Assembly Tool Selection and Sequence Optimisation

By
A.S.S. Lakshman [13MF3IM15] &
Waghe Shubham Yatindra [13MF3IM17]
Under the guidance of
Prof. Sankha Deb

Mechanical Engineering Department
Indian Institute of Technology, Kharagpur - 721302

Certificate

This is to certify that the project entitled “**Assembly Tool Selection and Sequence Optimisation**” being submitted by A.S.S.Lakshman [13MF3IM15] and Waghe Shubham Yatindra [13MF3IM17] is a bonafide work done by them in the department of Mechanical Engineering, Indian Institute of Technology, Kharagpur under my supervision and guidance, Prof. Sankha Deb for Bachelor's Project.

Submitted by:

A.S.S. Lakshman [13MF3IM15]

Waghe Shubham Yatindra [13MF3IM17]

Sign:

Prof. Sankha Deb
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur - 721302

Acknowledgement

With most of the projects, teamwork delivers the best results. This project is no exception and we wish to express my gratitude to the people below who participated in the successful completion of my work.

We take this opportunity to express my deepest gratitude to my project guide Prof. Sankha Deb, for giving us this opportunity to work under his esteemed guidance. We are greatly indebted to him for his invaluable advice and support.

We would also like to thank Research scholar, Atul Mishra, for his constant and instant help during the course of our project work.

A.S.S Lakshman [13MF3IM15]

Waghe Shubham Yatindra [13MF3IM17]

Place: Kharagpur

Contents:

Abstract:

In the present report, a knowledge based system has been used for selection of tools/ grippers for the parts of the assembly and determination and comparison of brute force approach and Genetic algorithm (a soft computing approach) for obtaining the optimal assembly sequence with least number of tools and assembly direction changes. Assembly process planning, tool/gripper selection is an important decision making task and becomes tedious and time consuming for an assembly with large number of components. Also, Assembly Sequence Planning (ASP) is regarded as a large-scale, highly constrained combinatorial optimization problem, and it is nearly impossible that all the assembly sequences are generated and evaluated to seek the optimal one, whether with manpower or through computer programs. This work solves the tools/gripper selection task using CLIPS (C Language Integrated Production System) and determination of optimal assembly sequence using brute force and Genetic algorithm and its comparison.

Introduction:

Assembly process is one of the most time-consuming and expensive manufacturing activities. Assembly sequence planning (ASP) is an NP-hard combinatorial optimization problem and the search space increases exponentially as the no of components increase. Assembly sequence planning is concerned with determination of sequence of assembly operations to assemble the components into the final product. Assembly plays a fundamental role in the manufacturing of most products. Parts that have been individually formed or machined to meet designed specifications are assembled into a configuration that achieves the functions of the final product or mechanism. The economic importance of assembly as a manufacturing process has led to extensive efforts to improve the efficiency and cost effectiveness of assembly operations. In recent years, the use of programmable and flexible automation has enabled the partial or complete automation of assembly of products in smaller volumes and with more rapid product changeover and model transition.

Numerous approaches have been proposed like mathematical algorithms, graph theoretic approaches, Artificial Intelligence (AI) techniques and various approaches like expert system, fuzzy logic, evolutionary soft computing based optimization techniques. Some of the mostly used soft computing algorithms are

Genetic algorithm, Particle Swarm Optimisation algorithm, Ant Colony Optimisation, hybrid cuckoo-search genetic algorithm.

In general, solving of ASP problem can be divided into two aspects, namely assembly modelling and problem solving algorithm. A broad variety of modelling theories had been introduced in previous researches. Since there are so many assembly factors to be considered, the most important step to create an appropriate model is to choose the right factors. An assembly product description consists of the geometric description of each of the individual parts, their geometric tolerances, and the configuration in which the parts fit together to form the final product. In practice, such a description is often incomplete or inexact and strongly relies on human experience and intuition for its full interpretation. These descriptions and relationships among components directly influence the feasible assembly sequences. Optimizations of assembly sequences are vital as it has important significance on productivity, product quality and manufacturing lead time.

Literature Review:

[13] In this paper, a knowledge based system has been developed for selection of assembly tools and grippers for performing the assembly, while a Genetic Algorithm (GA) based approach has been used to determine the feasible and optimal assembly sequences considering minimum number of tool changes and assembly direction changes.

Determination of Optimal Assembly plan (automation):

- Mathematical Algorithms
- Graph theory approaches
- Artificial Intelligence (AI) techniques various approaches like expert system, fuzzy logic
- Evolutionary soft computing based optimization techniques which include GA and swarm intelligence based approaches.

A knowledge-based system has been used to select the tools and grippers for performing the assembly.
A GA based optimisation approach has been used to determine the feasible and optimal assembly sequences.
It takes some input information and then automatically generates as output the feasible and optimal assembly sequences.

[12] This paper proposes Improved harmony search (IHS) algorithm, which has an outstanding global search ability to obtain the global optimum more efficiently. Modifying the traditional Harmonic Search for ASP problems which may inspire other studies on this problem or similar problems. The core of this improvements are as follows: first, a discretization encoding technique for ASP is proposed in this paper which has reduced the searching space effectively. Second, introducing a local search strategy to HIS which has balanced both of the global search and local search. The advantages of the proposed ASP algorithm over the competing algorithms like Genetic algorithm, Particle swarm optimization, memetic algorithm is also proved by couple of experiments on assemblies.

In this paper, the assembly directions are all along each axis. However, in practical assembly, some parts are not assembled along the axis direction, which should be studied in future work. Essentially, ASP is a multi-objective optimization problem (MOOP). For simplicity, this paper converts it to a single-objective optimal problem with linear weights. In the future, we will address it by using a multi-objective optimization algorithm.

[9] This paper presents a novel approach based on engineering assembly knowledge to the assembly sequence planning problem and provides an appropriate way to express both geometric information and non-geometric knowledge. In order to increase the sequence planning efficiency, the assembly connection graph is built according to the knowledge in engineering, design, and manufacturing fields. This paper though constructed the assembly connection graph to increase the efficiency of the assembly sequence planning it didn't applied to any real life assemblies to test the efficiency of the proposed method.

[3] This paper presents a knowledge-based approach to the assembly sequence planning problem. The CSBAT (connection-semantics-based assembly tree) hierarchy proposed in this paper provides an appropriate way to consider both geometric information and non-geometric knowledge. The approach proposed in this paper can generate assembly sequences for each CSBAT directly, without the problem of merging plans for different child CSBATs. The application shows that the knowledge-based approach can reduce the computational complexity drastically and obtain more feasible and practical plans. Proposed method gave a uniform representation of assemblies that containing all of the information required for assembly sequences planning.

Although KBASP generates a general-purpose geometric reasoning with the knowledge about how to build specific structure, there remains much to do. Future work should involve two main aspects:

1) More non-geometric information, such as the assembly intents, should be utilized in planning.
2) Find more robust assembly coefficients by applying the approach to more various assembly environments, and find better CSBATs by lifting or extending the selection criteria.

[1] This paper proposes an approach, which realises the benefits of this simultaneous consideration of product design phase and defining the sequence of assembly tasks. A constraint-based approach is used to confirm that the resulting assembly sequence is both feasible and practical and also gives guidance on the quality of the sequence.

Although this methodology has been shown to successfully overcome many of the issues with current sequence generation approaches, it might become more effective with the addition of further validation and evaluation criteria and the use of functional representations to enable methodology to be applied earlier in the design process.

[7] This paper attempts to optimise the Assembly Sequence Planning Problem (ASPP), a large scale, highly constrained combinatorial problem using Genetic algorithm. Paper deals with Modelling the assembly process, representation of assembly sequences as chromosomes, representation of assembly constraints as precedence relations and the algorithms developed to solve the ASPP.

[4] The paper addresses the assembly oriented design process, the knowledge based approach and framework for assembly oriented product design and the three stages of the assembly oriented design process.
It explains on how to approach a design problem and what factors are to be considered while assembly design is being made. Object-oriented knowledge representation of the problem.
Design task viewed as a functional decomposition process.
Knowledge-based approach and framework for intelligent assembly oriented design.

[5] Objective of this is to propose a knowledge based approach and develop an expert design system to support top-down design for assembled products.
Match-Select-Act cycle, Backtracking, heuristics graph search
Knowledge-based approach and framework for **assembly-oriented design**.

[6]In this paper, a method for disassembly sequence and path planning has been proposed.A disassembly task is said to be geometrically feasible if there is a collision-free path to bring the target subassembly or part out from the assembly. Set of hypothesis and lemmas.

The part's path planning algorithm for disassembly sequence planning.
Global interference check. The concept of DOFs is used to characterise the feasibility of the disassembly process.

[8]Given the product assembly model, assembly sequence planning (ASP) determines the sequences and paths of parts to obtain the assembly with minimum costs and shortest time using the new metaheuristic method called Ant Colony Optimization (ACO).

Three characteristics of the ACO algorithms, i.e., the positive feedback process, distributed computation, and the greedy constructive heuristic search, work together to find the optimal or near-optimal assembly sequences of mechanical products fast and efficiently.

[10] Assembly process constraints include topological relationships of the assembly, geometrical constraints and precedence relationships of parts or subassemblies, assembly direction changes, and issues regarding tools, accessibility, stability, and safety, which strictly limit the number of feasible assembly sequences.

BLS can be considered as an Iterated Local Search (ILS) algorithm that uses information of the search history for perturbing its solutions. The basic idea behind BLS is to use local search in each iteration to discover a local optimum and then employ adaptive diversification strategies to move from the current local optimum
to another in the search space. Thus, exploration of new search areas is achieved by continually and adaptively applying weak to strong perturbations, depending on the search state.

