TEB2023: Artificial Intelligence - September 2024

TITLE

Applying Genetics Algorithm to Solve Machine (Job Shop) Scheduling Problem

No	Name	ID	Prog	HP No.
1	LU HOU YANG	22012138	CS	0102580630
2	ARLEEN APRIL CHONG	22006582	CS	01119548078
3	MOHAMAD AKRAM BIN MOHD FAISAL	22006626	CS	0149730613
4	LIM JIA CHYUEN	22005444	CS	0184739166
5	SHARVIN A/L KANESAN	22006930	CS	01126268129
6	LESTER KHOO ZHEN JIE	22002950	CS	0163809348

INDEX

Source Code

Source Code GitHub Repository: https://github.com/luhouyang/AI_Assignment.git

Data and Analysis: Excel_Sheet

Industry Problem & Importance

The problem that has been chosen to tackle in this assignment is the 'Scheduling Problem in Manufacturing Systems' [1] also known as 'Scheduling in Job Shop Environment' [2]. The scheduling problem is usually described as the assignment of limited, available resources to activities in order to maximize productivity and resource utilization [1,2]. For manufacturing systems, it is manufacturing processes that are assigned to machines. This industry problem is important because of its wide-ranging use cases such as in production planning, process scheduling, workforce scheduling, etc. While these tasks can be solved using traditional methods that make use of conventional information systems and algorithms [1,2], the time and resources needed to compute the solution increases exponentially with the number of jobs and processes as the feasible schedules also increase exponentially. Hence, machine scheduling falls in the NP-hard class of combinatorial optimization problem [2].

Genetic algorithm (GA) can be used to effectively solve the machine scheduling problem, as optimal or near-optimal solutions can be found using less time and compute resources [1]. This provides benefits to sectors such as manufacturing, human resources, supply line management, etc.

Genetic Algorithm Design & How it Solve the Problem

Genetic Algorithm (GA)

Genetic Algorithms (GAs), a type of evolutionary algorithm (EA), are heuristic search methods inspired by the concept of survival-of-the-fittest from Darwinian evolution. They operate iteratively on a population of potential solutions, simulating natural processes like genetic crossover, mutation, and selection using mathematical operators. This stochastic optimization

technique effectively translates the principles of natural evolution into a robust algorithmic framework. Unlike other meta-heuristic methods, GAs are highly versatile, making them well-suited for tackling a wide range of optimization problems, including both single-objective and multi-objective challenges. Their adaptability has been particularly advantageous in complex applications such as scheduling problems in manufacturing systems, where they efficiently explore and exploit solution spaces [1,2].

Representation of the Problem

The specific scheduling problem tackled in this assignment involves scheduling ‘n’ products on ‘m’ machines. There are time slots ‘t’ in the schedule that can be changed dynamically i.e. 10 minutes per slot for 8 hours which amounts to 48 slots. Each product consists of ‘p’ number of processes that take ‘p_t’ time slots to carry out and must be carried out in a specified order. Furthermore, there is a demand ‘d’ for each product that must be met and there are a limited number of machines available for each process. The products and machines are bound by a few constraints:

- All demand ‘d’ for each product must be met.
- Products must be processed in the specified order.
- No two operations can be performed on a product simultaneously.
- Each machine can only carry out one type of process.
- A machine can process only one product at a time.
- Operations once started cannot be interrupted.

```
PROCESSES = ['Assembly', 'Testing', 'Packaging']
PROCESS_TIMES = {
    'Product 1': {
        'Assembly': 2,
        'Testing': 1,
        'Packaging': 1
    },
    'Product 2': {
        'Assembly': 3,
        'Testing': 2,
        'Packaging': 1
    },
    'Product 3': {
        'Assembly': 1,
        'Testing': 2,
        'Packaging': 2
    }
}
DEMAND = {'Product 1': 10, 'Product 2': 10, 'Product 3': 10}
MACHINES = {'Assembly': 7, 'Testing': 5, 'Packaging': 5}
WORK_HOURS = 8
TIME_SLOT_DURATION = 10 # minutes
```

Figure 1: Example problem 'Medium' taken from source code https://github.com/luhouyang/AI_Assignment.git

In the example above, there are 3 products (Product 1, Product 2, and Product 3) and each has 3 processes (Assembly, Testing, and Packaging). The processes of each product have different time requirements. The demand for each product is (10, 10, 10) for (Product 1, Product 2, and Product 3). The machines available for each process are (7, 5, 5) for (Assembly, Testing, and Packaging). The work hours are 8 and each time slot is 10 minutes, so there will be 48 time slots.

In a schedule all processes will be represented as a list, with each element having the product, process, machine, and time slot. For example, 'Assembly' of 'Product 1' using machine 3 at time slot 1 will be represented as ('Product 1', 'Assembly', 2, 0) using zero indexing for machine and time slot. Another example of 'Testing' of 'Product 3' using machine 1 at time slot 12 will be represented as ('Product 3', 'Testing', 0, 11).

```
PROCESSES = ['Assembly', 'Testing', 'Packaging', 'Loading']
PROCESS_TIMES = {
    'Cookie': {
        'Assembly': 2,
        'Testing': 1,
        'Packaging': 1,
        'Loading': 1
    },
    'EV car': {
        'Assembly': 10,
        'Testing': 2,
        'Packaging': 1,
        'Loading': 1
    },
    'Hose': {
        'Assembly': 1,
        'Testing': 2,
        'Packaging': 2,
        'Loading': 2
    },
    'Plumbus': {
        'Assembly': 4,
        'Testing': 5,
        'Packaging': 2,
        'Loading': 2
    },
    'Bomb': {
        'Assembly': 1,
        'Testing': 4,
        'Packaging': 5,
        'Loading': 2
    },
    'Cake': {
        'Assembly': 3,
        'Testing': 1,
        'Packaging': 2,
        'Loading': 1
    },
    'Bolts': {
        'Assembly': 1,
        'Testing': 1,
        'Packaging': 1,
        'Loading': 1
    }
}
DEMAND = {
    'Cookie': 15,
    'EV car': 10,
    'Hose': 14,
    'Plumbus': 7,
    'Bomb': 7,
    'Cake': 7,
    'Bolts': 20
}
MACHINES = {'Assembly': 22, 'Testing': 15, 'Packaging': 13, 'Loading': 13}
WORK_HOURS = 12
TIME_SLOT_DURATION = 5
```

Figure 2: Example problem 'Largest' taken from source code https://github.com/luhouyang/AI_Assignment.git

In the example above, there are 7 products (Cookie, EV car, Hose, Plumbus, Bomb, Cake, and Bolts) and each has 4 processes (Assembly, Testing, Packaging, and Loading). The processes of each product have different time requirements. The demand for each product is (15, 10, 14, 7, 7, 7, 20) for products (Cookie, EV car, Hose, Plumbus, Bomb, Cake, and Bolts). The machines available for each process are (22, 15, 13, 13) for (Assembly, Testing, Packaging, and Loading). The work hours are 12 and each time slot is 5 minutes, so there will be 144 time slots.

Schedules that are produced consist of a list of (product, process, machine, time slot), which are outputted in ascending order according to time slot together with relevant information as in Appendix 1.

Initial Population

Due to the nature of the problem, where processes of a product have to be carried out in a specific order. The optimal or near-optimal result usually takes a staircase shape, where later processes take up later time slots.

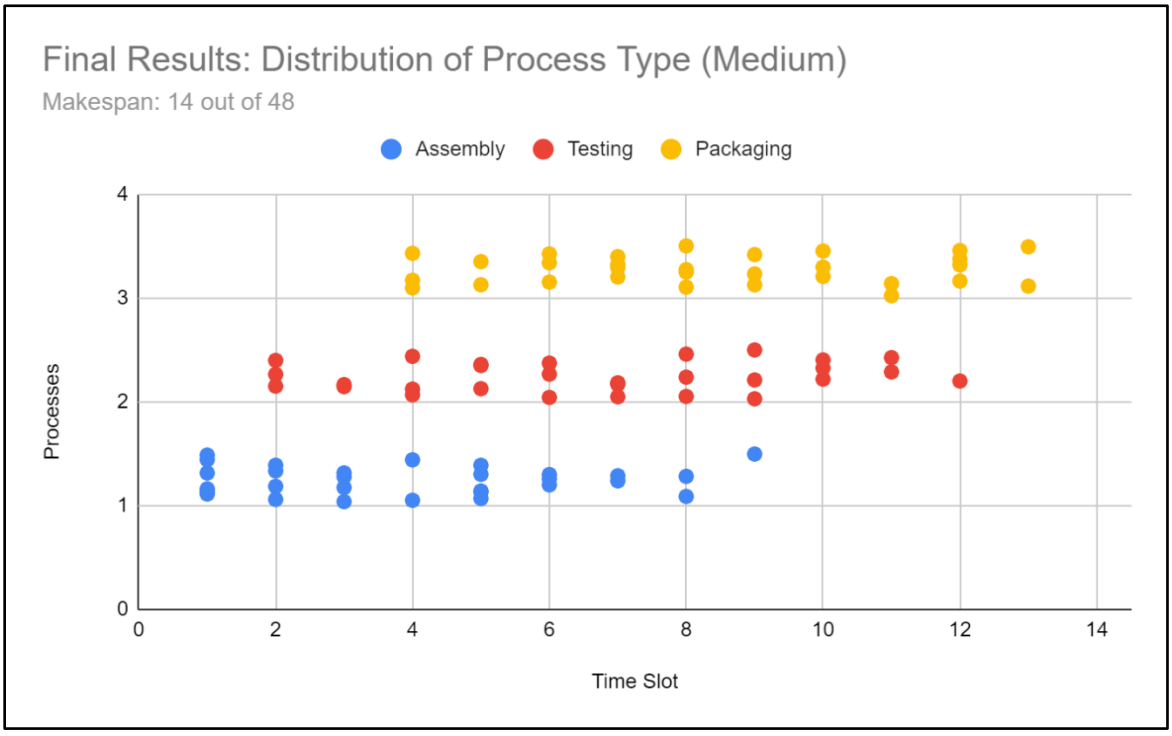


Figure 3: Final results of 'Medium' problem [Crossover: 0.85; Mutation: 0.01; Population: 100; Generations: 10000; Tournament Size: 4], distribution of process types vs time slot. [Chart Link] [Data]

Figure 3 shows an example solution for ‘Medium’ problem as described in Figure 1. The available time slots are 48 and the optimized solution achieved a makespan of 14. A staircase like distribution of the process according to order of process can be observed, with ‘Assembly’ being concentrated near the front time slots.

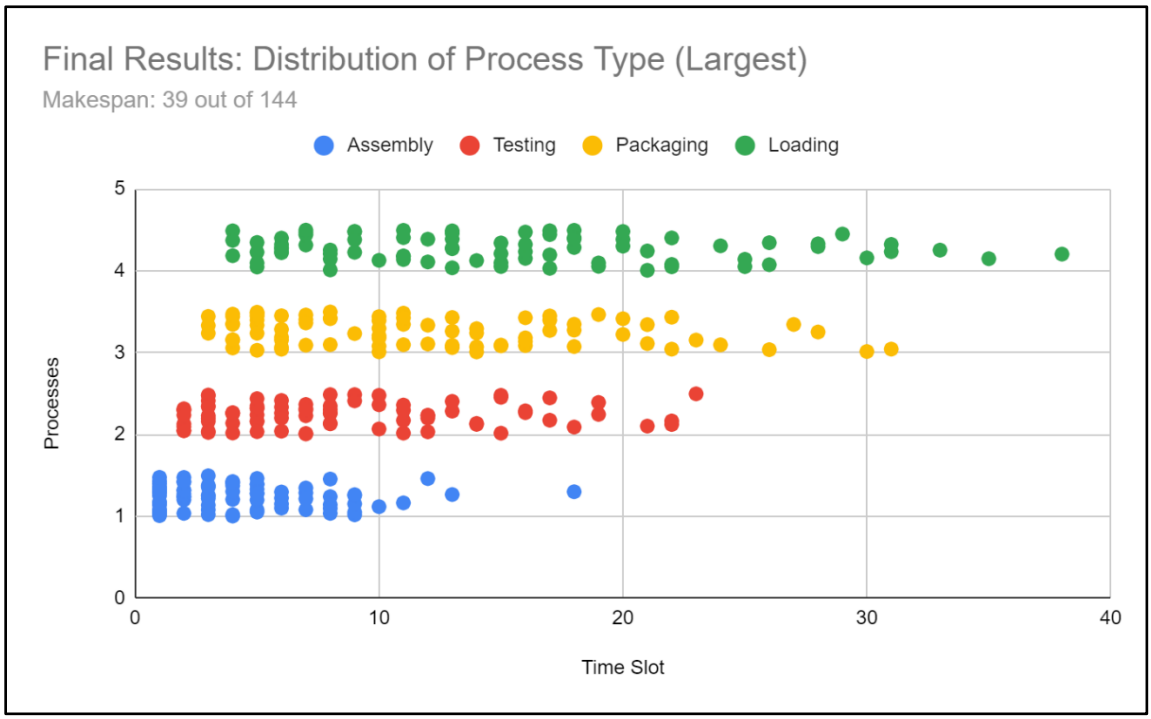


Figure 4: Final results of 'Largest' problem [Crossover: 0.85; Mutation: 0.01; Population: 150; Generations: 10000; Tournament Size: 4], distribution of process types vs time slot. [Chart Link] [Data]

Figure 4 shows an example solution for ‘Largest’ problem as described in Figure 2. The available time slots are 144 and the optimized solution achieved a makespan of 39. In Figure 4 the staircase like distribution of processes is also present. From the observations, a custom function to generate initial population that has similar distribution is created.

Pseudocode for initial population generation
1. INITIALIZE EPS=1.5, SCHEDULE=[]
2. FOR EACH product:
3. FOR EACH process:
4. FROM 1 to DEMAND of product:
5. machine = RANDOM INT BETWEEN 0 AND (available machines for process - 1)
6. CALCULATE TIME_SLOT:
7. lowest_index = PROCESS_LAG of product of process
8. highest_index = TIME_SLOTS - PROCESS_TIMES of product of process
9. extend_range = highest_index * ((index of process + EPS) / number of processes)

10. time_slot = RANDOM INT BETWEEN lowest_index AND MIN(lowest_index + extend_range, highest_index)
11. APPEND (product, process, machine, time_slot) to SCHEDULE
12. RETURN SCHEDULE

```python
def create_individual():
    eps = 1.5
    schedule = []
    for product in PROCESS_TIMES:
        for process in PROCESS_TIMES[product]:
            for _ in range(DEMAND[product]):
                machine = random.randint(0, MACHINES[process] - 1)

                lowest_index = process_lag[product][process]
                highest_index = TIME_SLOTS - PROCESS_TIMES[product][process]

                extend_range = int(highest_index * (PROCESSES.index(process) + eps) / (len(PROCESSES)))

                time_slot = random.randint(lowest_index, min(lowest_index + extend_range, highest_index))

                schedule.append((product, process, machine, time_slot))
    return schedule
```

Figure 5: Initial population generator function, source code https://github.com/luhouyang/AI_Assignment.git

Using the algorithm shown in Figure 5, an initial population that has a similar distribution to the observed solutions is created.

