Disassembly Sequencing using Artificial Bee Colony Algorithm

Ji-won Choi, ECE788: Computational Intelligence, Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, New Jersey 07102 USA, Email: jc423@njit.edu

***Abstract*—In disassembly process for end-of-life (EOL) product, optimization for sequence is important factor of efficiency. Optimization of disassembly sequencing is a challenging problem, as some components depend on the order of disassembly, and some components may be resold or recycled. In this paper, we attempt to apply Artificial Bee Colony (ABC) algorithm to optimize EOL product disassembly sequencing problem introduced by Elif Kongar [1]. The result and performance of ABC algorithm approach is compared to the Genetic algorithm (GA) approach of Kongar.**

***Index Terms*—Diassembly sequencing, artificial bee colony, optimization**

I. INTRODUCTION

As the time goes by, any product will reach its end-of-life stage, where the consumer deposes of the product. In the current years when electronics are becoming outdated quickly, some products may reach EOL stage when the product or the components of the product may still be in good condition. These EOL products can go through a disassembly process where components of the product may be resold, reused, or recycled, possibly bringing revenue.

Just as assembly sequence can be optimized to reduce the production time and cost, disassembly sequence can be optimized. In a certain aspect, disassembly can be more complicated; the components are collected for different causes (recycle or reuse), and the components have precedence relationships. To obtain the optimal disassembly operation, every possible disassembly sequencing method can be evaluated. However, this process may take a great amount of cost and time, and may not even be possible to do in real world case. The use of optimization algorithm allows efficient way to search optimal or near optimal disassembly operation. In this paper, we present a case study where product is disassembled into 10 components with certain restrictions such as precedence relationships. By using Artificial Bee Colony algorithm, this paper attempts to find the near-optimal or optimal disassembly operation to the disassembly sequencing problem.

A. Organization

The paper is organized as follows. Section II describes a disassembly sequencing problem and establishes its mathematical model. Section III describes design of Artificial Bee Colony Algorithm for the problem. Section IV provides the results of Artificial Bee Colony Algorithm. Section V compares the performance and the results of Artificial Bee Colony Algorithm to the performance and results of GA algorithm by Kongar. Lastly, conclusion is given in section VI.

II. PROBLEM STATEMENT

A. Notation

1) j : Index for component
2) s : Index for disassembly sequence ($s = 0, ..., 9$)
3) $t_{j,s}$: Disassembly time for component j in sequence s
4) $d_{j,s}$: Penalty for each direction change for component j in sequence s
5) $m_{j,s}$: Penalty for disassembly method change for component j in sequence s
6) $D_{j,s}$: Type of demand for component j in sequence s
7) $M_{j,s}$: Material type of component j in sequence s
8) T_s : Disassembly time after the disassembly operation sequence s
9) T_{total} : Total disassembly operation time j
10) n : Total number of components in EOL product
11) b : Index of bee
12) F_b : Fitness of bee b
13) P_b : Selection probability for employed bees
14) b_e : Number of onlooker bees

B. Problem Statement

The goal of ABC algorithm in this problem is to optimize the disassembly process. For this problem, we consider the disassembly operation which took the least time as the optimal operation. This EOL disassembly consists of 10 components, each of the component is associated with a direction of disassembly, method of disassembly, type of demand (resale, recycle, or none), and type of material. The total disassembly time, T_{total}, is associated with this information. A diagram in figure. 1 and table I shows an overview of properties of all the components. The 10 components are indexed from A to J, represented by set $j \in$\{A,B,C,...,J,I\}. Certain components must be processed prior to specific components, as shown in figure. 1; product B and C must be processed prior to any other products, product H must be processed prior to product D and G, and product G must be processed prior to product E and F.

Each of the components also must be disassembled in one of six directions: +x, +y, +z, -x, -y, or -z. Change of direction

