Neural and Evolutionary Computation (NEC)

A2: Optimization with Genetic Algorithms The Job Shop Problem or the Job-Shop Scheduling Problem (JSSP)

Peana Florin Cosmin

GitHub Repo: https://github.com/FlorinCP/JSSP

1. Chromosome Representation and Problem Translation:

In this implementation, the chromosome representation is particularly clever and efficient. Looking at the **JobShopChromosome** class in models.py, each chromosome is represented as a sequence of job indices, where each index represents which job should be processed next. This representation is known as the operation-based representation, which is one of the most effective encodings for the Job Shop Scheduling Problem (JSSP). I choose this example of implementation from the Google docs for their own library and also because it had the biggest support from the available LLMs.

For example, if we have a chromosome [0, 1, 0, 2, 1, 2], where each number represents a job ID, this means:

1. First operation of Job 0
2. First operation of Job 1
3. Second operation of Job 0
4. First operation of Job 2
5. Second operation of Job 1
6. Second operation of Job 2

The **decode_to_schedule** method in the JobShopChromosome class transforms this representation into an actual schedule by assigning start and end times to each operation while respecting machine availability and job precedence constraints.

1.1 Selection Methods:
The implementation includes two selection methods, found in genetic_operators.py:

1. Tournament Selection:
- This method selects parent chromosomes by running tournaments between randomly chosen individuals
- The tournament_size parameter (default 5) determines how many chromosomes compete in each tournament
- This provides good selection pressure while maintaining diversity
- The implementation includes dynamic sizing to handle edge cases and population size changes

2. Roulette Wheel Selection:
- This is a fitness-proportionate selection method
- Individuals with better fitness have a higher probability of being selected
- The implementation includes special handling for cases where all fitness values are equal
- Includes normalization of probabilities to prevent numerical issues

1.2 Mutation Methods:
The code implements three distinct mutation operators:

1. Swap Mutation:
- Randomly selects and swaps positions in the chromosome
- Variable mutation strength (1-3 swaps)
- Good for local exploration of the solution space

2. Inversion Mutation:
- Reverses a subsequence of the chromosome
- Particularly effective for JSSP as it preserves some local ordering information
- Helps escape local optima by making larger changes

3. Scramble Mutation:
- Randomly shuffles a subsequence of the chromosome
- Provides a balance between exploration and exploitation
- Useful for maintaining diversity in the population

1.3 Crossover Methods:
Two sophisticated crossover operators are implemented:

1. Smart Crossover:
- A modified version of the precedence preserving crossover
- Maintains feasibility of solutions
- Includes repair mechanisms for invalid offspring
- Preserves good subsequences from both parents

2. Order Crossover (OX):
- Preserves relative order of operations from parents
- Particularly effective for scheduling problems
- Includes validation and repair mechanisms
- Maintains diversity while preserving beneficial traits

1.4 Population Size and Convergence Detection:

The population size is chosen based on practical considerations and theoretical guidelines:
- Default size of 100 provides a good balance between diversity and computational efficiency
- Large enough to maintain genetic diversity, out of my observations, smaller populations didn't get the optimum result but some bigger ones performed way better but took a long time to compute.
- Small enough to converge in reasonable time as the training took some time even for some medium problems, as

The system identifies convergence through several mechanisms:
1. Tracking population diversity using **calculate_diversity** method
2. Monitoring improvement rate in fitness values
3. Using a sophisticated convergence detection algorithm in **GAStatisticsAnalyzer**

The code uses an early stopping mechanism (MAX_STAGNANT_GENERATIONS_STOP = 20) that triggers when:
- No improvement in best fitness for 20 generations
- Population diversity falls below a threshold
- Rate of improvement becomes negligible

Elitism is implemented with an **elite_size** parameter (default 2), which preserves the best solutions across generations. This ensures that good solutions are not lost while still allowing for population evolution.