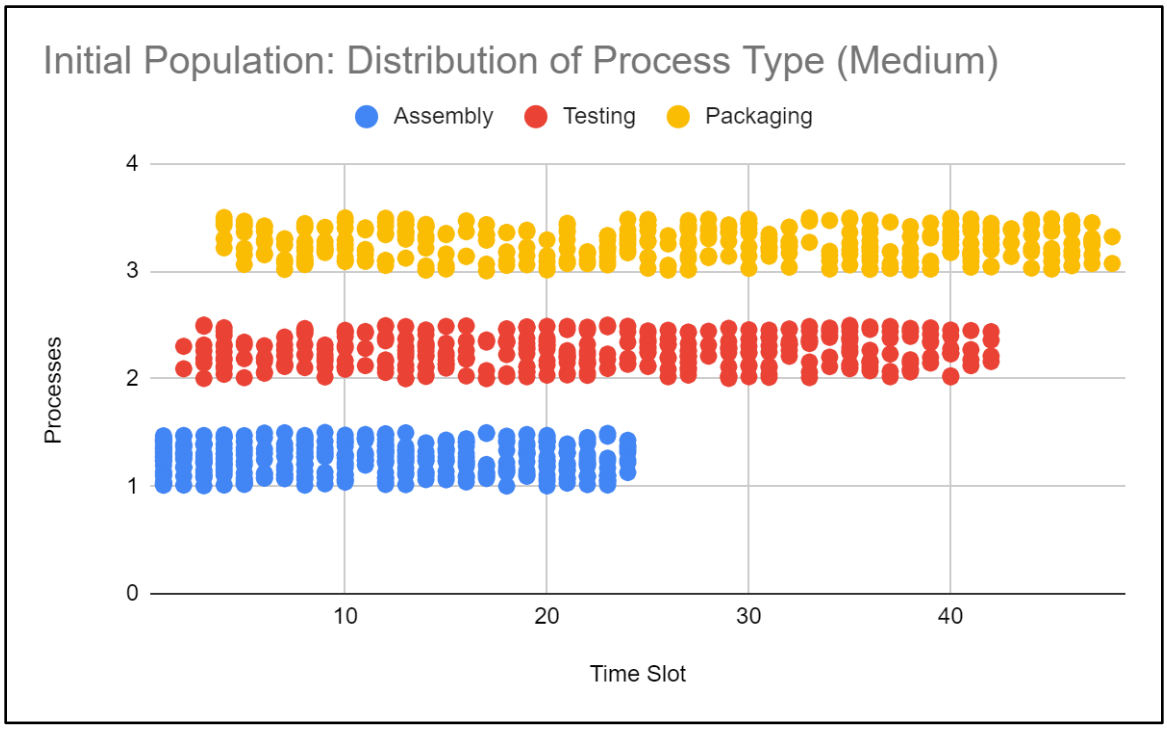


Figure 6: 'Medium' problem, 1000 data points from generated individuals [Chart] [Data]

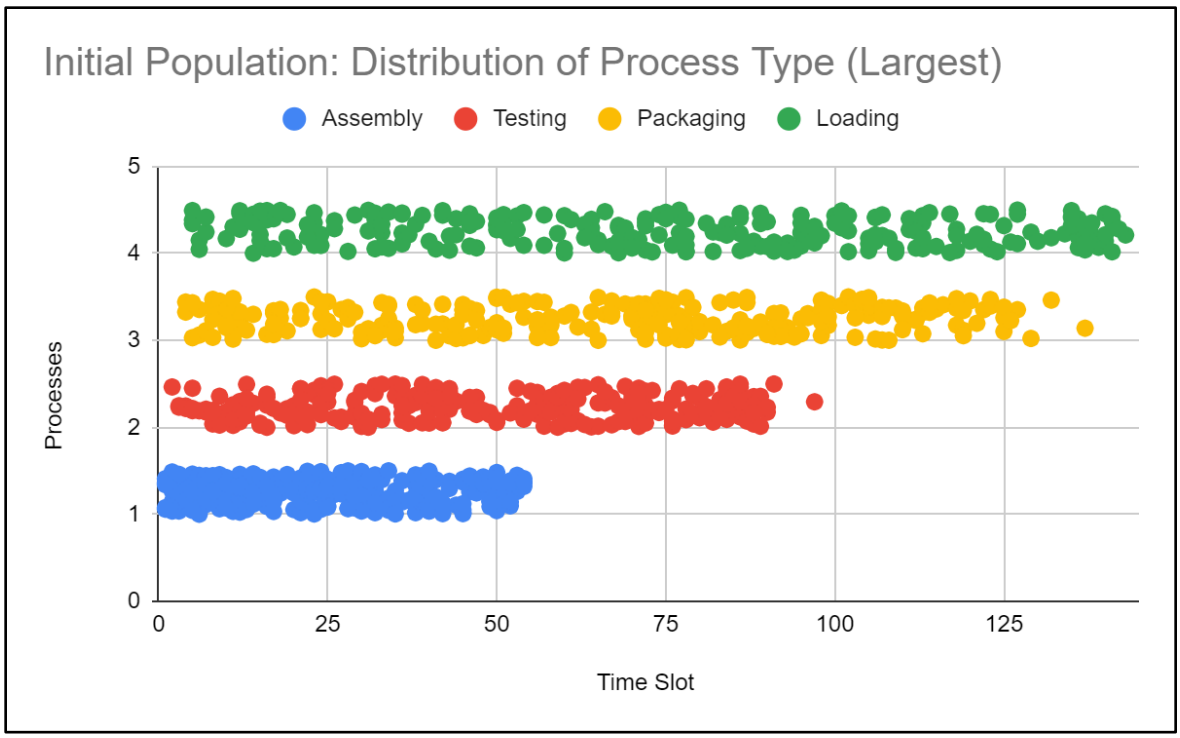


Figure 7: 'Largest' problem, 1000 data points from generated individuals [Chart] [Data]

Evaluation, Fitness Function

Pseudocode for evaluation of individuals
1. Initialize PENALTY=0, END_TIMES=[], EMPTY_MACHINES=0, PRODUCTS_WAITING=0, COMPLETED=[], MACHINE_STATE=[] 2. SORT all process in the indivisual according to TIME_SLOT 3. FOR EACH individual: 4. CALCULATE END_TIME = START_TIME + DURATION 5. INCREMENT the TIME_SLOT of products waiting for next process in COMPLETED 6. ADD amount of products that are waiting to PRODUCTS_WAITING 7. IF there is product that has completed previous process: 8. DECREMENT the product from COMPLETED 9. ELSE: 10. ADD 10000 to PENALTY 11. IF machine in current TIME_SLOT is not occupied (i.e. 0): 12. UPDATE MACHINE_STATE at TIME_SLOT with 1 13. ELSE: 14. ADD 10000 to PENALTY 15. FROM PRODUCT_LAG to END_TIME: 16. IF MACHINE_STATE for the process equal 0: 17. EMPTY_MACHINES += 1

```
18.     UPDATE END_TIMES list with END_TIME
19. REPEAT FOR all processes in individual
20. MAKESPAN = max(END_TIMES) + PENALTY
21. RETURN MAKESPAN, EMPTY_MACHINE, PRODUCTS_WAITING
```

Source code https://github.com/luhouyang/AI_Assignment.git

The evaluation function in the Genetic Algorithm (GA) for the Job Shop Machine Scheduling Problem is designed to assess the fitness of a given schedule based on three primary metrics: 'Makespan', 'Empty Machines', and 'Waiting Products'. These metrics work together to ensure that the schedule adheres to the problem's constraints while minimizing inefficiencies.

Makespan represents the total time required to complete all processes for all products. It is determined as the maximum end time across all processes, with penalties applied for scheduling violations such as overlapping machine usage or processing a product before completing its preceding steps. Minimizing the makespan ensures the schedule completes in the shortest possible time.

Empty Machines measure the total number of idle machine slots across all processes up to the current time. By encouraging the utilization of empty machine slots within the same time frame, this metric promotes vertical optimization, aiming to maximize resource usage and reduce inefficiencies.

Waiting Products quantifies the total number of products that have finished one process and are waiting for the next process to start. By encouraging the movement of processes to earlier time slots, this metric facilitates horizontal optimization, minimizing product that are left idle and improving overall flow.

The function evaluates each process sequentially, sorting the schedule by start times. It dynamically tracks the state of machines (occupied or idle) and the readiness of products for subsequent processes, applying heavy penalties (10000 to makespan) for violations such as overlapping machine usage or attempting to proceed out of sequence. By balancing these three metrics, the evaluation function identifies schedules that not only minimize makespan but also improve machine utilization and product flow, leading to a well-optimized solution that adheres to all problem constraints.

Two-Point Crossover

Pseudocode for two-point crossover
1. INITIALIZE SIZE = length of individual process list 2. SELECT 2 random integers in between 1 and (SIZE - 1) and set to POINT_1 and POINT_2 3. SWAP the intergers IF the POINT_1 is greter than POINT_2 4. FROM POINT_1 to POINT_2: 5. SWAP BOTH machine AND time_slot 6. RETURN both individuals

```
427  def cxSelectiveTwoPoint(ind1, ind2):
428      # choose crossover points
429      size = len(ind1)
430      cxpoint1 = random.randint(1, size - 1)
431      cxpoint2 = random.randint(1, size - 1)
432
433      # ensure cxpoint1 is less than cxpoint2
434      if cxpoint1 > cxpoint2:
435          cxpoint1, cxpoint2 = cxpoint2, cxpoint1
436
437      # swap the `machine` and `time_slot` between the two indivi
438      for i in range(cxpoint1, cxpoint2):
439          # keep `product` and `process` constant
440          product1, process1, machine1, time_slot1 = ind1[i]
441          product2, process2, machine2, time_slot2 = ind2[i]
442
443          ind1[i] = (product1, process1, machine2, time_slot2)
444          ind2[i] = (product2, process2, machine1, time_slot1)
445
446      return ind1, ind2
```

Figure 8: Source code https://github.com/luhouyang/AI_Assignment.git

In genetic algorithms, the two-point crossover function is a recombination operator designed to combine genetic material from two parent solutions, generating offspring that inherit features from both. This method helps explore the solution space by introducing new combinations of genes, potentially leading to better-performing solutions in subsequent generations. The selective two-point crossover function implemented here focuses on preserving the integrity of the schedule while exchanging information between two individuals.

The function begins by selecting two random crossover points within the length of the individual's process list, ensuring that the first point is less than or equal to the second. These points define the range within which the crossover occurs. For each gene (process) between these points, the function swaps the 'machine' and 'time_slot' attributes of the two individuals while keeping the 'product' and 'process' attributes unchanged. This approach ensures that the processes remain consistent with the scheduling constraints while introducing variability in the machine assignments and time slots.

The function returns two new individuals with recombined genetic material. By swapping specific attributes selectively, this method maintains the feasibility of the schedules and encourages diverse yet valid solutions, increasing the chances of finding an optimal or near-optimal solution in the evolving population.

Mutation

Pseudocode for mutation of machine and time slot
1. FOR EACH process in the individual: 2. IF random probability < MUTATION_RATE: 3. MUTATE by randomly assigning MACHINE 4. IF random pobability < MUTATION_RATE: 5. MUTATE by randomly assigning TIME_SLOT 6. RETURN individual

```
488  def mutate(individual, indpb):
489      for i in range(len(individual)):
490          # unpack the current schedule entry
491          product, process, machine, time_slot = individual[i]
492
493          # apply mutation based on the probability `indpb`
494          if random.random() < indpb:
495              machine = random.randint(0, MACHINES[process] - 1)
496
497          if random.random() < indpb:
498              time_slot = random.randint(
499                  process_lag[product][process],
500                  TIME_SLOTS - PROCESS_TIMES[product][process])
501
502          # update the individual's schedule with the mutated values
503          individual[i] = (product, process, machine, time_slot)
504
505      return (individual, )
```

Figure 9: Source code https://github.com/luhouyang/AI_Assignment.git

In genetic algorithms, mutation is a genetic operator that introduces diversity into the population by randomly altering the attributes of individuals. This prevents the algorithm from converging prematurely to suboptimal solutions by exploring less-visited areas of the solution space. The mutation function implemented here focuses on modifying the 'machine' and 'time_slot' attributes of processes in a scheduling problem.

The function iterates over every process in an individual's schedule. For each process, it applies mutations based on a predefined mutation probability ('indpb'). If a randomly generated number is less than 'indpb', the 'machine' is mutated by randomly selecting a new machine from those capable of performing the current process. Similarly, another random check determines if the 'time_slot' should be mutated. If so, the time slot is reassigned to a new value within valid bounds, considering the process lag and the process duration.

The mutated individual retains its feasibility while potentially introducing new combinations of machines and time slots. This approach encourages exploration of alternative schedules, which can lead to improved solutions over successive generations. By balancing the mutation probability, the algorithm ensures sufficient diversity without excessively disrupting well-performing individuals.

Tournament Selection

In genetic algorithms, 'selection' is the process of choosing individuals from the current population to create offspring for the next generation. Tournament selection is one of the most popular methods due to its simplicity and effectiveness in maintaining selective pressure. In this method, a fixed number of individuals, defined by the tournament size, are randomly chosen from the population. These individuals then compete based on their fitness, and the best among them is selected for reproduction.

In this implementation, 'tournament selection' is performed using the 'selTournament' function from the DEAP library, with the tournament size, 'tournsize' set to 4. Here, a tournament size of 4 means that, for each selection, four individuals are randomly drawn from the population, and the one with the best fitness is chosen. This approach strikes a balance between exploration and exploitation: it allows weaker individuals a chance to be selected (through randomness), while

still favoring fitter solutions. By iteratively applying this method, the algorithm constructs a mating pool of high-quality individuals, facilitating the generation of improved offspring in the subsequent evolution stages.

Results

The primary metric to measure success in solving the problem is to obtain the schedule that has the minimal makespan for a machine scheduling task of any size [1,2]. Meaning the ideal solution should make full use of available machines and schedule processes as early as possible. The results obtained by running the genetic algorithm (GA) on problems of different sizes and checking the solutions suggest that the algorithm created can reliably find near-optimal solutions.

