WARSAW UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRONICS AND INFORMATION TECHNOLOGY

The Evolutionary Algorithm for Solving Bipartite Graph Problem

AUTHOR

Salomea Grodzicka - KD-39

2025-01-23

Abstract

This report presents an evolutionary algorithm to solve the bipartite graph assignment problem efficiently. The problem involves finding an optimal matching between two disjoint sets of nodes, minimizing the total weight of the edges in the matching. The proposed algorithm incorporates key genetic operations, including selection (tournament, roulette, or random), crossover with validation and repair mechanisms, mutation, and elitism. A modular implementation in Python enables flexible parameter tuning, such as population size, crossover probability, and mutation probability, to analyze the algorithm's performance across different configurations. Experimental results on synthetically generated bipartite graphs demonstrate the algorithm's capability to converge to high-quality solutions while balancing exploration and exploitation. The findings highlight the impact of parameter choices and genetic operations on the efficiency and effectiveness of the algorithm.

Contents

1 Introduction

The bipartite graph assignment problem is a fundamental challenge in graph theory and combinatorial optimization. It involves finding the optimal matching between two disjoint sets of nodes, such that the total weight of the selected edges is minimized. This problem has numerous real-world applications, including task allocation, resource management, or network design.

This is a class P problem, so it can be solved by a deterministic algorithm in polynomial time, which means that the time required to solve the problem increases at most as a polynomial function of the size of the input. Traditional algorithms offer exact solutions to the assignment problem. However, these approaches can become computationally expensive when dealing with large-scale graphs or complex constraints. For example, the Hungarian method for the bipartite graph assignment problem has a time complexity of $O(n^3)$, where n is the number of nodes. This motivates the exploration of heuristic and meta-heuristic approaches, such as evolutionary algorithms, which are capable of providing near-optimal solutions within a reasonable computational budget.

Evolutionary algorithms draw inspiration from biological evolution, using mechanisms such as selection, crossover, and mutation to iteratively improve a population of candidate solutions. They are particularly well-suited for problems like bipartite graph assignment, where the solution space is vast and direct computation may be infeasible.

In this report, the evolutionary algorithm is tailored to solve the problem of bipartite graph assignment. It is implemented in Python and incorporates:

- **Selection strategies**: Random, tournament, and roulette selection methods to balance exploration and exploitation.
- **Crossover mechanisms**: Including validation and repair steps to ensure valid solutions after genetic recombination.
- **Mutation**: Randomized perturbations to introduce diversity into the population.
- **Elitism**: Preservation of high-quality solutions across generations.

The report is structured as follows: Section 2 reviews related work and the theoretical background of evolutionary algorithms. Section 3 outlines the methodology and algorithm design, including key genetic operations. Section 4 discusses the details of the implementation and the experimental setup,

followed by Section 5, which presents the results and analysis. Finally, Section 6 concludes with key findings and directions for future work.

2 Background and Related Work

The bipartite graph assignment problem has been extensively studied in the context of optimization and graph theory. A bipartite graph is a graph whose nodes can be divided into two disjoint sets such that every edge connects a node in one set to a node in the other. The assignment problem in bipartite graphs aims to find the optimal matching that minimizes the total weight of selected edges.

2.1 Classical Approaches

Traditional methods, such as the Hungarian algorithm [1], provide exact solutions to the assignment problem and have polynomial time complexity for small and medium-sized graphs. However, these methods become computationally expensive as the size of the graph grows, especially when constraints or additional requirements are introduced into the problem space.

2.2 Metaheuristic Approaches

To address the limitations of classical methods, metaheuristic algorithms such as genetic algorithms [2], simulated annealing, and particle swarm optimization have been applied to assignment problems. These methods leverage randomization and iterative improvement to efficiently explore the solution space. Among these, evolutionary algorithms are particularly popular due to their flexibility and adaptability to various types of problem.

