Solving the U-Shaped Assembly Line Balancing Problem Type-II Using a Genetic Algorithm

Project Final Report

EMU427 – Heuristic Methods for Optimization

Department of Industrial Engineering
Hacettepe University

Students

Firuze İpek Yıldırım	2220469028
Mustafa Alp Ulaş	2210469034
Beril Yıldız	2230469107
Pınar Ece Pank	2240469085
Tarık Buğra Birinci	2210469046

Instructor: Prof. Çağrı Koç

December 2025
Ankara, Türkiye

Contents

1 Introduction

Assembly line balancing is a fundamental optimization problem in industrial engineering. It aims to distribute tasks across workstations by trying to maximize productivity and minimize idle time. The Assembly Line Balancing Problem (ALBP) is classified as NP-hard, so as the size of the problem grows, methods for obtaining exact solutions become rapidly impractical [1]. As a result, heuristic and metaheuristic approaches have been developed and adapted to obtain high-quality results with reasonable computational effort. U-shaped assembly lines have significant popularity for being an effective alternative to traditional straight lines in production. They allow workers to work two sides of the line, so it reduces distance and eases multi-tasking.[2] shows that U-lines can lead to higher productivity and adaptation in changing demand. Type-2 assembly line balancing can be preferred when the number of workstations is fixed. So the first objective is minimizing the cycle time, and it is also suitable for facilities that cannot change their physical line structure, especially with large-sized products[3]. Since the possibilities of assembly line problems get exponentially larger because of their combinatorial structure and growing search area, deterministic optimization approaches remain limited and become computationally infeasible for medium to large scale problems. As a consequence, Genetic Algorithms (GAs) have become an alternative because of their stochastic search methods. In our project, numerous studies we researched mostly highlight that GAs can generate well-performing solutions without the problem needing to be convex or linear as in classical LP problems, so this makes them appropriate for NP-hard manufacturing problems. The essential consideration for U-shaped assembly lines is that the encoding part has to represent bidirectional task assignments and preserve precedence relationships in the evolutionary process. This project aims to develop and analyze a GA-based approach specifically for Type-2 U-shaped balancing. Even though various metaheuristic approaches are used in classical assembly line balancing problems and show very strong performance in different combinations of the classical assembly line problem, there are a limited number of studies that specifically work on the performance of these algorithms and their applicability for U-shaped Type-2 line balancing. There is a limited number of encoding schemes that represent forward and back task assignments and preserve precedence constraints, while effectively minimizing cycle time in a fixed number of workstations specifically for this problem.

2 Problem Formulation - Detailed Problem Description (UALBP-2)

2.1 Introduction to Assembly Line Balancing

Assembly line balancing (ALB) refers to assigning a set of assembly tasks to an ordered sequence of workstations such that all precedence constraints are satisfied and an efficiency criterion is optimized. Each task has a processing time, and some tasks must be completed before others (precedence relations). In straight-line assembly systems, workstations are arranged linearly, and a task can only be assigned after all of its predecessors are allocated to earlier stations.

Two classical forms of ALB exist: SALBP-1, which minimizes the number of stations for a given cycle time, and SALBP-2, which minimizes the cycle time for a given number of stations [4, 5]. Both are NP-hard combinatorial problems [6].

2.2 Characteristics of U-Shaped Assembly Lines

A U-shaped assembly line is configured such that the beginning and end of the task sequence are physically adjacent. In this layout, a worker can operate on both sides of the U, giving access to tasks from the front (forward direction) or the back (reverse direction) of the precedence graph.

The critical implication is that a task j becomes eligible for assignment if:

- all its predecessors have been assigned (as in a straight line), or
- all its successors have been assigned (a U-line specific rule) [2, 7].

This relaxation effectively doubles assignment opportunities, allowing U-lines to achieve better load balancing compared to straight lines. Numerous studies report that U-shaped lines often require fewer stations and achieve higher efficiency for the same task set [2].

2.3 UALBP-2 Definition

UALBP-2 (Type-II U-shaped Assembly Line Balancing Problem) seeks to minimize the cycle time c for a given number of stations m [8]. The input consists of:

1. a set of n tasks with processing times t_i,
2. a precedence graph represented as a DAG,
3. a fixed number of stations m.

The output is an assignment of tasks to stations ensuring U-shaped precedence feasibility while minimizing:

$$c = \max_{j=1,\ldots,m} \left(\sum_{i \in S_j} t_i \right),$$

where S_j denotes the set of tasks assigned to station j.

2.4 Assumptions and Constraints

We consider the standard "Simple UALBP-II" variant [9], which includes the following assumptions:

- Single-model production: Only one product type is assembled.
- Deterministic task times: Processing times t_i are fixed and known.
- Single-sided work: Each worker operates on the inner side of the U.
- Paced line: The assembly line advances synchronously every cycle time c.
- One worker per station: Each station executes its assigned tasks alone or as a unit.
- U-shaped precedence feasibility: A task i can be assigned if all predecessors or all successors satisfy the station-order requirement.
- Fixed number of stations: m is given and constant.

Under these assumptions, UALBP-2 becomes a partitioning problem: partition n tasks into m ordered groups while minimizing the maximum station load.

2.5 Computational Complexity

UALBP-2 is computationally challenging. Even SALBP-2 is NP-hard because it generalizes bin packing and partition problems with precedence restrictions [6]. UALBP-2 expands the search space due to bidirectional task eligibility.

Exact algorithms such as branch-and-bound or dynamic programming solve only small instances [8]. Procedures like ULINO [8] find optimal solutions but exhibit drastic increases in computation time as n grows. Due to this difficulty and the scarcity of exact solutions for larger problem sizes, heuristic and metaheuristic methods such as genetic algorithms are widely recommended.

