, assign tasks.
7. Compute the Balancing Efficiency
8. Repeat iterations of the BFA till satisfactory solution is reached.
9. Print out the final workstation assignment matrix of the optimal assembly line.

The algorithm is implemented in such a way that the input data of any kind of an assembly line layout can be imported in the form of a CSV file generated in MS Excel. The BFA parameters such as population size and no. of iterations can also be customized as per the desired CPU time.

The output of the algorithm is balancing efficiency, unassigned cycle times for each workstations and the final assignment matrix.

4. RESULTS AND DISCUSSION

4.1. Results and its Significance

The minimum no. of workstations required is computed using the relation:

$$N_{min} = \frac{Sum\ of\ task\ durations}{Cycle\ Time}$$

The value comes out to be 6.53. Hence, we can take 7, 8 or even 9 workstations.

If we take **N=8**; our optimal solution is as follows:

Work Station	Selected Work Element	Idle Time
1	1,2,3	3
2	4,5,6,7	15
3	8,9,10,11	0
4	12,13	10
5	14,15,16	61
6	20,21	30
7	17,18	30
8	19,22	104

Total Idle Time	253 seconds
Balancing Efficiency	**82.4306%**

The following result is equivalent to the values calculated using the conventional methods of **HAL** and **CWF** (Composite Weight Factor) which gives an efficiency of **82.4%**.

If we take **N=7**; our optimal solution is as follows:

Work Station	Selected Work Element	Idle Time
1	1,2,3	3
2	4,5,6,7	15
3	8,9,10,11	0
4	12,13	10
5	14,20	18
6	17,18,21	0
7	15,16,19,22	27

Total Idle Time	73 seconds
Balancing Efficiency	**94.2063%**

The above result is very similar to the one obtained using NEHU (New Efficient Heuristic), a technique developed by Prof. S. Narayanan as a part of his doctoral research.

For **this particular problem**, NEHU gave the **most optimal** arrangement. In order the compare the effectiveness of the proposed hybrid algorithm and NEHU, we need to try out more problems of a larger scale. The proposed algorithm also generated the most optimal solution possible for the above problem. Further experimentation is necessary. The algorithm, being modular in nature can easily be tweaked for further work. New data can be added simply by replacing the 'data.csv' file in the folder.

The fact that the hybrid algorithm is able to reproduce results of the tried and tested CWF as well as NEHU methods means that the technique is strongly grounded and is demonstrating practical results.

4.2. Scope for Improvement

The algorithm is primarily stochastic in nature as results are computed purely on the basis of random numbers generated in the various stages of the program. Due to this, the algorithm often gives layouts where all the tasks are not assigned, i.e. certain tasks do not belong to any workstation. Such results are useless from a practical perspective.

To eliminate these results, I used a **penalty approach** where efficiency is put to zero for all those solutions where complete assignment does not take place. Efficiency is only computed after checking complete assignment. Among those solutions, the best is chosen.

Due to lack of familiarity with the field of Algorithm Analysis from a computer science perspective, I am unable to measure the computing efficiency. But as a developer, it is very much possible to reduce the number of loops, iterations and variables with more careful thought.

5. CONCLUSIONS

The newly developed algorithm works and gives comparable results to conventionally used heuristics although the algorithm is far from complete perfection. Further experimentation needs to be done for various straight-type deterministic assembly line problems. Repeated trails with appropriate corrections to the code as and when erroneous results are discovered can lead the algorithm to perfection.

Development of an algorithm is different from its code implementation. Code implementation is carried out in a specific language where in this case I am using MATLAB. The code needs to be debugged meticulously while analyzing possible mistakes in the statements. The errors in programming don't show up in all test data. This can lead to unintended errors in the results. Therefore, converting this into a working application will take more time.

However, the algorithm is grounded soundly and demonstrates tremendous future scope. For instance, the BFA can be modified to use different weight vectors for different work stations. The BFA algorithm can be readily replaced with any other global search technique.