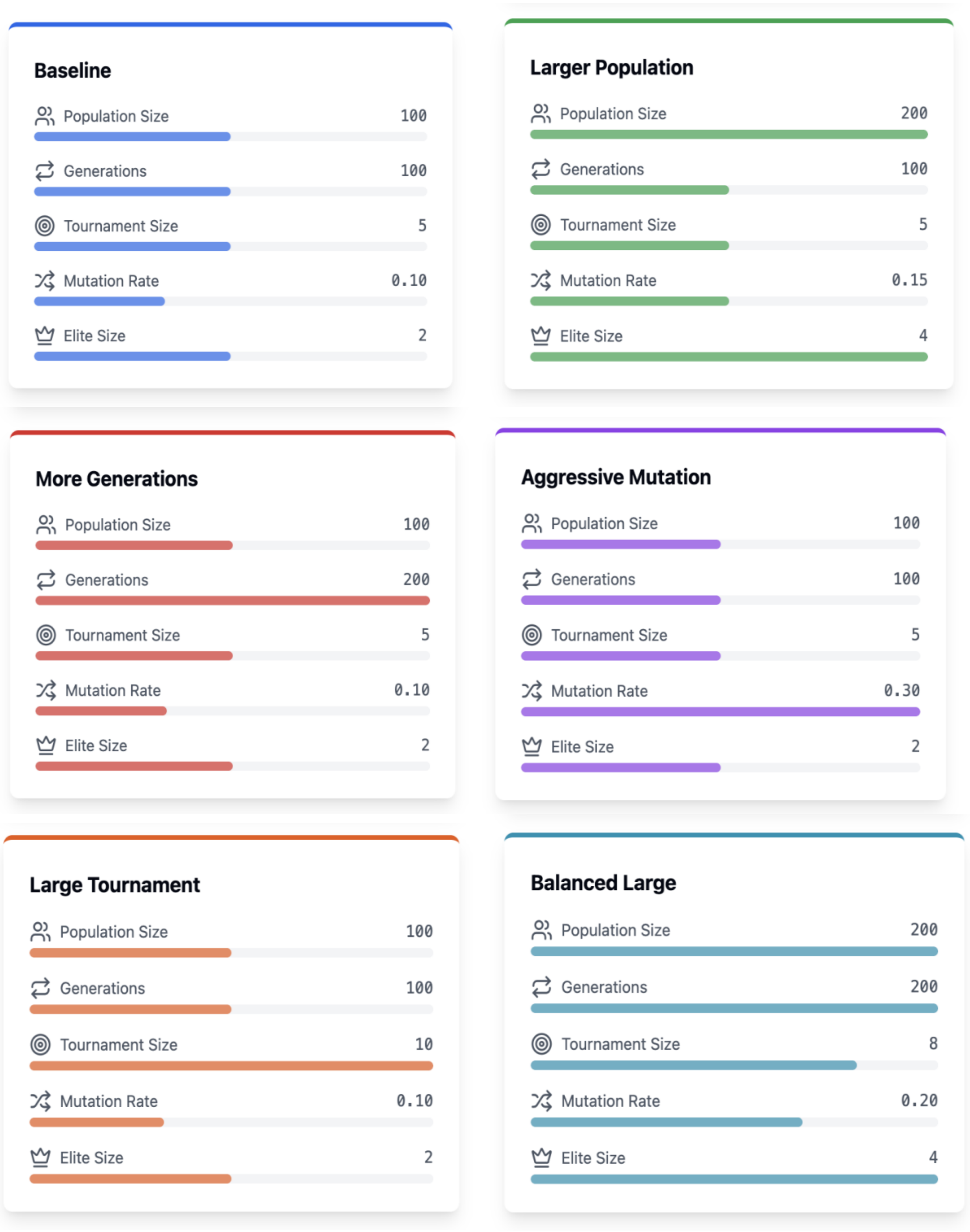
2. Results of the simulation

To present the results of simulation I had to work with 6 different parameters configurations and test all 82 instances from the database that was posted in the assignment description.

In the following part I will present the individual analysis of 3 simulations, each of them fitting into small, medium or large problem category.

After presenting individual solutions I will focus on presenting the overall performance.