2.6 Importance of Solving UALBP-2

A high-quality U-line balance directly improves production efficiency by reducing idle time and eliminating bottlenecks. Solving UALBP-2 provides actionable insights for:

- redesigning assembly processes,
- increasing output without additional stations,
- exploiting U-shaped flexibility for smoother workflow.

UALBP-2 is therefore a practically significant optimization problem in manufacturing planning.

2.7 MILP Formulation for UALBP-2

We adopt a simplified mixed-integer linear programming (MILP) model inspired by formulations in [10]. The objective is to assign tasks to m stations while satisfying U-shaped precedence and minimizing cycle time.

Sets and Parameters

- $i = 1, \ldots, n$: tasks
- $j = 1, \ldots, m$: workstations
- $\mathcal{P}$: set of precedence pairs (p, q)
- $Pred(i)$: set of predecessors of task i
- $Succ(i)$: set of successors of task i
- t_i: processing time of task i
- M: sufficiently large constant ($M \geq m$)

Decision Variables

- $X_{ij} \in \{0, 1\}$: equals 1 if task i is assigned to station j
- $y_i \in \{0, 1\}$: 0 for forward feasibility, 1 for backward feasibility
- $S_i \in \{1, \ldots, m\}$: station index of task i
- $c \geq 0$: cycle time (to be minimized)

Objective Function

$$\min \ c \tag{1}$$

Constraints

1. Task Assignment Each task must be assigned to exactly one station:

$$\sum_{j=1}^{m} X_{ij} = 1 \qquad \forall i \tag{2}$$

2. Station Index Definition The station index of each task is defined as:

$$S_i = \sum_{j=1}^{m} j X_{ij} \qquad \forall i \tag{3}$$

3. Cycle Time Constraint The total processing time at each station cannot exceed the cycle time:

$$\sum_{i=1}^{n} t_i X_{ij} \leq c \qquad \forall j \tag{4}$$

4. U-Shaped Precedence Constraints *Forward feasibility* for each predecessor $p \in Pred(q)$:

$$S_p \leq S_q + M y_q \tag{4a}$$

Backward feasibility for each successor $s \in Succ(q)$:

$$S_s \leq S_q + M(1 - y_q) \tag{4b}$$

These constraints ensure that task q is feasible in either the forward or backward direction.

5. Variable Domains

$$X_{ij},\, y_i \in \{0, 1\}, \quad S_i \in \{1, \ldots, m\}, \quad c \geq 0 \tag{5}$$

3 Description of the Genetic Algorithm

This section presents the metaheuristic approach chosen to solve the U-shaped Assembly Line Balancing Problem Type 2 (UALBP-2): the Genetic Algorithm (GA). Genetic Algorithm is an evolutionary optimization method that takes its inspiration from the concepts of natural selection and genetics. It is particularly suitable for problems in the field of combinatorial optimization where the search space is enormous and discrete.
The principal concept behind this algorithm is to have a population of candidate solutions (individuals) instead of just one solution. An iterative process, which is similar to evolution, undergoes selection, crossover (recombination), and mutation operations. This, in turn, helps the search to not only simultaneously explore different regions of the solution space but also to realize the benefits of good-quality partial solutions. In the UALBP-2 case, the algorithm works on producing a sequence of tasks (permutation), which is then decoded to find out the minimum feasible cycle time for a certain number of stations (m).

3.1 The Algorithm

The framework of our Genetic Algorithm is outlined in algorithm 1. The core component of the algorithm is the representation of the problem. We use a permutation-based representation, where an individual is a topological sort of the tasks. The algorithm begins by initializing a random population of feasible task sequences. In every generation, the fitness of each individual is evaluated. Since we are solving UALBP-2 (minimizing cycle time c for fixed m), the evaluation function involves a sub-procedure that calculates the minimum possible cycle time for a given sequence. To generate new offspring, we employ Tournament Selection to choose parents. These parents undergo POX (Precedence Operation Crossover) to produce a child that inherits structural characteristics from both parents. To prevent premature convergence, Swap Mutation is applied with a low probability. Finally, because random genetic operations may violate precedence constraints, a Repair Mechanism is applied to ensure the child remains a valid topological sort.

3.2 Implementation

The algorithm 1 has been developed using the programming language Python due to its excellent high-level data structure handling ability.

3.2.1 Constructive Heuristic

When tackling the Type-2 U-Shaped Assembly Line Balancing Problem (UALBP-2), where the aim is to reduce the cycle time for a predetermined number of stations, a major difficulty is that of making the sequence feasible. One of the solutions proposed

Algorithm 1 Genetic Algorithm for UALBP-2

Require: Task times, Precedence constraints, Number of stations (m), Parameters ($PopSize, MaxGen, P_c, P_m$)
Ensure: Best Found Permutation and Cycle Time

$P \leftarrow$ InitializePopulation($PopSize$)
$BestSol \leftarrow \emptyset$
For $gen \leftarrow 1$ **to** $MaxGen$ **do**
    For all individual $I \in P$ **do**
        $Fitness(I) \leftarrow$ EvaluateCycleTime(I, m)
    End For
    $P_{new} \leftarrow \emptyset$
    While $|P_{new}| < PopSize$ **do**
        $Parent1, Parent2 \leftarrow$ TournamentSelection($P, k = 2$)
        If RANDOM $< P_c$ **then**
            $Child \leftarrow$ POXCrossover($Parent1, Parent2$)
        Else
            $Child \leftarrow Parent1$
        End If
        If RANDOM $< P_m$ **then**
            $Child \leftarrow$ SwapMutation($Child$)
        End If
        $Child \leftarrow$ RepairToTopological($Child$)
        Add $Child$ to P_{new}
    End While
    $P \leftarrow P_{new}$
    Update $BestSol$
End For
Return $BestSol$

is to implement topological sorting-based constructive heuristic within the Genetic Algorithm, which draws upon the dependency graph to produce a precedence-feasible list of tasks. This topological ordering not only guides the decoding phase with its precedence constraints but also facilitates the search for the best cycle time by allowing task assignment at either end of the stations.