Further guidance and appropriate allocation of time can take this project a very long way and could possibly pave way for a new approach to Operations Research algorithms augmented by Soft Computing Heuristics.

REFERENCES

- Production and Operations Management by Dr. Panneerselvam
- Prof. S. Narayanan's Doctoral Thesis titled DEVELOPMENT OF NEW EFFICIENT HEURISTIC FOR DETERMINISTIC ASSEMBLY LINE BALANCING PROBLEM

APPENDIX I: MATLAB CODE

The code is divided into 3 components the scripts (.m files) of which must be in the same folder for the program to run.

a) Problem Definition and BFA

```
CLC
global T N C P D PR;

P=csvread('data.csv');
D=P(:,1); %duration of each task
P(:,1)=[]; %immediate predecessors of the task
[a1,b1]=size(P);
T=a1; %no. of tasks
PR=b1;
      C=180; %cycle time
      a2=(sum(D))/C;
      %N=1+a2-rem(a2,1); %no. of workstations
      N=9;
      %declared input data

      %BFA Begins
      iter=15;
      pop=3000;
      dim=T;
      cv=[];
      ibest=[];
      population=[];
      for p=1:pop
          particle(p).x= C*(rand(dim,1));
          population=[population particle(p).x];
          particle(p).c= cost(population(:,p));
          cv=[cv particle(p).c];
      end
      [least index]=min(cv);
      gmin=least;
      ibest=[ibest least];
      ibestx=population(:,index);
      gminx=population(:,index);
      %BFA trail solutions declared
      %BFA begins here
      for j=1:iter
          %deleting the worst 50 percent elements
          for p=1:(pop/2)
              [most imost]=max(cv);
              population(:,imost)=[];
              cv(imost)=[];
          end
          cv=[];
```

```
        %copying/reproducing the fittest five elements
        population=[population population];
        %make the bacteria tumble and run
        population=population+(C*rand(dim,pop));
        for p=1:pop
            particle(p).c= cost(population(:,p));
            cv=[cv particle(p).c];
        end
        [least index]=min(cv);
        ibest=[ibest least];
        if(least<gmin)
            gmin=[gmin least];
            gminx=population(:,index);
        end

    end
    %final output
    efficiency=(1-least)*100  %efficiency
    assignment=assign(population(:,index))    %final
assignment matrix
```

b) Cost the Assembly Line

```
function c = cost(x)

global T N C P D PR;
x=x.*x;
x=x';
a=zeros(1,T);
suact=[];
list=zeros(1,T);
assign=zeros(N,T);
    flag=zeros(1,T);

    %assign the first task to the first station
    list(1)=1;
    assign(1)=1;
    flag(1)=1;
    u=C-D(1);
    mintime=min(D); %smallest duration of task

    %start assigning the remaining
    for p=1:N
        flagw=0;
        if p==1
            UACT=u;
        else
            UACT=C;
        end
        j=2;
```

```
        while flagw==0
            if j<=T && j~=1
                if list(j)==1 %check if task already
assigned
                    j=j+1;
                    continue;
                elseif flag(j)==1 %check if task in list
A
                    j=j+1;
                    continue;
                elseif P(j,1)==0 %check if no prereq
needed
                    a(j)=1;
                    flag(j)=1;
                    j=j+1;
                    a(1)=0;
                    continue;
                else
                    flagt=1;
                    for q=1:PR
                        if P(j,q)==0  %check if prereq
exists in P matrix
                            a(j)=1;
                            flag(j)=1;
                            j=j+1;
                            %q=1;
                            flagt=1;
                            a(1)=0;
                            break;
                        elseif list(P(j,q))==0 %check if
prereq is assigned. reject if not
                            flagt=0;
                            break;
                        elseif a(P(j,q))==1 %check if
prereq in list A reject. if yes
                            flagt=0;
                            break;
                        end
                    end
                    if flagt==0
                        j=j+1;
                        continue;
                    end
                end
            end

            wa=zeros(1,T); %make a wieght matrix
generated using BFA with values for only list A tasks
            wa=(a.*x).*(ones(1,T)-list);

            wa(1)=-100;
                [wt index1]=max(wa);
```

```
                if UACT-D(index1)>=0 %if cycle time
permits, assign it to assignment matrix, remove from list
A, and tick on the assigned list
                    assign(p,index1)=1;
                    UACT=UACT-D(index1);
                    wa(index1)=0;
                    list(index1)=1;
                    a(index1)=0;
                    j=2;
                    if UACT<mintime
                        flagw=1;%move to next workstation
                        suact=[suact UACT];
                    end
                else
                    flagw=1;%move to next workstation
                    suact=[suact UACT];
                    a=zeros(1,T);
                end

            end
        end
        %final output
        if sum(list)==T && sum(sum(assign))==T %checking if
all tasks have been assigned and no task has been assigned
more than once
            idle=sum(suact);
            c=idle/(C*N);
        else
            c=1;
        end
```

