Intelligent Agents COMP2009

School of Electrical Engineering, Computing and Mathematical Sciences

Optimising Employee Task Assignments with Evolutionary Algorithms

Design and Evaluation of Genetic Algorithm, Particle Swarm Optimisation and Ant Colony Optimisation

Group 15

Saf Flatters 21827361

Thomas Sounness 21483734

Nandar Ko Ko Lynn 21072899

Lab: Monday, Tuesday 12pm-2pm

Contents

1. Introduction

1.1. Problem Statement

The Employee Task Assignment Optimisation problem involves assigning a set of tasks to a team of employees while meeting five key constraints. Each employee has limited working hours, specific skill sets and skill level. Each task requires a certain number of working hours to complete, has a deadline for completion, a difficulty level (matched with skill level) and a required skill.

The five key constraints to consider are;

- 'Unique Assignment' (Unique Assignment Violation Penalty) where each task should be assigned to exactly one employee,
- 'Capacity Constraint' (Overload Penalty) where the total working hours of the tasks assigned to an employee can not exceed their available working hours,
- 'Skill Level Constraint' (Difficulty Violation Penalty) where the difficulty of a task must be less or equal to the assigned employees' skill level
- 'Specialised Skill Matching' (Skill Mismatch Penalty) where the task's required skill must be in the assigned employees' specific skill set and,
- 'Deadline Consideration' (Deadline Violation Penalty) where the employees assigned tasks are scheduled into ascending order of processing time and the cumulative task time must not exceed the task's deadline

The Objective Function calculates penalties for violations of these constraints. We have been provided weighting factors for the respective penalties. The weighting factors α, β, γ, δ, σ all have a weighting factor of 0.2.

Objective Cost Function =

$$\alpha \times (\text{Overload Penalty}) + \beta \times (\text{Skill Mismatch Penalty}) + \gamma \times (\text{Deadline Violation}) + \delta \times (\text{Difficulty Violation Penalty}) + \sigma \times (\text{Unique Assignment Violation Penalty})$$

The goal is to find a solution that minimises this cost as much as possible.

1.2. Purpose

The purpose of this project is to design and analyse three different Evolutionary Algorithms; Genetic Algorithm (GA), Particle Swarm Optimisation (PSO) and Ant Colony Optimisation (ACO), to find the optimal assignment of tasks to employees that results in the lowest possible total penalty and the most feasible and efficient solution.

The algorithms are compared based on three performance parameters, using plots to visualise the results:

1) Solution Quality (Optimality): This compares how well each algorithm reduces the Objective Cost Function. A plot is created to analyse and compare convergence behaviour as the number of generations, iterations, or colonies increase. Additional plots are created to explore the optimal number of generations, iterations or colonies to set and the optimal population size of chromosomes, particles or ants to set. Each algorithm is run many times to account for randomness and to explore confidence intervals.
2) Computational Efficiency (Runtime): This compares how fast each algorithm can run, measured in seconds. Each algorithm is tested across different generation, iteration or colony settings. It also is run multiple times to account for randomness and explores confidence intervals. We also explore the impact of using a termination function that stops the algorithm early when convergence behaviour is detected. This is compared to the runtime when not using the termination function.
3) Constraint Satisfaction (Feasibility): This compares how quickly each algorithm produces solutions that minimise total constraint violations. The total number of penalties is summed after each generation, iteration or colony to track how constraint satisfaction improves over time.

Analysing these performance parameters enabled us to iteratively tune each algorithm and evaluate the strengths and limitations of each approach.

2. Methodology

2.1. Experimental Setup

Synthetic Data

We created two dictionaries, ‘tasks’ and ‘employees’, in a file called SyntheticData.py using the information provided in the assignment brief. This file was imported into all algorithm function files, GeneticFunctions.py, ParticleSwarmFunctions.py and AntColonyFunctions.py, and also into our main run file, Run.py.

```
# Synthetic Task Data #10 tasks
tasks = [
    {"id": "T1", "time": 4, "difficulty": 3, "deadline": 8, "skill": "A"},
    {"id": "T2", "time": 6, "difficulty": 5, "deadline": 12, "skill": "B"},
    {"id": "T3", "time": 2, "difficulty": 2, "deadline": 6, "skill": "A"},
    {"id": "T4", "time": 5, "difficulty": 4, "deadline": 10, "skill": "C"},
    {"id": "T5", "time": 3, "difficulty": 1, "deadline": 7, "skill": "A"},
    {"id": "T6", "time": 8, "difficulty": 6, "deadline": 15, "skill": "B"},
    {"id": "T7", "time": 4, "difficulty": 3, "deadline": 9, "skill": "C"},
    {"id": "T8", "time": 7, "difficulty": 5, "deadline": 14, "skill": "B"},
    {"id": "T9", "time": 2, "difficulty": 2, "deadline": 5, "skill": "A"},
    {"id": "T10", "time": 6, "difficulty": 4, "deadline": 11, "skill": "C"},
]

# Synthetic Employee Data #5 employees
employees = [
    {"id": "E1", "hours": 10, "skill_level": 4, "skills": {"A", "C"}},
    {"id": "E2", "hours": 12, "skill_level": 6, "skills": {"A", "B", "C"}},
    {"id": "E3", "hours": 8, "skill_level": 3, "skills": {"A"}},
    {"id": "E4", "hours": 15, "skill_level": 7, "skills": {"B", "C"}},
    {"id": "E5", "hours": 9, "skill_level": 5, "skills": {"A", "C"}},
]
```

Robustness

Testing was performed at different generations/iterations/colony numbers (10, 50, 100, 200, and 500) and at population sizes of 5, 10, 15, 20, and 30 to search for the optimal settings for each algorithm. Each setup was run 3 times to account for randomness and to allow confidence intervals to be plotted. See the Performance Evaluation section for this analysis and results. The outcomes of these experiments also informed the final parameter settings chosen for each algorithm, which are discussed in their respective sections.

Hardware/Software

Code was written and experimented using Python 3.11 with libraries numpy, pandas, time, seaborn, matplotlib.pyplot and random. Experiments were run on a Dell laptop with an Intel Core i7 processor and 16GB RAM. All experiments used seeds to ensure replication ability. Plots and larger experiments are commented out in Run.py which calls functions in VisualisationFunctions.py. The team used Git and Github heavily to manage version control and team collaboration.