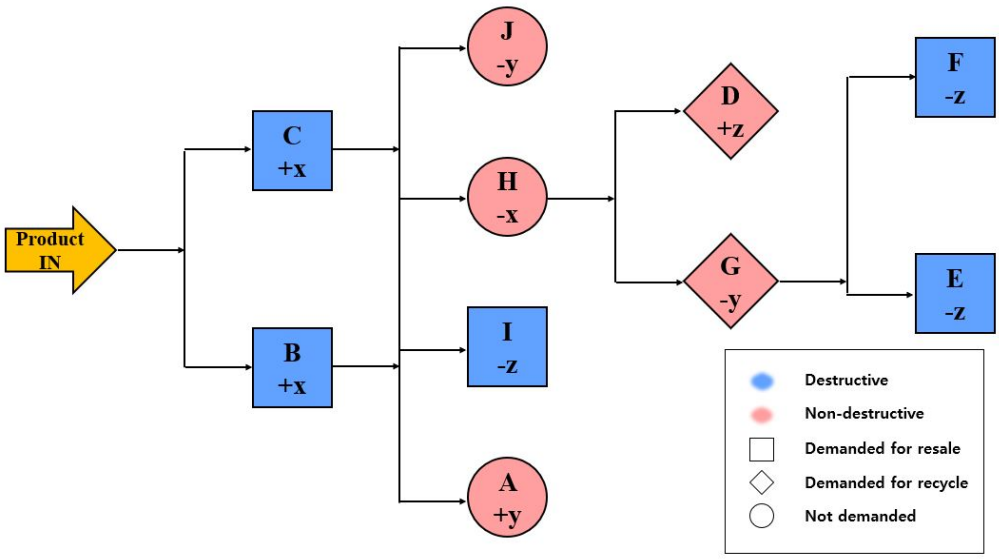

Fig. 1: Original product structure of the EOL product

TABLE I: Input data for disassembly sequencing example

Variable	Value									
j	A	B	C	D	E	F	G	H	I	J
$d_{j,s}$	2	3	3	2	3	4	2	1	3	2
Direction	+y	+x	+x	+z	-z	-z	-y	-x	-z	-y
Method	D	N	N	D	N	N	D	D	N	D
Demand	0	1	1	2	1	1	2	0	1	0
Material	A	S	S	S	P	P	S	P	P	P

in disassembly process is penalized, having a bigger penalty for direction change in completely opposite direction (e.g. from y to -y). The directional change penalty, $d_{j,s}$, is defined as

$$d_{j,s} = \begin{cases} 0 & \text{If no direction change required} \\ 1 & \text{If } 90^\circ \text{ change is required} \\ 2 & \text{If } 180^\circ \text{ change is required} \end{cases} \quad (1)$$

The disassembly method for each of the components can be either destructive (D) or non-destructive (N). The penalty for disassembly method is applied when there is a change in method of disassembly during the sequence (e.g. from N to D). The disassembly method change penalty, $m_{j,s}$, is defined as

$$m_{j,s} = \begin{cases} 0 & \text{If no method change required} \\ 1 & \text{If method is required} \end{cases} \quad (2)$$

Some components are demanded for reuse or recycle. Demand for reuse is represented by 1 and recycle is represented by 2. For components that are not demanded are represented by 0. Lastly, a component can be made up of aluminum (A), plastic (P), or steel (S). In the case of a sequence consisting of a pair of components of same material and demanded for recycling, the disassembly time and penalties are assigned to zero. In this problem, only component D and G fit the criterion. For example, if G is processed after D, the disassembly time and penalties for direction change and method change are assigned to zero. Using these information with disassembly time for component, $t_{j,s}$, the total time of the disassembly operation, T_{total}, can be calculated. For the case at which sequence does not consist of a pair of components of the same material and demanded recycling, the cumulative time function is as follows:

$$T_s = \begin{cases} T_{s-1} + t_{j,s} + d_{j,s} + m_{j,s} & s = 0, ..., n-2 \\ T_{s-1} + t_{j,s} & s = n-1 \end{cases} \quad (3)$$

The reason for last sequence not being penalized is because the last sequence does not have change in direction of method of disassembly for the next sequence. T_s for $s = n-1$ is T_{total}. The pseudocode for the cumulative time function is shown in Algorithm 1. After the total disassembly time has been calculated, to change the problem from minimum search to maximum search, the fitness is equal to a certain constant C subtracted by the total disassembly time. The constant C is set to be bigger than possible value of T_{total}. Following the Kongar's problem statement, C is set as 100.

Fig. 2: Overall flowchart for Artificial Bee Colony Algorithm)

$$F_b = C - T_{total} \tag{4}$$

Algorithm 1 Pesudo code of cumulative disassembly time function

for $s = 0$ to $n - 1$ **do**
  if $(D_{j,s} = D_{j,s+1} = 2)$ AND $(M_{j,s} = M_{j,s+1})$ **then**
    $T_s = T_{s-1}$
  else if $s \neq n - 1$ **then**
    $T_s = T_{s-1} + t_{j,s} + d_{j,s} + m_{j,s}$
  else
    $T_s = T_{s-1} + t_{j,s}$
  end if
end for
$T_{total} = T_s$

C. Fitness Value Calculation Examples

In this section, two examples of fitness evaluation is provided. The two disassembly operations are similar; by swapping 6^{th} and 7^{th} sequence (component J and D), one is able to obtain the other example. The second sample operation supplies an example of a sequence containing consecutive disassembly of components of same material and is demanded for recycling.