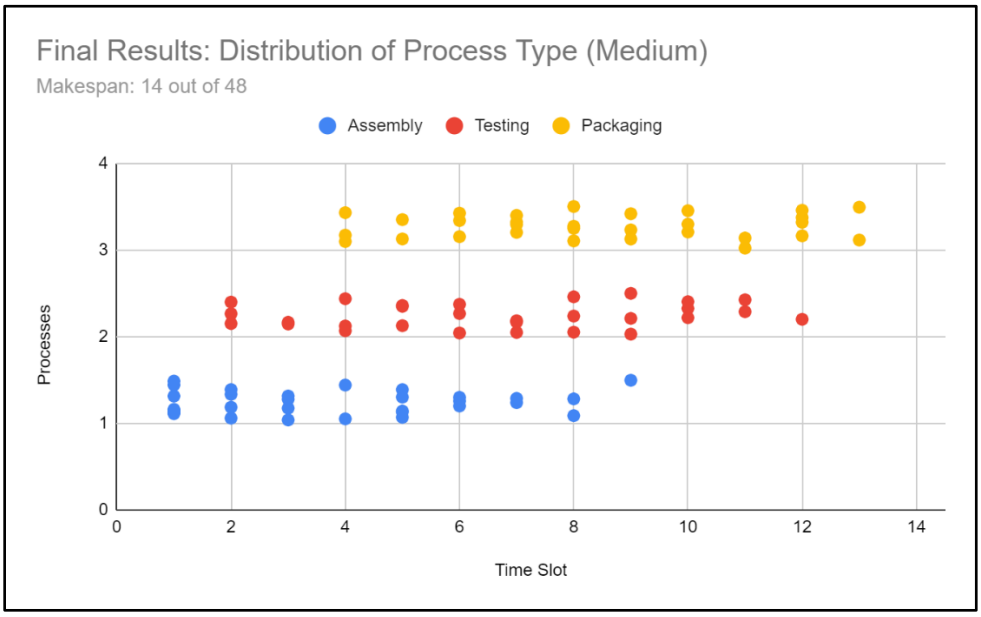
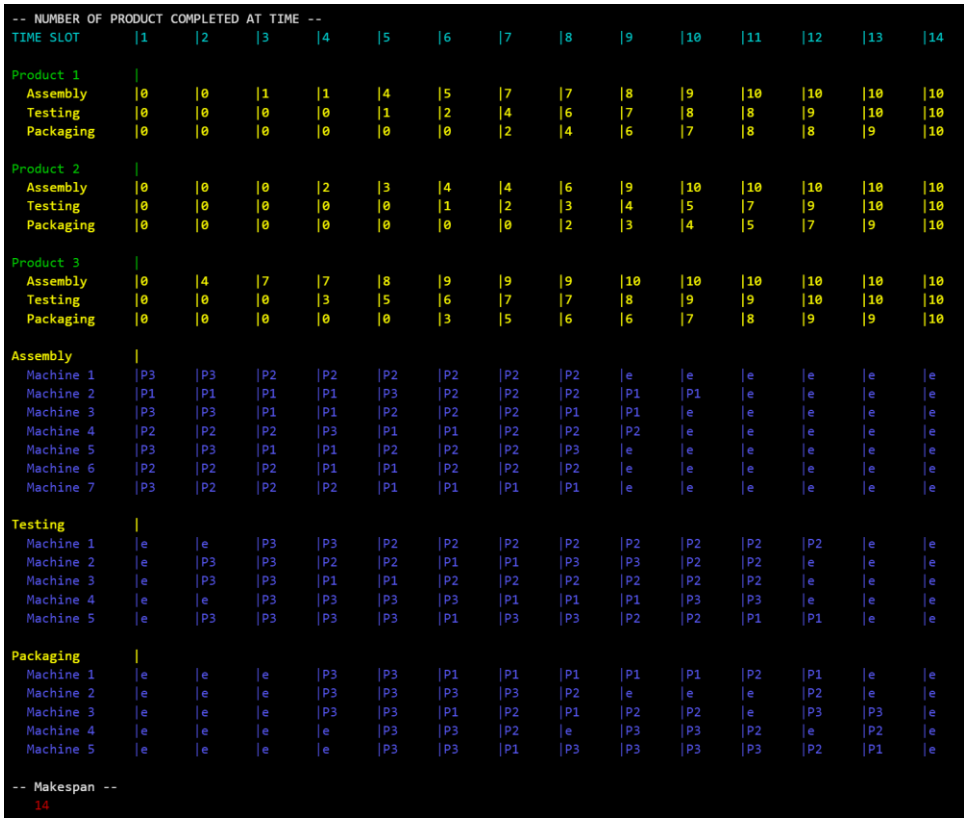


Figure 10: Final results of 'Medium' problem, same as Figure 3 [Crossover: 0.85; Mutation: 0.01; Population: 100; Generations: 10000; Tournament Size: 4], distribution of process types vs time slot. [Chart Link] [Data]



```
-- NUMBER OF PRODUCT COMPLETED AT TIME --
TIME SLOT       |1      |2      |3      |4      |5      |6      |7      |8      |9      |10     |11     |12     |13     |14

Product 1       |
  Assembly      |0      |0      |1      |1      |4      |5      |7      |7      |8      |9      |10     |10     |10     |10
  Testing       |0      |0      |0      |0      |1      |2      |4      |6      |7      |8      |8      |9      |10     |10
  Packaging     |0      |0      |0      |0      |0      |0      |2      |4      |6      |7      |8      |8      |9      |10

Product 2       |
  Assembly      |0      |0      |0      |2      |3      |4      |4      |6      |9      |10     |10     |10     |10     |10
  Testing       |0      |0      |0      |0      |0      |1      |2      |3      |4      |5      |7      |9      |10     |10
  Packaging     |0      |0      |0      |0      |0      |0      |0      |2      |3      |4      |5      |7      |9      |10

Product 3       |
  Assembly      |0      |4      |7      |7      |8      |9      |9      |9      |10     |10     |10     |10     |10     |10
  Testing       |0      |0      |0      |3      |5      |6      |7      |7      |8      |9      |9      |10     |10     |10
  Packaging     |0      |0      |0      |0      |0      |3      |5      |6      |6      |7      |8      |9      |9      |10

Assembly        |
  Machine 1     |P3     |P3     |P2     |P2     |P2     |P2     |P2     |P2     |e      |e      |e      |e      |e      |e
  Machine 2     |P1     |P1     |P1     |P1     |P3     |P2     |P2     |P2     |P1     |P1     |e      |e      |e      |e
  Machine 3     |P3     |P3     |P1     |P1     |P2     |P2     |P2     |P1     |P1     |e      |e      |e      |e      |e
  Machine 4     |P2     |P2     |P2     |P3     |P1     |P1     |P2     |P2     |P2     |e      |e      |e      |e      |e
  Machine 5     |P3     |P3     |P1     |P1     |P2     |P2     |P2     |P3     |e      |e      |e      |e      |e      |e
  Machine 6     |P2     |P2     |P2     |P1     |P1     |P2     |P2     |P2     |e      |e      |e      |e      |e      |e
  Machine 7     |P3     |P2     |P2     |P2     |P1     |P1     |P1     |P1     |e      |e      |e      |e      |e      |e

Testing         |
  Machine 1     |e      |e      |P3     |P3     |P2     |P2     |P2     |P2     |P2     |P2     |P2     |P2     |e      |e
  Machine 2     |e      |P3     |P3     |P2     |P2     |P1     |P1     |P3     |P3     |P2     |P2     |e      |e      |e
  Machine 3     |e      |P3     |P3     |P1     |P1     |P2     |P2     |P2     |P2     |P2     |P2     |e      |e      |e
  Machine 4     |e      |e      |P3     |P3     |P3     |P3     |P1     |P1     |P1     |P3     |P3     |e      |e      |e
  Machine 5     |e      |P3     |P3     |P3     |P3     |P1     |P3     |P3     |P2     |P2     |P1     |P1     |e      |e

Packaging       |
  Machine 1     |e      |e      |e      |P3     |P3     |P1     |P1     |P1     |P1     |P1     |P2     |P1     |e      |e
  Machine 2     |e      |e      |e      |P3     |P3     |P3     |P3     |P2     |e      |e      |e      |P2     |e      |e
  Machine 3     |e      |e      |e      |P3     |P3     |P1     |P2     |P1     |P2     |P2     |e      |P3     |P3     |e
  Machine 4     |e      |e      |e      |e      |P3     |P3     |P2     |e      |P3     |P3     |P2     |e      |P2     |e
  Machine 5     |e      |e      |e      |e      |P3     |P3     |P1     |P3     |P3     |P3     |P3     |P2     |P1     |e

-- Makespan --
  14
```

Figure 11: Schedule visualization of results in Figure 3, [Crossover: 0.85; Mutation: 0.01; Population: 100; Generations: 10000; Tournament Size: 4] [Record]

Figure 11 shows the visualization of the final schedule produced from the ‘Medium’ problem in Figure 1, and the results of Figure 10. The machines for ‘Assembly’ are almost fully occupied starting from the first time slot. Once products have completed the previous process, they are consumed by the next process. Machines for ‘Testing’ and ‘Packaging’ are empty near the start since there is a fixed lag from waiting for previous processes to finish. While there are some instances where machines are unoccupied, it can be said that the solution produced is near-optimal.

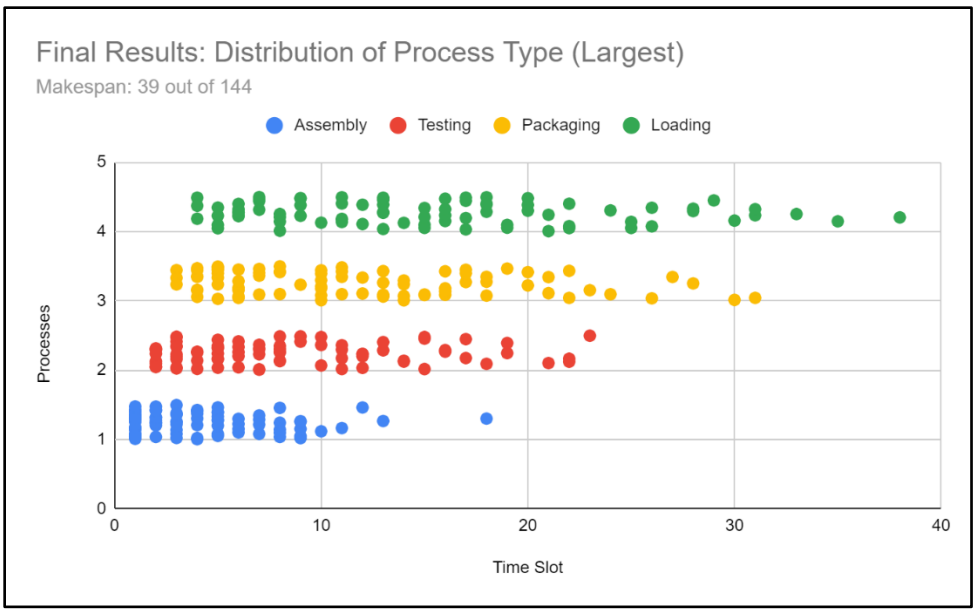


Figure 12: Final results of 'Largest' problem, same as Figure 4 [Crossover: 0.85; Mutation: 0.01; Population: 150; Generations: 10000; Tournament Size: 4], distribution of process types vs time slot. [Chart Link] [Data]

Assembly	|												
Machine 1	|P1	|P1	|P6	|P6	|P6	|e	|e	|e	|e	|e	|e	|e	|e
Machine 2	|P7	|P6	|P6	|P6	|P1	|P1	|e	|P2	|P2	|P2	|P2	|P2	|P2
Machine 3	|e	|P3	|P5	|P5	|P1	|P1	|e	|e	|e	|e	|e	|e	|e
Machine 4	|P7	|P7	|e	|e	|P7	|P7	|e	|P1	|P1	|e	|e	|e	|e
Machine 5	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|e	|P7	|e
Machine 6	|P7	|e	|P3	|P6	|P6	|P6	|e	|e	|P4	|P4	|P4	|P4	|e
Machine 7	|P4	|P4	|P4	|P4	|e	|e	|P3	|P2	|P2	|P2	|P2	|P2	|P2
Machine 8	|P1	|P1	|e	|P7	|P7	|P6	|P6	|P6	|P2	|P2	|P2	|P2	|P2
Machine 9	|P1	|P1	|P6	|P6	|P6	|e	|e	|P6	|P6	|P6	|e	|e	|e
Machine 10	|P3	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|e	|e
Machine 11	|P7	|P3	|e	|P4	|P4	|P4	|P4	|e	|P1	|P1	|e	|e	|e
Machine 12	|P5	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|e	|e
Machine 13	|P3	|e	|P3	|e	|e	|P7	|e	|e	|e	|e	|P4	|P4	|P4
Machine 14	|P4	|P4	|P4	|P4	|P1	|P1	|e	|e	|e	|e	|e	|e	|e
Machine 15	|P5	|P1	|P1	|P1	|P1	|e	|P1	|P1	|e	|e	|e	|e	|e
Machine 16	|P7	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|e	|e
Machine 17	|P1	|P1	|P1	|P1	|P1	|P1	|P1	|P1	|P7	|e	|e	|e	|P2
Machine 18	|e	|e	|P7	|e	|P3	|P3	|e	|e	|e	|P2	|P2	|P2	|P2
Machine 19	|P3	|P3	|P5	|e	|P7	|e	|P3	|P3	|P5	|e	|e	|e	|e
Machine 20	|P7	|P7	|P7	|P3	|P5	|e	|e	|e	|e	|e	|e	|e	|e
Machine 21	|P6	|P6	|P6	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2	|P2
Machine 22	|P7	|e	|P7	|P4	|P4	|P4	|P4	|e	|e	|e	|e	|e	|e
Testing	|												
Machine 1	|e	|e	|P3	|P3	|P4	|P4	|P4	|P4	|P4	|P1	|e	|P3	|P3
Machine 2	|e	|P7	|P5	|P5	|P5	|P5	|P1	|e	|e	|e	|e	|e	|e
Machine 3	|e	|e	|e	|e	|P4	|P4	|P4	|P4	|P4	|P4	|P4	|P4	|P4
Machine 4	|e	|e	|P7	|P7	|e	|P5	|P5	|P5	|P5	|e	|e	|e	|e
Machine 5	|e	|P7	|e	|e	|e	|P6	|P5	|P5	|P5	|P5	|P1	|e	|P7
Machine 6	|e	|P7	|P3	|P3	|P7	|e	|e	|e	|e	|e	|P7	|e	|e
Machine 7	|e	|P7	|P5	|P5	|P5	|P5	|P6	|P7	|e	|P3	|P3	|P5	|P5
Machine 8	|e	|e	|P7	|P1	|P7	|P3	|P3	|P1	|P6	|e	|e	|P2	|P2
Machine 9	|e	|P3	|P3	|e	|P7	|e	|e	|P6	|P4	|P4	|P4	|P4	|P4
Machine 10	|e	|P7	|P1	|P6	|P1	|P3	|P3	|P7	|e	|e	|P3	|P3	|e
Machine 11	|e	|e	|P1	|P1	|P6	|e	|e	|P3	|P3	|e	|e	|P2	|P2

Figure 13: Schedule visualization of results in Figure 3, [Crossover: 0.85; Mutation: 0.01; Population: 150; Generations: 10000; Tournament Size: 4] [Record]

Figure 13 shows the visualization of a section of the final schedule produced from the 'Largest' problem in Figure 2, and the results of Figure 12. The schedule generated also has a

similar arrangement as for the ‘Medium’ problem, where machines at the front most time slots are almost fully filled. Furthermore, the process distribution in Figure 13 indicates that once products complete one process, they are consumed relatively quickly by the proceeding processes. Together these results strongly suggest that the schedules generated follow initial intuition and are near-optimal. Similar results can be seen in another run, on another example problem in Appendix 2.