2.2 Implementation Details

Initialisation - How solutions are encoded

Chromosomes, Particles and Ants has been initialised and encoded as a list of employee IDs, where the index of the list represents the task number. For example; The chromosome [3, 2, 3, 2, 1, 2, 5, 4, 2, 1] means Task 1 is assigned to Employee 3, Task 2 is assigned to Employee 2, Task 3 is assigned to Employee 3 and so on.

Due to updates to the assignment brief after we had already built our algorithms, the Unique Assignment constraint was treated as a hard constraint rather than something that could be penalised. It

was built directly into the way we initialised chromosomes, particles, and ants, meaning no Unique Assignment violations could occur during solution generation. At the start of the project, we considered several ways of handling initialisation. In hindsight, we would have chosen another approach such as initialising a chromosome, particle or ant as a binary list of tasks (where 1 indicates a task is assigned and 0 is not assigned) for each employee, allowing multiple employees to be assigned to the same task from the beginning.

Fitness Function - How constraints are managed

The Objective Cost Function was calculated almost identically across all three algorithms - but each had its own function module (in the algorithms function .py file). The main function, called FitnessCost(), would pass in the chromosome, particle position or ant representing a proposed solution.

A task_counter dictionary was created for each employee, initialised with four zeros to track penalties for overload, mismatch, difficulty, and deadline violations. The first three penalties (overload, mismatch and difficulty) were calculated by iterating through each task index of the chromosome, particle, or ant, using helper functions: calcOverload_Emp(), calcMismatch_Emp() and calcDifficulty_Emp().
The fourth penalty (deadline violation) was calculated by collecting all tasks assigned to an employee, sorting them by processing time (Shortest Job First), and computing cumulative finish times with the calcDeadline_Task() function.

We also implemented calcUniqueAssign_Violation() which checked if any task was assigned to more than one employee. However, as discussed earlier, the way chromosomes, particles, and ants were initialised meant that unique assignment violations would always be zero.

Once all penalties were counted, the objectiveFunction() module calculated the total cost by multiplying each penalty by a weighting factor of 0.2 and summing them together. This cost was then paired with the corresponding chromosome, particle position or ant into a tuple (cost, solution). The cost component was used by each algorithm to determine the ranking of their solutions before selection.

Termination Condition

For all three algorithms, we implemented a termination condition designed to improve computational efficiency without sacrificing solution quality. In addition to running for a maximum number of generations, iterations, or colonies (e.g. 500), the algorithm could also stop early if it detected convergence.

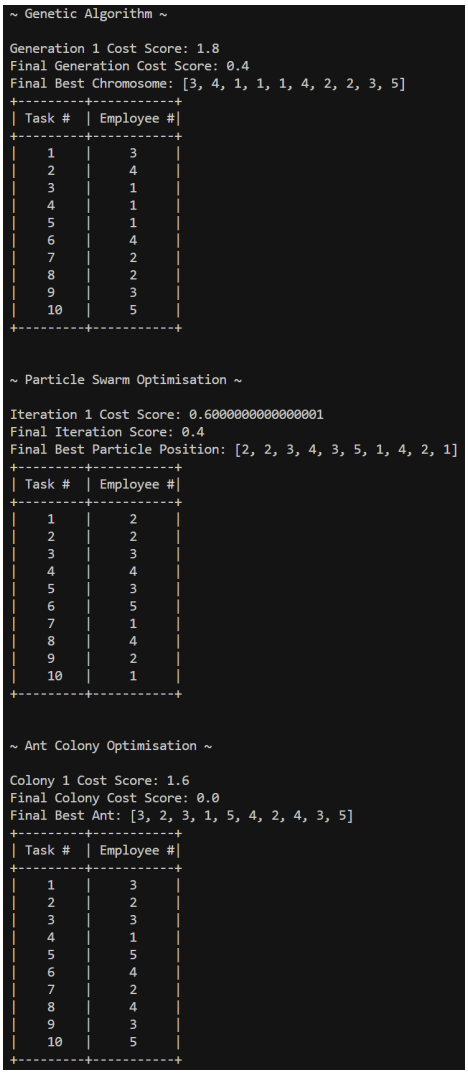
Convergence was defined as no improvement in the best objective function score for a set number of consecutive generations (specifically, 10% of the maximum number of generations, with a minimum of 10 generations). If the best solution remained unchanged for this threshold, the algorithm would

automatically terminate early. This approach helped to avoid unnecessary computation when no further progress was being made.

The termination setting could be turned on by passing a term=True argument when calling the algorithm's main function in Run.py (or for experiments in VisualisationFunctions.py) otherwise its default is False which allows the Generations / Iterations / Colonies to run for the entire amount specified. See Performance Evaluation section for experiments run on this Termination condition.

Dummy Output

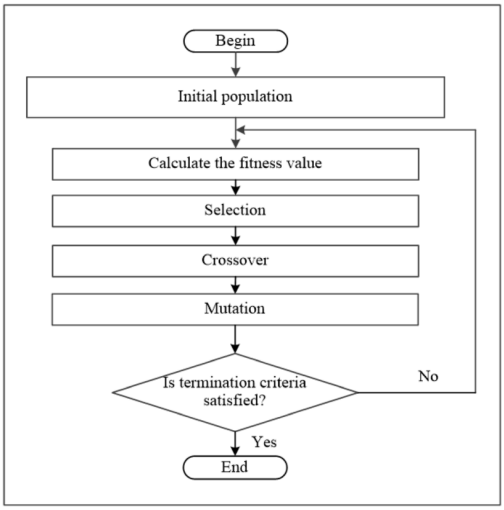
Run.py runs each algorithm and prints an output for 50 generations / Iterations / Colonies with population size 20 (chromosomes, particles, ants). This was chosen to show a significant difference from the first generation to the last one in the dummy output. This setting was not the basis of our testing or experiments. The dummy output uses a function called humanReadable() that prints the final solution Task-Employee assignments into a table in the terminal.



```
~ Genetic Algorithm ~

Generation 1 Cost Score: 1.8
Final Generation Cost Score: 0.4
Final Best Chromosome: [3, 4, 1, 1, 1, 4, 2, 2, 3, 5]
+---------+-----------+
| Task #  | Employee #|
+---------+-----------+
|    1    |     3     |
|    2    |     4     |
|    3    |     1     |
|    4    |     1     |
|    5    |     1     |
|    6    |     4     |
|    7    |     2     |
|    8    |     2     |
|    9    |     3     |
|    10   |     5     |
+---------+-----------+


~ Particle Swarm Optimisation ~

Iteration 1 Cost Score: 0.6000000000000001
Final Iteration Score: 0.4
Final Best Particle Position: [2, 2, 3, 4, 3, 5, 1, 4, 2, 1]
+---------+-----------+
| Task #  | Employee #|
+---------+-----------+
|    1    |     2     |
|    2    |     2     |
|    3    |     3     |
|    4    |     4     |
|    5    |     3     |
|    6    |     5     |
|    7    |     1     |
|    8    |     4     |
|    9    |     2     |
|    10   |     1     |
+---------+-----------+


~ Ant Colony Optimisation ~

Colony 1 Cost Score: 1.6
Final Colony Cost Score: 0.0
Final Best Ant: [3, 2, 3, 1, 5, 4, 2, 4, 3, 5]
+---------+-----------+
| Task #  | Employee #|
+---------+-----------+
|    1    |     3     |
|    2    |     2     |
|    3    |     3     |
|    4    |     1     |
|    5    |     5     |
|    6    |     4     |
|    7    |     2     |
|    8    |     4     |
|    9    |     3     |
|    10   |     5     |
+---------+-----------+
```