3.2.2 Chromosome Representation

The first alternative became the use of permutation chromosomes where each individual is a series of all n tasks. The order of the genes from left to right indicates the priority that will be applied in the decoding process. This encoding has become the most popular one in assembly-line balancing because it allows the genetic operators to perform the whole task order manipulation without resulting in a non-valid permutation.[11]. Nevertheless, a chromosome in UALBP-2 does not associate with the station assignment directly like in straight-line SALBP-2. On the contrary, the permutation gives an ordered list that the U-line decoding algorithm uses to create a feasible solution. The reason is that U-shaped lines allow the assignments to be done in both directions, thus the decoding phase evaluates the permutation in a manner that shows this unique bidirectional feasibility.[11].

3.2.3 U-Shaped Decoding

Given a task permutation, the decoding procedure assigns tasks to stations progressively from station 1 to station m. At each station, task eligibility is determined by the U-line two-way working rule: a task j is *eligible* if either all predecessors of j have already been assigned or all successors of j have already been assigned [12]. This rule reflects the fact that, in a U-shaped layout, tasks can be performed from both the forward and the backward directions.

Among the currently eligible tasks, those appearing earliest in the chromosome are considered first and are assigned to the current station as long as the cumulative station load does not exceed the current cycle-time estimate. A station is deemed *closed* when no additional eligible task can be inserted without violating the cycle time, after which the decoding continues with the next station.

To evaluate a permutation efficiently, the decoding method incorporates a binary search to determine the smallest cycle time that yields a feasible assignment for the given permutation. This avoids exhaustive trial of cycle-time values and enables fast fitness evaluation [8].

A task j is eligible if

$$\text{Pred}(j) \subseteq A \quad \text{or} \quad \text{Succ}(j) \subseteq A,$$

where A is the set of already assigned tasks, and $\text{Pred}(j)$ and $\text{Succ}(j)$ denote the prede-

cessor and successor sets of task j, respectively.

3.2.4 Fitness Function

Taking into consideration that UALBP-2 is a Type-II problem, the aim is to minimize the cycle time c while the number of stations m is given. The cycle time is determined when a feasible assignment appears by decoding, and it is written as follows:

$$c = \max_{k=1,\dots,m} \sum_{j \in S_k} t_j, \tag{6}$$

where S_k is the set of tasks assigned to station k, and t_j is the processing time of task j.

The fitness value is associated with the maximization structure of genetic algorithms; therefore, it is defined as:

$$\text{Fitness} = \frac{1}{c}. \tag{7}$$

Therefore, solutions with smaller cycle times are awarded higher fitness values. On the other hand, if a permutation does not lead to any feasible assignment for any cycle time (i.e., decoding fails), then its fitness is set to 0, penalizing such individuals as infeasible. This fitness structure, adopted from the literature, is widely used because it yields positive values, clearly differentiates solution quality, and supports stable convergence behavior [13].

3.2.5 Parent Selection: Tournament Selection ($k = 2$)

The selection of parents is done by using the tournament selection method of size 2.This has been found efficient for combinatorial optimization problems where the fitness landscape includes many plateaus. The strategy of selection by tournaments is regarded as highly effective for combinatorial optimization problems that have tough fitness landscapes with many plateaus. Tournament selection not only preserves genetic diversity but also avoids a common problem seen in roulette-wheel selection, which is the sensitivity to scaling. [14]

3.2.6 Crossover Operator: Precedence-Preserving Order Crossover (POX-like)

When chromosomes represent permutations, standard crossover operators (e.g., one-point or two-point crossover) may easily introduce infeasibility and duplicated tasks. Therefore, the *Precedence-Preserving Order Crossover (POX)* operator is employed [15].

The POX operator proceeds as follows:

1. A random subset of tasks is selected.

2. The offspring chromosome is initialized as an empty sequence.

3. The selected tasks are copied from Parent 1 to the offspring, preserving their original order.

4. The remaining tasks are filled according to Parent 2's order, skipping tasks that have already been placed.

This procedure preserves major subsequences from both parents and typically reduces the need for extensive repair operations. POX is particularly suitable for UALBP-2 because tight precedence constraints make it crucial to keep coherent task blocks intact.

3.2.7 Mutation Operator: Swap Mutation

Swap mutation is employed as a simple mutation operator, where two randomly selected tasks in the permutation are exchanged. This operator introduces small but effective perturbations, preserving most of the chromosome structure while preventing premature stagnation of the population. Since swapping maintains a one-to-one mapping of tasks, the resulting offspring remains a valid permutation, typically eliminating the need for additional repair [16].

In permutation-based genetic algorithms, mutation is generally applied with a low probability (e.g., 0.05–0.10) to balance exploration and stability [17].

3.2.8 Repair Mechanism

After crossover and mutation, a lightweight repair operator is applied to every permutation. If a task is found to violate precedence by appearing before any of its predecessors, it is gradually shifted to the right until the violation is removed. This procedure restores a valid topological order while preserving the original permutation structure as much as possible.

3.2.9 Termination Criteria

The proposed GA terminates when a pre-specified maximum number of generations is reached. In this study, the generation limit is set to 300, which is consistent with common practice in the assembly line balancing literature. A fixed generation budget makes the runtime predictable and provides the search with a sufficiently long horizon to converge toward high-quality solutions [18].