Operation 1: BCAIHGJDFE
(Shown in TABLE II)
*Initial time is set as 0 ($T_{-1} = 0$)
$F = 100 - 36 = 64$

Operation 2: BCAIHGDJFE
(Shown in TABLE III)
$F = 100 - 33 = 67$

D. Assumptions

- The disassembly direction of each components in an operation is independent of component in the previous or the next sequence.
- The disassembly of a component does not destroy the component in the previous or the next disassembly sequence.
- The disassembly operation must disassemble all 10 components from the EOL product.

III. Artificial Bee Colony Algorithm Design

Artificial bee colony algorithm is a swarm-based optimization algorithm proposed by Karaboga [2], which uses the nature of bees to optimize the disassembly operation. Bees in nature have a foraging area which extends for two miles [3]. In their search for food, food sources which contain more food supply attracts more bees than the food sources which contain small amounts of food supply. Bee has an interesting method of communicating with other bees to let the other bees know of which site has more food supply, and this is done by the waggle dance in the dance floor in the beehive. Not only does the bees communicate to tell the other bees of the direction and the distance of the food source, the waggle dance is also able to let the other bees know of the quality of the food source. The bee can communicate the quality of the duration of the dance. It is told that duration of the dance is related to the quality of the food source, bee dancing for a longer time for a better food source. Therefore, when the other bees are observing the dancing bees, there is a higher probability of sighting a bee which is dancing for a long time, attracting more bees

TABLE II: Fitness calculation of operation 1

s	j	$d_{j,s}$	Dir	Met	$D_{j,s}$	$M_{j,s}$	T_s
0	B	3	+x	N	1	S	$0+3+0+0=3$*
1	C	3	+x	N	1	S	$3+3+1+1=8$
2	A	2	+y	D	0	A	$8+2+1+1=12$
3	I	3	-z	N	1	P	$12+3+1+1=17$
4	H	1	-x	D	0	P	$17+1+1+0=19$
5	G	2	-y	D	2	S	$19+2+0+0=21$
6	J	2	-y	D	0	P	$21+2+1+0=24$
7	D	2	+z	D	2	S	$24+2+2+1=29$
8	F	4	-z	N	1	P	$29+4+0+0=33$
9	E	3	-z	N	1	P	$33+3=36$

TABLE III: Fitness calculation of operation 2

s	j	$d_{j,s}$	Dir	Met	$D_{j,s}$	$M_{j,s}$	T_s
0	B	3	+x	N	1	S	$0+3+0+0=3$
1	C	3	+x	N	1	S	$3+3+1+1=8$
2	A	2	+y	D	0	A	$8+2+1+1=12$
3	I	3	-z	N	1	P	$12+3+1+1=17$
4	H	1	-x	D	0	P	$17+1+1+0=19$
5	G	2	-y	D	2	S	$19=19$
6	D	2	+z	D	2	S	$19+2+1+0=22$
7	J	2	-y	D	0	P	$22+2+1+1=26$
8	F	4	-z	N	1	P	$26+4+0+0=30$
9	E	3	-z	N	1	P	$30+3=33$

Fig. 3: Detailed flowchart of Employed, Onlooker, and Scout Bee phase)

to a food source with high quality, which will eventually end up in more bees dancing to advertise the better food sites.

For the application of the ABC algorithm to this problem, we divide the process into two general parts, initialization and optimization, as shown in figure. 2. The pseudo code explaining the implementation of ABC for disassembly sequencing example is shown as Algorithm 2.

A. *Initialization*

In the initialization phase, we perform a random search for all the bees. One possible approach that could be made during initialization is having only feasible food sources, which are food sources that meet the precedence requirement of disassembly process, e.g. product B and C must be processed prior to any other products. However, it is possible for a operation that does not meet the precedence requirement to find the optimal operation in its neighbor. Therefore, completely random initial search is performed.

The initial food sources are evaluated for its fitness, and the few of the most optimal food sources, or best disassembly operation, are listed as possible food sources for employed bees and onlooker bees. Due to random selection, it is possible that no operation was found to be better than the others. In this case, the food sources selected for employed bees and onlooker bees to exploit was chosen at random. The first three lines of algorithm 2 represent initialization.

B. *Optimization*

Optimization is composed of 3 phases: employed bee phase, onlooker bee phase, and scout bee phase. Employed bees are the type of bees which are sent to occupy all the best food sources. Onlooker bees are the type of bees which observe the waggle dance of employed bees to decide the destination. Due to this reason, the onlooker bees visit food sources with a probability related to the best food sources. By this kind of behavior, it is possible that onlooker bees will not visit the most optimal site out of the best site list. The employed bees play a crucial role of making sure all the sites in the list is considered. Scout bees are the type of bees that are sent to random location to search for new food sources. These three bee phases are repeated until termination criterion, such as maximum number of iteration, has been met. The optimization process is shown as the main for loop in algorithm 2. More descriptive explanation about the bee phases is stated in the subsections below.