2.3 Algorithm Overview

Genetic Algorithm (GA)



The Genetic Algorithm (GA) was built to solve the Employee Task Assignment Problem by evolving a population of candidate solutions 'chromosomes' over multiple generations. Each chromosome is made up of task-employee assignments called 'genomes'.
The design of our GA followed the general evolutionary principles outlined by Mirjalili (2019), where the main operators are Selection, Crossover, and Mutation and it was guided by the flow chart above proposed by Albadr et al. (2020).

The GA, implemented in GeneticFunctions.py, is called from Run.py using the function geneticAlgorithm(size, generations, term=False, seed=None). It first generates an initial population using buildPopulation(), which calls createChromosome() for each solution. The initial Objective Cost Function is then calculated by the fitnessCost() function described earlier to establish a baseline.
For the number of generations specified, the following steps are repeated:

- Chromosomes are ranked from best to worst based on their cost function value.
- Elite-ism is used to clone the top 20% of chromosomes directly into the next generation.
- The top 80% of chromosomes (excluding the worst 20%) are placed into a breeding pool.
- Using Roulette Wheel Selection (rouletteSelection() function), chromosomes are selected for breeding (crossover) based on their probability (higher ranked chromosomes have a higher chance of being selected).
- Two parents are selected and Single-Point Crossover (singlePointCrossover()) is performed, where a random crossover point is selected to swap genetic material between parents and create two children.
- The resulting offspring undergo Mutation (reassignMutation()), where at a 20% mutation rate, a task may be reassigned to a different employee randomly.
- Mutated children are added to the new population.

This process repeats until the new population reaches the specified size.
At the end of all generations (or early termination if convergence is detected), the best solution found is returned.

GA Parameter Settings

The parameters used for the Genetic Algorithm were not chosen arbitrarily. They were selected through an iterative process of testing, plotting and analysing performance trends. We experimented with different settings and observed their effect of Optimality, Efficiency and Feasibility - discussed in the Performance Evaluation section of this report.

Parameter	Setting
Selection	Roulette Wheel Selection & Elite-ism (top 20% cloned & bottom 20% discarded)
Crossover	Single-Point Crossover - point chosen at random
Mutation Rate	0.2

Particle Swarm Optimisation (PSO)

Particle Swarm Optimisation (PSO) is a metaheuristic inspired by the behavior of flocking birds (Kennedy & Eberhart, 1995). While originally designed for continuous optimisation, we adapted it to our discrete task assignment problem. In this implementation, each “particle” represents a complete task assignment, where each position corresponds to an employee-task pairing. The algorithm iterates through particles, adjusting their positions over time to explore the solution space and improve task assignments.

As described by Mirjalili (2017) in Chapter 2, PSO relies on two key memories for each particle: Personal Best (pBest), the best solution the particle has encountered, and Global Best (gBest), the best solution found by any particle in the swarm. To initialise the swarm, tasks are randomly assigned to employees through the createParticle() function, and multiple particles are generated using buildSwarm(size), where the size controls the diversity of the population.

Our particle updating mechanism modifies the traditional PSO velocity update formula to better fit the discrete task assignment problem. The standard PSO velocity update formula (Mirjalili, 2017) is:

$$v_i^{(t+1)} = w.v_i^t + c_1.r_1.(pBest_i - x_i^t) + c_2.r_2.(gBest - x_i^t)$$

However, in our implementation, we simplified this by eliminating the previous velocity term $v_i^{(t)}$ resulting in the following update equation:

$$v_i^{(t+1)} = w.(pBest_i - x_i^t) + c_1.r_1.(pBest_i - x_i^t) + c_2..(gBest - x_i^t)$$

This simplification was made because, in our discrete assignment problem, maintaining the historical velocity was less important than the influences of the personal and global best positions. By eliminating the previous velocity component and directly using the differences between the current position and the best positions, we created a more direct "attraction" mechanism toward promising solutions. This approach draws from heuristic methods used for discrete optimisation problems, such as the work of Rezaee Jordehi and Jasni (2011).

The inertia weight *w* has been adjusted to influence the personal best difference rather than the previous velocity, which facilitates faster convergence toward the optimal areas of the solution space in our discrete task assignment problem.

The parameters had been configured as follows: w was fixed at 0.5 for exploration and refinement balance; c_1 and $c_2$were both set to 1.5 so that personal and global bests would have equal weight; the random parameters were selected randomly to provide some stochastic effects, r_1 and r_2.

The early termination mechanism would halt the algorithm if it finds a perfect solution, or if no improvement is observed for 10% of iterations (minimum 10 iterations).

After velocity calculation, particle position is updated by adding velocity to the current position:

$$particle[i] \leftarrow particle[i] + velocity$$

The position is then discretized to valid employee indices. The fitnessCost() method evaluates the task assignment, while particles are updated based on historical bests.

The parameters are set up to balance exploration and exploitation, and further fine-tuning is required, as the algorithm has, however, not converged to the perfect solution values 0.0 for objective function value.

PSO Parameter Settings

The parameters used for the Particle Swarm Optimisation were not chosen arbitrarily. They were selected through an iterative process of testing, plotting and analysing performance trends. We experimented with different settings and observed their effect of Optimality, Efficiency and Feasibility - discussed in the Performance Evaluation section of this report.