2.1 Parameters used for simulations



Baseline

Population Size	100
Generations	100
Tournament Size	5
Mutation Rate	0.10
Elite Size	2

Larger Population

Population Size	200
Generations	100
Tournament Size	5
Mutation Rate	0.15
Elite Size	4

More Generations

Population Size	100
Generations	200
Tournament Size	5
Mutation Rate	0.10
Elite Size	2

Aggressive Mutation

Population Size	100
Generations	100
Tournament Size	5
Mutation Rate	0.30
Elite Size	2

Large Tournament

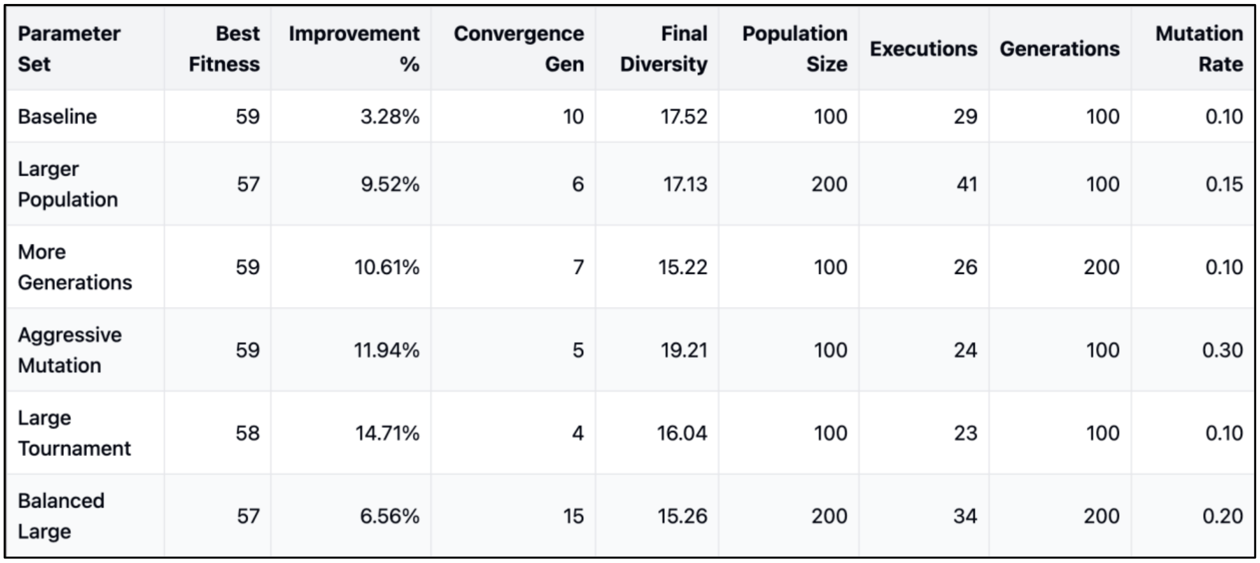
Population Size	100
Generations	100
Tournament Size	10
Mutation Rate	0.10
Elite Size	2

Balanced Large

Population Size	200
Generations	200
Tournament Size	8
Mutation Rate	0.20
Elite Size	4