2.3 Evolutionary Algorithms for Graph Problems

Evolutionary algorithms mimic natural selection processes by evolving a population of candidate solutions over generations. They rely on genetic operations such as selection, crossover, and mutation to explore the solution space. For bipartite graph problems, these algorithms must address unique challenges, such as ensuring valid matchings and handling duplicate nodes in offspring.

Recent studies have demonstrated the effectiveness of evolutionary algorithms in solving graph-related problems, including network optimization and resource allocation.

2.4 Proposed Approach

This implementation integrates additional features into an evolutionary algorithm especially designed for bipartite graph assignment.

- Validation and repair mechanisms to ensure valid offspring.
- Flexible selection methods, including random, tournament, and roulette strategies.
- Basic parameter tuning to analyze the impact of various configurations on performance.

3 Methodology

The methodology for solving the bipartite graph assignment problem using an evolutionary algorithm involves the following key steps: population initialization, selection, elitism, crossover, and mutation . These steps collectively guide the population of candidate solutions toward an optimal or near-optimal solution.

3.1 Problem Representation

The bipartite graph consists of two disjoint sets of nodes, A and B, where every edge connects a node in A to a node in B and is associated with a weight. A solution (or *genotype*) is represented as a list of pairs, where each pair (a, b) indicates that node $a \in A$ is matched with node $b \in B$.

To better illustrate this, consider a simple example of a 5×5 bipartite graph with randomly assigned weights between the nodes as shown in the Figure 1. The nodes are divided into two disjoint sets, A and B, and the edges between the nodes represent possible assignments with specific weights.

3.2 Initialization

The initial population of solutions is generated randomly by shuffling the nodes from B and pairing them with nodes in A. This ensures that the population is diverse and covers a broad spectrum of potential solutions.

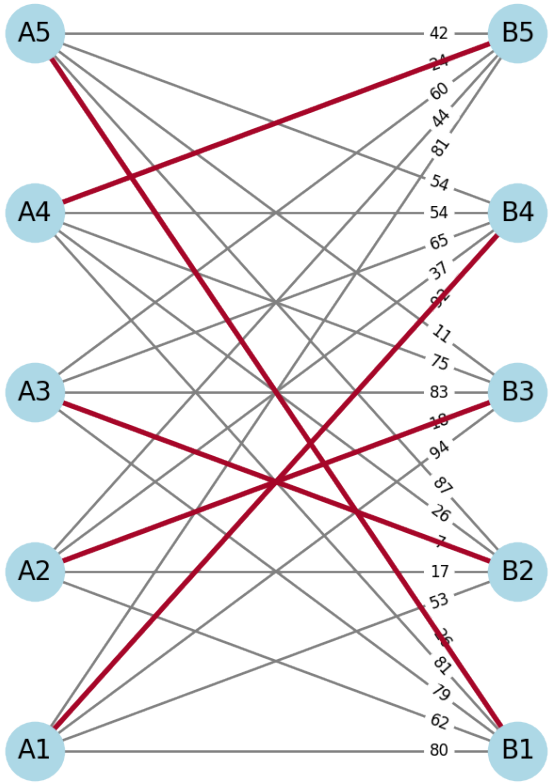


Figure 1: Example of a Bipartite Graph

3.3 Fitness Evaluation

The fitness function evaluates the quality of each solution based on the total weight of the matching Invalid solutions are penalized heavily to ensure the algorithm prioritizes valid matchings.

3.4 Selection

Selection determines which individuals are chosen to contribute to the next generation. Three strategies are implemented:

- **Random Selection**: Chooses individuals randomly, irrespective of fitness, to ensure diversity
- **Tournament Selection**: Randomly selects pairs of individuals to compete, with the better individual proceeding to the next generation (Algorithm 3).

- **Roulette Selection**: Assigns selection probabilities based on fitness, favoring fitter individuals but allowing diversity (Algorithm 4).

3.5 Elitism

Elitism preserves the best solutions across generations by directly copying the top n_e solutions into the next population. Two types of elitism are implemented:

- **Strict Elitism**: Selects the top solutions based on fitness.
- **Partial Elitism**: Selects a subset of the population randomly and applies elitism to the top solutions within this subset.