1) Employed Bee Phase: In the employed bee phase, employed bees are sent to the best food sources in the list, and the employed bees explore the neighboring sites. In our design, the neighboring sites are chosen by INSERT method or SWAP method. Each of the employed bees will have equal chance of choosing either SWAP or INSERT method. The INSERT method takes a certain component and insert the component in a random sequence location, where the SWAP method takes two different components and swap their sequences. Visual explanation for SWAP and INSERT method is shown in Figure. 4.

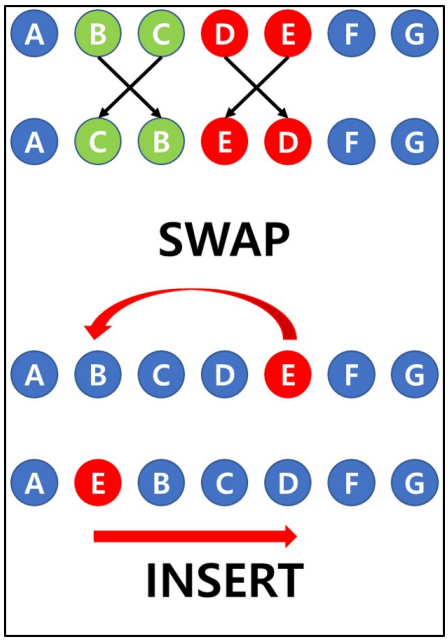

Fig. 4: SWAP and INSERT visual diagram

Similar to the initialization phase, one approach that can be made is that SWAP and INSERT methods to only SWAP or INSERT to a ”feasible” food source. However, this approach faces many problems. The problem occurs due to the precedence requirements. For example, operation JHIADEFGBC has many issues. Not only is component B and C listed last in the sequence, but also component E and F is listed before G. In the case where only one SWAP or INSERT method can be applied, it is unclear what should be chosen, as only one application of either method does not provide a feasible disassembly operation. Therefore, SWAP and INSERT method is performed ignoring the precedence requirements.

On the other hand, this can also cause different issues. A unfeasible disassembly operation should not have a high fitness value and should not be considered as one of the optimal disassembly operation. Therefore, the fitness function (4) is modified to:

Algorithm 2 Pseudo code of the Artificial Bee Colony Algorithm

```
Initialize random food sources
Evaluate fitness function of each food sources
Create list of best food sources (operations)
for i = 1 : max-iteration do
  for each best food source do
    Send employed bee to the source
    Find neighboring food source by using SWAP or INSERT (method selected with equal 0.5 probability)
    Evaluate fitness of neighboring food source
    Update best operations
  end for
  Calculate P_b = F_b/(sum_{i=1}^{b_e} F_i) (selection probability for each best food sources)
  for each onlooker bee do
    Select food source with r ~ U(0,1) within the cumulative P_b (selection probability for best food sources)
    Send on looker bee to the selected food source
    Find neighboring food source by using SWAP or INSERT (method selected with equal 0.5 probability)
    Evaluate fitness of neighboring food source
    Update best operations
  end for
  for each scout bee do
    Send scout bees to the random food source
    Evaluate fitness of neighboring food source
    Update best operations
  end for
end for
```

$$F_b = \begin{cases} C - T_{total} & \text{feasible operation} \\ 0 & \text{unfeasible operation} \end{cases} \quad (5)$$

These neighboring sites found by the employed bees are evaluated for fitness, and the list of best food source is updated. First for loop inside of the optimization phase in algorithm 2 represents the employed bee phase.

2) Onlooker Bee Phase: In the onlooker bee phase, a roulette wheel with the selections proportional to the fitness values is set up for the onlooker bees:

$$P_b = \frac{F_b}{\sum_{i=1}^{b_e} F_i} \quad (6)$$

This roulette wheel is used to select the destination of onlooker bees. The food sources of higher fitness value will therefore have higher probability to be visited by the onlooker bees. Then, the onlooker bees explore the neighboring sites chosen by SWAP or INSERT method, like the employed bees. The fitness values of these neighboring values are calculated, and best food source list is updated. Second for loop inside of the optimization phase in algorithm 2 represents the onlooker bee phase.

3) Scout Bee Phase: As for the scout bees, the best food source site list is not considered. Scout bees conduct a random search, being sent to random food sources similar to the initialization phase. Scout bees, although may seem useless, serves a crucial role in optimization. By performing a random search, all different types of disassembly operations are considered, reducing the possibility of the algorithm converging to a local maxima. The new sources found by the scout bees are evaluated for its fitness, and the list is updated if any better food source has been found. Last for loop inside of the optimization phase in algorithm 2 represents the scout bee phase.