Problem Description:

The determinations of assembly plans have a strong influence on the cost of assembly processes. Given the product-assembly model, determining the tools to be used and assembly sequence planning (ASP) determines the sequences and paths of parts to obtain the assembly with minimum costs and shortest time. Also, study of variation of fitness value in genetic algorithm on varying different GA parameters.
Furthermore, exploring more Metaheuristic methods of finding the optimal sequence like Ant-colony and Particle Swarm Optimization (PSO) techniques.

Work Plan and Methodology:

We have taken an assembly of “Punching machine” for the use case.
It consists of **16 parts** with the following exploded view showing the direction of assembly of parts.

Scope:

This methodology can be used in industries for automating tool/gripper selection and optimal sequence generation for an assembly given the precedence constraints and assembly directions for the parts which eventually leads to cost and time efficiency of assembly process.

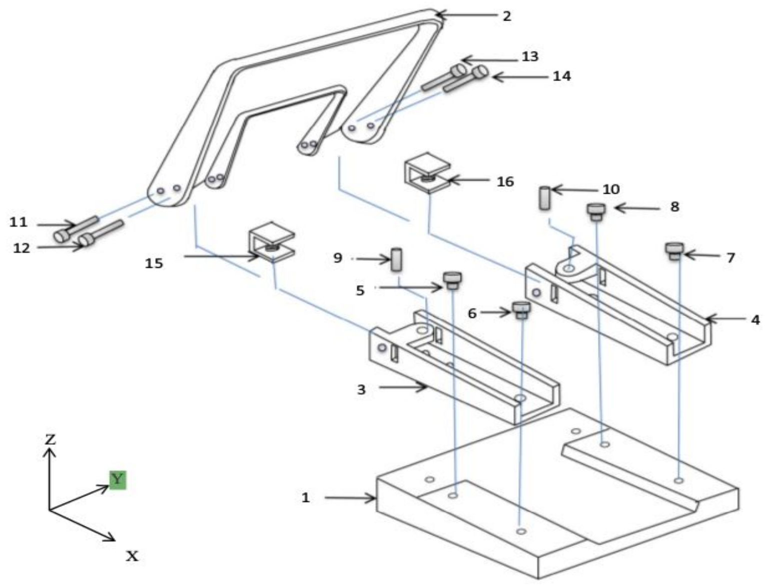


Figure 1: Punching Machine Assembly [14]

For the “Punching machine” assembly, it is required to find the tools for the task and the optimal assembly sequence i.e the sequence requiring the least number of tool gripper changes and orientation changes.

For finding the tools:

1. A database set has been taken which contains all the tool set and grippers.
2. Writing the rules to get the tools and grippers of the assembly components.

For finding the assembly sequence:

1. A brute force algorithm approach is tried to find all the optimal feasible sequences for the assembly.
2. In real world, since this problem of ASP is an NP hard highly complex and constrained combinatorial problem, it is not feasible to search the entire solution space for the optimal sequence.
3. This problem is thus also solved by implementation of a soft computing algorithm called Genetic Algorithm for finding the optimal sequence saving the time and resources.

Methodology:

CLIPS:

CLIPS is a public domain software tool for building expert systems. An expert system is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge, represented mainly as if-then rules rather than through conventional procedural code. Expert systems were among the first truly successful forms of artificial intelligence(AI) software. An expert system is divided into two subsystems: the inference engine and the knowledge base. The knowledge base represents facts and rules. The inference engine applies the rules to the known facts to deduce new facts.

There are few things which one has to understand before using the CLIPS software.

For example,

1) What is a Template
2) What is a Fact
3) What is a Rule

<u>DefTemplate:</u> deftemplate which stands for define template is analogous to a struct definition in C. That is, the deftemplate defines a group of related fields in a pattern similar to the way in which a C struct is a group of related data. A deftemplate is a list of named fields called slots. Deftemplate allows access by name rather than by specifying the order of fields. Deftemplate contributes to good style in expert systems programs and is a valuable tool of software engineering.

As an example, the deftemplate will have the following format:

```
(deftemplate <name>
(slot-1)
(slot-2)
.
.
.
(slot-N))
```

Every slot can have an extra argument like default, allowed only, type.

Fact: The basic concepts of an expert system which is anything just a piece of information. Facts are created by asserting them onto the fact database using the assert command. Here's an example, complete with the response from CLIPS.

```
CLIPS> (assert (colour green))
<Fact-0>
```

There are many others commands for facts like clear which clears all the facts from the memory, retract which will remove the fact which we want to by mentioning the indices of the facts in the command. (facts) command which will show up all the facts which are stored in the memory.

<u>Rule:</u> A rule is similar to an IF THEN statement in a procedural language like Java, C, or Ada. An IF THEN rule can be expressed in a mixture of natural language and computer language as follows:

```
IF certain conditions are true
THEN execute the following actions
```

The CLIPS format for the rule will be

```
(defrule duck
(animal-is duck)
=>
(assert (sound-is quack)))
```

CLIPS tries to match the pattern (if conditions) to facts which are already in memory or when inserted and run the program. If the patterns match, then the action part will take place otherwise no action will take place.

We are using an intelligent methodology for Tool selection/Gripper selection by using CLIPS expert system. As described before our assembly Punching machine consists of 16 components. At First,

1) We have to update our database of components and available Gripper and Tools into CLIPS memory.
2) Then we will be writing the rules for getting the tools/grippers for specific component.

A snippet of code for updating database is shown below.

```
(deffacts MAIN::components_present
(component (number 1)(component_type functional_part)(name base)(mass 10)(shape
cuboidal)(length 100.990)(breadth 56.007)(height 24.994)(diameter 0)(first_contact_with
13)(first_contact_along  Y+ Y- Z+ Z-)(first_disassembly_dir_along X- )))
(component (number 8)(component_type fastener)(name motor_screw_1)(type
screw)(specification M5)(head_type slotted)(head_size_1 0.6)(head_size_2 0.4))
```

Deffacts is a function command for asserting facts all at a time. The word follows that is the name of the function. In the same way, all components we want to update are added to deffacts function. Similarly, all the available grippers and Tools are also updated.

```
(deffacts MAIN::gripper_data
(gripper_available(number 1)(name concentric_3finger)(max_opening 15)(gripping_force
260)(shape cylindrical)(actuation_type mechanical)(fingers three))
(gripper_available(number 2)(name concentric_3finger)(max_opening 30)(gripping_force
650)(shape cylindrical)(actuation_type mechanical)(fingers three)))
```

```
(deffacts MAIN::tool_data
(tool_available (assembly_type manual)(number 1)(name slotted_screw_driver1)(type slotted_screw_driver)(size_1 0.6)(size_2 0.4))
(tool_available (assembly_type manual)(number 2)(name slotted_screw_driver1)(type slotted_screw_driver)(size_1 1.6)(size_2 0.4)))
```

Before updating the database, we have to give the facts the template or format they will be written. For that, DefTemplate command will be used. A snippet of code is shown below.

```
(deftemplate MAIN::component
        (slot number (type INTEGER)(default ?NONE))
        (slot component_type(type SYMBOL)(allowed-symbols functional_part fastener))
        (slot name (type SYMBOL))
        (slot mass (type NUMBER)(default 0))
        (slot shape(type SYMBOL)(allowed-symbols cuboidal cylindrical spherical disc_shaped flat others))
        (slot length(type NUMBER)(default 0)))
```

Similarly, we can write for gripper_available, Tools_available. Then we have written the rules for selecting the gripper/Tools for components of the assembly.

Rules for the tools/ gripper selection has been included in the Appendix [1]

After we get the tools from the CLIPS code, the next task is to determine the assembly sequence i.e the order in which the parts will be assembled.

Brute Force Algorithm:

Brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.
In the real world, assembly sequence planning is regarded as a large-scale, highly constrained combinatorial optimization problem, and it is nearly impossible that all the assembly sequences are generated and evaluated to seek the optimal one,

whether with manpower or through computer programs. For this reason, we use intelligent methodologies to solve this problem of assembly sequence planning.
Below is a Python Code implementation for a 12 parts Brute force for a motor drive assembly with the following precedence matrix and tools.

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0
4	1	1	1	0	1	0	1	1	0	0	1	0
5	1	0	0	0	0	0	0	0	0	0	0	0
6	1	1	1	1	1	0	1	1	1	0	1	1
7	1	0	1	0	0	0	0	0	0	0	0	0
8	1	0	0	0	1	0	0	0	0	0	0	0
9	1	1	1	1	1	0	1	1	0	0	1	0
10	1	1	1	1	1	1	1	1	1	0	1	1
11	1	0	0	0	0	0	0	0	0	0	0	0
12	1	1	1	1	1	0	1	1	1	0	1	0
Assembly directions	−z	−z	−z	−z	−z	+z	−z	+x	−z	−y	−z	−z

Table 1 : Precedence Matrix for Motor Drive Assembly [13]

Python code implementation of brute force algorithm for Motor driver assembly (12 parts) is included in the Appendix [2]
It prints all the possible feasible sequences with satisfy the precedence matrix.

There are many soft computing algorithms to solve these problems such as Ant Colony Optimisation, Genetic Algorithm, Improved Harmonic Search, Particle Swarm Optimisation etc. We have used Genetic Algorithm in our work.

Genetic Algorithm:

Genetic algorithm (GA) is a method for solving both constrained and unconstrained optimization problems based on a natural selection process that mimics biological evolution. The algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm randomly selects individuals from the current population and uses them as parents to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution.

The algorithm is started with a set of solutions (represented by **chromosomes**) called **population**. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness - the more suitable they are the more chances they have to reproduce. This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:
Selection rules select the individuals, called parents, that contribute to the population at the next generation.
Crossover rules combine two parents to form children for the next generation.
Mutation rules apply random changes to individual parents to form children.