Finally, the algorithm is able to solve the machine scheduling problem with O(n) time complexity, where ‘n’ is the number of processes that have to be scheduled. Test cases of various sizes were used and the time to produce schedules are recorded in Table 1. Figure 14 shows that the solution utilizing genetic algorithm for the machine scheduling problem has significantly better improvement than traditional methods that make use of conventional information systems and algorithms [1,2]. Allowing the solution to solve the scaling problem related with machine scheduling which is a NP-hard class of combinatorial optimization problem [2].

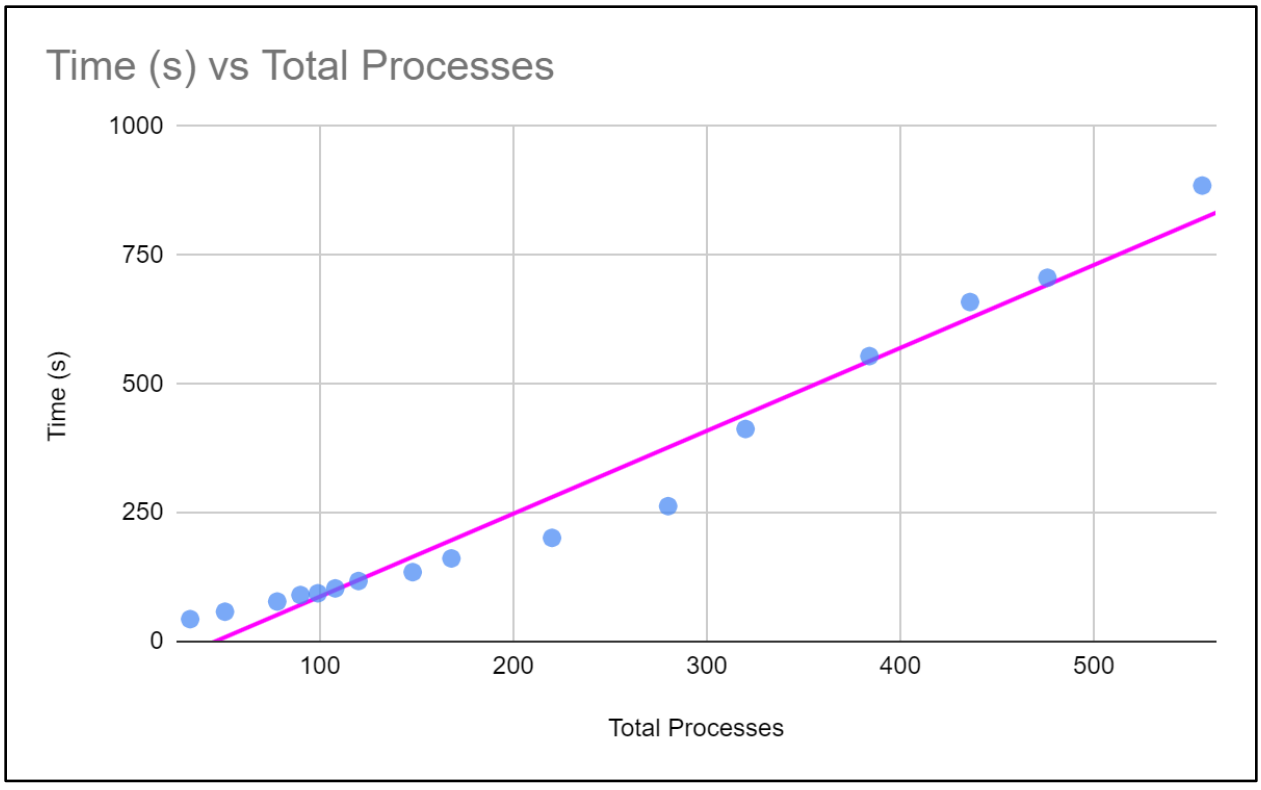


Figure 14: Time to produce schedule in seconds vs total processes.

Table 1: Test ran on 1 '13th Gen Intel® Core™ i7-13700F'

TIME	TOTAL_PROCESSES	TOTAL_PRODUCTS	TOTAL_MACHINES	POSSIBLE_TIME_SLOTS	MAX_TIME_SLOT	MAKESPAN
43.42856789	33	11	17	816	48	8
57.83863235	51	17	17	816	48	12
77.59825444	78	26	17	816	48	14
90.22689438	90	30	17	816	48	15
93.63609457	99	33	17	816	48	18
103.1303611	108	36	17	816	48	19
117.4117837	120	30	28	2016	72	21
134.6964982	148	37	28	2016	72	23
161.3110452	168	42	28	2016	72	33
201.2264462	220	55	28	2016	72	35
262.7495768	280	70	28	2016	72	44
412.5972083	320	80	63	9072	144	82
554.0825584	384	96	63	9072	144	112
659.0527923	436	109	63	9072	144	111
706.2064574	476	119	63	9072	144	122
885.1886935	556	139	63	9072	144	126

In conclusion, the genetic algorithm can reliably generate near-optimal schedules in an acceptable amount of time with O(n) time complexity. Providing a usable and scalable solution to the machine scheduling problem. All source code and logs can be found in the GitHub repository: https://github.com/luhouyang/AI_Assignment.git and Excel sheet with data analysis: https://docs.google.com/spreadsheets/d/1xaZo5W7p0s90UqtUo0wsylv6MDIH3nIWH1oN1GzGBvQ/edit?usp=sharing

Parameter Tuning

The parameters that are tuned for optimization are crossover rate, mutation rate, population size, generation, and tournament size. First a random test by changing tournament size, population size, crossover rate and mutation rate was carried out, using values recommended in a paper by A.

Hassanat, et. al. [4]. The number of generations was fixed at 4000 to allow solutions to converge. Results for the 'Medium' problem recorded at Appendix 3 showed five sets of values that tied with the lowest makespan of 15. Below presents a summary of the values:

Tournament Size	Population Size	Crossover Rate	Mutation Rate	Makespan
3	100	0.7	0.01	15
4	50	0.9	0.01	15
4	100	0.8	0.01	15
4	100	0.85	0.01	15
4	100	0.9	0.01	15

Then, the 'Largest' problem with maximum makespan of 144 was used to test the five set of values, with the lowest makespan being 46 achieved with parameters (Crossover Rate: 0.85; Mutation Rate: 0.01; Tournament Size: 4; Population Size: 100). Data

Tournament Size	Population Size	Crossover Rate	Mutation Rate	Makespan
3	100	0.7	0.01	48
4	50	0.9	0.01	79
4	100	0.8	0.01	84
4	100	0.85	0.01	46
4	100	0.9	0.01	56

After the initial exploration, a tournament size of 4, and population size of 100 was selected, while crossover rate and mutation rate were further tuned by varying both variables, creating a contour map with makespan as the height. Data collected at Appendix 4 is first plot using log scale for makespan to include incomplete solution (schedules with error).

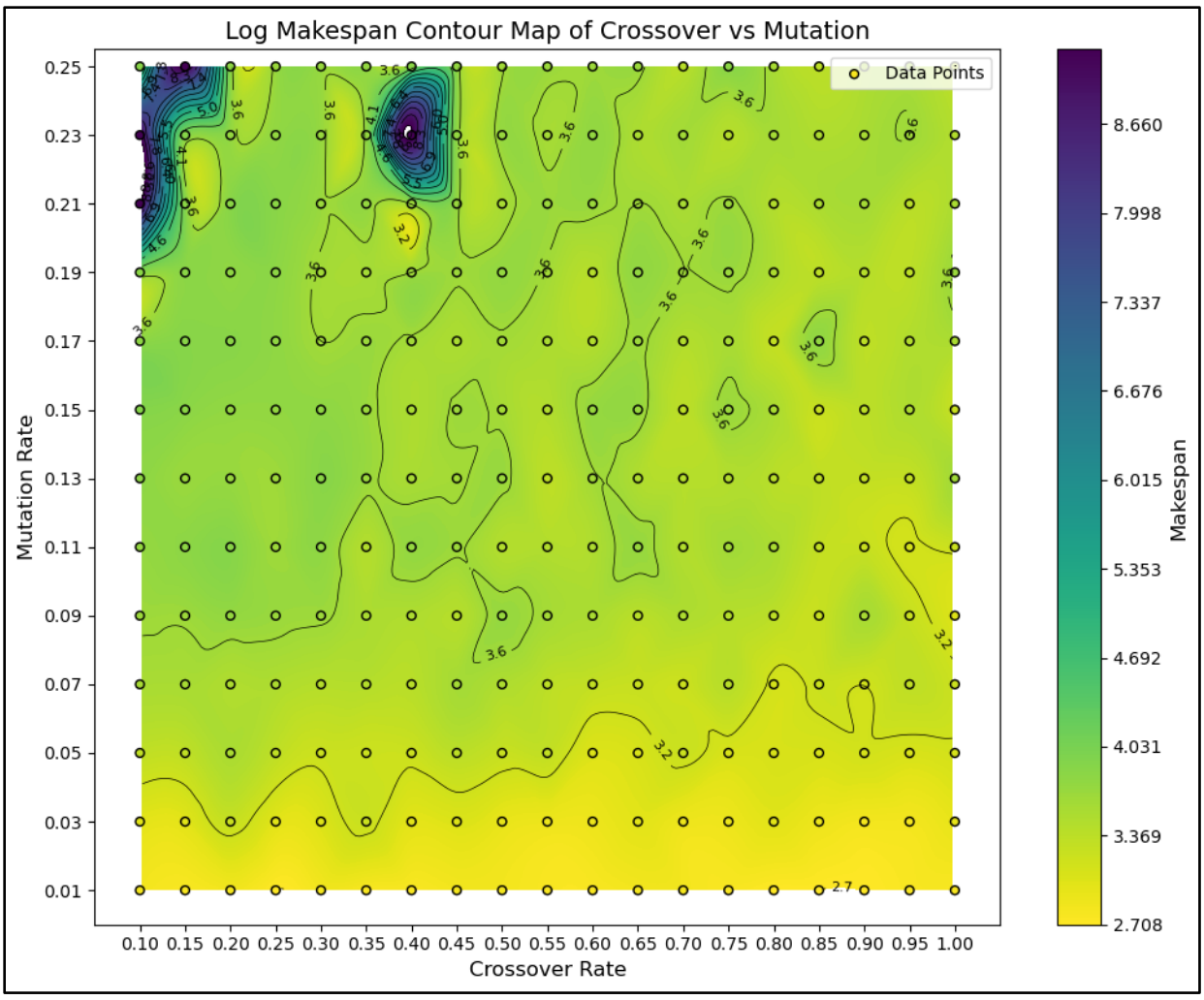


Figure 15: Log makespan contour map of crossover rate vs mutation rate.'Medium' problem. Data at Appendix 4.

From Figure 15, height difference is concentrated at extreme values, reducing the distinguishability of most of the data. To address this, the data is cleaned by removing extreme values, makespans greater than 10000 (not complete solutions) which consist of four points of data. The remaining data is used to plot a contour map without using log scale of makespan in Figure 16.

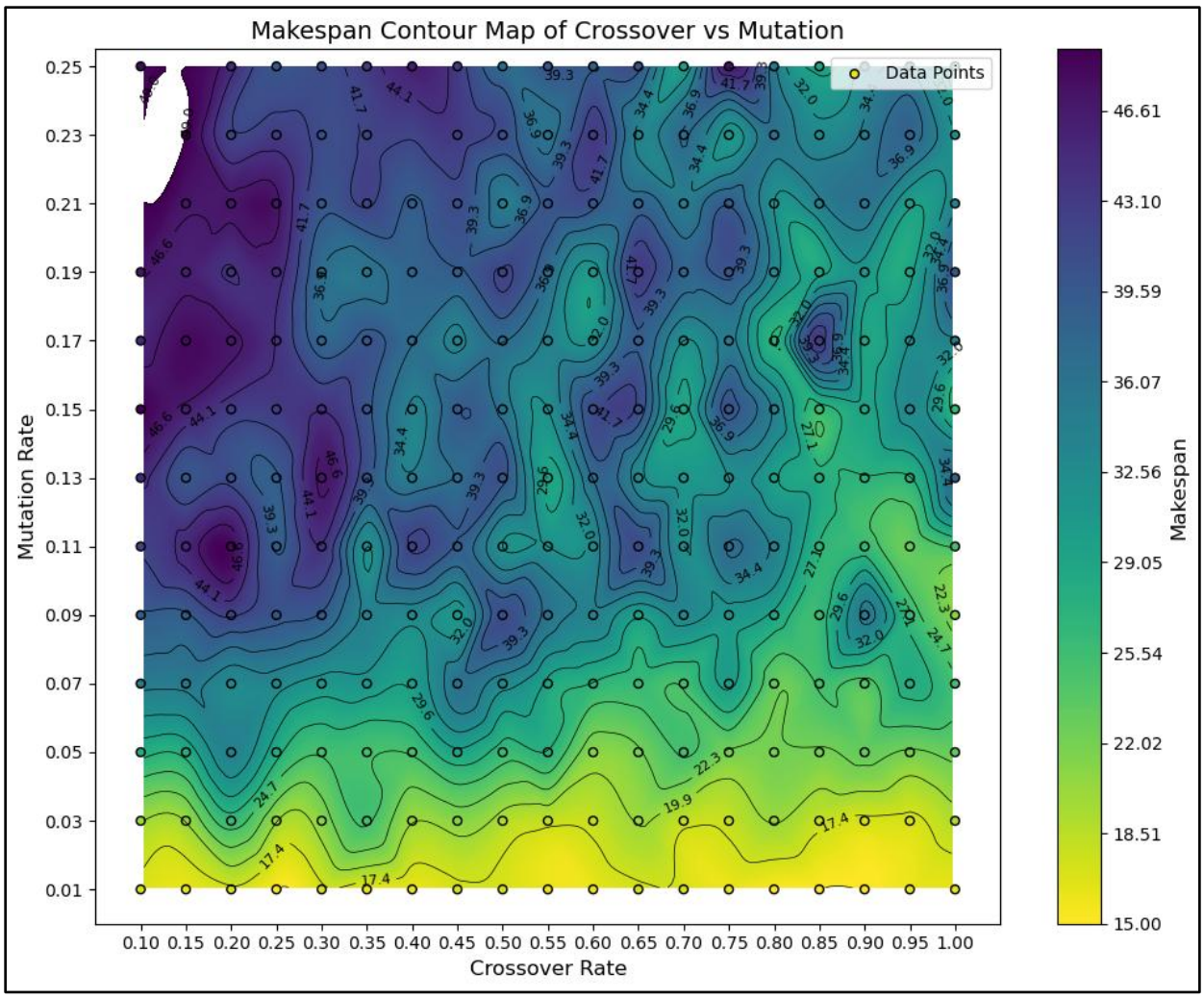


Figure 16: Makespan contour map of crossover rate vs mutation rate. Data at Appendix 4. 'Medium' problem.