Parameter	Setting
Inertia Weight *(w)*	0.5
Cognitive Component *(c1)*	1.5
Social Component *(c2)*	1.5

Ant Colony Optimisation (ACO)

The Ant Colony Optimisation (ACO) algorithm was built around the concept of stigmergy, where indirect communication between agents happens through modifications to the environment. In ACO, this is represented by pheromone trails, which guide the colony toward better solutions over time (Dorigo, et al., 2000). Unlike GA and PSO, where the agents themselves are updated directly, in ACO we maintained a Pheromone Matrix with employees on one axis and tasks on the other.

This Pheromone matrix is initialised by buildPhermoneMatrix(). Each cell in the matrix represents the pheromone strength (the probability) of assigning a particular employee to a particular task. Initially, all pheromone values are set to 1.0, giving each possible assignment an equal chance.

For the specified number of generations (colonies) we constructed solutions by generating a new set of ants through the buildColony() function, which internally calls createAnt() function.

Solution Construction: Each ant constructs a solution by probabilistically selecting an employee for each task using a roulette wheel selection approach based on the current pheromone levels. The probability of an employee being assigned to a task (t, e) is calculated as:

$$P(t,\ e)\ =\ \frac{Pheromone\ level(t,e)^{\alpha}}{\sum(Pheromone\ level(t,e)^{\alpha}}$$

where α = 1 (in our code) and the heuristic is omitted for simplicity.

Each ant's solution (a full assignment of tasks to employees) is evaluated using the fitnessCost() function, identical in structure to GA and PSO but adapted for ants. Solutions are ranked based on their objective cost.

Global Best: After evaluating all ants in the colony, the best solution (lowest cost) is compared against the global best solution. If the new solution is better, it updates the global best. The violations associated with the global best are also recorded at each generation.

Evaporation: evaporationPheromone() decay the pheromone levels over time to avoid early convergence on suboptimal solutions. We used:

$$Pheromone\ level(t,e)\ \leftarrow\ (1\ -\ \rho)\ \times\ Pheromone\ level(t,\ e)$$

where ρ was eventually set to 0.2 after testing for a balance between exploration and exploitation.

Deposit: After evaporation, depositPheromone() uses elitism selection to allow the top 25% of ants to deposit their pheromones according to:

$$Pheromone\ level\ (t,\ e)\ \leftarrow Pheromone\ level\ (t,\ e)\ +\ \frac{Q}{fitness\ cost\ of\ ant}$$

where Q is a constant we eventually set to 0.04 after testing how much we should reinforce good paths.

We chose Elitism after iteratively attempting to get an acceptable solution (compared to the other algorithms) within 100 generations.

The amount of pheromone deposited is inversely proportional to the cost of the chosen ants so that lower cost is reinforced more strongly (Mirjalili, 2019). This elitist approach allows the algorithm to converge quicker by focusing on high quality solutions while allowing the weaker solutions to be forgotten over time due to evaporation.

The colony generation process continues until the maximum number of generations is reached or an early stopping criterion is satisfied, as described in the Termination section of this report.

At the end of the process, the global best solution is returned.

ACO Parameter Settings

The parameters used for the Ant Colony Optimisation were not chosen arbitrarily. They were selected through an iterative process of testing, plotting and analysing performance trends. We experimented with different settings and observed their effect of Optimality, Efficiency and Feasibility - discussed in the Performance Evaluation section of this report.

Parameter	Setting
Pheromone Evaporation Rate ρ	0.2
Deposit Constant Q	0.04
Deposit selection	Elitism (top 25%) can only deposit

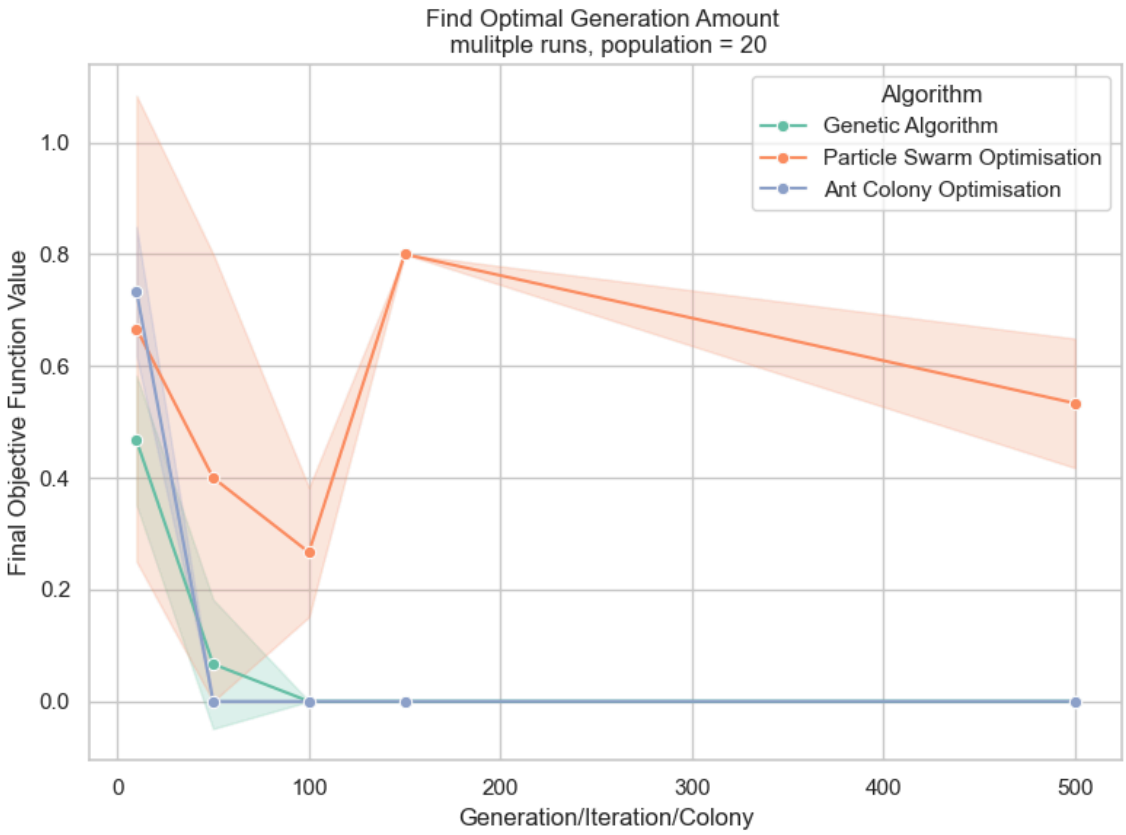
3. Performance Evaluation

To assess Solution Quality, Computational Efficiency, and Feasibility, we first created two plots to determine the optimal number of generations and the optimal population size. We ran each algorithm (without applying any termination condition) multiple times and plotted the mean and confidence

intervals at one standard deviation to account for randomness. A hashed seed was used for reproducibility across multiple runs.