3.6 Crossover

Crossover combines two parent solutions to create offspring. A random cutting point divides the parent genotypes, and the segments are swapped to generate two children. To ensure valid solutions, a repair mechanism checks for duplicate nodes and replaces them with unassigned nodes.

3.7 Mutation

Mutation introduces diversity by randomly swapping the assignments of two nodes in the solution. The mutation probability p_m controls how frequently this operation occurs. Mutation helps the algorithm avoid premature convergence by exploring new areas of the solution space.

3.8 Algorithm Flow

The algorithm proceeds as follows:

1. Generate an initial population of solutions.
2. Evaluate the fitness of each individual.
3. Apply elitism to retain the best individuals.
4. Use the chosen selection method to determine parents.

5. Apply crossover and mutation to generate offspring.

6. Repeat steps from 2 to 5 for the specified number of generations.

A high-level flowchart of the algorithm is presented in the Figure 2.

3.9 Parameter Configuration

The algorithm allows tuning of several parameters:

- Population Size: Number of solutions in each generation.
- Number of Generations: Limits number of the genetic operations.
- Crossover Probability (p_k): Likelihood of applying crossover.
- Mutation Probability (p_m): Likelihood of applying mutation.
- Elitism Type: Strategy for retaining top solutions.
- Selection Method: Choice between tournament, roulette, or random selection.

These parameters enable experimentation to balance exploration and exploitation, ensuring the algorithm converges effectively.

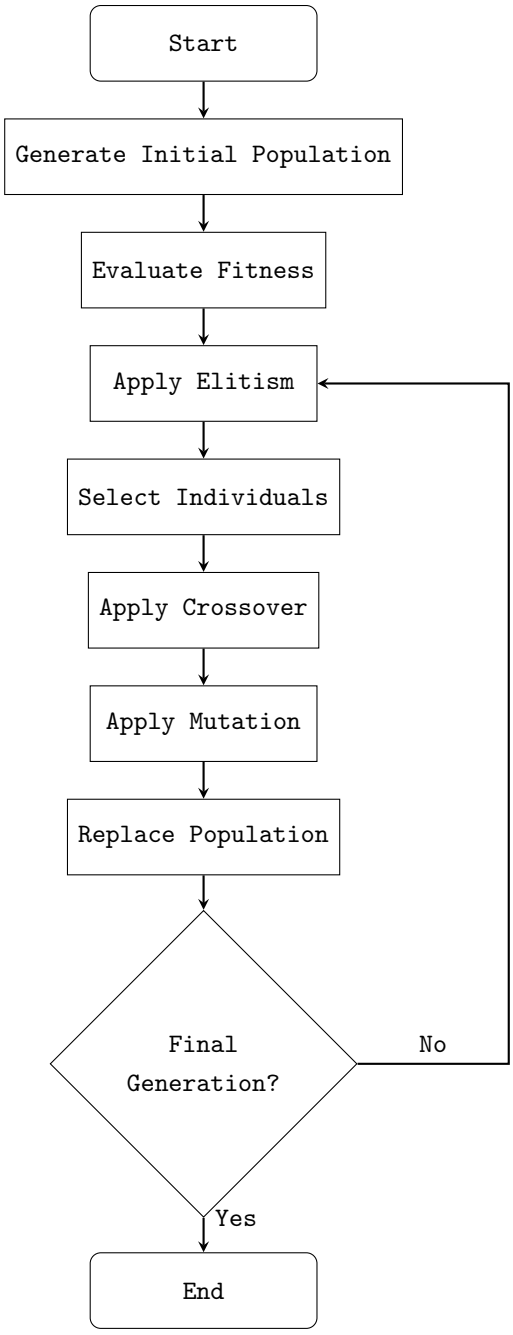


Figure 2: Flowchart of the Evolutionary Algorithm for Bipartite Graph Assignment Problem

4 Implementation Details

The repository for the implementation of this evolutionary algorithm is hosted at: `https://github.com/salomeaaleksandra/EvolutionaryAlgorithm`. It includes:

- Graph Construction Functions: Tools to build bipartite and plot graphs, which are the input for the algorithm.
- Algorithm Class: A Python class implementing the evolutionary algorithm, designed specifically for solving the assignment problem in bipartite graphs.
- Notebook: A Jupyter Notebook (`EvolutionaryAlgorithmTest.ipynb`) to test the algorithm and visualize results, such as the evolution of fitness scores across generations.