IV. Results

The initial optimization trial parameters is shown in table IV.

TABLE IV: Algorithm parameters

Parameter	value
Number of employed bees	5
Number of onlooker bees	10
Number of scout bees	5
Maximum number of iteration	500

First, all the bees perform a random search. The initial operation list is given in table V, and list of operations after 500 iterations of ABC algorithm is given in table VI. The growth of fitness value over the number of iteration is given in figure 5.

TABLE V: List of initial operations

Number	Operation	Fitness
1	ICEFBHDGJA	0
2	HEFJCGABDI	0
3	DJGIECHABF	0
4	HFECAGIBDJ	0
5	DGEBCAFHIJ	0
6	EBIJCHADFG	0
7	GDIHCFAEJB	0
8	JIBCHDGAFE	0
9	ECDJFGIBHA	0
10	IFBEDCGAJH	0
11	JGAIECDBFH	0
12	AIDHJFBGCE	0
13	CJGBDFHAEI	0
14	BHGDECAFIJ	0
15	BHGCIFADEJ	0
16	BCHDGIAEFJ	65
17	FDEGJBHICA	0
18	IFBJGADHEC	0
19	IEAJGHCBFD	0
20	CIGFBDAHJE	0

TABLE VI: List of operations after ABC algorithm

Number	Operation	Fitness
1	BCAHDGJIEF	71
2	CBAHDGJIEF	71
3	CBAHDGJEFI	71
4	CBAHDGJEIF	71
5	CBAHDGJEFI	71

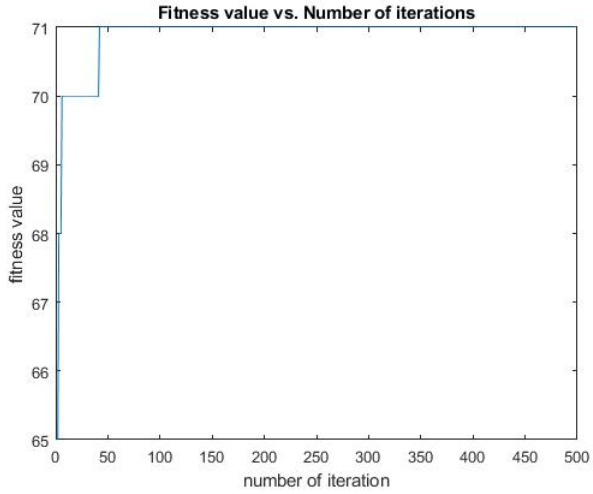

Fig. 5: History of fitness value during ABC optimization

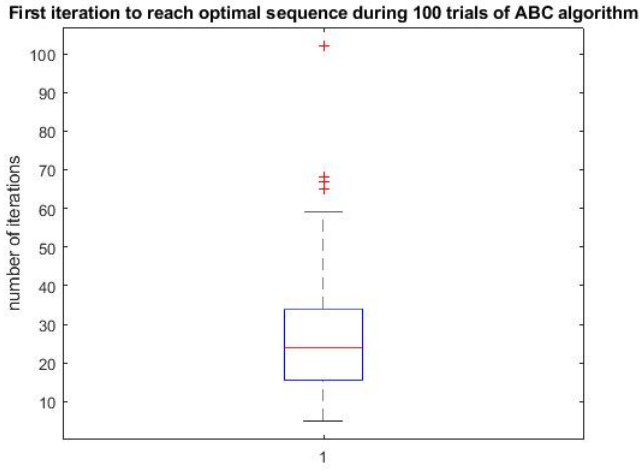

Fig. 6: Box plot of first iteration number for 100 trials of ABC algorithm optimization

As shown in figure 5, the algorithm converged to a fitness value of 71 around 50^{th} iteration. This is consistent with the results of Kongar, using both genetic algorithm and exhaustive search algorithm. To observe the performance of ABC algorithm, the optimization is tested 100 times. Boxplot that shows the result is shown in figure 6.

As it turns out, the first optimization trial was not within the 50^{th} percentile from the center. The average number of iteration it took to first reach the optimal sequencing was around 25, and the smallest possible number of iteration it took to reach optimal sequencing was 7.

Interestingly, as seen in table VI, there isn't a unique optimal operation. From table VI, it seems that interchanging $s = 0$ and $s = 1$ (component B and C), and interchanging $s = 7$, $s = 8$, and $s = 9$ (component E, F, and I) does not affect the fitness values. To confirm this, few additional "optimal" operations are given in table VII.

By observing table VIII and IX, the reason is quite clear. Component B and C have same direction of disassembly, method of assembly, type of demand, and type of material. Likewise, Component E, F, and I have same direction of disassembly, method of assembly, type of demand, and type of material. Therefore, switching the order of these sequences does not effect the fitness value.