This assessment was performed iteratively, testing for the Optimal Number of Generations using the Optimal Population Size, and vice versa, to ensure consistency. The resulting test parameters were then used for all subsequent tests.

Optimal Number of Generations

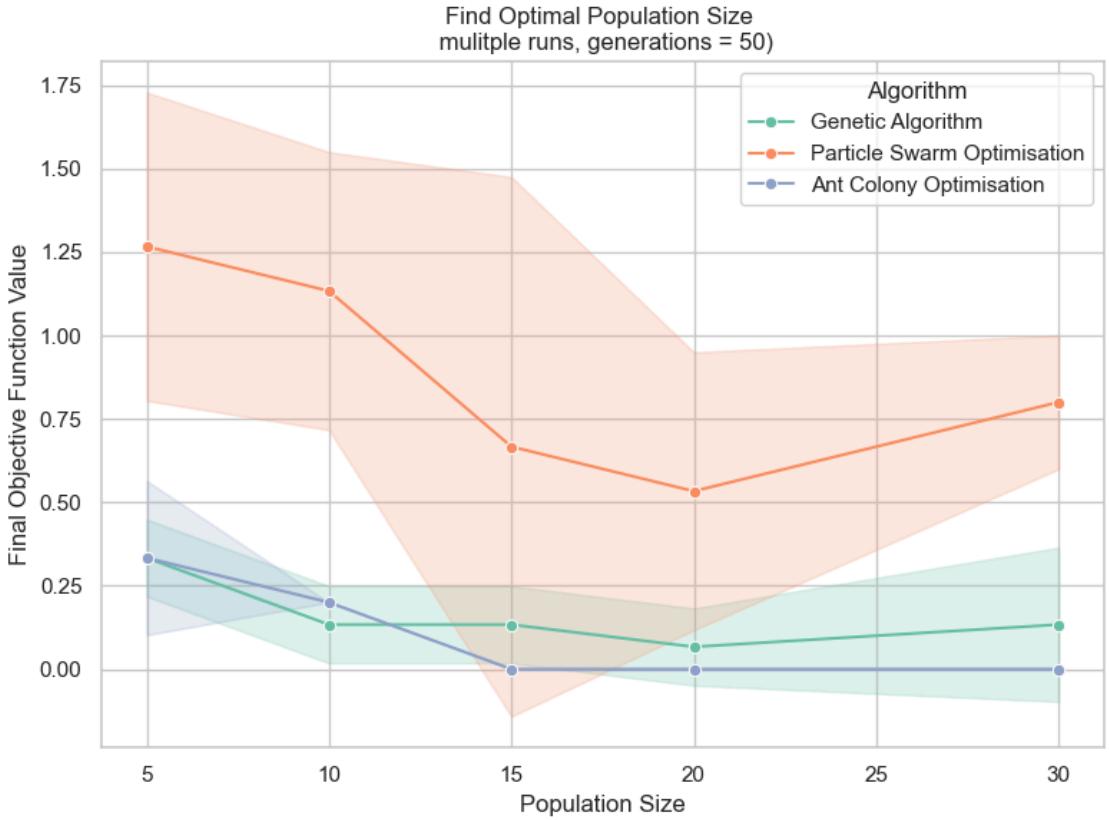


Testing was conducted at 10, 50, 100, 200, and 500 generations/iterations/colony numbers. As seen in the plot above, the best Final Objective Cost Function values were achieved at 100 generations across all three algorithms (population size = 20), where:

- GA (green) achieved 0.00
- PSO (orange) achieved 0.25 but could go lower,
- ACO achieved 0.00 for colony numbers over 50.

For further analysis, 100 generations were chosen to allow a fair comparison across all algorithms.

Optimal Population Size

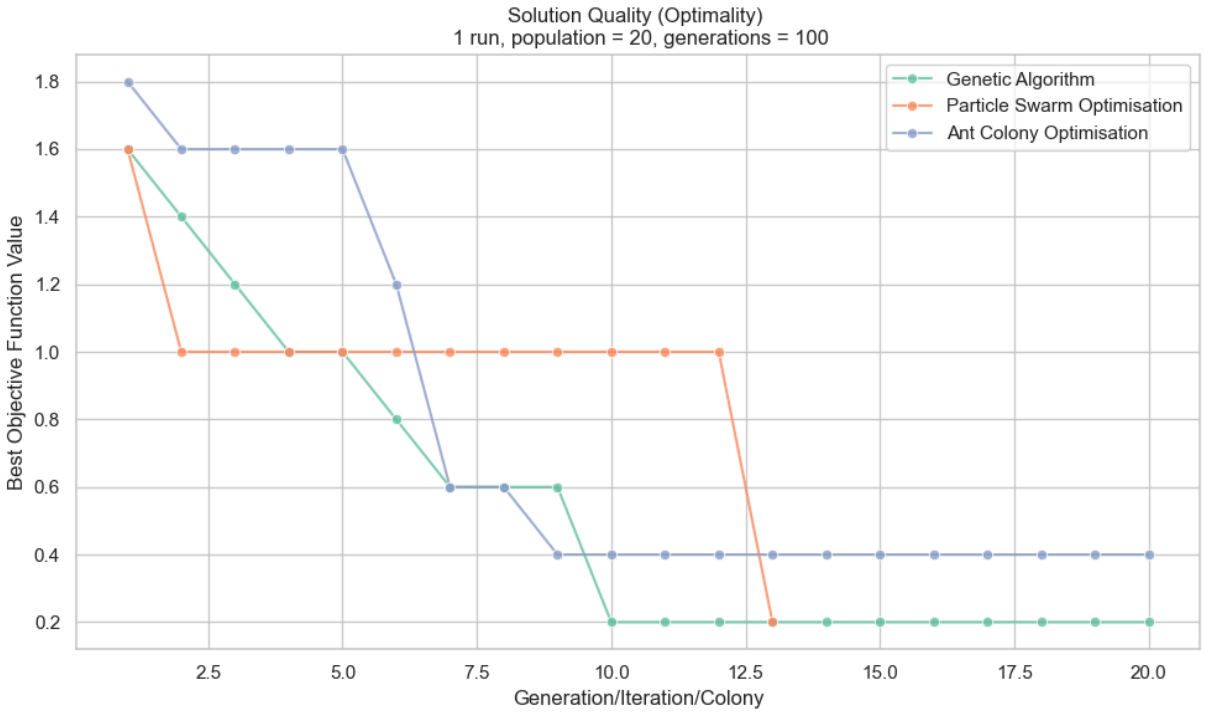


Similarly, population size testing was performed at 5, 10, 15, 20, and 30. As shown in the plot above, the best Final Objective Cost Function values for all algorithms were found at a population size of 20.