Thee pseudo-code for each function involved in the algorithm's initialization, genetic operations, and succession mechanisms is following:

- Initialization generates the initial population of solutions (genotypes) by matching nodes from the two sides of the bipartite graph (refer to Algorithm 1).

Algorithm 1: Initialization of Population

Input: Bipartite graph G with disjoint sets of nodes (`side1`, `side2`), population size `population_size`

Output: Initial population `population` of individuals

```
Initialize an empty population;
for i ← 1 to population_size do
    Shuffle the nodes in side2 randomly;
    Create a matching by pairing nodes from side1 with shuffled nodes from side2;
    Add the matching to population;
end
return population
```

- The fitness function evaluates each genotype by calculating the total weight of valid edges and penalizing invalid connections (refer to Algorithm 2).

Algorithm 2: Fitness Evaluation

Input: Genotype `individual` (matching), Bipartite graph `G`

Output: Fitness score `fitness_score` (lower is better)

Initialize `total_weight` $\leftarrow 0$;
Initialize `penalty` $\leftarrow 0$;
Initialize an empty set `used_nodes`;
foreach *edge* (u, v) *in* `individual` **do**
    if v *is in* `used_nodes` ***or*** *edge* (u, v) *does not exist in* `B` **then**
        Increment `penalty` by 1;
    end
    else
        Add v to `used_nodes`;
        Increment `total_weight` by the weight of edge (u, v);
    end
end
`fitness_score` $\leftarrow$ `total_weight` $+$ `penalty` $\times$ `LARGE_PENALTY`;
return `fitness_score`

- Selection mechanism that determines which individuals are chosen for reproduction (refer to Algorithm 3.

Algorithm 3: Tournament Selection

Input: Population of individuals `population`, Fitness scores `scores`

Output: Selected individual `winner`

Randomly select two individuals `individual1` and `individual2` from `population`;
Get the fitness scores `score1` and `score2` corresponding to `individual1` and `individual2`;
if *score1* $<$ *score2* **then**
    `winner` $\leftarrow$ `individual1`;
end
else
    `winner` $\leftarrow$ `individual2`;
end
return *winner*

Algorithm 4: Roulette Wheel Selection

Input: Population of individuals `population`, Fitness scores `scores`

Output: A new population `selected_population` based on roulette wheel selection

Compute `total_fitness` $\leftarrow \sum$(`scores`);
Initialize an empty `selected_population`;
for $i \leftarrow 1$ ***to*** *len(population)* **do**
    Generate a random number `r` in the range $[0,$ `total_fitness`$]$;
    Initialize `cumulative_fitness` $\leftarrow 0$;
    foreach *individual in population* **do**
        Increment `cumulative_fitness` by the fitness score of `individual`;
        if *cumulative_fitness* $>$ *r* **then**
            Add `individual` to `selected_population`;
            break;
        end
    end
end
return *selected_population*

- Elitism that ensures the best-performing solutions are preserved in each generation (refer to Algorithm 5).