One interesting study is the performance of method of finding the neighbor. Instead of using both SWAP and INSERT, instances of using only INSERT or SWAP is tested.

Table 7 shows a box plot of first iteration number to reach optimal value for 100 trials. As it can be seen, one instance of ABC algorithm which only uses INSERT method has converged to the optimal value close to 150 iterations.

Table 8 shows a box plot of first iteration number to reach optimal value for 100 trials. For SWAP method,

TABLE VII: Additional list of operations after ABC algorithm

Number	Operation	Fitness
1	CBAHDGJIEF	71
2	BCAHDGJFEI	71
3	BCAHDGJEFI	71
4	BCAHDGJFEI	71
5	CBAHDGJFEI	71
6	CBAHDGJFEI	71
7	BCAHDGJEFI	71
8	BCAHDGJEIF	71
9	BCAHDGJEFI	71
10	CBAHDGJFIE	71

TABLE VIII: Input data for component B and C

Variable	Value	
j	B	C
$d_{j,s}$	3	3
Direction	+x	+x
Method	N	N
Demand	1	1
Material	S	S

TABLE IX: Input data for component E, F, and I

Variable	Value		
j	E	F	I
$d_{j,s}$	3	4	3
Direction	-z	-z	-z
Method	N	N	N
Demand	1	1	1
Material	P	P	P

the data seem to be consistent throughout the trials unlike the INSERT and combination of the two methods. The comparison of the three different ABC algorithm methods are shown in figure 9.

As seen in figure 9, the box plot of SWAP method data seem to span much larger than INSERT and original method. However, it seem to show less outliers, suggesting that the method is more consistent. The INSERT method performs much better than SWAP method, showing smaller range of box plot. Nevertheless, it seems that combining the two method seem to show better performance than only using one method.

Another interesting study is finding the worst performing feasible operation. To do this, we change the fitness equation (5) to be equal to the cost of the operation:

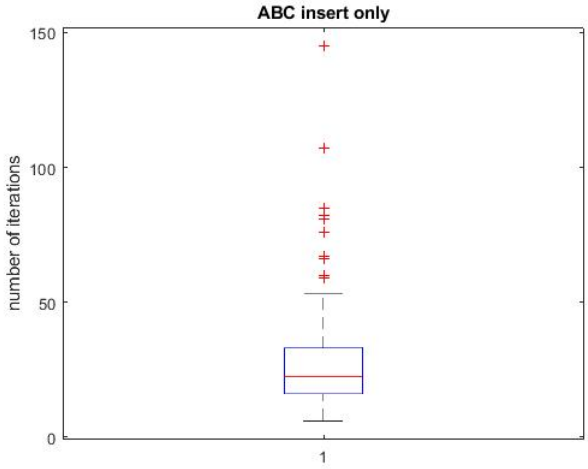

Fig. 7: Box plot of 100 trials with only INSERT method

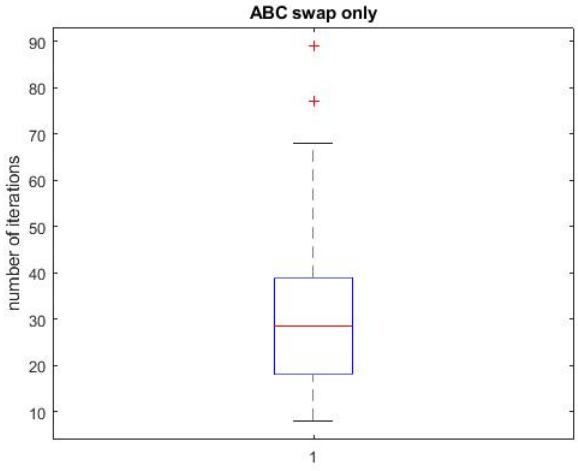

Fig. 8: 100 trials with only SWAP method

$$F_{bad} = \begin{cases} T_{total} & \text{feasible operation} \\ 0 & \text{unfeasible operation} \end{cases} \quad (7)$$

The result is given in table X

TABLE X: List of worst operations found using (7)

Number	Operation	Fitness
1	BCHIGEDFJA	44
2	CBHIGEDFJA	44
3	BCHIGEDFJA	44
4	CBHIGFDEAJ	44
5	BCHIAGEDFJ	44

Which means that the optimal solution's cost is equal to 29, and the worst solution's cost is 44. Therefore, we save

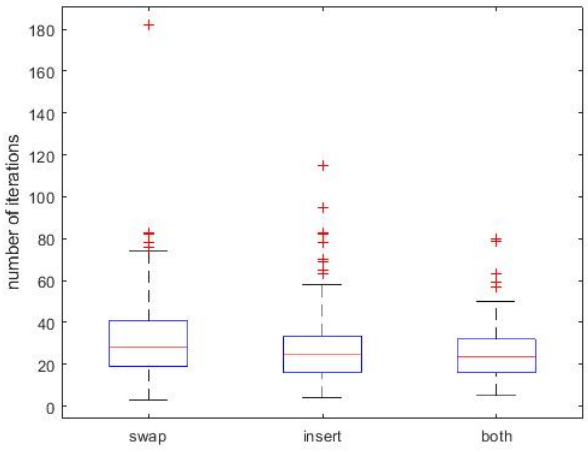

Fig. 9: Comparison of methods for 100 trials

cost of 15 by using optimization, which is a significant save in the cost.