3.1. Solution Quality (Optimality)

The plots above also provide insight into algorithm optimality. They indicate that all algorithms are capable of achieving a Final Objective Cost Function value of 0.00, with Ant Colony Optimisation (ACO) and Genetic Algorithm (GA) demonstrating higher consistency, as reflected by their narrower confidence intervals.

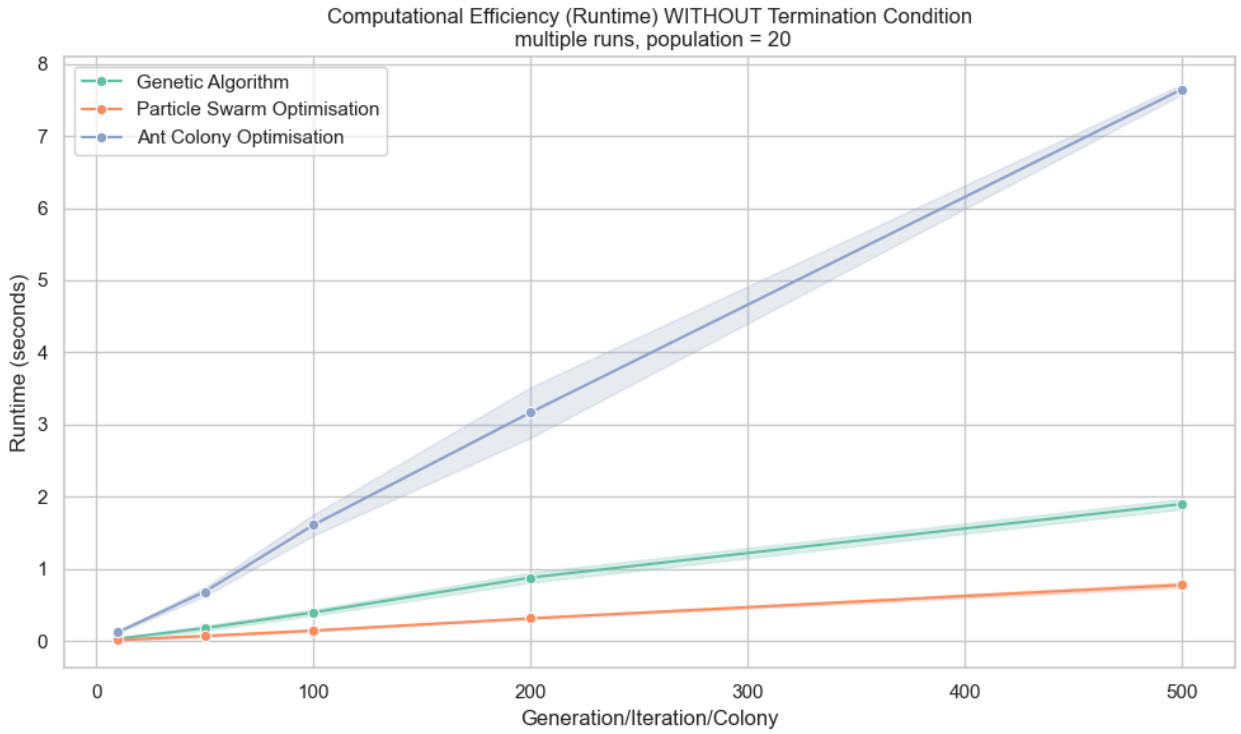
Particle Swarm Optimisation (PSO) shows greater variability possibly due to its stochastic velocity and position update equations. This inherent randomness contributes to larger confidence intervals, suggesting that while PSO can reach near-optimal solutions, its performance is less stable. Further tuning of PSO parameters may reduce variance and improve solution reliability.