4 Results

In this part, the paper showcases the outcomes derived from the Genetic Algorithm (GA) that was utilized for solving the ARC83 U-shaped Assembly Line Balancing Problem Type II (UALBP-2) benchmark instance. The results are based on several CSV outputs generated by the implementation, including global summaries, run-by-run statistics, detailed descriptions of the best solution, Top-10 individuals of the best run, and parameter tuning logs.

4.1 Global Performance Summary

The final parameter settings, which were in use during all the experiments, are written below as per GA_summary_results.csv:

- Population size: 40
- Generations: 300
- Crossover rate: 0.7
- Mutation rate: 0.1
- Selection method: Tournament (k = 2)
- Crossover method: POX (precedence-preserving)
- Mutation method: Swap
- Number of independent runs: 10

4.2 Run-by-Run Analysis

Table 1: Cycle Times Obtained from 10 Independent GA Runs

Run	Cycle Time
1	6498
2	6476
3	6467
4	6544
5	6529
6	6510
7	6562
8	6522
9	6514
10	6562
Best	6467
Mean	6518.4
Worst	6562

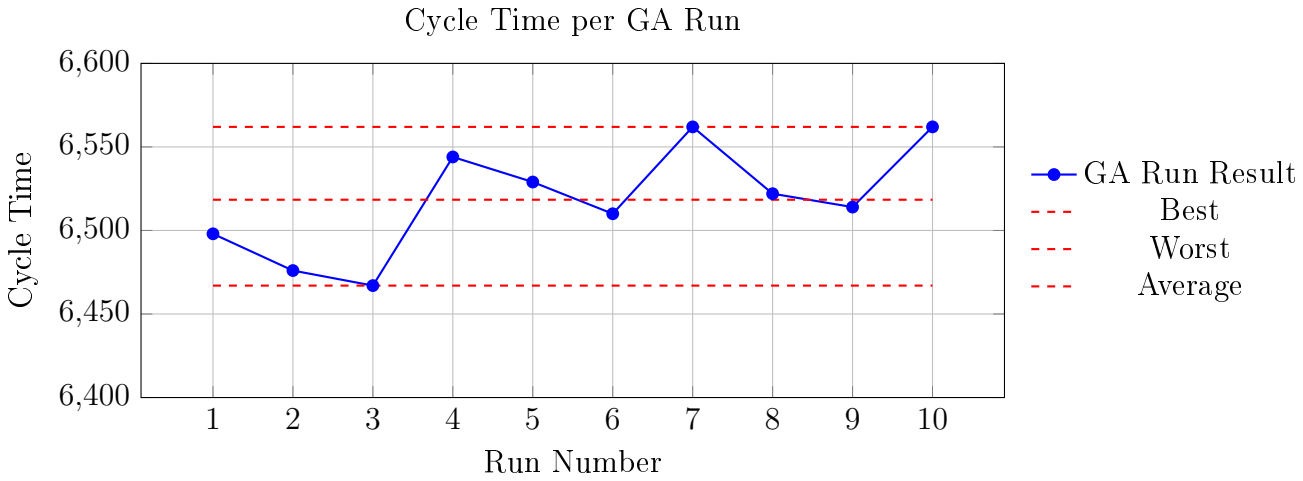

Figure 1: Instance ARC83 cycle time obtained in each of the 10 independent GA runs, including best, worst, and average values.