The **assignment code** is similar to the previous code, only it returns the assignment matrix, unassigned cycle time and list of assign tasks when called instead of cost.

APPENDIX III: GENERALIZED ALGORITHM

7.2.1 Generalized Algorithm [Panneerselvam, et al., 1993]

Step 0. Input: cycle time (CT), precedence matrix, total number of work elements (M), work elements times.
Initialize: unassigned cycle time (UACT) = CT, station number (SN = 1) and total number of assigned work elements (N) = 1.

Step 1. Print the station number (SN)

Step 2. Initialize the total number of assigned work elements counter to one, i.e. $I = 1$.

Step 3. If the Ith work element is already assigned, go to Step 7, otherwise go to Step 4.

Step 4. If all the immediate precedors of the Ith work elements are assigned, then go to Step 5; otherwise go to Step 7.

Step 5. If the Ith work element time is less than or equal to the unassigned cycle time (UACT), then go to Step 6, otherwise go to Step 7.

Step 6. Include the Ith work element in the List A which consists of work elements whose all immediate precedors are assigned and the processing time of each work element is less than the unassigned cycle time (UACT).

Step 7. Increment the total number of assigned work elements counter by one, i.e. $I = I + 1$.

Step 8. If $I \leq M$, then go to Step 3; otherwise go to Step 9.

Step 9. If the number of work elements in the List A is more than 0, then go to Step 11; otherwise go to Step 10.

Step 10. Perform the following and then go to Step 2.

(i) Add UACT to the sum of unassigned cycle time (SUACT)

(ii) Update UACT = CT

(iii) Update SN = SN + 1

(iv) Print SN.

Step 11. If the number of work elements in the List A is equal to one, then assign that work element to the current station; otherwise select a work element from the List A using a given HEURISTIC.

Step 12. Subtract the recently assigned work element time from UACT and print the recently assigned work element's number.

Step 13. Update $N = N + 1$

Step 14. If $N \leq M$, then go to Step 2; otherwise go to Step 15.

Step 15. Compute the Balancing Efficiency (BE) in percentage.

$$BE = \left[1 - \frac{SUACT}{CT*SN}\right] * 100$$

Step 16. Stop.

APPENDIX III: MATLAB OUTPUT

For N=7

```
efficiency =

   94.2063

suact =

     3    15     0     10     18      0     27

list =

     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1

assignment =

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


     1     2     3     4     5     6     7     8     9    10    11    12    13    14    15    16    17    18    19    20    21    22
```

For N=8

```
efficiency =

    82.4306

suact =

    3    15     0    10    61    30    30   104

list =

    1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1     1

assignment =


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


    1     2     3     4     5     6     7     8     9    10    11    12    13    14    15    16    17    18    19    20    21    22
```