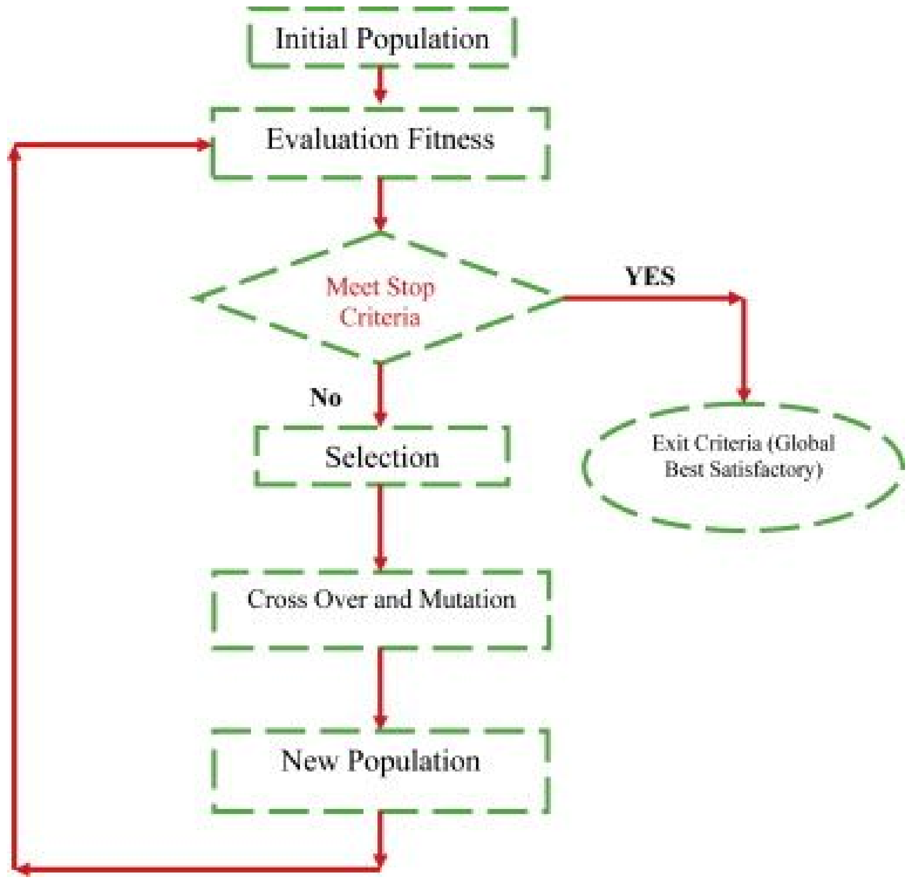


Figure 2: A flowchart for Genetic Algorithm implementation

Selection:
Selection is the stage of a genetic algorithm in which individual genomes are chosen from a population for later breeding (using the crossover operator).

In selection the offspring producing individuals are chosen. The first step is fitness assignment. Each individual in the selection pool receives a reproduction probability depending on the own objective value and the objective value of all other individuals in the selection pool. This fitness is used for the actual selection step afterwards.
There are many selections schemes such as:
Roulette-wheel selection (also called stochastic sampling with replacement) , local selection, truncation selection, tournament selection etc.
Tournament Selection is giving the best results for the use case.

Tournament selection is a method of selecting an individual from a population of individuals in a genetic algorithm. Tournament selection involves running several "tournaments" among a few individuals (or chromosomes) chosen at random from the population. The winner of each tournament (the one with the best fitness) is selected for crossover. Selection pressure is easily adjusted by changing the tournament size. If the tournament size is larger, weak individuals have a smaller chance to be selected.

Crossover:
Crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. Cross over is a process of taking more than one parent solutions and producing a child solution from them. There are methods for selection of the chromosomes. Those are also given below:
1. Single-point crossover
2. Two-point crossover
3. Uniform crossover
4. Arithmetic crossover etc.

For our implementation purpose, we have used **Single Point - 1D List Crossover CutCrossfill** for the sequence crossover. Crossover rate has been set to a default value of **0.9**

Mutation:
Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely

from the previous solution. Hence GA can come to a better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability. This probability should be set low. If it is set too high, the search will turn into a primitive random search.
In our implementation, we have used **Swap Mutation** with mutation probability set to a default value of **0.02**

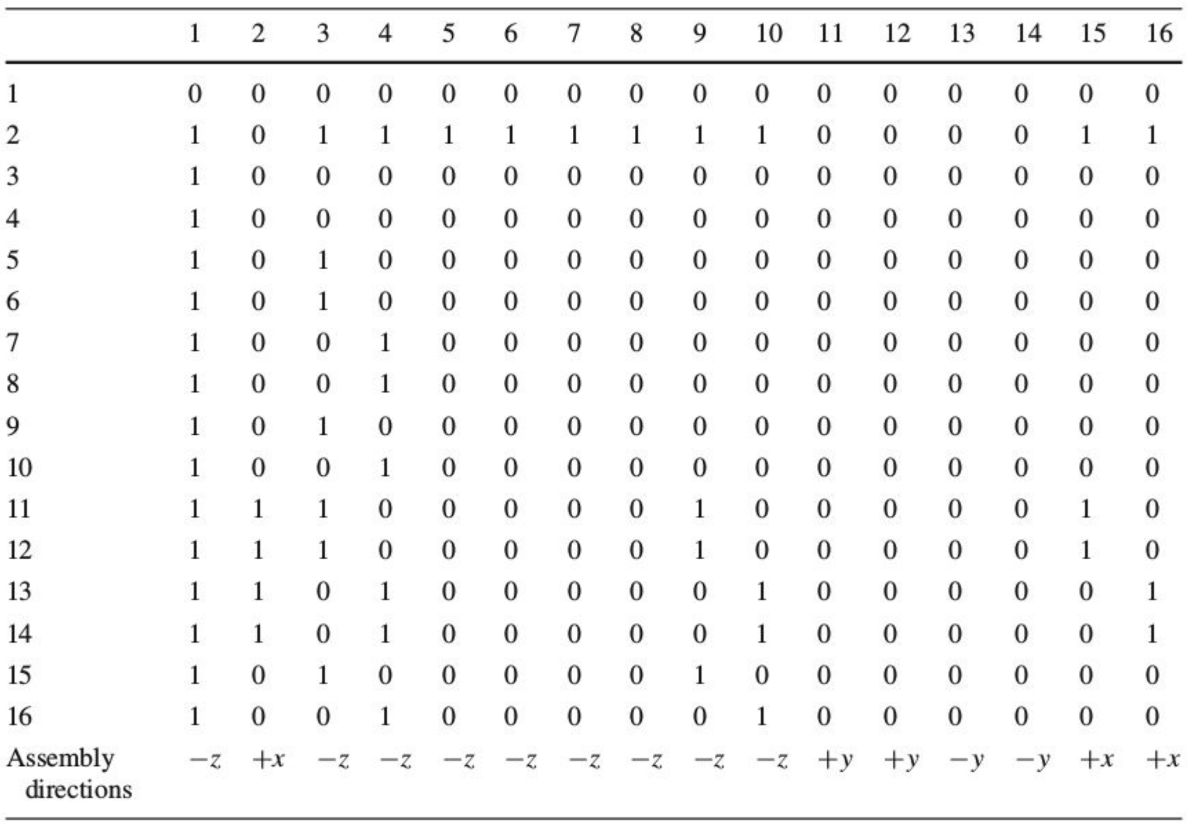
Fitness Function:
A **fitness function** is a particular type of objective **function** that is used to summarise, as a single figure of merit, how close a given design solution is to achieving the set aims. In our implementation the fitness function returns value proportional to the number of violations in precedence matrix if the sequence is infeasible and returns number of tool and orientation changes if the sequence is feasible. In both cases proper weightages have been assigned so as to give importance to feasibility and the ultimate aim to minimize the value of fitness function.

Termination Criteria:
The above steps for reproduction are repeated until the termination criteria is reached.
There can be many termination criterias for different types of problem. Some of these are –

1. Achievement of maximum no of generations as given in problem.
2. No improvement for some generations.
3. A solution is found that satisfies minimum criteria.

In our implementation, we have set the maximum number of generations to repeat the reproduction process for, i.e for **200 generations**.

Finally after the termination criteria is reached, we get the optimal sequence as the healthiest chromosome among the population.
The python implementation for **Genetic Algorithm** for “**Punching Machine**” assembly with **16 parts**, is included in the Appendix [3]



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	1	1	1	1	1	1	1	1	0	0	0	0	1	1
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
6	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
8	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
9	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
11	1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0
12	1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0
13	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	1
14	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	1
15	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
16	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
Assembly directions	$-z$	$+x$	$-z$	$-z$	$-z$	$-z$	$-z$	$-z$	$-z$	$-z$	$+y$	$+y$	$-y$	$-y$	$+x$	$+x$

Table 2: Precedence Matrix for Punching Machine [14]

Genetic Algorithm Parameter variation effect:

There are two basic parameters of GA - crossover probability and mutation probability.

Crossover probability says how often will be crossover performed. If there is no crossover, offspring is exact copy of parents. If there is a crossover, offspring is made from parts of parent's' chromosome. If crossover probability is 100%, then all offspring is made by crossover. If it is 0%, whole new generation is made from exact copies of chromosomes from old population (but this does not mean that the new generation is the same!). Crossover is made in hope that new chromosomes will have good parts of old chromosomes and maybe the new chromosomes will be better. However it is good to leave some part of population survive to next generation.