From Figure 16, the optimal range of value for mutation rate showing the best results is on the lower end (0, 0.03]. Whereas for crossover rate, performance increases as crossover rate increase for all value of mutation rate. Therefore, a higher value in the range of [0.7, 0.95] is favorable. Hence, the parameters (Crossover Rate: 0.85; Mutation Rate: 0.01; Tournament Size: 4; Population Size: 100) are maintained.

Next, the effects of the number of generations on makespan were investigated. This is done by running the algorithm with 250 increments to the number of generations, starting with an initial value of 250.

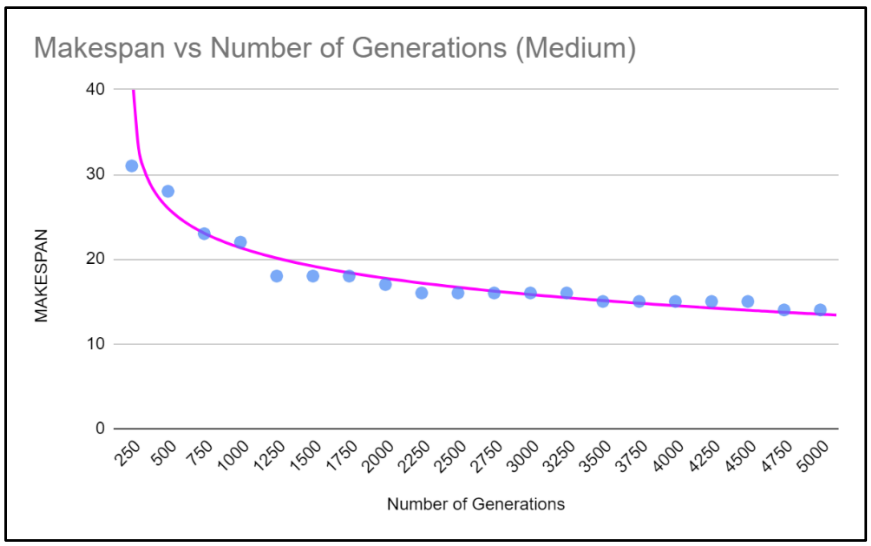


Figure 17: Makespan vs Number of generations, 'Medium' problem. [Chart] [Data]

From Figure 17 a logarithmic trend between makespan and number of generations is observed, with decreasing makespan as number of generations increase. When choosing the number of generations, the time taken to run the algorithm should also be considered, since there are diminishing returns from increasing number of generations. The same trend can be seen in the graph for 'Largest' problem in Figure 18. Hence for a number of generations 4000 was chosen since it allowed for generation of near-optimal solutions with acceptable time.

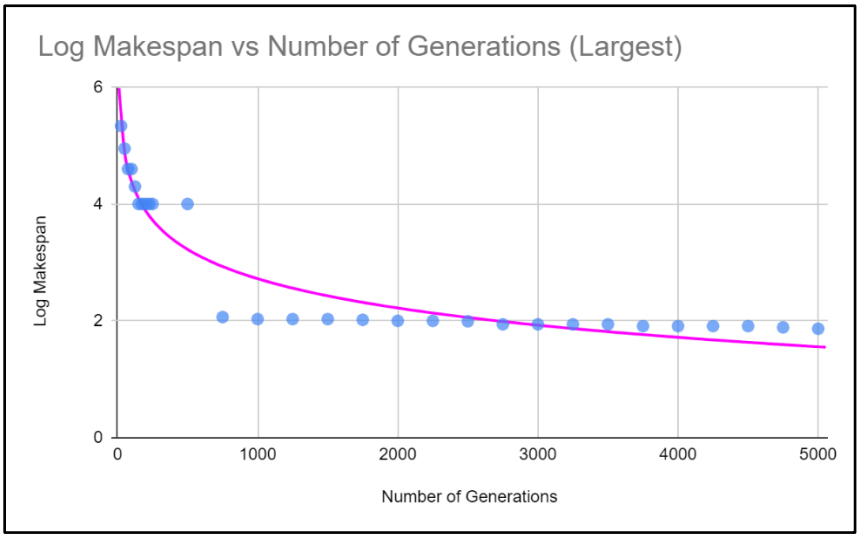


Figure 18: Makespan vs Number of generations, 'Largest' problem. [Chart] [Data]

Selection of tournament size and population size was also done by first plotting a contour map, using the 'Medium' problem and range of values of 1 to 10 for tournament size, and 50 to 500 for population size. The results are recorded at Appendix 5.

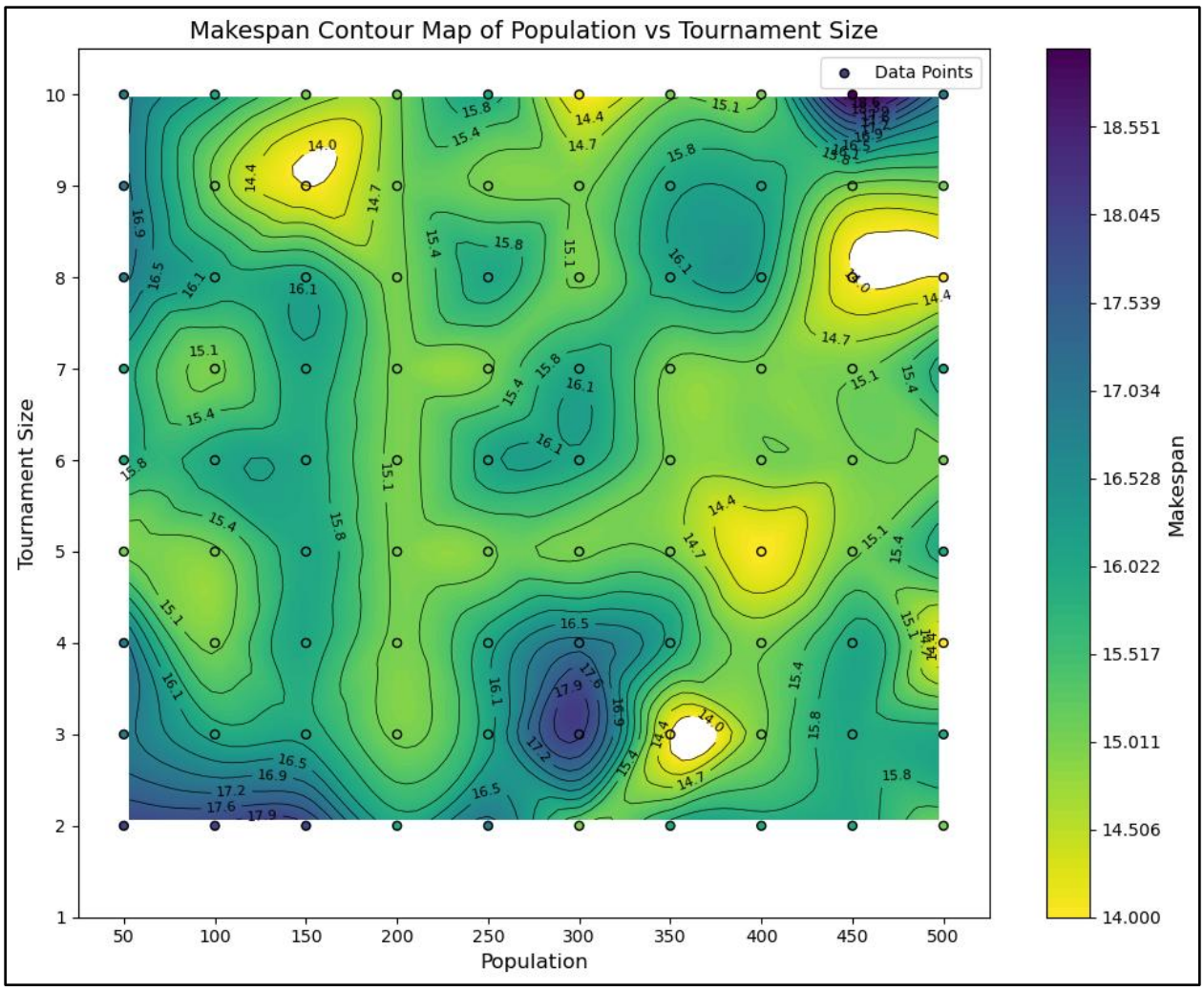


Figure 19: Makespan contour map of tournament size vs population. 'Medium' problem. Data at Appendix 5

From Figure 19 the best performing values of tournament size and population pairs are:

Tournament Size	Population
4	100
5	100
9	150
3	350
5	400
8	450
8	500

While larger tournament size and population give better results, the time needed to compute each generation increases dramatically. Hence, for the optimal performance to time 4 was selected for tournament size and 100 for population size.

Insights on the Performance of Genetic Algorithm

The application of genetic Algorithms (GAs) as versatile and robust optimization methods is inspired by the principles of natural selection and genetics. They differ from traditional algorithms that follow a predefined and deterministic step by step procedure and adopt evolutionary processes adaptively to complex problem space. The core mechanisms of Gas selection, crossover, and mutation mimic biological evolution and facilitate the exploration of a wide range of solutions. This evolutionary approach enables GAs to find optimal or near-optimal solutions, making them particularly effective for solving problems that may be infeasible for traditional algorithms [10].

One of the critical advantages of GAs is their capacity to explore multiple solution pathways simultaneously. Traditional algorithms often converge to a single solution, which may limit their applicability in problems requiring diverse exploration. By maintaining a population of solutions and iteratively improving them through genetic operators, GAs can explore a broader solution space and generate multiple high-quality outcomes over successive generations. This property makes them especially valuable in applications such as machine learning and artificial intelligence, where diversity in solutions is critical [11].

GAs became incredibly effective in the realm of cybersecurity. Traditional cryptographic algorithms frequently rely on fixed structures, which later tend to be recognizable and susceptible to attack. GAs, however, can dynamically generate variable key sizes and structures, significantly enhancing security measures against potential threats. For example, research has shown that GA-based cryptosystems provide a higher degree of robustness against unauthorized access by increasing the complexity and variability of encryption methods [12].

Furthermore, GAs is highly adaptable and can solve a diverse range of problems, from optimization challenges in logistics to scheduling problems in manufacturing. Their ability to operate without requiring derivative information or specific problem assumptions allows them to address problems with discontinuities, noise, or multiple local optima. This flexibility positions GAs as a powerful tool across various fields, including artificial intelligence, finance, and cryptography [4].

Overall, the literature demonstrates that Genetic Algorithms are a transformative approach to problem-solving. By evolving solutions iteratively and adapting dynamically to the problem

space, they have become a preferred method for tackling complex and multidimensional challenges. As research continues to refine and enhance their mechanisms, the potential applications of GAs are expected to expand further, cementing their role in advancing technological innovation [11].

Advantages of Genetic Algorithm

The numerous advantages that Genetic Algorithms possess have made them very effective along a wide range of problems focused on optimization in different areas of application. Their adaptability is one of the greatest strengths of their own. Unlike traditional optimization methods that often require specific derivative or auxiliary information, GAs rely solely on objective functions to evaluate fitness scores. This independence allows GAs to excel in solving problems with discontinuous, noisy, or non-smooth objective functions. Their applicability across fields such as engineering, logistics, and artificial intelligence underscores their versatility [1].

A key advantage of GAs is their ability to find global optima instead of getting stuck in local minima, a common issue with traditional algorithms. By exploring a broad solution space through random mutations and crossover operations, GAs increases the probability of converging to an optimal solution. This characteristic makes them particularly effective in solving highly complex problems, such as scheduling in manufacturing systems and multi-objective optimization challenges [11].

Another notable strength is the parallelism inherent in GA computations. The algorithm operates on a population of solutions, allowing multiple individuals to be processed simultaneously. This parallel structure not only enhances computational efficiency but also makes GAs well-suited for distributed computing environments. As a result, they can handle large-scale optimization problems more efficiently than traditional methods [4].

Additionally, GAs is application-agnostic, meaning they can be tailored to solve problems in a wide range of domains without needing significant modifications. This separation of the algorithm from the application allows researchers and practitioners to use GAs for various tasks, including financial modeling, medical diagnostics, and cryptographic key generation [12].

Finally, GAs is highly robust in handling problems with multiple objectives, constraints, or parameters. They excel in optimizing functions that are mixed (discrete/continuous), stochastic,

or involve many variables. This robustness, combined with their evolutionary improvement process, ensures that GAs remain a reliable tool for tackling real-world challenges effectively [11].

Limitations of Genetic Algorithm

Genetic Algorithms are powerful, but have limitations, too. Probably their most salient negative is computational expenses. GAs requires extensive exploration of the search space and perform numerous evaluations across successive generations. This iterative process, though necessary for their success, can demand significant computational resources, making GAs impractical for time-sensitive or resource-constrained applications.

Another limitation is the time-consuming nature of GAs. Due to their stochastic processes and reliance on multiple iterations to achieve convergence, GAs can take considerably longer than traditional algorithms to find an optimal solution. This characteristic makes them unsuitable for simpler problems where traditional algorithms can provide faster and equally effective results [4].

The implementation of GAs also presents challenges. The conceptual framework for GAs is not terribly difficult, but it is important to get correct since an intelligent algorithm will consider many variables: the representation of solutions, the design of the fitness function, the design of genetic operators. The poor implementation can thus yield suboptimal solutions or result in the algorithm's early convergence [12].

Moreover, GAs is less predictable in terms of the quality of results. Unlike deterministic algorithms that provide guaranteed solutions under certain conditions, GAs may yield inconsistent outcomes depending on their configuration and the randomness of their operations. This lack of assurance can be a disadvantage in critical applications requiring high reliability [11].

Lastly, while GAs requires minimal initial information about the problem, this advantage is often offset by the challenges associated with designing the objective function and selecting appropriate parameters. These difficulties can make GAs less accessible to practitioners without specialized knowledge, thereby limiting their widespread adoption.

Suggestion for Improvement

One improvement that can be made to the solution is using numbers or shorter length symbols to represent the ‘products’ and ‘processes’. This will reduce the overhead when running the program and reduce run time. Encoding schemes in [1,2] can be explored and adapted to the specific use case.

Another potential improvement is modifying the evaluation function to favor schedules where the same products are assigned to machines continuously. For instance, a machine for processing ‘Assembly’ is assigned to produce many ‘Product 1’ without suddenly switching to another product. In a real-world scenario this will be helpful, since a machine may require set up or modification for every different product, which will certainly take time.