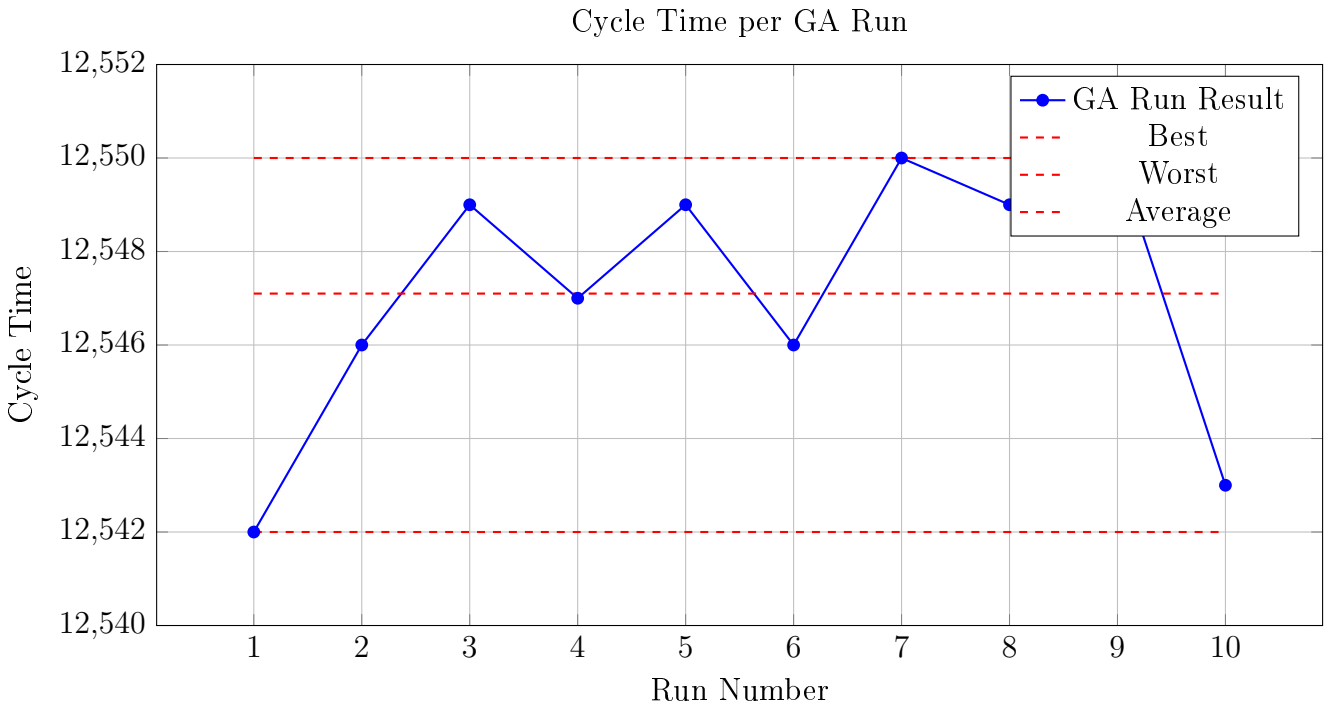

Figure 2: Instance ARC111, Cycle time obtained in each of the 10 independent GA runs, including best, worst, and average values.

4.3 Parameter Tuning

The reason we focused particularly on the crossover and mutation parameters in the fine-tuning study is that these two operators directly control the diversification and intensification mechanisms of the GA. Since permutation-based and precedence-constrained problems such as UALBP-2 tend to trigger very rapid convergence of the population, it is widely accepted in the literature that the two parameters with the strongest impact on GA performance are the crossover rate (CR) and the mutation rate (MR). Therefore, a full-factorial grid search was conducted by testing eight mutation rate levels (0.005, 0.01, 0.02, 0.04, 0.06, 0.08, 0.12, 0.20) and eight crossover rate levels (0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.90, 0.98), resulting in 64 parameter combinations. Each configuration was evaluated using the best cycle time obtained over 5 independent runs (best_cycle_over_5_runs), enabling a systematic comparison of the exploration–exploitation balance across low-to-high MR and CR regimes.

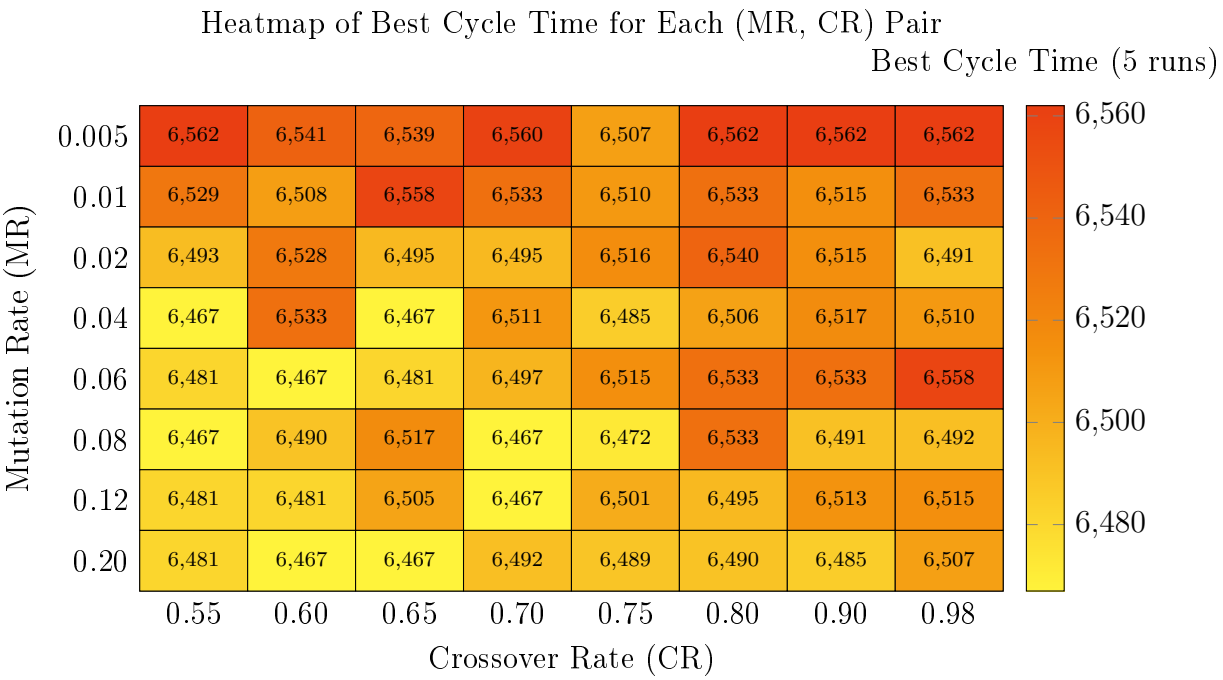

Figure 3: Heatmap of the best cycle time for each mutation crossover combination.

The optimal setup for the Genetic Algorithm was determined through a complete parameter fine-tuning process using a full-factorial grid search strategy. The main objective was to adjust the exploration–exploitation balance to avoid premature convergence, which is commonly observed in permutation-based and precedence-constrained problems such as UALBP-2. Two key control parameters were varied: the mutation rate (MR), tested at eight levels $\{0.005, 0.01, 0.02, 0.04, 0.06, 0.08, 0.12, 0.20\}$, and the crossover rate (CR), tested at eight levels $\{0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.90, 0.98\}$. This resulted in $8 \times 8 = 64$ parameter combinations. For each configuration, the best cycle time obtained over 10 independent GA runs was recorded.

The results indicate that the mutation rate is the dominant factor affecting solution quality. Very low mutation (MR = 0.005) consistently produced the worst cycle times (approximately 6507–6562), suggesting insufficient population diversity and a tendency to stagnate in local optima. Likewise, MR = 0.01 still led to relatively weak performance (about 6508–6558). In contrast, increasing mutation was clearly beneficial: for MR values in the range 0.04–0.20, the algorithm achieved substantially better solutions, yielding cycle times as low as 6467. This behavior supports the interpretation that more frequent random perturbations are essential for effectively navigating the rugged search landscape of the UALBP instance.