Mutation probability says how often will be parts of chromosome mutated. If there is no mutation, offspring is taken after crossover (or copy) without any change. If mutation is performed, part of chromosome is changed. If mutation probability is 100%, whole chromosome is changed, if it is 0%, nothing is changed. Mutation is made to prevent falling GA into local extreme, but it should not occur very often, because then GA will in fact change to random search.

There are also some other parameters of GA. One also important parameter is population size.

Population size says how many chromosomes are in population (in one generation). If there are too few chromosomes, GA have a few possibilities to perform crossover and only a small part of search space is explored. On the other hand, if there are too many chromosomes, GA slows down. Research shows that after some limit (which depends mainly on encoding and the problem) it is not useful to increase population size, because it does not make solving the problem faster.

Particle swarm optimization:

Particle swarm optimization (**PSO**) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality.It solves a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity. Each particle's movement is influenced by its local best known position, but is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions.

PSO is a metaheuristic as it makes few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics such as PSO do not guarantee an optimal solution is ever found. Also, PSO does not use the gradient of the problem being optimized, which means PSO does not require that the optimization problem be differentiable as is required by classic optimization methods such as gradient descent and quasi-newton methods.

Let S be the number of particles in the swarm, each having a position $\mathbf{x}_i \in \mathbb{R}^n$ in the Search space and a velocity $\mathbf{v}_i \in \mathbb{R}^n$. Let $\mathbf{p}_i$ be the best known position of particle i and let $\mathbf{g}$ be the best known position of the entire swarm. A basic PSO algorithm is then

```
for each particle $i$ = 1, ..., $S$ do
    Initialize the particle's position with a uniformly-distributed random vector: $\mathbf{x}_i \sim U(\mathbf{b}_{lo}, \mathbf{b}_{up})$
```

```
  Initialize the particle's best known position to its initial position: $\mathbf{p}_i \leftarrow \mathbf{x}_i$
  if $f(\mathbf{p}_i) < f(\mathbf{g})$ then
     update the swarm's best known position: $\mathbf{g} \leftarrow \mathbf{p}_i$
  Initialize the particle's velocity: $\mathbf{v}_i \sim U(-|\mathbf{b}_{up}-\mathbf{b}_{lo}|, |\mathbf{b}_{up}-\mathbf{b}_{lo}|)$
while a termination criterion is not met do:
  for each particle $i = 1, ..., S$ do
    for each dimension $d = 1, ..., n$ do
      Pick random numbers: $r_p, r_g \sim U(0,1)$
      Update the particle's velocity: $\mathbf{v}_{i,d} \leftarrow \omega\ \mathbf{v}_{i,d} + \phi_p r_p (\mathbf{p}_{i,d}-\mathbf{x}_{i,d}) + \phi_g r_g (\mathbf{g}_d-\mathbf{x}_{i,d})$
    Update the particle's position: $\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$
    if $f(\mathbf{x}_i) < f(\mathbf{p}_i)$ then
      Update the particle's best known position: $\mathbf{p}_i \leftarrow \mathbf{x}_i$
      if $f(\mathbf{p}_i) < f(\mathbf{g})$ then
        Update the swarm's best known position: $\mathbf{g} \leftarrow \mathbf{p}_i$
```

While applying **particle swarm optimization** to **Assembly Sequence Optimization** we are assuming particle position as assembly sequence. To make sure that the position of particle are of integer values we have taken the values of ω , ϕ_p , ϕ_g to be 1 , 1 , 1 in the updation of velocity and position of particles. The range of ω is $0 \le w \le 1.2$, ϕ_p is $0 \le \phi_p \le 2$ and ϕ_g is $0 \le \phi_g \le 2$. At the end we made sure that the updated positions of particles are having distinct, non-negative, non-repetitive values as it is not the case with sequences.

We are given the precedence matrix, orientation of particular component while assembly, and Tool/grippers for a particle component assembly. We are taking the fitness function as the sum of of orientation changes and Tool/Grippers changes for a particular sequence. The various parameters for **particle swarm optimization** are maximum Iterations, Population size and dimensions which is no of components of the assembly.

We are applying the Particle swarm Optimization on 12 components assembly Motor Driver. The precedence matrix, Orientation list, Tool/Gripper list are already mentioned before.

Ant colony optimization algorithms:

The **ant colony optimization** algorithm (**ACO**) is a probabilistic technique for solving computational problems which can be reduced to finding good paths through graphs.

The algorithm was aiming to search for an optimal path in a graph, based on the behavior of ants seeking a path between their colony and a source of food. In the natural world, ants of some species (initially) wander randomly, and upon finding food return to their colony while laying down pheromone trails. If other ants find such a path, they are likely not to keep travelling at random, but instead to follow the trail, returning and reinforcing it if they eventually find food.

Over time, however, the pheromone trail starts to evaporate, thus reducing its attractive strength. The more time it takes for an ant to travel down the path and back again, the more time the pheromones have to evaporate. A short path, by comparison, gets marched over more frequently, and thus the pheromone density becomes higher on shorter paths than longer ones. Pheromone evaporation also has the advantage of avoiding the convergence to a locally optimal solution. If there were no evaporation at all, the paths chosen by the first ants would tend to be excessively attractive to the following ones. In that case, the exploration of the solution space would be constrained. The influence of pheromone evaporation in real ant systems is unclear, but it is very important in artificial systems.

In this paper we have implemented the algorithm proposed by the **"J.F. Wang · J.H. Liu · Y.F. Zhong"** in their paper "**A novel ant colony algorithm for assembly sequence planning**". Ant colony Optimization is applied to Assembly Sequence Optimization based on assembly by disassembly philosophy.

We are given the disassembly matrix of the assembly. Given that an assembly comprises "n" components represented as (e1 , e2 , . . . , en). DM denotes the interference relationship of any component with all other components in the assembly during its extraction from the assembly along ±k-axis, where k ∈ (x, y, z). If the component e i of the assembly does not interfere with another component "ej" in the direction of the +k-axis, the component "ei" can be disassembled freely from the component "ej" in the direction of the +k-axis.

DM is described as an n × 3n matrix, where I ijk is equal to 1 if component “ei” interferes with the component “ej” during the move along direction +k-axis; otherwise, I ijk is equal to 0. As a convention, I iik is always equal to 0.

The initial DM of the assembly (4 components) we are dealing with and the assembly a

$$\begin{array}{c|ccc|ccc|ccc|ccc}
 & & 1 & & & 2 & & & 3 & & & 4 & \\
 & X & Y & Z & X & Y & Z & X & Y & Z & X & Y & Z \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}.$$

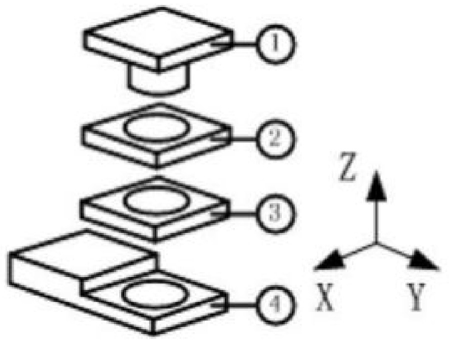


Fig. 1. An illustrative assembly

Figure 3: Initial Disassembly Matrix of the assembly with illustrative assembly diagram

The algorithm used for Ant Colony Optimization in the proposed paper based on assembly by disassembly philosophy is:

```
Begin
Step 1. Generate the initial feasible DOs and compute their quantity ma;
Step 2. Set cycle counter NC := 1;
Step 3. While NC < NCmax
DO {
    Step 4. Place ma ants on the initial feasible nodes of the DCG;
    Step 5. While each ant has not completed its tour
    DO {
        Step 6. Put current DO into sequence of the ant;
        Step 7. Generate candidate list of the ant by Eq. 2 and Eq. 3;
        Step 8. Calculate pk(i, j) of each candidate using Eq. 4;
        Step 9. Choose next DO j based on a random roulettewheel mode;
        Step 10. Move the ant to DO j;
        Step 11. Add the component number of DO j to the tabu list of the ant;
        Step 12. Locally update PM according to Eq. 6;
    }
    Step 13. Evaluate all solutions taking into account their reorientations;
    Step 14. Globally update PM using iteration-best solutions by Eq. 7;
    Step 15. Update the best sequence of each ant if its iteration sequence is the best one found so far;
    Step 16. Empty the sequence, candidate list, and tabu list of each ant;
Step 17. Set NC := NC +1;
}
Step 18. Output the reversed best sequence of each ant.
End
```

Pheromone Matrix:

In ACO algorithms, a pheromone matrix (PM) is used as a shared memory of all ants. It records the pheromone amount deposited on all edges to guide the search of ants. In this article, PM is expressed as a 6n × 6n matrix. An element of PM denoted as τ (i, j) indicates the visiting status from DO i to DO j. In the initial Pheromone matrix the element with name of column and row are same in n*n matrix is a 6*6 matrix with all zeroes. The element with different column and row names is a 6*6 matrix of all 1's.

Calculating Feasible Disassembly Operation:

From the DM, all the next feasible DO's after a DO of a component can be obtained . For example, the feasible DO's of component i along with ±k-axis are computed as follows:

$$\mathrm{DO}_{i,(+k)} = \bigcup_{j=1}^{n} I_{ijk} \ ,$$

$$\mathrm{DO}_{i,(-k)} = \bigcup_{j=1}^{n} I_{jik} \ ,$$

where ∪ is the Boolean operator OR. The component i can be disassembled in direction k if the first equation equal to 0 (or) in the (-k) direction if 2nd equation equal to 0. In similar way the Initial feasible DO's are (1, +z) and (4, −z).

The no of **ants** in the algorithms is decided by the no of initial DO's

Updating the Disassembly Matrix:

After the DO of a component the disassembly matrix is updated before calculating the feasible DO's in the next step. During updation all the elements in the rows and columns of the component which is disassembled just now are made to "0". Next again the feasible DO's are calculated based on this updated matrix.