Algorithm 5: Elitism

Input: Population `population`, Fitness scores `scores`, Elitism type `type`, Number of elites `elite_count`, Fraction for random selection `fraction_random`

Output: Elite individuals `elites` and their scores `elite_scores`

if *type* = *strict* **then**
    Sort `population` and `scores` in ascending order of `scores`;
    Select the top `elite_count` individuals as `elites`;
    Select their corresponding scores as `elite_scores`;
end
else if *type* = *partial* **then**
    Compute `subset_size` $\leftarrow$ `len(population)` $\times$ `fraction_random`;
    Randomly select `subset_size` individuals and their scores from `population`;
    Sort the selected individuals and scores in ascending order of `scores`;
    Select the top `elite_count` individuals as `elites`;
    Select their corresponding scores as `elite_scores`;
end
else
    return *Error: Unsupported elitism type*;
end
return *elites, elite_scores*

- Crossover which combines segments from two parent genotypes to create offspring.It includes validation and repair mechanisms to ensure valid solutions (refer to Algorithm 6).

Algorithm 6: Crossover

Input: Parent genotypes `parent1`, `parent2`, Crossover probability `pk`, Nodes `side1` and `side2`, Validation flag `if_validate`

Output: Two child genotypes `child1`, `child2` or `None` if crossover is skipped

Generate a random number `r` in the range $[0, 1]$;

if $r > pk$ **then**

  return *parent1, parent2* ; `// Skip crossover`

end

Randomly select a `cutting_point` between 1 and the length of `parent1`;

Create `child1` by combining:

- The first `cutting_point` elements from `parent1`
- The remaining elements from `parent2`

Create `child2` by combining:

- The first `cutting_point` elements from `parent2`
- The remaining elements from `parent1`

if *if_validate* **then**

  Repair duplicates in `child1` and `child2` using `repair_duplicates`;

end

if *child1 and child2 are valid solutions* **then**

  return *child1, child2*;

end

else

  return *None* ; `// Invalid crossover`

end

- Mutation to introduce random changes in genotypes to maintain diversity and avoid premature. (refer to Algorithm 7).

Algorithm 7: Mutation

Input: Genotype `individual`, Mutation probability `pm`, Nodes `side2`, Index of the individual `index` (optional for logging)

Output: Mutated genotype `individual`

Generate a random number `r` in the range $[0, 1]$;
if $r > pm$ **then**
    return *individual* ; `// Skip mutation`
end
Randomly select two indices `idx1` and `idx2` in the genotype;
Swap the values of `side2` at `idx1` and `idx2`;
if *index is provided (optional)* **then**
    Log the mutation details: `index`, `idx1`, `idx2`, and resulting `individual`;
end
return *individual*

- Main evolutionary loop that applies initialization, fitness evaluation, genetic operations (selection, crossover, mutation), and succession until the stopping criterion is met (number of generations is reached).

5 Experimental Setup

5.1 Graph Generation

To evaluate the performance of the evolutionary algorithm, we generated a weighted bipartite graph G with 20 nodes on each side. The graph generation was carried out using the appropriate function from the accompanying implementation.

5.2 Parameter Configurations

Three sets of experiments were conducted to investigate the impact of various parameters on the algorithm's performance. Each configuration tested a different combination of population size,

crossover probability, mutation probability, and selection type. The experiments were executed for 5000 generations.

Experiment 1: Impact of Selection Method and Population size. The parameters:

- Crossover probability (p_k): 0.8
- Mutation probability (p_m): 0.15
- Elitism type: Strict elitism
- Population sizes: 50, 100, and 150
- Selection methods: Tournament, Roulette, and Random.

Experiment 2: Impact of Crossover Probability. The parameters:

- Mutation probability (p_m): 0.15
- Elitism type: Strict elitism
- Population sizes: 150
- Selection methods: Tournament
- Crossover probability (p_k): 0.7, 0.8, and 0.9

Experiment 3: Impact of Mutation Probability. The parameters:

- Crossover probability (p_k): 0.8
- Elitism type: Strict elitism
- Population sizes: 150
- Selection methods: Tournament
- Mutation probability (p_m):
- Mutation probability (p_m): 0.15

5.3 Evaluation Criteria

The algorithm's performance was evaluated using the following metrics:

1. Best Score per Generation: The minimum weight matching obtained by the algorithm in each generation.

2. Convergence Speed: The rate at which the algorithm approaches the optimal solution over generations.

3. Comparison with Hungarian Algorithm: The results were compared with the optimal solution obtained using the Hungarian algorithm.