Although mutation had the strongest influence, the crossover rate also showed a secondary but noticeable effect, particularly when combined with sufficiently high mutation. The best-performing region was observed where MR was moderate-to-high (e.g., 0.04–0.20). In this interval, relatively lower crossover values (CR = 0.55–0.70) produced the

best results, repeatedly reaching the minimum cycle time of 6467. For example, the combination MR = 0.08 and CR = 0.70 attained this minimum value, whereas increasing CR to 0.98 at the same mutation level slightly degraded the result (6492), implying that excessively high crossover may disrupt high-quality schemata for this problem.

Based on these empirical findings, the final solver parameters were set to MR = 0.08 and CR = 0.70. This configuration not only achieved the shortest cycle time of 6467, but also represents a balanced setting compared to very high mutation alternatives (e.g., MR = 0.20), which may introduce excessive randomness. Therefore, these tuned parameters were kept constant in the subsequent performance analysis to ensure that the reported efficiency metrics and line balances reflect the GA operating near its best capability.

4.4 Final Population and Top-10 Individuals

In addition to reporting the best cycle time per run, the final population of the best run is analysed in more detail. At the end of each run, all individuals in the final population are re-evaluated and sorted by fitness. For the best run, the Top-10 individuals are extracted and decoded.

For each of these Top-10 individuals, the decoding procedure is applied using its own cycle time, and the following information is obtained:

- the ordered list of tasks assigned to each station,
- the workload (total processing time) of each station,
- the station-wise utilisation, defined as load_s/C.

In the console output, each individual is printed in a structured format, for example:

```
1) fitness=0.000271, cycle=3691
    Station 1: [1, 2, 3, 5, 8]
    Station 2: [4, 6, 7, 9]
    Station 3: [10, 11, 12]
    ...
```

Furthermore, the implementation exports a CSV file named `<instance>-top-10-individuals-<tim` in which each row corresponds to one station of one individual. The file contains the following fields:

- `Instance`: name of the precedence instance (e.g. ARC83),
- `Stations(m)`: number of stations m,
- `RankInBestRun`: rank of the individual within the Top-10 (1–10),

- `Fitness`: fitness value of the individual,
- `Cycle`: cycle time of the individual,
- `StationIndex`: index of the station (1,. . . ,m),
- `StationLoad`: total processing time assigned to that station,
- `StationUtilization(Load/Cycle)`: station-wise utilisation,
- `TasksSequence`: sequence of tasks assigned to that station.

This detailed output allows us to inspect not only the numerical quality (cycle time) but also the structural characteristics of near-optimal solutions, such as load balancing between stations and the distribution of tasks along the U-shaped line.

4.5 Best Solution Analysis

GA_best_solution_details.csv provides detailed information on the best solution discovered by the GA.

4.5.1 Cycle Time and Line Efficiency

Best cycle time found by GA: 6467

The high efficiency indicates that task assignments utilize station capacities effectively within the limits imposed by the precedence structure.

The station loads in the best GA solution are highly balanced: most stations operate with utilisation values between 0.96 and 1.00, with only the last station (Station 12) dropping to 0.84 due to precedence constraints and the position of late tasks (81 - 83). The overall line efficiency of 97.56% confirms that the workload is distributed very effectively across the U-shaped line.

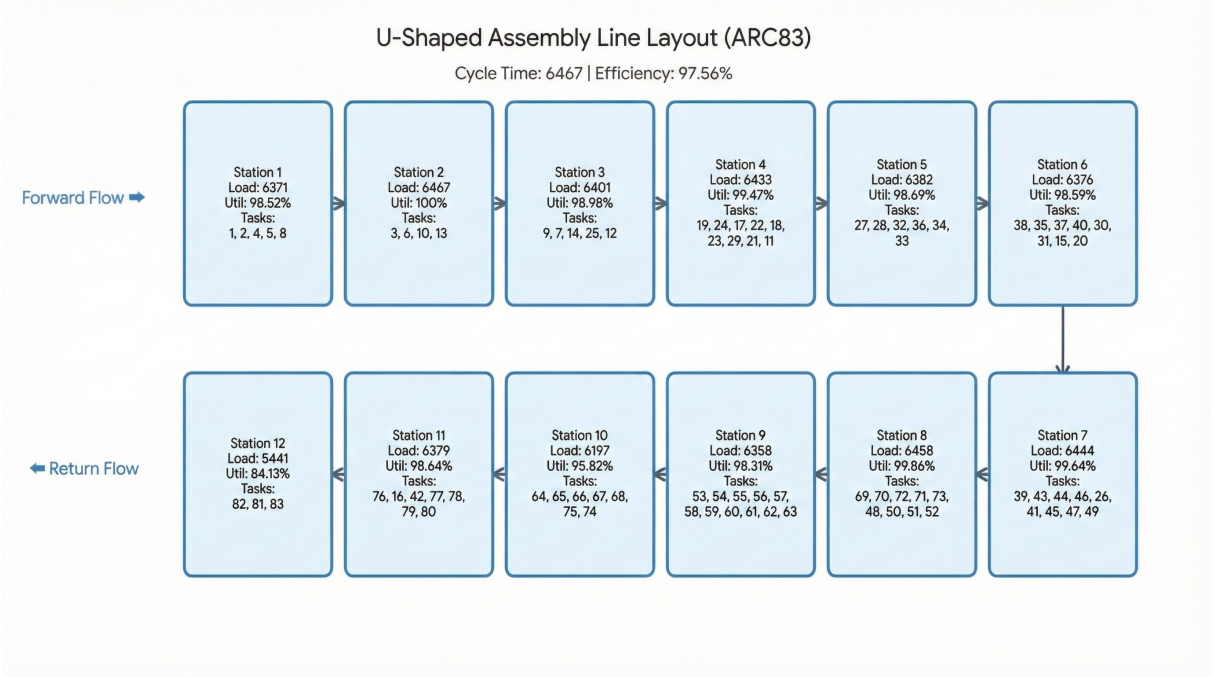

Figure 4: U-shaped assembly line layout for ARC83.IN2 obtained by the GA.