Probability of DO’s:

After calculating all the feasible DO’s from the updated matrix we have to calculate the probability of each DO out of which we will choose the highest probability DO as the next DO. The formula for calculating the probability of a DO is

$$p_k^{(i,j)} = \begin{cases} \dfrac{\tau(i,j)[\eta(i,j)]^\beta}{\sum\limits_{u \in C_k(i)} \tau(i,u)[\eta(i,u)]^\beta}, & \text{if } j \in C_k(i) \\ 0, & \text{otherwise,} \end{cases}$$

where τ (i, j) is the quantity of pheromone corresponding to the positive feedback, η (i, j) is the problem-dependent heuristic information corresponding to the greedy search, C k (i) is the candidate list generated by DM after the component of DO i has been disassembled, and β is a parameter that determines the relative importance of pheromone versus heuristic information.

η (i, j) is equivalent to the distance between two cities of TSP. In this article, it concerns changes of the disassembly direction (that is, the assembly reorientations) from DO i to DO j.

$$\eta(i,j) = \begin{cases} \eta_1, & \text{if the direction does not change} \\ \eta_2, & \text{otherwise,} \end{cases} \tag{5}$$

where η_1 is a large positive constant, η_2 is a small one. In this article, $\eta_1 = 1$ and $\eta_2 = 0.2$.

Pheromone Matrix Updation:

While constructing the sequence, if an ant carries out the transition from DO i to DO j, the pheromone level of the corresponding edge in PM is modified using the local updating rule.

τ (i, j) ← (1 – ρ) τ (i, j) + ρ τ 0

where 0 < ρ < 1 is the rate of pheromone evaporation and τ 0 is the initial value of pheromone trail. The local updating of the pheromone shuffles the sequence construction so that the early DOs in one ant’s sequence may be explored later in other ants’ sequences. Every time, an edge used by an ant becomes slightly less desirable because it loses some of its pheromone, thus avoiding premature convergence. All ants would search in a narrow neighborhood of the best sequences found previously without local updating. Once all ants have built their sequences completely, the edges belonging to the iteration-best solutions with the least reorientations are reinforced with extra amounts of pheromone. At the same time, evaporation of pheromone on all edges is also performed. The global updating rule is

$$\tau(i, j) \leftarrow (1-\gamma)\tau(i, j) + \sum_{k=1}^{m} \Delta\tau_k(i, j) ,$$

where 0 < γ < 1 is the pheromone decay parameter and m is the number of ants that find the iteration-best sequences. The quantity of pheromone deposited by each ant is calculated as

$$\Delta\tau_k(i, j) = \begin{cases} \dfrac{Q}{R+1} , & \text{if } (i, j) \in \text{ sequence of ant } k \\ 0 , & \text{otherwise,} \end{cases}$$

Results and Discussion:

CLIPS Results: After running the CLIPS rules we selected the following Grippers/Tools for the 16 components of Punching Machine given in [Table 3]

Part No.	Part Name	Gripper/Tool
1	Base	2-finger adaptive gripper
2	Top Cover	3-finger adaptive gripper 1
3	U-Shaped Bracket a	2-finger parallel gripper
4	U-Shaped Bracket b	2-finger parallel gripper
5	Rivet a	Rivet gun1
6	Rivet b	Rivet gun1
7	Rivet c	Rivet gun1
8	Rivet d	Rivet gun1
9	Cutter a	Rivet gun2
10	Cutter b	Rivet gun2
11	Rivet e	Rivet gun2
12	Rivet d	Rivet gun2
13	Rivet f	Rivet gun2

14	Rivet g	Rivet gun2
15	Clip a	3-finger adaptive gripper 2
16	Clip b	3-finger adaptive gripper 2

Table 3: Results of Grippers/ Tools used in Punching machine assembly from CLIPS

Assembly Sequence Planning Results:

Brute Force Algorithm:

For “Motor drive assembly” with 12 parts (N=12) -
Time required: **27.68 minutes**
Sequences:
[1, 2, 3, 5, 7, 8, 11, 4, 9, 12, 6, 10]
[1, 2, 3, 5, 11, 8, 7, 4, 9, 12, 6, 10]
.
... total **48 sequences**, with
Orientation changes: **4**
Tool gripper changes: **8**

Genetic Algorithm:

Swap Mutation rate: **0.02**
Crossover rate: **0.9**

For “Motor drive assembly” with 12 parts (N=12) -
Number of Generations: **150**
Time required: **0.738 seconds**
Fitness value: **0.015234**
Sequence: **[1, 5, 3, 2, 11, 7, 8, 4, 9, 12, 6, 10]**
Orientation changes: **4**
Tool gripper changes: **8**

For “Punching Machine” with 16 parts (N=16) -
Number of Generations: **200**
Time required: **1.314 seconds**
Fitness value: **0.030644**
Sequence: **[1, 3, 4, 10, 9, 6, 7, 8, 5, 16, 15, 2, 13, 14, 12, 11]**
Orientation changes: **3**
Tool gripper changes: **6**

Fitness value variation with parameters:

For “**Motor Drive**” assembly with 12 parts - (lower fitness value is better)

a) **CrossOver:** Crossover rate, Fitness value: [**0.9**, 0.01068376681614] (figure 4)

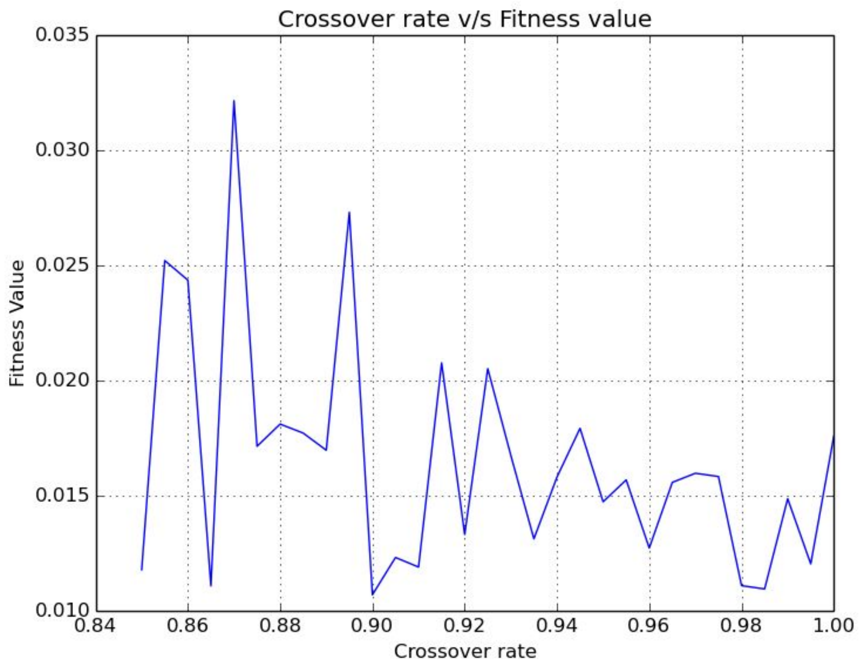


Figure 4: Variation of fitness value with crossover rate (12 parts)

b) **Mutation:** Mutation rate, Fitness value: [**0.013**, 0.008113684210526] (figure 5)

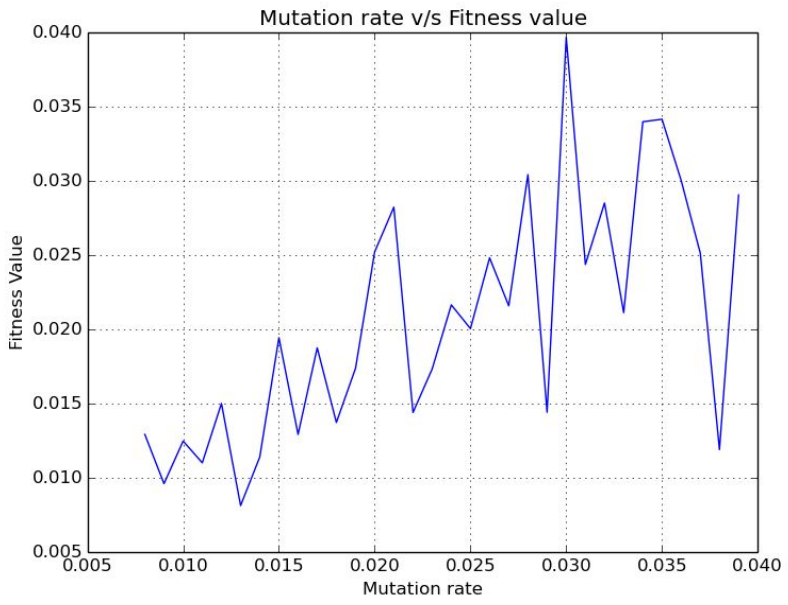


Figure 5: Variation of fitness value with mutation rate (12 parts)

c) **Generations:** No. of generations, Fitness value: [**410**, 0.00845] (figure 6)