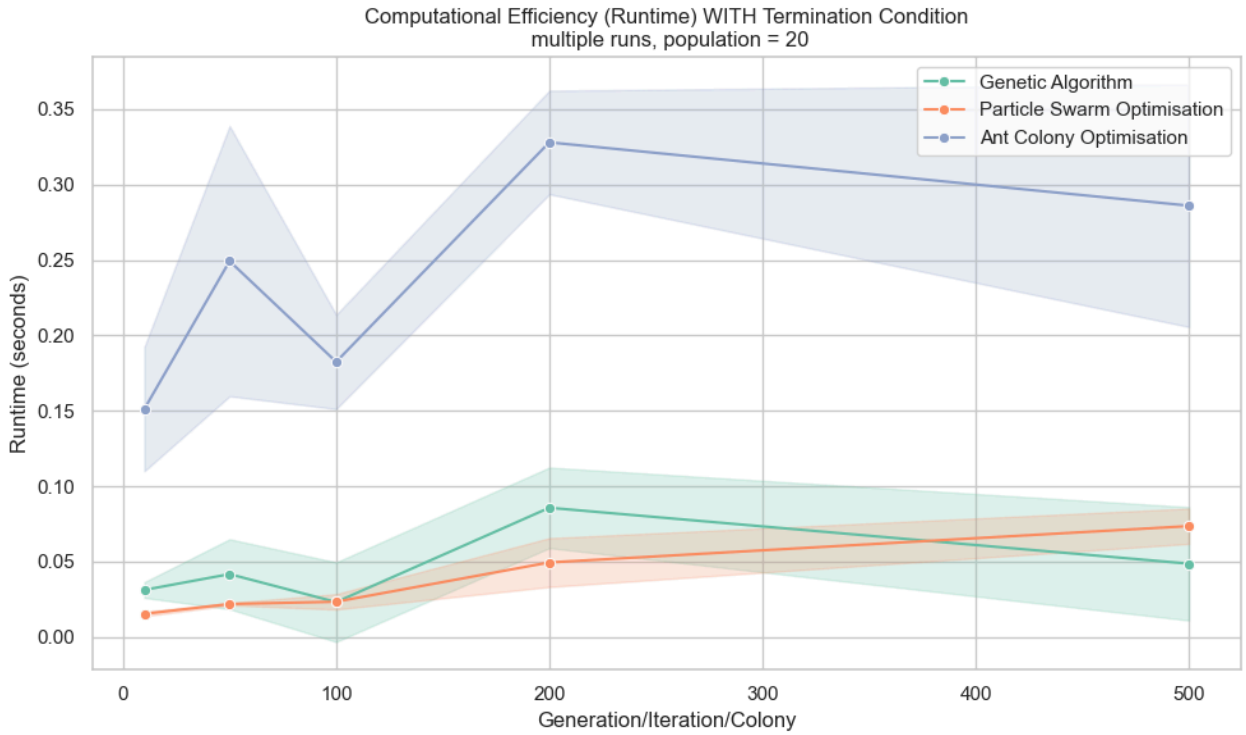


A convergence analysis was conducted using a single seeded run for each algorithm, with a population size of 20 and 100 generations. The fixed seed ensures reproducibility. In all cases, the Termination Condition triggered at Generation 20, causing the algorithms to stop early. This early termination pattern was consistent across multiple tests.

Observations showed that PSO tended to converge toward improved solutions later than GA and ACO. This behaviour is consistent with previous performance plots, where PSO exhibited a higher final objective cost and greater variance.

3.2 Computational Efficiency

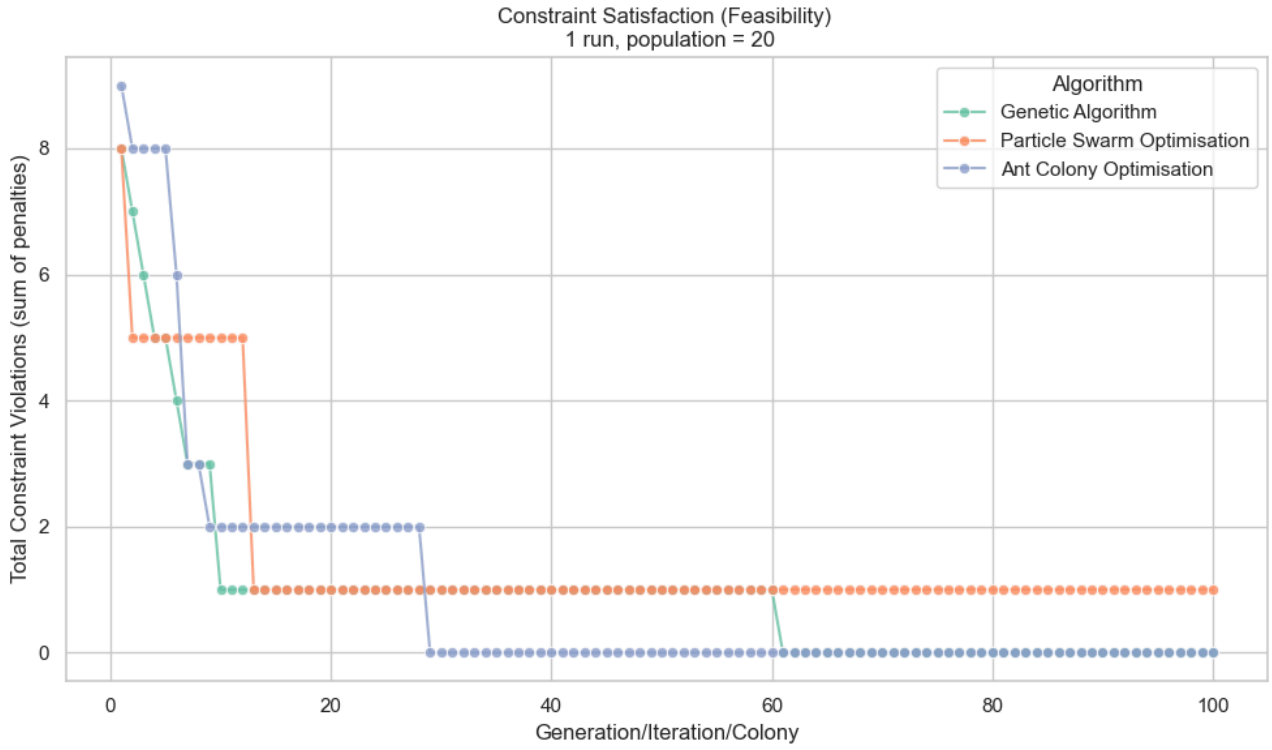




To assess Computational Efficiency, we created two plots showing the runtime in seconds for each algorithm across multiple runs, with means and confidence intervals plotted to account for variability. The first plot represents runtime without the Termination Condition applied, while the second plot includes early stopping based on the Termination Condition. A clear reduction in runtime was observed when the Termination Condition was active, all algorithms were significantly faster showing they all terminated significantly earlier than the maximum allowed generations. ACO exhibited the largest relative reduction - we believe this is because ACO has less inherent randomness and biases good pathway solutions early.

The trade-off between solution quality and computational cost is less evident as the number of generations does not necessarily mean a better solution and even without the Termination Condition, the longer runtime for 500 generations may not produce a much better solution than 100 generations.

3.3 Constraint Satisfaction (Feasibility)



In the constraint violations graph, all three algorithms have decreasing violations over generations and different trajectories: The Genetic Algorithm has a quick reduction of constraint violations at the beginning, and by generation 20 there is a high level of feasibility. Ant Colony Optimisation shows consistent improvements in feasibility, with the pheromone mechanisms of the ACO effectively directing the solutions toward satisfying the constraints. Particle Swarm Optimisation shows a slow but still

consistent improvement in constraint satisfaction compared to the GA and ACO, which directly relates to the nature of the PSO and its convergence.

By the early termination of the algorithms (around generation 20) the GA and ACO have eliminated almost all constraint violations, while the PSO still has constraint violations present and could benefit from continuing for additional generations to achieve the same level of feasibility exhibited by the other two algorithms.