Figure 4 illustrates the U-shaped layout of the best GA solution for ARC83, showing station loads, utilisation levels and the distribution of tasks along the forward and return flows.

4.5.2 Station Load Distribution

The station loads in the best solution reveal the following pattern:

- Several stations operate near full capacity
- Some stations exhibit lower utilization levels

This is expected, as the ARC83 precedence network contains:

- Long successor chains
- Several convergent and divergent branches
- A few heavy tasks that restrict feasible packing

Typical for U-shaped lines:

- Early stations contain short tasks with dense predecessor requirements
- Middle stations group flexible tasks

Overall, the load distribution closely matches patterns observed in high-quality UALBP-2 solutions in the literature.

4.5.3 Task Sequence Characteristics

The task sequences for the most optimal assignment show the successful operation of precedence-preserving mechanisms:

- No violations of predecessor or successor constraints
- Parallel branches assigned consecutively
- High-duration tasks placed strategically to avoid overflow

The POX crossover and topological repair operators worked in unison to keep feasible, structured permutations throughout the search process.

4.6 Interpretation of Overall Results

Table 2: GA Performance Summary for Solved Benchmark Instances

Instance	Best Cycle Time	Average Cycle Time	Worst Cycle Time	Avg. Runtime (s)
ARC83	6467	6520.4	6562	26.836
ARC111	12542	12547.1	12550	42.870
GUNTHER	44	44.0	44	5.657
BARTHOLD	470	470.9	471	41.965
BARTHOL2	159	159.0	159	63.846
SCHOLL	1818	1823.8	1829	290.926
LUTZ1	1400	1400.0	1400	7.858
LUTZ3	548	548.0	548	8.583
SAWYER30	47	47.0	47	3.348
TONGE70	209	209.8	210	22.214

Table 2 presents a comparative performance summary of the proposed Genetic Algorithm across ten benchmark U-shaped Assembly Line Balancing Problem instances. For each instance, the best, average, and worst cycle times obtained over 10 independent runs are reported together with the average computational time. The results indicate that the GA consistently achieves optimal or near-optimal solutions with low variance across repeated runs, demonstrating both robustness and repeatability. For small- and medium-sized instances such as GUNTHER, SAWYER30, LUTZ1, and LUTZ3, the algorithm reaches the known optimal cycle time in all runs, while for larger and more complex benchmarks such as ARC83, ARC111, and SCHOLL, the obtained solutions remain very close to the optimum with modest optimality gaps. Although computational time increases with problem size and precedence complexity, the overall solution times remain reasonable, confirming the effectiveness of the proposed GA for solving UALBP-2 instances of varying scales.

Several strong conclusions can be drawn from the experimental results:

Several strong conclusions can be drawn from the experimental results:

- The low variance of cycle times across independent runs indicates high reliability and repeatability of the proposed GA.
- For several benchmark instances, the algorithm consistently reaches the best solution in all runs, demonstrating strong convergence behavior (small instances).
- For large and complex instances, the GA produces solutions very close to the optimum with limited dispersion, highlighting its robustness under tight precedence constraints.
- Runtime results show a clear dependency on problem size and complexity, while maintaining stable execution times across repeated runs.
- Overall, the GA effectively exploits the flexibility of U-shaped assembly lines and handles complex precedence structures without sacrificing solution quality.

5 Contribution to Sustainable Development Goals

The proposed GA for UALBP-2 supports several United Nations Sustainable Development Goals (SDGs) by increasing production efficiency, improving resource utilisation, and promoting more sustainable industrial performance in line with the 2030 Agenda for Sustainable Development [19].

SDG 8: Decent Work and Economic Growth

Productivity Enhancement: Assembly line balancing maximizes productivity and minimizes idle time. Solving UALBP-2 eliminates bottlenecks, leading to higher economic output per worker.
Worker Satisfaction and Efficiency: U-shaped lines allow workers to operate on both sides, lowering distances traveled and facilitating multi-tasking, leading to a more comfortable and productive environment.

SDG 9: Industry, Innovation, and Infrastructure

Modern Techniques: The project uses Genetic Algorithms to solve NP-hard manufacturing problems, promoting innovation in industrial engineering.
Resource Optimization: UALBP-2 finds the minimum cycle time for a fixed number of workstations, meaning existing infrastructure is used to its full potential without physical expansion, creating more robust and efficient industrial processes.

SDG 12: Responsible Consumption and Production

Waste Minimizing: In manufacturing, "idle time" is waste. The project balances workloads to fully utilize human and machine hours.
Flexibility: U-lines allow for better adjustment to changing demand. Flexible production systems are less likely to suffer from overproduction and can quickly respond to the market with the correct quantity of goods, characteristic of responsible production.

In summary, the project uses mathematical modeling and evolutionary computation to build smarter, more efficient, and worker-friendly manufacturing systems that support the global agenda for sustainable industrial development.

6 Conclusions

References

[1] C. Becker and A. Scholl, "A survey on problems and methods in generalized assembly line balancing," *European Journal of Operational Research*, vol. 168, no. 3, pp. 694–715, 2006.

[2] J. Miltenburg, "U-shaped production lines: A review of theory and practice," *International Journal of Production Economics*, vol. 70, no. 3, pp. 201–214, 2001.