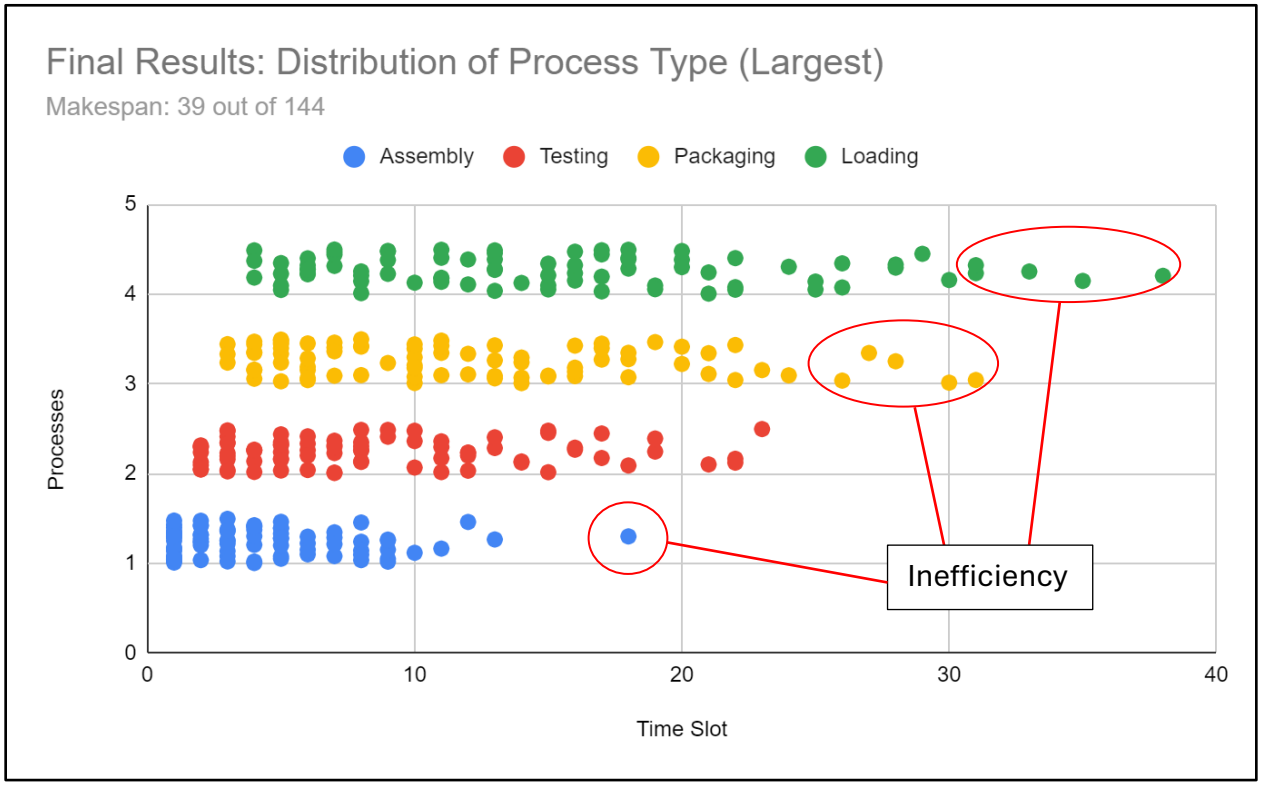


Figure 20: Results of 'Largest' problem [Crossover: 0.85; Mutation: 0.01; Population: 150; Generations: 10000; Tournament Size: 4], showing inefficiency in 'Packaging' and 'Loading' process. [Chart Link] [Data]

Figure 13 and Figure 20 indicate that there are inefficiencies in the solution. In Figure 13, there are instances where ‘Assembly’ machines that carry out the first process are not occupied. This will lead to delays in proceeding processes, increasing makespan. In Figure 20, there are trailing processes that are scheduled later than necessary, also leading to unwanted delays.

Alternative Optimization Techniques

While Genetic Algorithms (GAs) offer robust solutions for complex optimization problems like scheduling in manufacturing systems, alternative optimization techniques and improvements to GAs can provide significant performance enhancements. Below are several techniques, along with examples of their application:

1. Modified Genetic Algorithms

Standard GAs are also effective but adaptive cross and mutation rates which are included as an enhancement are able to be adjusted during execution. These changes ensure that the algorithm does not converge to sub-optimal parts of the solution space while hastening the search for the global optimum. For example, a genetic algorithm that is able to adapt the search area to refine the solution is demonstrated, making it better suited to job-shop scheduling problems [7].

2. Parallel Genetic Algorithms

In parallel GAs, the population is divided into multiple subpopulations, each processed independently. These subpopulations occasionally exchange individuals (migration), enabling a broader exploration of the solution space. A parallel genetic algorithm is developed to optimize scheduling by employing distinct populations and enhancing performance through elitist selection and crossover diversity [6]

3. Heuristic-Based Enhancements

Integrating heuristics into optimization algorithms can reduce computational overhead and improve solution quality. Priority-rule scheduling and neighbourhood search crossovers are examples of such heuristics. These techniques guide the GA by leveraging problem-specific

knowledge to restrict the search space and focus on promising regions. For example, heuristic-based crossovers are used to achieve efficient scheduling for job-shop problems [9].

4. Alternative Encoding Schemes

Encoding plays a critical role in the effectiveness of genetic algorithms. The operation-based encoding method, a direct representation of schedules, is commonly used but can be computationally intensive for large problems. As an alternative, the priority sequence-based encoding scheme represents operations as priorities [5]. This method simplifies decoding and ensures that precedence constraints are met while improving the search for optimal solutions.

5. Hybrid Optimization Techniques

Integrating GAs with alternative optimization strategies, which may include Simulated Annealing (SA) or Tabu Search, could result in better quality of the solution and quicker convergence time. These hybrid strategies leverage the advantages of each technique. For instance, the feature of Tabu Search to jump over local minima augments the searching tendency of GAs and produces good results in multi-objective optimization problems [5].

6. Simulated Annealing

Simulated Annealing is a probabilistic technique that mimics the cooling process of metals. It has been successfully applied to scheduling problems as it explores the solution space randomly but focuses increasingly on areas close to the global optimum as iterations progress. SA can also be combined with GAs to enhance its effectiveness by injecting diversity into the genetic population.

7. Swarm-Based Algorithms

Effective algorithms, including Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO) that employ ideas of collective behavior from nature, have emerged. These techniques excel in dynamic environments where adaptability and quick convergence are required. ACO, for instance, has shown success in solving scheduling problems by mimicking the way ants optimize paths to food [5].

By implementing these alternative optimization techniques and improvements, scheduling challenges in manufacturing and other industries can be addressed more efficiently. These

approaches not only enhance solution quality but also reduce computational costs, enabling practical applications in real-world scenarios.

References

[1] A. Ławrynowicz, “Genetic Algorithms for Solving Scheduling Problems in Manufacturing Systems,” *Foundations of Management*, vol. 3, no. 2, pp. 7–26, Jan. 2011, doi: https://doi.org/10.2478/v10238-012-0039-2.

[2] K. RITWIK and S. DEB, “A GENETIC ALGORITHM-BASED APPROACH FOR OPTIMIZATION OF SCHEDULING IN JOB SHOP ENVIRONMENT,” *Journal of Advanced Manufacturing Systems*, vol. 10, no. 02, pp. 223–240, Dec. 2011, doi: https://doi.org/10.1142/s0219686711002235.

[3] S. Katoch, S. S. Chauhan, and V. Kumar, “A review on genetic algorithm: past, present, and future,” *Multimedia Tools and Applications*, vol. 80, no. 5, Oct. 2020, doi: https://doi.org/10.1007/s11042-020-10139-6.

[4] A. Hassanat, K. Almohammadi, E. Alkafaween, E. Abunawas, A. Hammouri, and V. B. S. Prasath, “Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach,” *Information*, vol. 10, no. 12, p. 390, Dec. 2019, doi: https://doi.org/10.3390/info10120390.

[5] R. Kumar and S. Deb, "A genetic algorithm-based approach for optimization of scheduling in job shop environment," *Journal of Advanced Manufacturing Systems*, vol. 10, no. 2, pp. 223–240, 2011, doi: 10.1142/S0219686711002235.

[6] B. J. Park, H. R. Choi, and H. S. Kim, "A hybrid genetic algorithm for the job shop scheduling problem," *Computers & Industrial Engineering*, vol. 45, no. 4, pp. 597–604, 2003.

[7] M. Watanabe, K. Ida, and M. Gen, "A genetic algorithm with modified crossover operator and search area adaptation for the job-shop scheduling problem," *Computers & Industrial Engineering*, vol. 48, no. 4, pp. 743–748, 2005.

[9] T. Yamada and R. Nakano, "Genetic algorithms for job shop scheduling problems," in *Proceedings of Modern Heuristic for Decision Support*, 1997, pp. 68–77.

[10] *What is the Difference Between Genetic Algorithm and Traditional Algorithm - Pediaa.com*. (2019, March 26). Pediaa.com. https://pediaa.com/what-is-the-difference-between-genetic-algorithm-and-traditional-algorithm/

[11] *Genetic Algorithm in Machine Learning - Javatpoint*. (n.d.). Www.javatpoint.com. https://www.javatpoint.com/genetic-algorithm-in-machine-learning

[12] Bagane, P., & Kotrappa, S. (2021). COMPARISON BETWEEN TRADITIONAL CRYPTOGRAPHIC METHODS AND GENETIC ALGORITHM BASED METHOD TOWARDS CYBER SECURITY. In *International Journal of Advanced Research in Engineering and Technology (IJARET)* (Vol. 12, Issue 2, pp. 676–682) [Journal-article]. https://doi.org/10.34218/IJARET.12.2.2021.066

Appendix

Appendix 1

```
-- PRODUCT DETAILS --
1. Product 1
        Assembly: 2
        Testing: 1
        Packaging: 1
2. Product 2
        Assembly: 3
        Testing: 2
        Packaging: 1
3. Product 3
        Assembly: 1
        Testing: 2
        Packaging: 2

-- MACHINE TYPES --
1. Assembly: 7
2. Testing: 5
3. Packaging: 5

-- TIME SLOTS --
48 time slots. 10 mins each

-- SCHEDULE FORMAT --
[ product, process, machine_num, time_slot ]
```

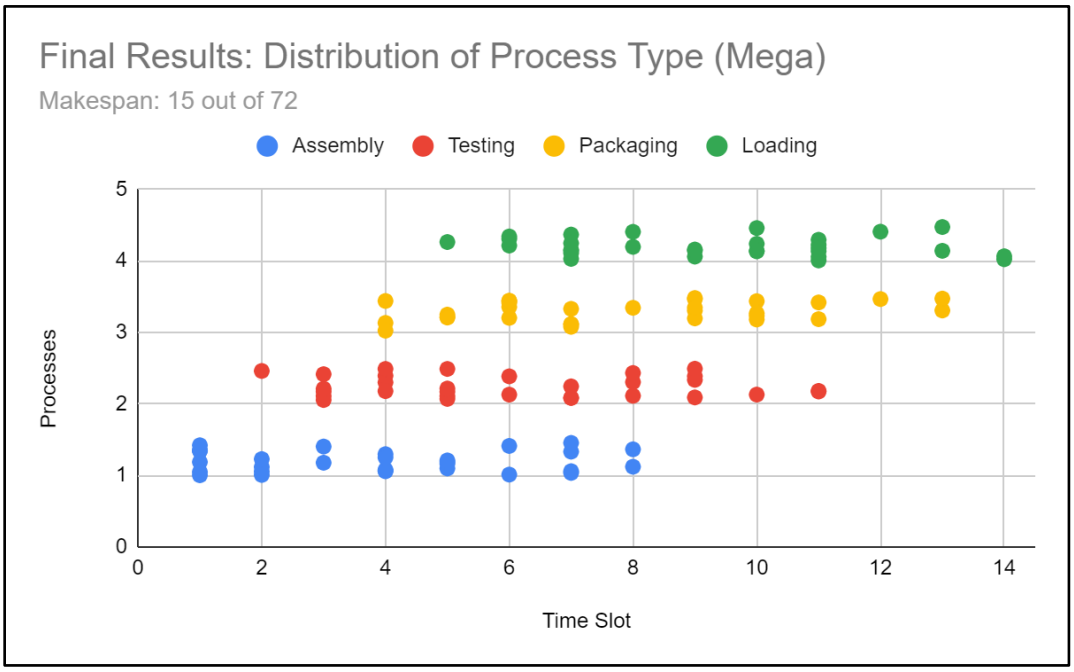
```
-- SCHEDULE --
[['Product 1' 'Assembly' '1' '1']
 ['Product 1' 'Assembly' '6' '1']
 ['Product 2' 'Assembly' '4' '1']
 ['Product 2' 'Assembly' '2' '1']
 ['Product 3' 'Assembly' '3' '1']
 ['Product 3' 'Assembly' '7' '1']
 ['Product 3' 'Assembly' '5' '1']
 ['Product 2' 'Assembly' '7' '2']
 ['Product 2' 'Assembly' '3' '2']
 ['Product 3' 'Assembly' '5' '2']
 ['Product 1' 'Assembly' '1' '3']
 ['Product 1' 'Testing' '3' '3']
 ['Product 3' 'Assembly' '5' '3']
 ['Product 3' 'Assembly' '6' '3']
 ['Product 3' 'Testing' '2' '3']
 ['Product 3' 'Testing' '1' '3']
 ['Product 3' 'Testing' '4' '3']
 ['Product 1' 'Assembly' '6' '4']
 ['Product 1' 'Packaging' '2' '4']
 ['Product 2' 'Assembly' '4' '4']
 ['Product 2' 'Assembly' '2' '4']
 ['Product 2' 'Assembly' '5' '4']
 ['Product 2' 'Testing' '3' '4']
 ['Product 3' 'Testing' '5' '4']
 ['Product 1' 'Assembly' '3' '5']
 ['Product 1' 'Assembly' '7' '5']
 ['Product 1' 'Testing' '2' '5']
 ['Product 2' 'Assembly' '1' '5']
 ['Product 3' 'Testing' '4' '5']
 ['Product 3' 'Testing' '1' '5']
 ['Product 3' 'Packaging' '5' '5']
 ['Product 3' 'Packaging' '3' '5']
 ['Product 3' 'Packaging' '1' '5']
 ['Product 1' 'Assembly' '6' '6']
 ['Product 1' 'Testing' '2' '6']
 ['Product 1' 'Testing' '5' '6']
 ['Product 1' 'Packaging' '4' '6']
 ['Product 2' 'Testing' '3' '6']
 ['Product 3' 'Packaging' '2' '6']
 ['Product 1' 'Assembly' '2' '7']
 ['Product 1' 'Assembly' '3' '7']
 ['Product 1' 'Testing' '1' '7']
 ['Product 1' 'Testing' '2' '7']
 ['Product 1' 'Packaging' '3' '7']
 ['Product 1' 'Packaging' '4' '7']
 ['Product 2' 'Assembly' '4' '7']
 ['Product 2' 'Testing' '5' '7']
 ['Product 2' 'Testing' '4' '7']
 ['Product 2' 'Packaging' '5' '7']
 ['Product 3' 'Assembly' '5' '7']
 ['Product 3' 'Assembly' '7' '7']
 ['Product 3' 'Packaging' '1' '7']
 ['Product 1' 'Assembly' '6' '8']
 ['Product 1' 'Testing' '2' '8']
 ['Product 1' 'Packaging' '3' '8']
 ['Product 1' 'Packaging' '4' '8']
 ['Product 2' 'Assembly' '1' '8']
 ['Product 2' 'Testing' '1' '8']
 ['Product 2' 'Packaging' '5' '8']
 ['Product 3' 'Assembly' '7' '8']
 ['Product 3' 'Assembly' '5' '8']
 ['Product 3' 'Testing' '3' '8']
 ['Product 1' 'Testing' '5' '9']
 ['Product 1' 'Packaging' '4' '9']
 ['Product 2' 'Testing' '4' '9']
 ['Product 2' 'Packaging' '5' '9']
 ['Product 2' 'Packaging' '2' '9']
 ['Product 3' 'Testing' '2' '9']
 ['Product 3' 'Packaging' '1' '9']
 ['Product 1' 'Packaging' '2' '10']
 ['Product 2' 'Testing' '3' '10']
 ['Product 2' 'Testing' '1' '10']
 ['Product 2' 'Packaging' '5' '10']
 ['Product 3' 'Testing' '5' '10']
 ['Product 3' 'Packaging' '3' '10']
 ['Product 1' 'Testing' '4' '11']
 ['Product 2' 'Packaging' '1' '11']
 ['Product 3' 'Testing' '2' '11']
 ['Product 3' 'Packaging' '2' '11']
 ['Product 1' 'Testing' '1' '12']
 ['Product 1' 'Packaging' '5' '12']
 ['Product 2' 'Testing' '3' '12']
 ['Product 2' 'Testing' '4' '12']
 ['Product 2' 'Packaging' '4' '12']
 ['Product 2' 'Packaging' '3' '12']
```