4. Results and Discussion

4.1. Comparison

Comparing performance across our metrics revealed clear differences between the algorithms:

Metrics	**Genetic Algorithms (GA)**	**Particle Swarm Optimisation (PSO)**	**Ant Colony Optimisation (ACO)**
Solution Quality	Consistently achieved cost of 0.00 within 100 generations	Found good but more variable solutions (average cost 0.25)	Consistently achieved cost of 0.00 within 100 generations
Computational Efficiency (without early termination)	Moderate runtime	Most efficient runtime	Most computationally expensive
Computational Efficiency (with early termination)	Significant runtime improvement; typically terminated around generation 20	Significant runtime improvement; typically terminated around generation 20	Significant runtime improvement; typically terminated around generation 20
Constraint Satisfaction	Quickly eliminated violations	Improved gradually over time	Quickly eliminated violations

The delayed convergence of PSO suggests that the algorithm may require more iterations to reliably reach optimal or near-optimal solutions compared to GA and ACO. Early termination may disadvantage PSO's Optimality. Future work could explore tuning PSO-specific parameters (e.g., inertia weight, cognitive and social coefficients) or adopting adaptive strategies to speed up convergence.

4.2. Insights

Genetic Algorithm: Displayed a quick jump upwards in the first few iterations and then stable performance. A possible weakness could be premature convergence if the elite selection becomes too dominant. Key parameters were the elite percentage of 20% and mutation rate of 0.2, which allowed the GA to conserve generations while keeping a healthy balance of exploration and exploitation. As a future study - we could expand our testing to plot changes in these parameters.

Particle Swarm Optimisation: Displayed good performance with respect to computational efficiency in each iteration, and ease of implementation. Also, PSO struggled with a slower convergence rate, and also showed higher variability in performance. With respect to c1=c2=1.5, cognitive weight and social weight were moderately balanced, while the inertia weight (w=0.5) kept a major role in the trade off between exploratory and exploitative processes in PSO during searching. As a future study - we could expand our testing to plot changes to these parameters.

Ant Colony Optimisation: Displayed consistent quality solutions and good performance in each run, but incurred a higher computational cost when not using termination conditions. The main parameters in ACO were the pheromone evaporation rate (ρ=0.2), deposit constant (Q=0.04), and elitist selection (top25%), which kept a focused search on more promising solutions. As a future study - we could expand our testing to plot changes to these parameters.

By enforcing unique assignments during initialisation, we made sure the algorithms could focus on optimising the remaining constraints, which ended up improving performance of all algorithms. The early termination method helped cut out wasted computation time once the solution quality smoothed out, but it might have held PSO back a bit — given more iterations, it may have found even better solutions.

4.3. Recommendations

For algorithm selection, we recommend GA for scenarios requiring quick, high-quality solutions with consistent performance. When computational resources are limited but slight compromise in solution quality is acceptable, PSO offers a reasonable trade-off. For applications where solution quality is of most importance and we have computational power resources, ACO provides the most consistent optimal results.

Regarding parameter configuration, population sizes around 20 appear optimal for problems of this scale. Early termination mechanisms should be implemented with a threshold of approximately 10% of maximum generations. Algorithm-specific parameters that provided good results include GA's mutation rate around 0.2, PSO's inertia weight near 0.5, and ACO's evaporation rate of 0.2.

Potential improvements to the models include:

- Implementing adaptive parameter control to enhance performance. For example, we could set the inertia weight of PSO high to start and then slow it down as the algorithm runs, this would encourage exploration at the start more and then settle the particles down to focus on exploitation as the iterations get higher.
- Developing hybrid approaches and using two or three of the models on the same problem. An example could be combining GA's rapid initial convergence with ACO's solution quality (tweaking at the end).
- Testing alternative PSO velocity update equations tailored to discrete assignment problems.
- Incorporate the unique assignment flaws into the chromosome, particle, and ant structures. We could do this by using a binary set for each employee at initialisation, so multiple employees could be assigned to the same task from the beginning.

5. Conclusion

The employee task assignment problem was solved by all three evolutionary algorithms, where GA and ACO found optimal solutions with an objective function value of 0.0, and PSO 0.25, almost optimal. Early termination at generation 20 considerably reduced the computational costs while maintaining solution accuracy, demonstrating the effectiveness of stopping criteria that utilize domain knowledge.

Analysis revealed the best combination of solution diversity and computational efficiency existed at a population size of 20. While GA and ACO quickly resolved constraint violations, constraint violations persisted for longer in PSO. The performance of PSO was measured in terms of iterations, taking only a few per iteration but requiring many iterations to converge, while ACO benefited the most from early termination, indicating it quickly detected promising solutions and gravitated toward them.

Future work should direct the following topics:

- Test on larger assignment problems for scalability evaluation
- Implementation of mechanisms for controlling adaptive parameters
- Hybridisation schemes combining strengths of multiple algorithms

All three algorithms solved the problem, but GA and ACO managed to maintain better solution quality and constraint satisfaction, with ACO showing the most consistency across runs. Choice of an algorithm depends on the application requirements with respect to solution quality, computational efficiency, and implementation difficulty.

This assignment also provided an opportunity to collaborate as a team, similar to how teams operate in a professional environment. Throughout the project, we developed skills in using and managing GitHub, resolving coding issues collaboratively, and setting and meeting project goals. Future work will include seeking further opportunities to work on team projects to continue building practical experience and preparing for careers in AI.

References

Albadr, M., Tiun, S., Ayob, M., & Al-Dhief, F. (2020). Genetic algorithm based on natural selection theory for optimization problems. *Symmetry, 12*(11), 1–31. https://doi.org/10.3390/sym12111758

Baeldung. (n.d.). *Particle swarm optimization (PSO) explained*. Baeldung. Retrieved April 26, 2025, from https://www.baeldung.com/cs/pso

Mirjalili, S. (2019). *Evolutionary algorithms and neural networks: Theory and applications.* Springer. https://doi.org/10.1007/978-3-319-93025-1

Dorigo, M., Bonabeau, E., & Theraulaz, G. (2000). Ant algorithms and stigmergy. *Future Generation Computer Systems, 16*(8), 851–871. https://doi.org/10.1016/S0167-739X(00)00042-X

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks*, 4, 1942–1948. https://doi.org/10.1109/ICNN.1995.488968

Rezaee Jordehi, A., & Jasni, J. (2011). Heuristic methods for solution of FACTS optimization problem in power systems. *Proceedings - 2011 IEEE Student Conference on Research and Development, SCOReD 2011*, 30–35. https://doi.org/10.1109/SCOReD.2011.6148703