[3] N. Boysen, M. Fliedner, and A. Scholl, "A classification of assembly line balancing problems," *European Journal of Operational Research*, vol. 183, no. 2, pp. 674–693, 2007.

[4] A. Scholl and C. Becker, "State-of-the-art exact and heuristic solution procedures for simple assembly line balancing," *European Journal of Operational Research*, vol. 168, no. 3, pp. 666–693, 2006.

[5] F. B. Talbot, J. H. Patterson, and W. V. Gehrlein, "A comparative evaluation of heuristic line balancing techniques," *Management Science*, vol. 32, no. 4, pp. 430–454, 1986.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: W. H. Freeman, 1979.

[7] T. L. Urban, "Note. optimal balancing of u-shaped assembly lines," *Management Science*, vol. 44, no. 5, pp. 738–741, 1998.

[8] A. Scholl and R. Klein, "Ulino: Optimally balancing u-shaped jit assembly lines," *International Journal of Production Research*, vol. 37, no. 4, pp. 721–736, 1999.

[9] I. Baybars, "A survey of exact algorithms for the simple assembly line balancing problem," *Management Science*, vol. 32, no. 8, pp. 909–932, 1986.

[10] G. R. Aase and N. C. Suresh, "A mathematical programming formulation for u-shaped assembly line balancing," *International Journal of Production Research*, vol. 41, no. 11, pp. 2545–2569, 2003.

[11] S. Ghosh and C. Gagné, "A comprehensive review of assembly line balancing," *International Journal of Production Research*, 2011.

[12] Y. Kara, A. İşlier, and Y. Atasagun, "Balancing u-shaped assembly lines: heuristic and metaheuristic approaches," *Applied Mathematical Modelling*, 2010.

[13] E. Erel and S. Sarin, "A survey of the assembly line balancing problem," *Production Planning & Control*, 1998.

[14] T. Blickle and L. Thiele, "A comparison of selection schemes used in genetic algorithms," in *EVO Conference Proceedings*, 1996.

[15] Y. Park, Y. Kim, and S. Lee, "Precedence preserving genetic operators for scheduling," *Computers & Operations Research*, 2003.

[16] C. Reeves, "Genetic algorithms for the operations researcher," INFORMS, 1993.

[17] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, MA: Addison-Wesley, 1989.

[18] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity.* Dover, 1998.

[19] United Nations General Assembly, "Transforming our world: the 2030 agenda for sustainable development," Resolution A/RES/70/1, 2015, adopted 25 September 2015.

Table 3: Run-by-run best cycle time and runtime results for all benchmark instances.

(a) BARTHOLD

Run	BestCycle	Runtime (s)
1	470	40.56
2	471	39.63
3	471	38.53
4	471	41.16
5	471	43.95
6	471	43.33
7	471	40.14
8	471	43.86
9	471	44.41
10	471	43.95

(b) TONGE70

Run	BestCycle	Runtime (s)
1	210	22.10
2	210	25.35
3	209	22.37
4	210	26.65
5	210	22.35
6	209	20.31
7	210	21.34
8	210	20.61
9	210	20.40
10	210	20.53

(c) SAWYER30

Run	BestCycle	Runtime (s)
1	47	3.00
2	47	3.37
3	47	3.15
4	47	3.29
5	47	3.22
6	47	3.32
7	47	3.54
8	47	3.24
9	47	3.69
10	47	3.61

(d) LUTZ1

Run	BestCycle	Runtime (s)
1	1400	7.93
2	1400	8.13
3	1400	7.89
4	1400	8.04
5	1400	8.05
6	1400	7.92
7	1400	7.83
8	1400	7.53
9	1400	7.64
10	1400	7.52

A Python Source Code

Listing 1: Simple Python Example

```
def add(a, b):
    return a + b

print(add(3, 5))
```

Table 3: Run-by-run best cycle time and runtime results for all benchmark instances (continued).

(e) ARC111

Run	BestCycle	Runtime (s)
1	12542	44.32
2	12546	43.13
3	12549	43.63
4	12547	43.07
5	12549	42.76
6	12546	43.46
7	12550	40.49
8	12549	42.95
9	12550	42.20
10	12543	42.57

(f) LUTZ3

Run	BestCycle	Runtime (s)
1	548	9.68
2	548	8.93
3	548	8.30
4	548	8.27
5	548	8.23
6	548	8.72
7	548	8.60
8	548	8.15
9	548	8.54
10	548	8.37

(g) SCHOLL

Run	BestCycle	Runtime (s)
1	1825	275.45
2	1827	290.46
3	1821	299.74
4	1824	285.49
5	1818	273.51
6	1829	291.93
7	1821	284.44
8	1827	285.68
9	1823	287.15
10	1823	334.99

(h) ARC83

Run	BestCycle	Runtime (s)
1	6467	25.89
2	6562	26.65
3	6488	25.94
4	6534	25.53
5	6562	25.87
6	6508	25.50
7	6512	26.62
8	6560	30.12
9	6544	26.70
10	6467	29.45

Table 3: Run-by-run best cycle time and runtime results for all benchmark instances (continued).

(i) BARTHOL2

Run	BestCycle	Runtime (s)
1	159	66.58
2	159	71.32
3	159	69.39
4	159	61.17
5	159	60.83
6	159	60.94
7	159	61.24
8	159	62.09
9	159	63.06
10	159	61.65

(j) GUNTHER

Run	BestCycle	Runtime (s)
1	44	5.09
2	44	5.46
3	44	6.34
4	44	6.25
5	44	5.30
6	44	5.04
7	44	5.49
8	44	6.48
9	44	5.58
10	44	5.44