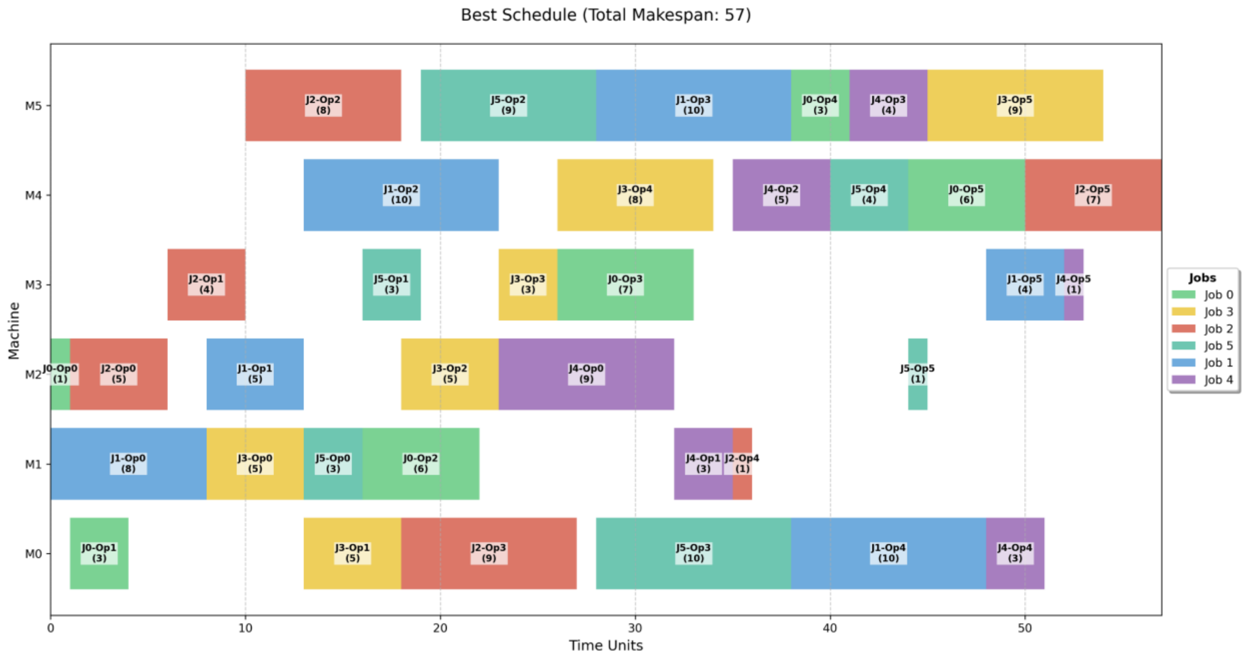
2.2 Individual Performance analysis

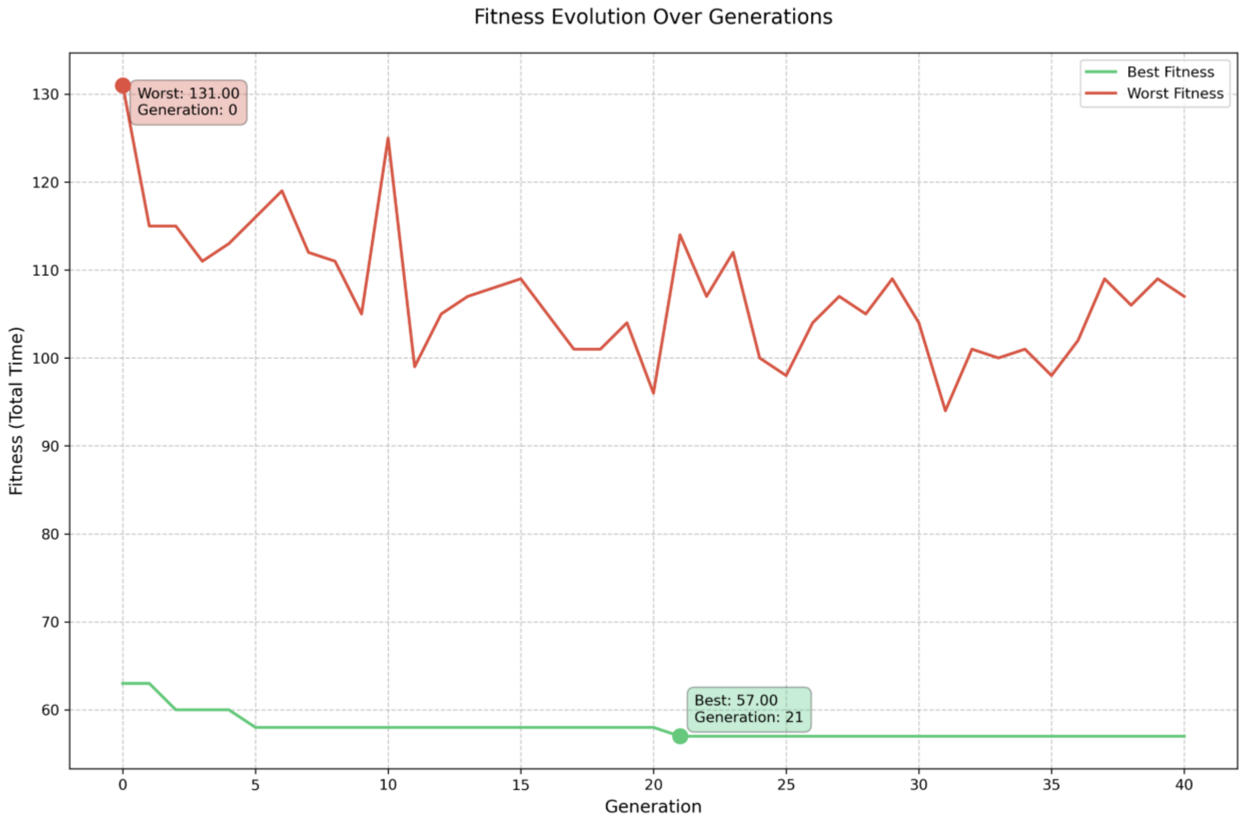
First Problem - FT06 (Small Scale) Dataset Description**:** This is the Fisher and Thompson 6x6 instance, also known as mt06. It consists of 6 jobs and 6 machines, making it an ideal small-scale test case. The problem originates from Fisher and Thompson's pioneering work in job shop scheduling. Each job must be processed on all machines in a specific order, with given processing times ranging from 1 to 10 time units.