5.4 Visualization

The experimental results were visualized in the Figures 3 and 4 to show the evolution of the best score across generations for each parameter configuration. The optimal solution (calculated using the Hungarian algorithm) was indicated by a dashed horizontal line in the plots for comparison purposes.

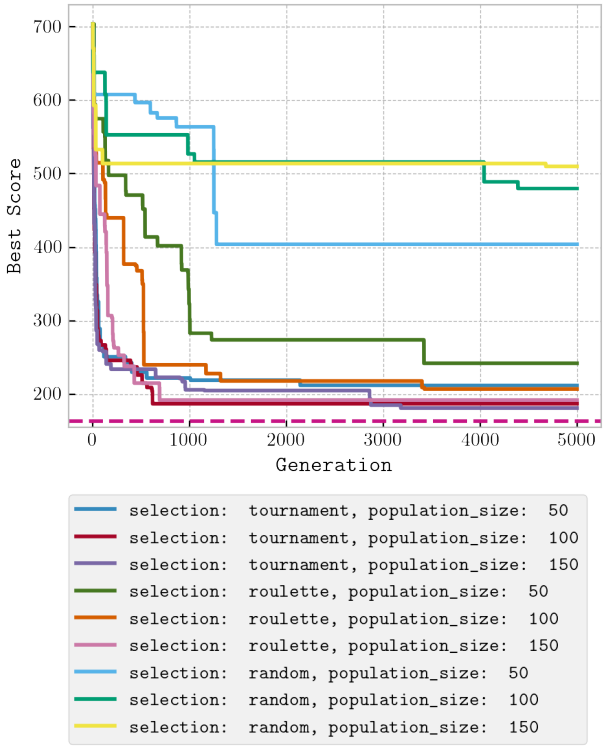
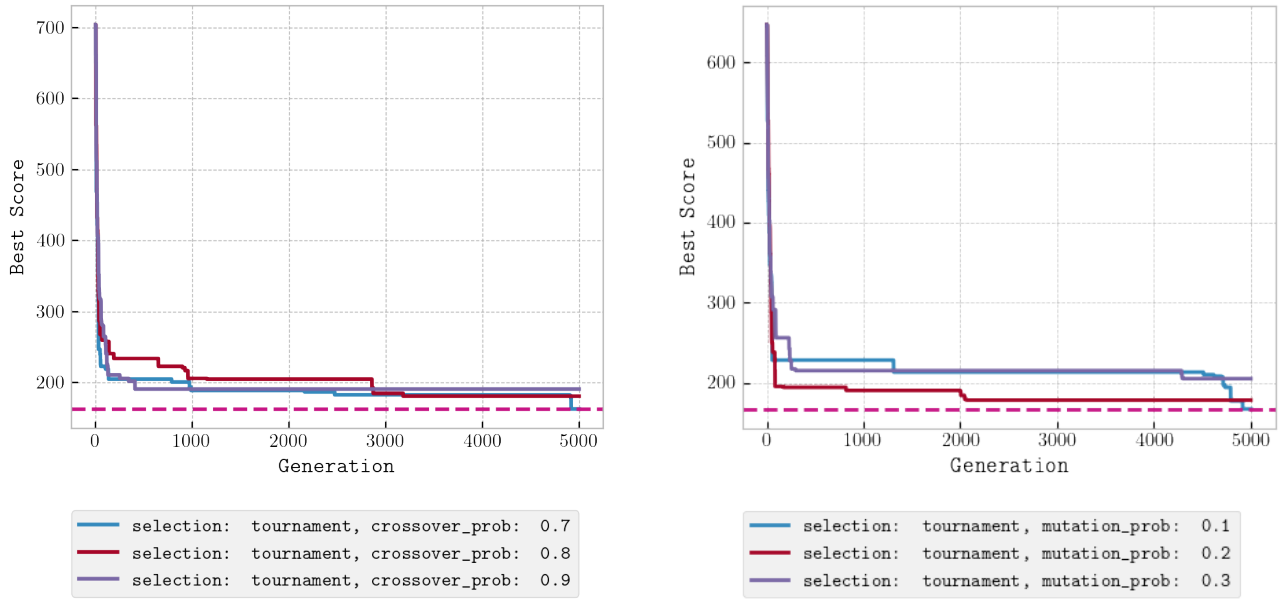