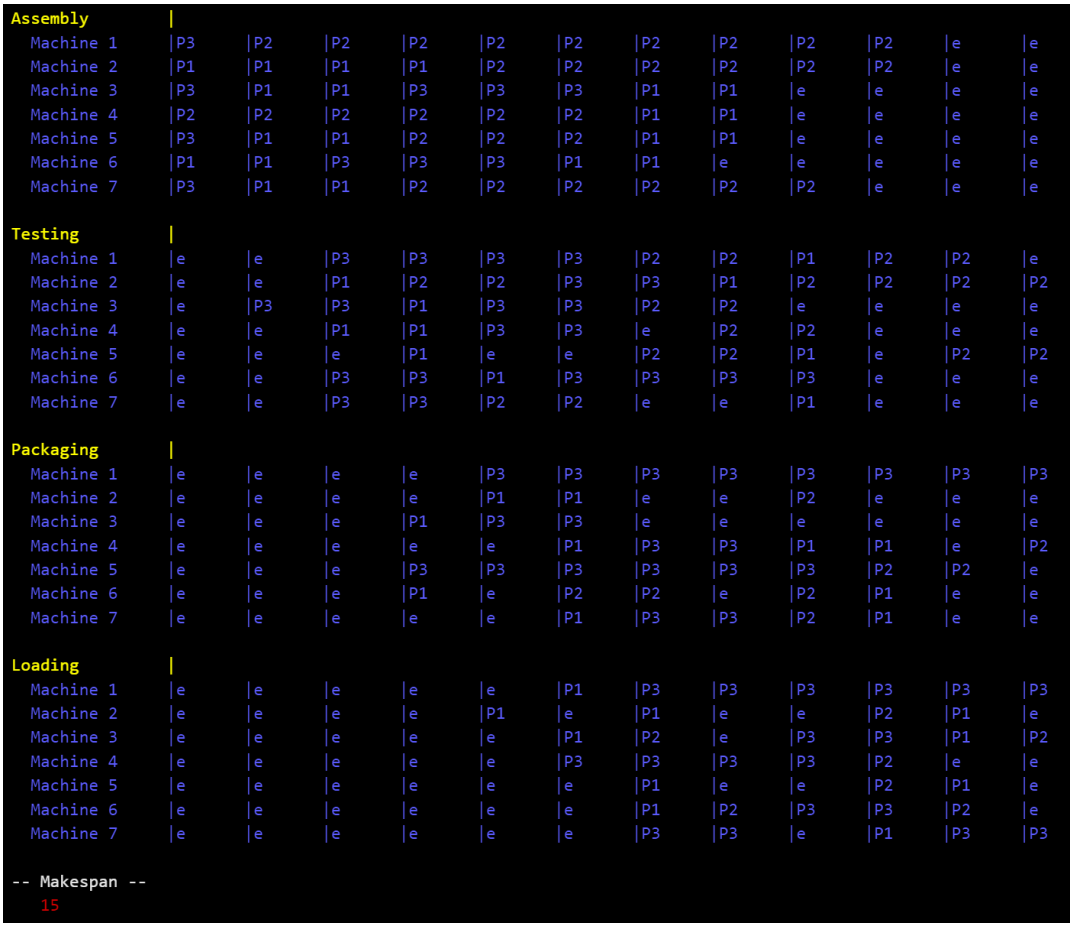
```
 ['Product 3' 'Packaging' '1' '12']
 ['Product 1' 'Packaging' '5' '13']
 ['Product 3' 'Packaging' '3' '13']
 ['Product 2' 'Packaging' '4' '14']
 ['Product 2' 'Packaging' '2' '14']]

-- NUMBER OF PRODUCT COMPLETED AT TIME --
TIME SLOT       |1      |2      |3      |4      |5      |6      |7      |8      |9      |10     |11     |12     |13     |14

Product 1       |
  Assembly      |0      |0      |2      |2      |3      |4      |6      |7      |9      |10     |10     |10     |10     |10
  Testing       |0      |0      |0      |1      |1      |2      |4      |6      |7      |8      |8      |9      |10     |10
  Packaging     |0      |0      |0      |0      |1      |1      |2      |4      |6      |7      |8      |8      |9      |10

Product 2       |
  Assembly      |0      |0      |0      |2      |4      |4      |7      |8      |8      |9      |10     |10     |10     |10
  Testing       |0      |0      |0      |0      |0      |1      |1      |2      |4      |5      |6      |8      |8      |10
  Packaging     |0      |0      |0      |0      |0      |0      |0      |1      |2      |4      |5      |6      |8      |8

Product 3       |
  Assembly      |0      |3      |4      |6      |6      |6      |6      |8      |10     |10     |10     |10     |10     |10
  Testing       |0      |0      |0      |0      |3      |4      |6      |6      |6      |7      |8      |9      |10     |10
  Packaging     |0      |0      |0      |0      |0      |0      |3      |4      |5      |5      |6      |7      |8      |9
```

```
Assembly        |
  Machine 1     |P1     |P1     |P1     |P1     |P2     |P2     |P2     |P2     |P2     |P2     |e      |e      |e      |e
  Machine 2     |P2     |P2     |P2     |P2     |P2     |P2     |P1     |P1     |e      |e      |e      |e      |e      |e
  Machine 3     |P3     |P2     |P2     |P2     |P1     |P1     |P1     |P1     |e      |e      |e      |e      |e      |e
  Machine 4     |P2     |P2     |P2     |P2     |P2     |P2     |P2     |P2     |P2     |e      |e      |e      |e      |e
  Machine 5     |P3     |P3     |P3     |P2     |P2     |P2     |P3     |P3     |e      |e      |e      |e      |e      |e
  Machine 6     |P1     |P1     |P3     |P1     |P1     |P1     |P1     |P1     |P1     |e      |e      |e      |e      |e
  Machine 7     |P3     |P2     |P2     |P2     |P1     |P1     |P3     |P3     |e      |e      |e      |e      |e      |e

Testing         |
  Machine 1     |e      |e      |P3     |P3     |P3     |P3     |P1     |P2     |P2     |P2     |P2     |P1     |e      |e
  Machine 2     |e      |e      |P3     |P3     |P1     |P1     |P1     |P1     |P3     |P3     |P3     |P3     |e      |e
  Machine 3     |e      |e      |P1     |P2     |P2     |P2     |P2     |P3     |P3     |P2     |P2     |P2     |P2     |e
  Machine 4     |e      |e      |P3     |P3     |P3     |P3     |P2     |P2     |P2     |P2     |P1     |P2     |P2     |e
  Machine 5     |e      |e      |e      |P3     |P3     |P1     |P2     |P2     |P1     |P3     |P3     |e      |e      |e

Packaging       |
  Machine 1     |e      |e      |e      |e      |P3     |P3     |P3     |P3     |P3     |P3     |P2     |P3     |P3     |e
  Machine 2     |e      |e      |e      |P1     |e      |P3     |P3     |e      |P2     |P1     |P3     |P3     |e      |P2
  Machine 3     |e      |e      |e      |e      |P3     |P3     |P1     |P1     |e      |P3     |P3     |P2     |P3     |P3
  Machine 4     |e      |e      |e      |e      |e      |P1     |P1     |P1     |P1     |e      |e      |P2     |e      |P2
  Machine 5     |e      |e      |e      |e      |P3     |P3     |P2     |P2     |P2     |P2     |e      |P1     |P1     |e

-- Makespan --
   15
```

Appendix 2

Schedule visualization of results for ‘Mega’ problem, [Crossover: 0.85; Mutation: 0.01; Population: 150; Generations: 10000; Tournament Size: 4] [Record]





Appendix 3

Initial tuning results, varying crossover rate, mutation rate, tournament size, and population size with fixed number of generations of 4000

Tournament Size	Population Size	Crossover Rate	Mutation Rate	Makespan
3	50	0.7	0.01	16
3	50	0.7	0.03	27
3	50	0.75	0.01	18
3	50	0.75	0.03	24
3	50	0.8	0.01	17
3	50	0.8	0.03	39
3	50	0.85	0.01	17
3	50	0.85	0.03	29
3	50	0.9	0.01	17

3	50	0.9	0.03	22
3	100	0.7	0.01	15
3	100	0.7	0.03	20
3	100	0.75	0.01	17
3	100	0.75	0.03	19
3	100	0.8	0.01	16
3	100	0.8	0.03	18
3	100	0.85	0.01	16
3	100	0.85	0.03	17
3	100	0.9	0.01	16
3	100	0.9	0.03	20
4	50	0.7	0.01	17
4	50	0.7	0.03	24
4	50	0.75	0.01	16
4	50	0.75	0.03	20
4	50	0.8	0.01	18
4	50	0.8	0.03	24
4	50	0.85	0.01	17
4	50	0.85	0.03	20
4	50	0.9	0.01	15
4	50	0.9	0.03	19
4	100	0.7	0.01	17
4	100	0.7	0.03	18
4	100	0.75	0.01	16
4	100	0.75	0.03	18
4	100	0.8	0.01	15
4	100	0.8	0.03	20
4	100	0.85	0.01	15
4	100	0.85	0.03	18
4	100	0.9	0.01	15

4	100	0.9	0.03	17

Appendix 4

Data of makespan contour map of crossover rate vs mutation rate

POPULATION_SIZE	CXPB	MU_INDPB	NGEN	TOURNAMENT_SIZE	MAKESPAN	EMPTY_MACHINE	PRODUCTS_WAITING
100	1	0.01	4000	4	16	214	38
100	1	0.03	4000	4	19	422	62
100	1	0.05	4000	4	23	625	73
100	1	0.07	4000	4	25	813	83
100	1	0.09	4000	4	21	866	73
100	1	0.11	4000	4	25	1109	92
100	1	0.13	4000	4	38	1335	98
100	1	0.15	4000	4	26	1345	106
100	1	0.17	4000	4	34	1543	87
100	1	0.19	4000	4	40	1689	131
100	1	0.21	4000	4	34	1712	146
100	1	0.23	4000	4	32	1664	133
100	1	0.25	4000	4	27	1417	119
100	0.95	0.01	4000	4	16	208	32
100	0.95	0.03	4000	4	17	345	49
100	0.95	0.05	4000	4	22	537	79
100	0.95	0.07	4000	4	28	1007	99
100	0.95	0.09	4000	4	26	879	72
100	0.95	0.11	4000	4	23	1004	104
100	0.95	0.13	4000	4	27	1515	109
100	0.95	0.15	4000	4	33	1536	126
100	0.95	0.17	4000	4	32	2276	146
100	0.95	0.19	4000	4	29	1239	116
100	0.95	0.21	4000	4	30	1637	108
100	0.95	0.23	4000	4	38	2019	126
100	0.95	0.25	4000	4	34	2003	161
100	0.9	0.01	4000	4	15	231	36
100	0.9	0.03	4000	4	17	294	52
100	0.9	0.05	4000	4	23	826	82
100	0.9	0.07	4000	4	24	917	89
100	0.9	0.09	4000	4	35	1194	100
100	0.9	0.11	4000	4	26	1208	118
100	0.9	0.13	4000	4	27	1203	105

100	0.9	0.15	4000	4	29	1437	126
100	0.9	0.17	4000	4	29	1453	105
100	0.9	0.19	4000	4	31	1767	132
100	0.9	0.21	4000	4	36	2001	144
100	0.9	0.23	4000	4	35	2098	152
100	0.9	0.25	4000	4	33	2196	158
100	0.85	0.01	4000	4	15	110	29
100	0.85	0.03	4000	4	18	478	54
100	0.85	0.05	4000	4	22	599	65
100	0.85	0.07	4000	4	25	863	87
100	0.85	0.09	4000	4	26	990	91
100	0.85	0.11	4000	4	27	1103	107
100	0.85	0.13	4000	4	28	1843	129
100	0.85	0.15	4000	4	26	1366	101
100	0.85	0.17	4000	4	43	2195	175
100	0.85	0.19	4000	4	29	1593	155
100	0.85	0.21	4000	4	34	2050	140
100	0.85	0.23	4000	4	33	1881	206
100	0.85	0.25	4000	4	29	1742	132
100	0.8	0.01	4000	4	15	199	43
100	0.8	0.03	4000	4	20	474	67
100	0.8	0.05	4000	4	22	771	79
100	0.8	0.07	4000	4	23	924	105
100	0.8	0.09	4000	4	29	1370	103
100	0.8	0.11	4000	4	34	1570	104
100	0.8	0.13	4000	4	30	1863	132
100	0.8	0.15	4000	4	34	1602	109
100	0.8	0.17	4000	4	27	1479	114
100	0.8	0.19	4000	4	33	2039	174
100	0.8	0.21	4000	4	32	1572	119
100	0.8	0.23	4000	4	36	2134	153
100	0.8	0.25	4000	4	35	1821	147
100	0.75	0.01	4000	4	16	197	36
100	0.75	0.03	4000	4	18	408	50
100	0.75	0.05	4000	4	22	801	70
100	0.75	0.07	4000	4	30	1204	107
100	0.75	0.09	4000	4	31	1671	104
100	0.75	0.11	4000	4	37	1658	116
100	0.75	0.13	4000	4	32	2429	187
100	0.75	0.15	4000	4	40	2023	115