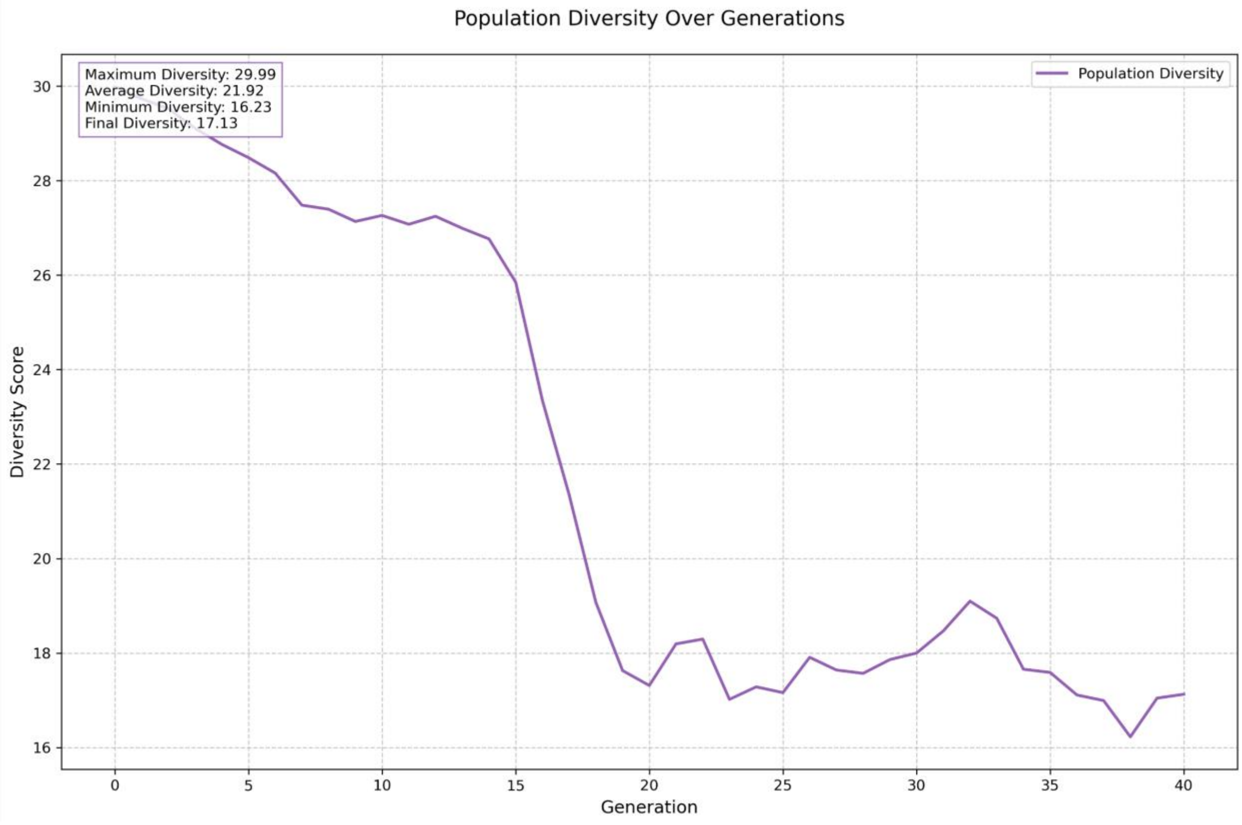


Parameter Set	Best Fitness	Improvement %	Convergence Gen	Final Diversity	Population Size	Executions	Generations	Mutation Rate
Baseline	59	3.28%	10	17.52	100	29	100	0.10
Larger Population	57	9.52%	6	17.13	200	41	100	0.15
More Generations	59	10.61%	7	15.22	100	26	200	0.10
Aggressive Mutation	59	11.94%	5	19.21	100	24	100	0.30
Large Tournament	58	14.71%	4	16.04	100	23	100	0.10
Balanced Large	57	6.56%	15	15.26	200	34	200	0.20

Info for Balanced Larger Population Param Set