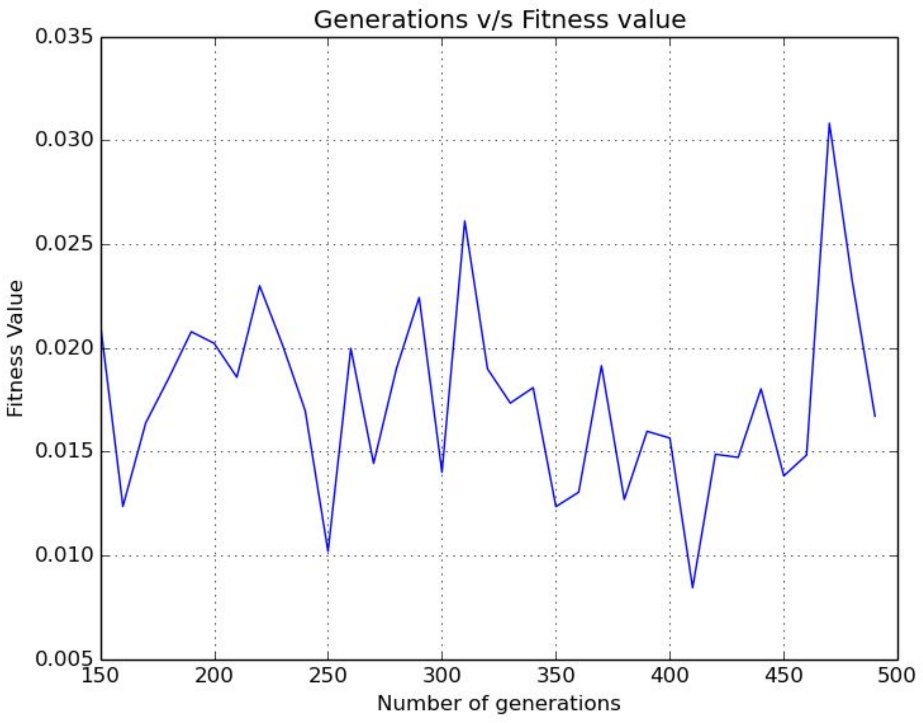


Figure 6: Variation of fitness value with number of generations(12 parts)

d) **Population Size:** Population Size, Fitness value: [**80**, 0.01108] (figure 7)

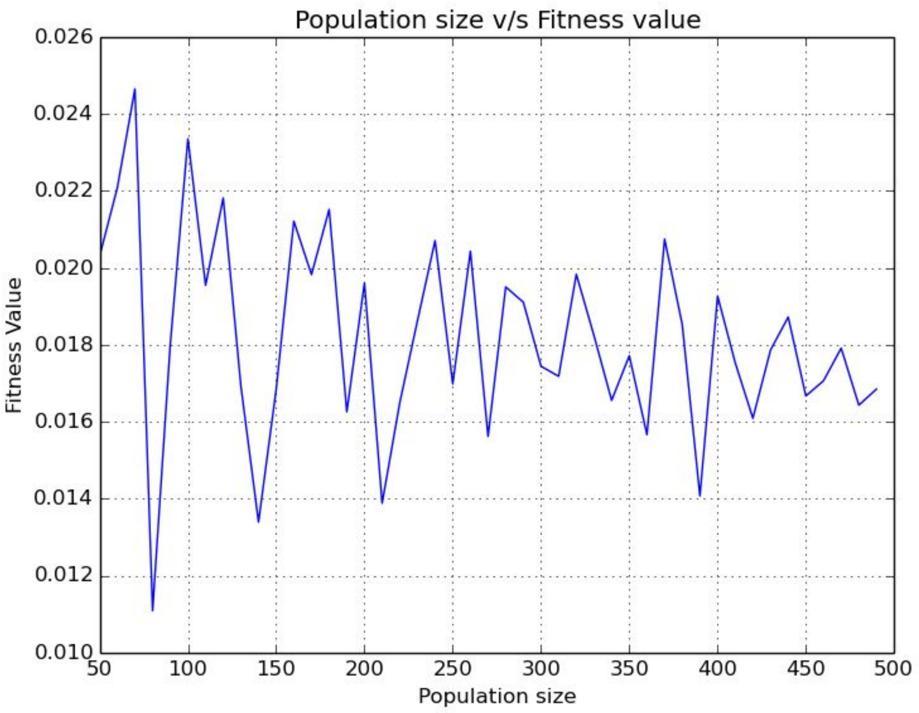


Figure 7: Variation of fitness value with population size(12 parts)

For “**Punching Machine**” assembly with 16 parts -

a) **CrossOver:** Crossover rate, Fitness value: [**0.945**, 0.009658] (figure 8)

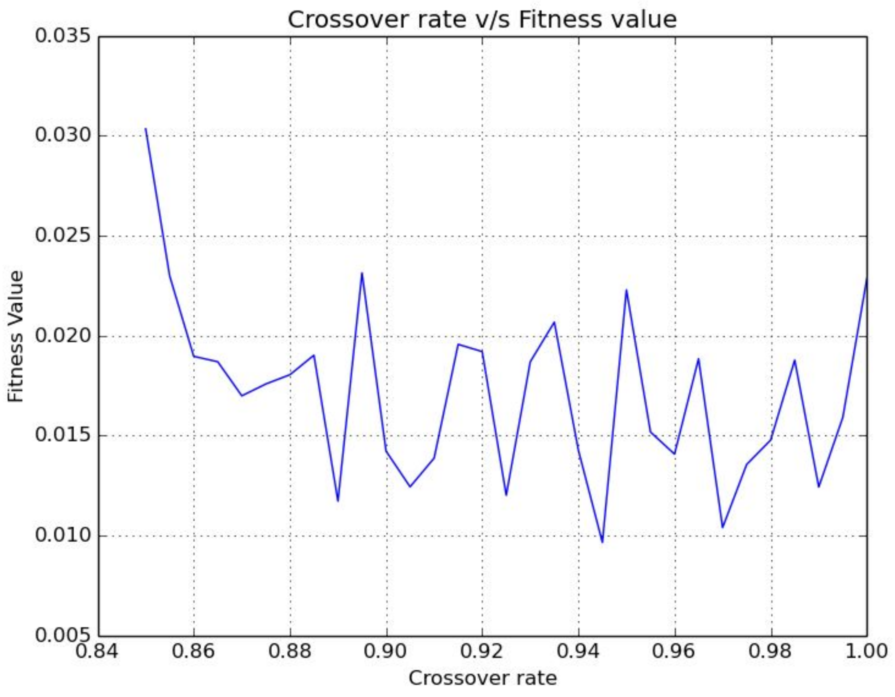


Figure 8: Variation of fitness value with crossover rate (16 parts)

b) **Mutation:** Mutation rate, Fitness value: [**0.012**, 0.007328] (figure 9)

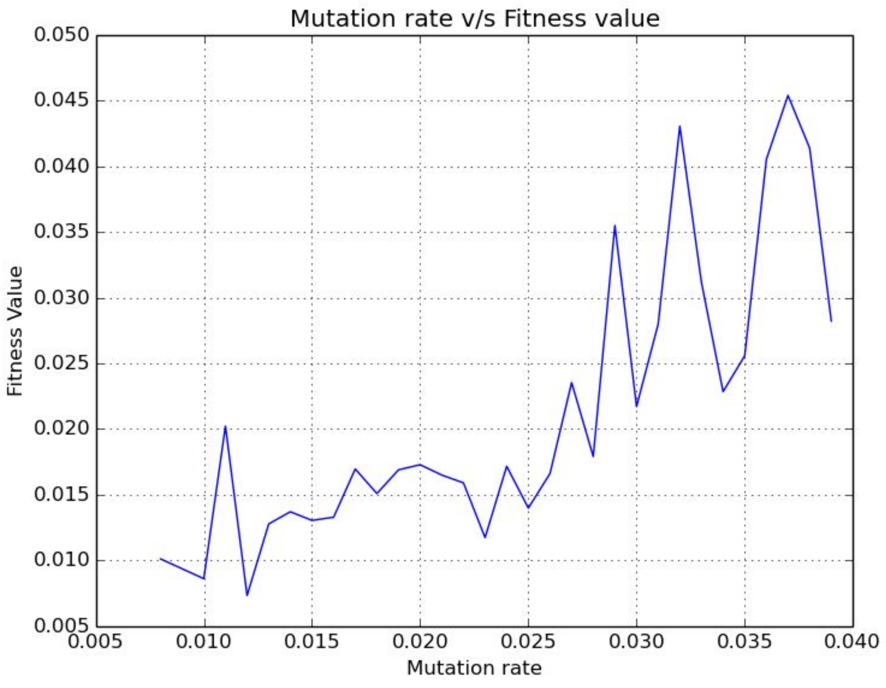


Figure 9: Variation of fitness value with mutation rate (16 parts)

c) **Generations:** No. of generations, Fitness value: [**260**, 0.00832174] (figure 10)

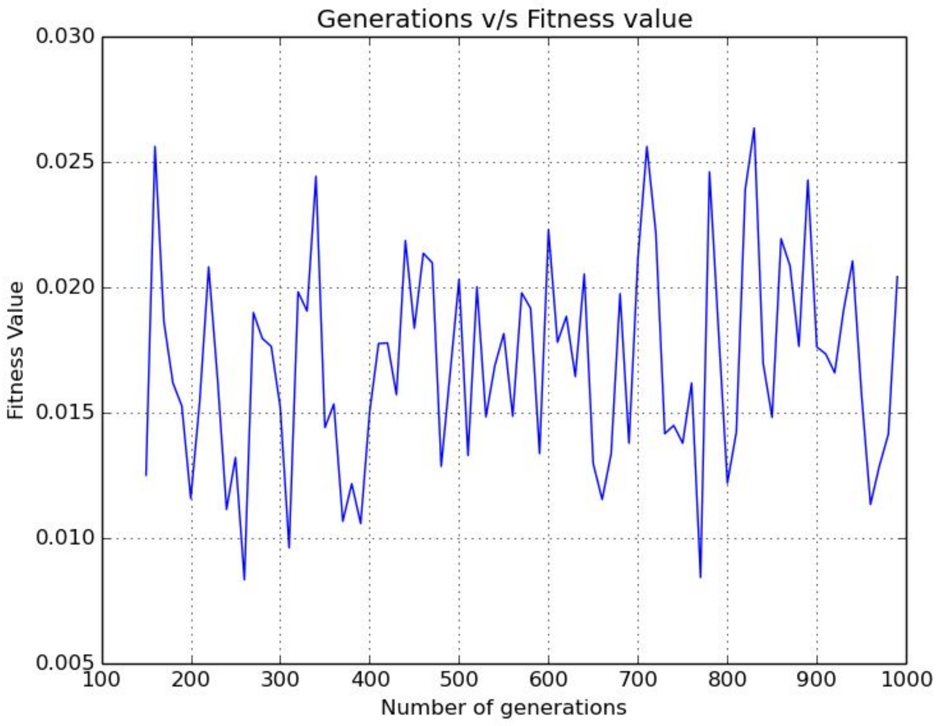


Figure 10: Variation of fitness value with number of generations (16 parts)

d) **Population Size:** Population Size, Fitness value: [**180**, 0.0115376] (figure 11)