V. Comparison to GA proposed by Kongar

For Kongar, next generation of chromosomes (new sets of operations in the next iteration) is determined by the use of precedence preservative crossover (PPX), and SWAP method which only swaps the component B and C. In [1], the paper claims convergence to the optimal solution within the 3rd generation (or iteration). This is significantly better performance than the ABC algorithm proposed on this paper. This paper has utilized total 20 bees, and have convergence at optimal solution at average number of iteration above 20.

However, Kongar makes very specific requirements for generating the new operations to be evaluated. These requirements are:

- Swap is only applied for component B and C.
- First half of the "sorted" population is selected and cloned.
- Based on fitness, two best-fit chromosomes in the first half of the "sorted" population are selected as parents to produce the next child. Based on probability of 0.6, parents are just selected as the child.

One crucial difference is that for Kongar's algorithm, the operations are "sorted," where as the ABC algorithm proposed in this paper does not attempt to sort the sequence to fit the precedence requirement. The sorting method is also undefined; for a sequence that would have an issue during generation of next operation (e.g. JHIADEFGBC), it is unclear how it was sorted.

Additionally, for mutation, only component B and C are swapped. It is difficult to say that switching only two components can be called "mutation;" regardless, it seems that this was a result of optimizing the mutation algorithm to give most favorable result. It should be noted that in result section of this paper, we have discovered that changing the sequence of B and C does not effect the fitness value of a feasible operation. Due to this, this type of mutation is useless in optimization algorithm.

To summarize, it is highly unfair to say GA proposed by Kongar is better. The act of sorting is not a method random search. It is possible by "sorting" the GA algorithm may have already arrived at the optimal solution. One conclusion that can be made is that the GA algorithm proposed by Kongar is "optimal" for solving this specific disassembly sequencing problem. However, this is due to prior knowledge and testing with the optimization methods.

VI. Conclusion

In this paper, the design of artificial bee colony algorithm for disassembly sequencing is presented. Without any prior knowledge of the disassembly sequencing (e.g. working with only sorted solution will result in faster convergence), the proposed ABC algorithm is able to successfully and consistently find the optimal solutions within approximately 100 iterations.

The use of optimization algorithm allows the user to find the optimal or near optimal solutions in relatively fast speed and low number of computation, reducing the time and work required for manual analysis.

References

[1] E. Kongar, S. M. Gupta, "Disassembly sequencing using genetic algorithm," The international Journal of Advanced Manufacturing Technology, vol. 20, Issue 5-6, pp 497-506, 2005

[2] D. Karaboga, "An idea based on honey bee swarm for numerical optimization," Technical report TR06, Computer Engineering Department, Engineering Faculty, Erciyes University, 2005

[3] J. E. Eckert, "The flight range of the honeybee," Journal of Agricultural Research, vol. 47, no. 8, pp. 257-285, 1933