100	0.75	0.17	4000	4	35	2164	176
100	0.75	0.19	4000	4	40	1893	152
100	0.75	0.21	4000	4	39	2541	142
100	0.75	0.23	4000	4	31	2157	154
100	0.75	0.25	4000	4	47	3014	171
100	0.7	0.01	4000	4	17	261	37
100	0.7	0.03	4000	4	18	374	52
100	0.7	0.05	4000	4	25	635	68
100	0.7	0.07	4000	4	25	936	84
100	0.7	0.09	4000	4	33	1393	137
100	0.7	0.11	4000	4	32	1361	108
100	0.7	0.13	4000	4	31	1556	103
100	0.7	0.15	4000	4	29	1550	141
100	0.7	0.17	4000	4	32	1745	125
100	0.7	0.19	4000	4	38	2439	126
100	0.7	0.21	4000	4	35	2456	189
100	0.7	0.23	4000	4	37	2683	173
100	0.7	0.25	4000	4	30	2521	158
100	0.65	0.01	4000	4	17	197	34
100	0.65	0.03	4000	4	20	386	53
100	0.65	0.05	4000	4	22	852	87
100	0.65	0.07	4000	4	26	1133	106
100	0.65	0.09	4000	4	30	1380	121
100	0.65	0.11	4000	4	41	1843	90
100	0.65	0.13	4000	4	33	1817	152
100	0.65	0.15	4000	4	41	2473	166
100	0.65	0.17	4000	4	39	2487	140
100	0.65	0.19	4000	4	43	2782	142
100	0.65	0.21	4000	4	38	2486	182
100	0.65	0.23	4000	4	34	2096	150
100	0.65	0.25	4000	4	40	2363	164
100	0.6	0.01	4000	4	16	170	40
100	0.6	0.03	4000	4	18	381	44
100	0.6	0.05	4000	4	21	571	80
100	0.6	0.07	4000	4	27	1042	122
100	0.6	0.09	4000	4	34	1330	107
100	0.6	0.11	4000	4	33	1654	159
100	0.6	0.13	4000	4	37	2421	156
100	0.6	0.15	4000	4	41	2119	164
100	0.6	0.17	4000	4	32	1356	105

100	0.6	0.19	4000	4	31	2105	136
100	0.6	0.21	4000	4	40	2807	190
100	0.6	0.23	4000	4	43	2554	149
100	0.6	0.25	4000	4	39	2690	203
100	0.55	0.01	4000	4	16	210	30
100	0.55	0.03	4000	4	18	396	57
100	0.55	0.05	4000	4	27	1307	127
100	0.55	0.07	4000	4	29	1476	122
100	0.55	0.09	4000	4	36	1735	163
100	0.55	0.11	4000	4	32	1791	116
100	0.55	0.13	4000	4	29	1818	161
100	0.55	0.15	4000	4	32	1824	133
100	0.55	0.17	4000	4	34	1903	142
100	0.55	0.19	4000	4	37	2269	152
100	0.55	0.21	4000	4	40	3071	193
100	0.55	0.23	4000	4	36	2855	125
100	0.55	0.25	4000	4	40	2851	198
100	0.5	0.01	4000	4	17	268	46
100	0.5	0.03	4000	4	19	364	46
100	0.5	0.05	4000	4	26	931	65
100	0.5	0.07	4000	4	33	1397	115
100	0.5	0.09	4000	4	41	1686	128
100	0.5	0.11	4000	4	32	1744	160
100	0.5	0.13	4000	4	40	1668	126
100	0.5	0.15	4000	4	36	2399	219
100	0.5	0.17	4000	4	37	2802	202
100	0.5	0.19	4000	4	42	2231	165
100	0.5	0.21	4000	4	35	2932	173
100	0.5	0.23	4000	4	41	3116	181
100	0.5	0.25	4000	4	37	2713	152
100	0.45	0.01	4000	4	17	323	42
100	0.45	0.03	4000	4	21	436	56
100	0.45	0.05	4000	4	29	1148	103
100	0.45	0.07	4000	4	36	1914	142
100	0.45	0.09	4000	4	31	1710	128
100	0.45	0.11	4000	4	39	1836	115
100	0.45	0.13	4000	4	37	1825	158
100	0.45	0.15	4000	4	39	2611	227
100	0.45	0.17	4000	4	34	2410	191
100	0.45	0.19	4000	4	39	2463	174

100	0.45	0.21	4000	4	41	2663	128
100	0.45	0.23	4000	4	43	2868	224
100	0.45	0.25	4000	4	43	3290	198
100	0.4	0.01	4000	4	16	126	32
100	0.4	0.03	4000	4	20	448	58
100	0.4	0.05	4000	4	25	930	102
100	0.4	0.07	4000	4	30	1429	111
100	0.4	0.09	4000	4	33	1796	156
100	0.4	0.11	4000	4	42	2054	147
100	0.4	0.13	4000	4	35	2276	142
100	0.4	0.15	4000	4	34	2735	201
100	0.4	0.17	4000	4	37	2583	186
100	0.4	0.19	4000	4	38	3495	233
100	0.4	0.21	4000	4	39	2687	185
100	0.4	0.23	4000	4	10044	3716	220
100	0.4	0.25	4000	4	46	3447	195
100	0.35	0.01	4000	4	16	160	35
100	0.35	0.03	4000	4	25	653	70
100	0.35	0.05	4000	4	25	913	86
100	0.35	0.07	4000	4	31	1855	136
100	0.35	0.09	4000	4	34	1767	146
100	0.35	0.11	4000	4	32	2201	156
100	0.35	0.13	4000	4	40	2626	153
100	0.35	0.15	4000	4	40	3086	219
100	0.35	0.17	4000	4	40	2389	180
100	0.35	0.19	4000	4	36	3025	207
100	0.35	0.21	4000	4	42	3428	225
100	0.35	0.23	4000	4	42	3464	218
100	0.35	0.25	4000	4	43	3828	267
100	0.3	0.01	4000	4	17	314	47
100	0.3	0.03	4000	4	21	592	69
100	0.3	0.05	4000	4	25	885	81
100	0.3	0.07	4000	4	31	1929	137
100	0.3	0.09	4000	4	39	3178	176
100	0.3	0.11	4000	4	44	2921	159
100	0.3	0.13	4000	4	47	3973	232
100	0.3	0.15	4000	4	44	3081	228
100	0.3	0.17	4000	4	38	2787	215
100	0.3	0.19	4000	4	37	3307	237
100	0.3	0.21	4000	4	40	3780	285

100	0.3	0.23	4000	4	41	3967	249
100	0.3	0.25	4000	4	41	4183	203
100	0.25	0.01	4000	4	15	169	40
100	0.25	0.03	4000	4	20	410	63
100	0.25	0.05	4000	4	28	1448	126
100	0.25	0.07	4000	4	33	1911	143
100	0.25	0.09	4000	4	40	2702	208
100	0.25	0.11	4000	4	39	2632	161
100	0.25	0.13	4000	4	39	3471	234
100	0.25	0.15	4000	4	43	3709	219
100	0.25	0.17	4000	4	44	3672	202
100	0.25	0.19	4000	4	45	3306	228
100	0.25	0.21	4000	4	47	3535	228
100	0.25	0.23	4000	4	43	4124	227
100	0.25	0.25	4000	4	40	4205	233
100	0.2	0.01	4000	4	17	294	42
100	0.2	0.03	4000	4	26	631	80
100	0.2	0.05	4000	4	34	1377	108
100	0.2	0.07	4000	4	33	1644	142
100	0.2	0.09	4000	4	42	2827	186
100	0.2	0.11	4000	4	48	4819	358
100	0.2	0.13	4000	4	41	3333	226
100	0.2	0.15	4000	4	43	3548	274
100	0.2	0.17	4000	4	47	4347	265
100	0.2	0.19	4000	4	44	3440	197
100	0.2	0.21	4000	4	47	4265	266
100	0.2	0.23	4000	4	42	3992	237
100	0.2	0.25	4000	4	45	4499	257
100	0.15	0.01	4000	4	16	235	38
100	0.15	0.03	4000	4	20	548	89
100	0.15	0.05	4000	4	28	1451	127
100	0.15	0.07	4000	4	35	2125	165
100	0.15	0.09	4000	4	39	3557	216
100	0.15	0.11	4000	4	45	4922	307
100	0.15	0.13	4000	4	40	3702	203
100	0.15	0.15	4000	4	46	4094	279
100	0.15	0.17	4000	4	48	3722	190
100	0.15	0.19	4000	4	46	4007	271
100	0.15	0.21	4000	4	47	4611	264
100	0.15	0.23	4000	4	49	5733	300

100	0.15	0.25	4000	4	10049	4810	259
100	0.1	0.01	4000	4	17	161	33
100	0.1	0.03	4000	4	20	437	49
100	0.1	0.05	4000	4	28	1425	98
100	0.1	0.07	4000	4	35	2578	187
100	0.1	0.09	4000	4	39	3407	179
100	0.1	0.11	4000	4	42	4432	226
100	0.1	0.13	4000	4	45	4905	293
100	0.1	0.15	4000	4	49	4558	280
100	0.1	0.17	4000	4	45	4866	300
100	0.1	0.19	4000	4	47	5720	318
100	0.1	0.21	4000	4	10048	5330	337
100	0.1	0.23	4000	4	10047	5295	215
100	0.1	0.25	4000	4	48	5304	355

Appendix 5

Data of makespan contour map of population vs tournament size

POPULATION_SIZE	CXPB	MU_INDPB	NGEN	TOURNAMENT_SIZE	MAKESPAN	EMPTY_MACHINE	PRODUCTS_WAITING
50	0.85	0.01	4000	1	110048	6982	473
50	0.85	0.01	4000	2	18	644	61
50	0.85	0.01	4000	3	17	279	44
50	0.85	0.01	4000	4	17	283	36
50	0.85	0.01	4000	5	15	187	35
50	0.85	0.01	4000	6	16	211	42
50	0.85	0.01	4000	7	16	159	34
50	0.85	0.01	4000	8	17	300	43
50	0.85	0.01	4000	9	17	256	39
50	0.85	0.01	4000	10	17	311	41
100	0.85	0.01	4000	1	120048	7174	388
100	0.85	0.01	4000	2	18	471	55
100	0.85	0.01	4000	3	16	193	30
100	0.85	0.01	4000	4	15	110	29
100	0.85	0.01	4000	5	15	147	34
100	0.85	0.01	4000	6	16	231	34
100	0.85	0.01	4000	7	15	158	31
100	0.85	0.01	4000	8	16	178	31
100	0.85	0.01	4000	9	15	136	25
100	0.85	0.01	4000	10	16	307	39
150	0.85	0.01	4000	1	120049	8989	292
150	0.85	0.01	4000	2	18	471	62
150	0.85	0.01	4000	3	16	172	40
150	0.85	0.01	4000	4	16	166	38
150	0.85	0.01	4000	5	16	256	47
150	0.85	0.01	4000	6	16	214	47
150	0.85	0.01	4000	7	16	169	29
150	0.85	0.01	4000	8	16	161	37
150	0.85	0.01	4000	9	14	120	35
150	0.85	0.01	4000	10	15	112	29
200	0.85	0.01	4000	1	110049	8624	327
200	0.85	0.01	4000	2	16	380	64
200	0.85	0.01	4000	3	15	116	34
200	0.85	0.01	4000	4	15	140	32
200	0.85	0.01	4000	5	15	118	24
200	0.85	0.01	4000	6	15	197	41

200	0.85	0.01	4000	7	15	109	29
200	0.85	0.01	4000	8	15	193	34
200	0.85	0.01	4000	9	15	113	35
200	0.85	0.01	4000	10	15	199	40
250	0.85	0.01	4000	1	70049	8264	282
250	0.85	0.01	4000	2	17	212	46
250	0.85	0.01	4000	3	16	195	29
250	0.85	0.01	4000	4	16	141	32
250	0.85	0.01	4000	5	15	127	32
250	0.85	0.01	4000	6	16	157	34
250	0.85	0.01	4000	7	15	182	31
250	0.85	0.01	4000	8	16	225	37
250	0.85	0.01	4000	9	15	164	30
250	0.85	0.01	4000	10	16	364	53
300	0.85	0.01	4000	1	110049	8523	324
300	0.85	0.01	4000	2	15	236	34
300	0.85	0.01	4000	3	18	127	31
300	0.85	0.01	4000	4	17	285	42
300	0.85	0.01	4000	5	15	237	34
300	0.85	0.01	4000	6	16	237	38
300	0.85	0.01	4000	7	16	361	49
300	0.85	0.01	4000	8	15	199	45
300	0.85	0.01	4000	9	15	191	35
300	0.85	0.01	4000	10	14	105	24
350	0.85	0.01	4000	1	110049	6467	417
350	0.85	0.01	4000	2	16	185	31
350	0.85	0.01	4000	3	14	86	25
350	0.85	0.01	4000	4	16	224	37
350	0.85	0.01	4000	5	15	119	24
350	0.85	0.01	4000	6	15	162	28
350	0.85	0.01	4000	7	15	88	26
350	0.85	0.01	4000	8	16	166	35
350	0.85	0.01	4000	9	16	273	42
350	0.85	0.01	4000	10	15	83	27
400	0.85	0.01	4000	1	90049	8051	290
400	0.85	0.01	4000	2	16	220	48
400	0.85	0.01	4000	3	15	175	47
400	0.85	0.01	4000	4	15	93	25
400	0.85	0.01	4000	5	14	135	32
400	0.85	0.01	4000	6	15	100	25

400	0.85	0.01	4000	7	15	112	34
400	0.85	0.01	4000	8	16	230	44
400	0.85	0.01	4000	9	16	233	40
400	0.85	0.01	4000	10	15	134	38
450	0.85	0.01	4000	1	100047	6921	323
450	0.85	0.01	4000	2	16	234	42
450	0.85	0.01	4000	3	16	131	30
450	0.85	0.01	4000	4	16	257	42
450	0.85	0.01	4000	5	15	129	32
450	0.85	0.01	4000	6	15	223	33
450	0.85	0.01	4000	7	15	159	32
450	0.85	0.01	4000	8	14	160	36
450	0.85	0.01	4000	9	15	120	33
450	0.85	0.01	4000	10	19	305	45
500	0.85	0.01	4000	1	100048	6582	453
500	0.85	0.01	4000	2	15	192	47
500	0.85	0.01	4000	3	16	121	30
500	0.85	0.01	4000	4	14	142	41
500	0.85	0.01	4000	5	16	287	43
500	0.85	0.01	4000	6	15	142	36
500	0.85	0.01	4000	7	16	154	33
500	0.85	0.01	4000	8	14	85	18
500	0.85	0.01	4000	9	15	172	29
500	0.85	0.01	4000	10	17	288	32