Figure 11: Variation of fitness value with population size (16 parts)

Ant Colony Optimization:

The best sequence for the 4 component assembly is [[0, 5], [1, 5], [2, 5], [3, 5]] where the number of orientation changes is “0”. [0,5] in the solution means component 1 in the direction of ‘-Z’. Six directions[+X,+Y,+Z,-X,-Y,-Z] are represented as [0,1,2,3,4,5]

Particle Swarm Optimization:

The attached code for PSO is not converging for our use case of Motor driver assembly. Further work on the algorithm will give feasible sequences as we have got by using Genetic Algorithm.

Conclusions:

Finally, we obtain the tools required and the optimal sequence for the given assembly by using CLIPS and Genetic Algorithm. The methodology can be used so as to automate the task and reduce manual work.
The variation of parameters of Genetic Algorithm gives an insight of the crucial parameter values that needs to be used for successful, faster and efficient search of optimal sequence. We observe that for the parameters that we have taken - Crossover rate, Mutation rate, Population size and Number of Generations the graph is neither increasing nor a decreasing function. This implies a smaller value or a very large value may not be beneficial for optimal running of the algorithm. Crossover operator and Mutation Operator used was same during all the variations.

It was found that increasing the mutation rate resulted in increase of the fitness value function (lower is better) which was not desired. Furthermore, other parameters had an unusual trend in the variation of fitness value.

Choosing the right parameters for the algorithm thus plays an important role in developing an optimal sequence of the assembly. Future scope includes -
The parameters of GA can be tuned simultaneously using other meta heuristic approaches. This parameter tuning can help in determining all parameters of GA simultaneously.
There are various parameters to tune in Ant Colony Optimization which can be studied in future.
Reasons for PSO not converging might be that
1) We are considering very few sequences (maximum_iterations*initial_population) out of all possible sequences.
2) Even if we want to increase both maximum_iterations and initial_population then the algorithm is taking so much time.

References:

1) Barnes, C. J., Jared, G. E. M., & Swift, K. G. (2004). Decision support for sequence generation in an assembly oriented design environment. Robotics and Computer-Integrated Manufacturing, 20, 289–300.

2) Hsu, Y. Y., Tai, P.-H., Wang, M. W., & Chen, W. C. (2011). A knowledge-based engineering system for assembly sequence planning. International Journal of Advanced Manufacturing Technology, 55, 763–782.

3) Dong, T., Tong, R., Zhang, L., & Dong, J. (2007). A knowledge-based approach to assembly sequence planning. International Journal of Advanced Manufacturing Technology, 32, 1232– 1244.

4) X.F. Zhaa,*, H.J. Dua, J.H. Qiub. (2001) Knowledge-based approach and system for assembly oriented design, Part I: the approach, (vol. 14(1), pp. 61-75).

5) X.F. Zhaa,*, H.J. Dua, J.H. Qiub. (2001)Knowledge-based approach and system for assembly-oriented design, Part II: the system implementation. (vol. 14(2), pp. 239–254).

6) D. Hu, Y. Hu, C. Li. (2002). Mechanical Product Disassembly Sequence and Path Planning Based on Knowledge and Geometric Reasoning. (vol. 19(9), pp. 688–696).

7) Romeo M. Marian a,b,*, Lee H.S. Luonga, Kazem Abhary a. (2003)Assembly sequence planning and optimisation using genetic algorithms Part I. Automatic generation of feasible assembly sequences. (vol. 2(3), pp. 223–253).

8) J.F. Wang · J.H. Liu · Y.F. Zhong.(2005) A novel ant colony algorithm for assembly sequence planning. (vol. 25(11), pp. 1137–1143).

9) Meiping Wu, Yi Zhao, and Chenxin Wang, (2013) "Knowledge-Based Approach to Assembly Sequence Planning for Wind-Driven Generator," Mathematical Problems in Engineering, vol. 2013, Article ID 908316, 7 pages, 2013. doi:10.1155/2013/908316

10) Somayé Ghandi, Ellips Masehian. (2015). A breakout local search (BLS) method for solving the assembly sequence planning problem. (vol. 39, pp. 245–266).
11) M V A Raju Bahubalendruni Deepak B B V L Bibhuti Bhusan Biswal. (2016). An Advanced Immune Based Strategy to Obtain an Optimal Feasible Assembly Sequence. (vol. 36(2), pp. 127 - 137).

12) Xinyu Li1 & Kai Qin1 & Bing Zeng1 & Liang Gao1 & Jiezhi Su2. (2015). Assembly sequence planning based on an improved harmony search algorithm. (vol. 84(9), pp. 2367–2380).

13) Atul Mishra and Sankha Deb. An Intelligent Methodology for Assembly Tools Selection and Assembly Sequence Optimisation. (pp. 323-333).

14) Atul Mishra, Sankha Deb.(2016). Assembly sequence optimization using a flower pollination algorithm-based approach. (pp. 1-22).

15) Kalyanmoy Deb and Samir Agrawal. Understanding Interactions among Genetic Algorithm Parameters

16) Mahsa Sadat Emami Taba. Solving Traveling Salesman Problem With a Non-complete Graph

Appendix:

[1] CLIPS code for Rules for selection of tools/grippers for the assembly parts -

```
(defrule MAIN::gripper_selection_1
(component (number ?A)(component_type functional_part)(name ?B)(mass ?M)(shape
cuboidal)(breadth ?C))
(gripper_available(number ?a)(name ?n)(max_opening ?b)(gripping_force ?f)(actuation_type
mechanical)(fingers two)(finger_tiptype flat))
(test (< ?C ?b))
(test (>= ?f (/ (* ?M 9.81) (* 2 0.3))))
=>
(assert(gripper_selected
(number ?a)(name ?n)(for_comp ?B)(shape cuboidal)(gripping_force ?f)(actuation_type
mechanical)(fingers two)(finger_tiptype flat)(max_opening ?b))))

(defrule MAIN::gripper_selection_1
(component (number ?A)(component_type functional_part)(name ?B)(mass ?M)(shape
cuboidal)(breadth ?C))
(gripper_available(number ?a)(name ?n)(max_opening ?b)(gripping_force ?f)(actuation_type
mechanical)(fingers three)(finger_tiptype flat))
(test (< ?C ?b))
(test (>= ?f (/ (* ?M 9.81) (* 2 0.3))))
=>
(assert(gripper_selected
(number ?a)(name ?n)(for_comp ?B)(shape cuboidal)(gripping_force ?f)(actuation_type
mechanical)(fingers three)(finger_tiptype flat)(max_opening ?b))))

(defrule MAIN::tool_selection_7
(component(number ?a)(component_type fastener)(name ?c)(type bolt)(specification
?s)(head_type hexagonal)(head_size_1 ?f))

(tool_available (assembly_type ?z)(number ?m)(name ?d)(type open_end_wrench)(size_1 ?e))
(test (= ?f ?e))
=>
(assert(tool_selected (assembly_type ?z)(number ?m)(name ?d)(type
open_end_wrench)(for_fastener ?a)(size_1 ?e))
))

(defrule MAIN::tool_selection_8
(component(number ?a)(component_type fastener)(name ?c)(type bolt)(specification
?s)(head_type hexagonal)(head_size_1 ?f))
(tool_available (assembly_type ?z)(number ?m)(name ?d)(type socket_wrench)(size_1 ?e))
(test (= ?f ?e))
=>
(assert(tool_selected (assembly_type ?z)(number ?m)(name ?d)(type
socket_wrench)(for_fastener ?a)(size_1 ?e))
))

(defrule MAIN::tool_selection_9
(component(number ?a)(component_type fastener)(name ?c)(type bolt)(specification
?s)(head_type hexagonal)(head_size_1 ?f))
(tool_available (assembly_type ?z)(number ?m)(name ?d)(type adjustable_wrench)(size_1 ?e))
(test (= ?f ?e))
=>
(assert(tool_selected (assembly_type ?z)(number ?m)(name ?d)(type
adjustable_wrench)(for_fastener ?a)(size_1 ?e))
))
```

```
(defrule MAIN::tool_selection_10
(component(number ?a)(component_type fastener)(name ?c)(type nut)(specification
?s)(head_type hexagonal)(head_size_1 ?f))
(tool_available (assembly_type ?z)(number ?m)(name ?d)(type open_end_wrench)(size_1 ?e))
(test (= ?f ?e))
=>
(assert(tool_selected (assembly_type ?z)(number ?m)(name ?d)(type
open_end_wrench)(for_fastener ?a)(size_1 ?e))
))
```