Matlab Code

A. Main ABC algorithm

```
close all
clear all
clc
input_data

seq = ['A';'B';'C';'D';'E';'F';'G';'H';'I';'J'];

n_E = 5; % number of employed bees
n_O = 10; % number of onlooker bees
n_S = 5; % number of scout bees
n = n_E + n_O + n_S; % total number of bees

for pop = 1:n
    Food(:,pop) = seq(randperm(10));
end
```

```
max_search = 200;


hWaitbar = waitbar(0, 'Artificial Bee Colony Algorithm Optimization in process... ');

for trials = 1:max_search
    fitness = [];
    for u = 1:size(Food,2)
        fitness(u) = cost2(Food(:,u), INPUT);
    end

    if trials == 1
        [dvals, dindex] = sort(fitness, 'descend');
        large = dvals(1:n_E);
        Index = dindex(1:n_E);
        super = Food(:,Index);
        super_f = large;
    else
        fitness = [fitness super_f];
        Food = [Food super];
        [dvals, dindex] = sort(fitness, 'descend');
        large = dvals(1:n_E);
        Index = dindex(1:n_E);
        super = Food(:,Index);
        super_f = large;
    end
    fitness_Max_history(trials) = max(fitness);
    fitness_Mean_history(trials) = mean(fitness);
    if ~any(super_f)
        super_f = [1:n_E];
    end
    Roulette = cumsum(super_f/sum(super_f));
    new = [];
    if trials ~=max_search  % to find new food sources
        for j = 1:n_E   % employed bees
            if rand>0.5
                new = [new SWAP2(super(:,j))];
            else
                new = [new Insertion(super(:,j))];
            end
        end
        for k = 1:n_O   % onlooker bees
            if rand>0.5
                new = [new SWAP2(super(:,find(Roulette>rand,1)))];
            else
                new = [new Insertion(super(:,find(Roulette>rand,1)))];
            end
        end
        for L = 1:n_S
            new = [new seq(randperm(10))];
        end
        Food = new;
    end
    waitbar(trials/max_search, hWaitbar, 'Artificial Bee Colony Algorithm Optimization in process... ');
end
close(hWaitbar)
superior_Food = super(:,1);

cost2(superior_Food,INPUT)

figure;
plot(fitness_Max_history);
% hold on;
% plot(fitness_Mean_history);
% legend('Maximum','Mean');
xlabel('number of iteration');
ylabel('fitness value');
title('Fitness value vs. Number of iterations');

super'
```

B. Fitness calculation

```
function f = cost2(seq,INPUT)

placement_order = ['A';'B';'C';'D';'E';'F';'G';'H';'I';'J'];

T = 0;

A = ismember(seq,'A');
B = ismember(seq,'B');
C = ismember(seq,'C');
D = ismember(seq,'D');
E = ismember(seq,'E');
F = ismember(seq,'F');
G = ismember(seq,'G');
H = ismember(seq,'H');
I = ismember(seq,'I');
J = ismember(seq,'J');
```

```
First = B + C;
Last  = E + F;
Middle = D + G;
Check = H + D + G + E + F;

if find(H == 1) > min(find(Middle+Last
   == 1)) || find(G == 1) > min(find(
   Last == 1)) || ~all(find(First == 1)
   < 3)
    f = 0;
else
    for i = 1:length(seq)
        c_number = ismember(
            placement_order,seq(i));
        if i ~= length(seq)
        c_next = ismember(
            placement_order,seq(i+1));
            if INPUT(4,c_number) == 2 &&
                INPUT(4,c_next) == 2 &&
               INPUT(5,c_number) ==
               INPUT(5,c_next)
                T = T;
            else
                if INPUT(2,c_number) ==
                   INPUT(2,c_next)
                    penalty_d = 0;
                else
                    if mod(INPUT(2,
                       c_number),2) == 0
                        && INPUT(2,
                       c_next) ~= INPUT
                       (2,c_number)-1 ||
                        mod(INPUT(2,
                       c_number),2) == 1
                        && INPUT(2,
                       c_next) ~= INPUT
                       (2,c_number)+1
                        penalty_d = 1;
                    else
                        penalty_d = 2;
                    end
                end
                if INPUT(3,c_number) ==
                   INPUT(3,c_next)
                    penalty_m = 0;
                else
                    penalty_m = 1;
                end
                T = T + INPUT(1,c_number
                   ) + penalty_d +
                   penalty_m;
            end
        else
            T = T + INPUT(1,c_number);
        end

    end
    f = 100-T;
end
```

C. Insert

```
function array=Insertion(test1)

    n=numel(test1);

    I=randsample(n,2);

    j=I(1);
    k=I(2);
    test1 = test1';
    if j<k
        array=test1([1:j-1 j+1:k j k+1:
           end]);
    else
        array=test1([1:k-1 j k:j-1 j+1:
           end]);
    end

    array = array';
end
```

D. Swap

```
function new = SWAP2(Test)

swap_index = ceil(rand*10);
swap_target = swap_index;

while swap_target == swap_index
    swap_target = ceil(rand*10);
end

temp = Test(swap_target);
Test(swap_target) = Test(swap_index);
Test(swap_index) = temp;
new = Test;
```

E. Input data

```
INPUT = [
    2 3 3 2 3 4 2 1 3 2;
    3 1 1 5 6 6 4 2 6 4;
    1 0 0 1 0 0 1 1 0 1;
    0 1 1 2 1 1 2 0 1 0;
    1 2 2 2 3 3 2 3 3 3];

% contents are ordered from A to J
```

```
% first row = dt_j

% second row (Direction)
% +x = 1
% -x = 2
% +y = 3
% -y = 4
% +z = 5
% -z = 6

% Third row (Method)
% D = 1
% N = 0

% Fourth row (Demand)
% No-demand = 0
% Reuse     = 1
% Recycle   = 2

% Fifth row (Material)
% A = 1
% S = 2
% P = 3
```