Figure 3: Evolutionary Algorithm results for the Experiment 1



(a) Experiment 2

(b) Experiment 3

Figure 4: Evolutionary Algorithm results for the Experiment 2 (4a) and 3 (4b)

6 Results and Discussion

6.1 Experiment 1

Influence of Population Size and Selection Methods

The Figure 3 illustrates the performance of the algorithm with varying population sizes and selection methods. As observed:

- Random selection failed to optimize the solution effectively, even after 5000 generations. This highlights its inefficiency for this type of problem due to the lack of guided selection pressure.
- Surprisingly, the smallest population size (50) achieved the best results in some cases for random selection. However, this is likely due to randomness rather than systematic behavior.
- Surprisingly, the smallest population size (50) achieved the best results in some cases for random selection. However, this is likely due to randomness rather than systematic behavior.

- The tournament selection demonstrated the best convergence speed and solution quality across all configurations. It outperformed both roulette and random selection consistently.
- For tournament selection, population sizes of 100 and 150 achieved slightly better results and faster convergence than 50 individuals, emphasizing the importance of maintaining diversity in the population.

6.2 Experiment 2

Effect of Crossover Probability

Figure 4a explores the impact of varying crossover probabilities (p_k). Key insights include:

- The lowest crossover probability (0.7) resulted in the best solution in some cases, likely because it preserved more diversity in the population.
- Higher probabilities (0.8 and 0.9) showed faster initial convergence but sometimes resulted in premature convergence, as diversity was reduced too quickly.
- No explicit trend could be drawn between crossover probability and the quality of the final solution, indicating that further iterations may be needed for a clearer understanding.

6.3 Experiment 3

Effect of Mutation Probability

Figure 4b demonstrates the impact of mutation probability (p_m) on convergence and final solutions:

- Similar to crossover probability, the lowest mutation probability (0.1) achieved the best final result in some cases, potentially due to a reduced likelihood of disrupting good solutions.
- The middle value (0.2) yielded the best convergence speed, suggesting that a balanced approach to exploration and exploitation was most effective.
- The highest mutation probability (0.3) maintained diversity but delayed convergence, emphasizing the trade-off between exploration and convergence.

While the evolutionary algorithm achieved near-optimal solutions in most cases, random selection consistently failed to converge to a comparable value. The results validate the evolutionary algorithm's flexibility but also highlight its sensitivity to parameter settings and selection methods.

7 Conclusion

This study demonstrated the potential of an evolutionary algorithm for solving the bipartite graph assignment problem. The key findings are:

- Tournament selection outperformed other methods, achieving the best convergence speed and solution quality, particularly for population sizes of 100 and 150 individuals.
- Random selection consistently failed to optimize effectively, even after 5000 generations, demonstrating its unsuitability for this problem.
- While crossover probability and mutation probability influenced results, no explicit trends were observed. However, the best convergence speed for mutation was achieved with the middle value (0.2), indicating a balance between exploration and exploitation.

What is important to mention, the experiments were very basic and simple. Multiple iterations of the same parameter configurations are necessary to derive more robust conclusions. Moreover, the experiments were conducted on a small 20×20 graph. To validate the algorithm's scalability and robustness, future experiments should involve larger graphs with more complex structures.

The results emphasize the importance of balanced parameter settings and highlight the flexibility of evolutionary algorithms in solving combinatorial optimization problems. However, further work is needed to explore adaptive parameter tuning, hybrid optimization methods, and applications to larger, real-world graphs.

List of Figures

References

[1] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[2] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.