[2] Python - Brute force algorithm for Motor Driver Assembly (12 parts) -

```
#Assembly sequence planning - Brute Force
import csv, itertools, time
NUM_PARTS = 12
#Precedence Matrix
with open('precedence_matrix_12.csv', 'rb') as csvfile:
   precedence_list = []
   for line in csvfile.readlines():
      array = line.strip().encode('utf-8').split(',')
      precedence_list.append(array)
#Directions / Orientations
orientations = ['-z','-z','-z','-z','-z','+z','-z','+x','-z','-y','-z','-z']
#Tool Grippers
tools_grippers = ['A','A','A','A','A','A','B','C','D','E','F','G']
#Parts Naming
parts = [ i for i in range(0,NUM_PARTS) ]

#Probable Sequences :: Sequences satisfying precedence matrix
prob_sequences = []

def check_precedence_criteria(sequence):
   for i,part in enumerate(sequence):
      to_search = precedence_list[part]
      done_sequence = sequence[0:i]
      to_search_final = [x for j, x in enumerate(to_search) if j not in done_sequence]
      for a in to_search_final:
         if a == '1':
            return False
   return True
```

```
def calc_orientation_changes(sequence):
    orien_changes = 0
    current_orientation = orientations[sequence[0]]
    for part in sequence[1:]:
        if orientations[part] != current_orientation:
            orien_changes += 1
            current_orientation = orientations[part]
    return orien_changes

def calc_tool_changes(sequence):
    tool_changes = 0
    current_tool = tools_grippers[sequence[0]]
    for part in sequence[1:]:
        if tools_grippers[part] != current_tool:
            tool_changes += 1
            current_tool = tools_grippers[part]
    return tool_changes

for i in itertools.permutations(parts):
    sequence = list(i)
    if check_precedence_criteria(sequence) == True:
        prob_sequences.append(sequence)
        print "Sequence: " + str(sequence)
        print "Orientation changes: " + str(calc_orientation_changes(sequence))
        print "Tool gripper changes: " + str(calc_tool_changes(sequence)) + "\n"

print "Total sequences satisfying precedence matrix: " + str(len(prob_sequences))


print "Total Time Taken", str(round(time.time() - start_time, 1))

with open("opt.txt", "a") as text_file:
    for seq in prob_sequences:
        text_file.write("Sequence:  %s\n" % str(seq))
        text_file.write("Orientation changes:   %s\n" % str(calc_orientation_changes(seq)))
        text_file.write("Tool gripper changes:  %s\n\n" % str(calc_tool_changes(seq)))
    text_file.write("Total sequences:   %s\n" % str(len(prob_sequences)))
    text_file.write("Total Time Taken:  %s" % str(round(time.time() - start_time, 1)))
```

[3] Python - Genetic algorithm for Punching machine (16 parts) -

```
import csv
from random import shuffle
from pyevolve import G1DList, GSimpleGA, Consts, Initializators, Selectors, Mutators, Crossovers
#Number of Parts
NUM_PARTS = 16
#Precedence Matrix
with open('precedence_matrix_16.csv', 'rb') as csvfile:
   precedence_list = []
   for line in csvfile.readlines():
      array = line.strip().encode('utf-8').split(',')
      precedence_list.append(array)

#Directions / Orientations
orientations = ['-z','+x','-z','-z','-z','-z','-z','-z','-z','-z','+y','+y','-y','-y','+x','+x']
#Tool Grippers
tools_grippers = ['A','B','C','C','D','D','D','D','E','E','E','E','E','E','F','F']

def check_precedence_criteria(sequence):
   for i,part in enumerate(sequence):
      to_search = precedence_list[part]
      done_sequence = sequence[0:i]
      to_search_final = [x for j, x in enumerate(to_search) if j not in done_sequence]
      for a in to_search_final:
         if a == '1':
            return False
   return True

def check_precedence_swaps(sequence):
   score = 0
   for i,part in enumerate(sequence):
      to_search = precedence_list[part]
      done_sequence = sequence[0:i]
      to_search_final = [x for j, x in enumerate(to_search) if j not in done_sequence]
      for a in to_search_final:
         if a == '1':
            score += 1
   return score

def fitness_func(chromosome):
   chromosome = list(chromosome)
   if check_precedence_criteria(chromosome):
      return 0.0005*(calc_tool_changes(chromosome) + calc_orientation_changes(chromosome))
   else:
      return check_precedence_swaps(chromosome)*0.01
def calc_orientation_changes(sequence):
    orien_changes = 0
    current_orientation = orientations[sequence[0]]
    for part in sequence[1:]:
        if orientations[part] != current_orientation:
            orien_changes += 1
            current_orientation = orientations[part]
    return orien_changes
def calc_tool_changes(sequence):
    tool_changes = 0
    current_tool = tools_grippers[sequence[0]]
    for part in sequence[1:]:
        if tools_grippers[part] != current_tool:
            tool_changes += 1
            current_tool = tools_grippers[part]
    return tool_changes
def init_pop(genome, **args):
    genome.genomeList = range(0, NUM_PARTS)
    shuffle(genome.genomeList)
genome = G1DList.G1DList(NUM_PARTS)
genome.setParams(rangemin=0, rangemax=NUM_PARTS-1)
genome.initializator.set(init_pop)
genome.mutator.set(Mutators.G1DListMutatorSwap)
genome.crossover.set(Crossovers.G1DListCrossoverCutCrossfill)
genome.evaluator.set(fitness_func)
ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(200)
ga.setPopulationSize(60)
ga.selector.set(Selectors.GTournamentSelector)
# Set type of objective/ fitness function: Convergence
ga.setMinimax(Consts.minimaxType["minimize"])
ga.evolve(freq_stats=50)
if check_precedence_criteria(list(ga.bestIndividual())) == True:
    print "Final Sequence: " , [ m+1 for m in list(ga.bestIndividual()) ]
    print "Tool Changes: ", calc_tool_changes(list(ga.bestIndividual()))
    print "Orientation Changes: ", calc_orientation_changes(list(ga.bestIndividual()))
else:
    print "Genetic Algorithm - Unsuccessful!"
```

[4] Python - Genetic Algorithm Parameter Variation code (16 parts) -

```
# Code For Mutation Rate Parameter Variation v/s Fitness value
import csv
```

```
from random import shuffle
from pyevolve import G1DList, GSimpleGA, Consts, Initializators, Selectors, Mutators, Crossovers
import matplotlib.pyplot as plt
import numpy


#Number of Parts
NUM_PARTS = 16

#Precedence Matrix
with open('precedence_matrix_16.csv', 'rb') as csvfile:
  precedence_list = []
  for line in csvfile.readlines():
    array = line.strip().encode('utf-8').split(',')
    precedence_list.append(array)

#Directions / Orientations
orientations = ['-z','+x','-z','-z','-z','-z','-z','-z','-z','-z','+y','+y','-y','-y','+x','+x']
#Tool Grippers
tools_grippers = ['A','B','C','C','D','D','D','D','E','E','E','E','E','E','F','F']

def check_precedence_criteria(sequence):
  for i,part in enumerate(sequence):
    to_search = precedence_list[part]
    done_sequence = sequence[0:i]
    to_search_final = [x for j, x in enumerate(to_search) if j not in done_sequence]
    for a in to_search_final:
      if a == '1':
        return False
  return True

def check_precedence_swaps(sequence):
  score = 0
  for i,part in enumerate(sequence):
    to_search = precedence_list[part]
    done_sequence = sequence[0:i]
    to_search_final = [x for j, x in enumerate(to_search) if j not in done_sequence]
    for a in to_search_final:
      if a == '1':
        score += 1
  return score

def fitness_func(chromosome):
  chromosome = list(chromosome)
  if check_precedence_criteria(chromosome):
    return 0.0005*(calc_tool_changes(chromosome) + calc_orientation_changes(chromosome))
  else:
    return check_precedence_swaps(chromosome)*0.01

def calc_orientation_changes(sequence):
    orien_changes = 0
    current_orientation = orientations[sequence[0]]
    for part in sequence[1:]:
        if orientations[part] != current_orientation:
            orien_changes += 1
            current_orientation = orientations[part]
    return orien_changes

def calc_tool_changes(sequence):
    tool_changes = 0
    current_tool = tools_grippers[sequence[0]]
    for part in sequence[1:]:
        if tools_grippers[part] != current_tool:
            tool_changes += 1
            current_tool = tools_grippers[part]
    return tool_changes

def init_pop(genome, **args):
    genome.genomeList = range(0, NUM_PARTS)
    shuffle(genome.genomeList)

genome = G1DList.G1DList(NUM_PARTS)
genome.setParams(rangemin=0, rangemax=NUM_PARTS-1)
genome.initializator.set(init_pop)
# Set mutator function
genome.mutator.set(Mutators.G1DListMutatorSwap)
# Set Crossover function
genome.crossover.set(Crossovers.G1DListCrossoverCutCrossfill)
genome.evaluator.set(fitness_func)

mutation_rate = []
fitness_value = []
best_value = [1,1]
for x in numpy.arange(0.008,0.04,0.001):
    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(200)
    ga.setPopulationSize(50)
    ga.setCrossoverRate(0.9)
    ga.selector.set(Selectors.GTournamentSelector)
    # Set type of objective/ fitness function: Convergence
    ga.setMinimax(Consts.minimaxType["minimize"])
    ga.setMutationRate(x)
    ga.evolve()
    best = ga.bestIndividual()
    # print best.score
    if check_precedence_criteria(list(best)) == True:
        print "X: ", x, best.fitness
        if best.fitness < best_value[1]:
            best_value = [x,best.fitness]
        mutation_rate.append(x)
        fitness_value.append(best.fitness)
print "Best Fitness: ", best_value
plt.plot(mutation_rate, fitness_value)
plt.xlabel('Mutation rate')
plt.ylabel('Fitness Value')
plt.title('Mutation rate v/s Fitness value')
plt.grid(True)
plt.show()
```