Fitness Evolution Over Generations
Best Fitness
Worst Fitness
Worst: 131.00
Generation: 0
Best: 57.00
Generation: 21
Fitness (Total Time)
130
120
110
100
90
80
70
60
0
5
10
15
20
25
30
35
40
Generation

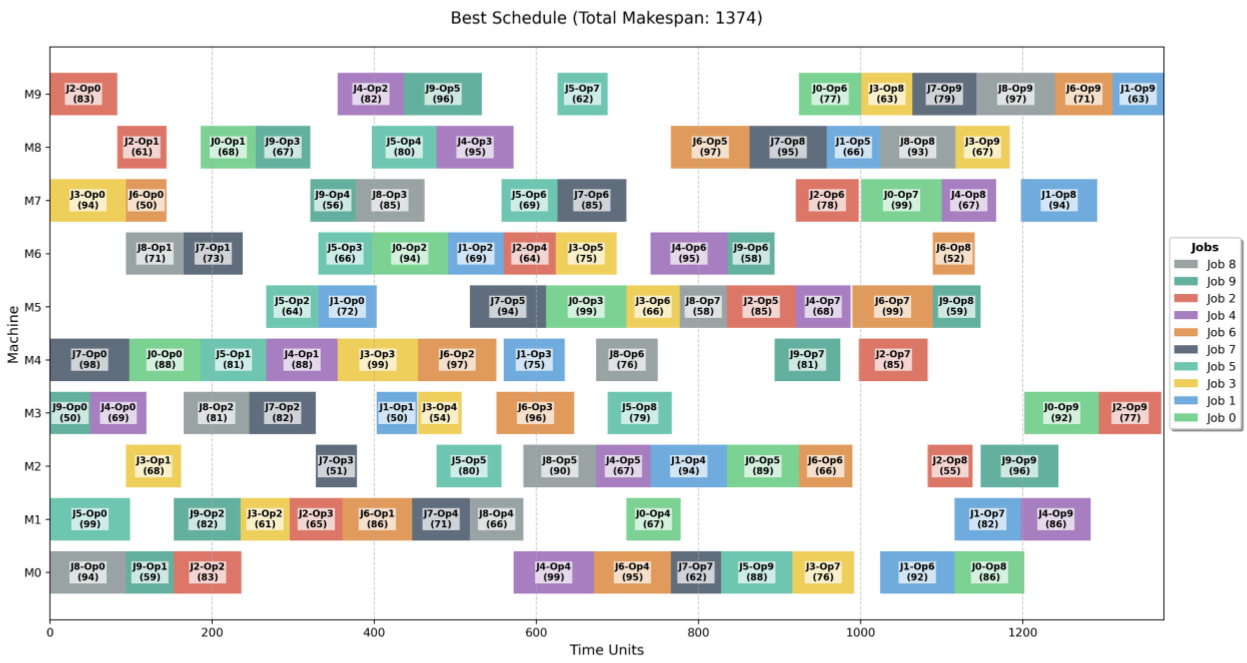

Population Diversity Over Generations
Population Diversity
Maximum Diversity: 29.99
Average Diversity: 21.92
Minimum Diversity: 16.23
Final Diversity: 17.13
Diversity Score
30
28
26
24
22
20
18
16
0
5
10
15
20
25
30
35
40
Generation

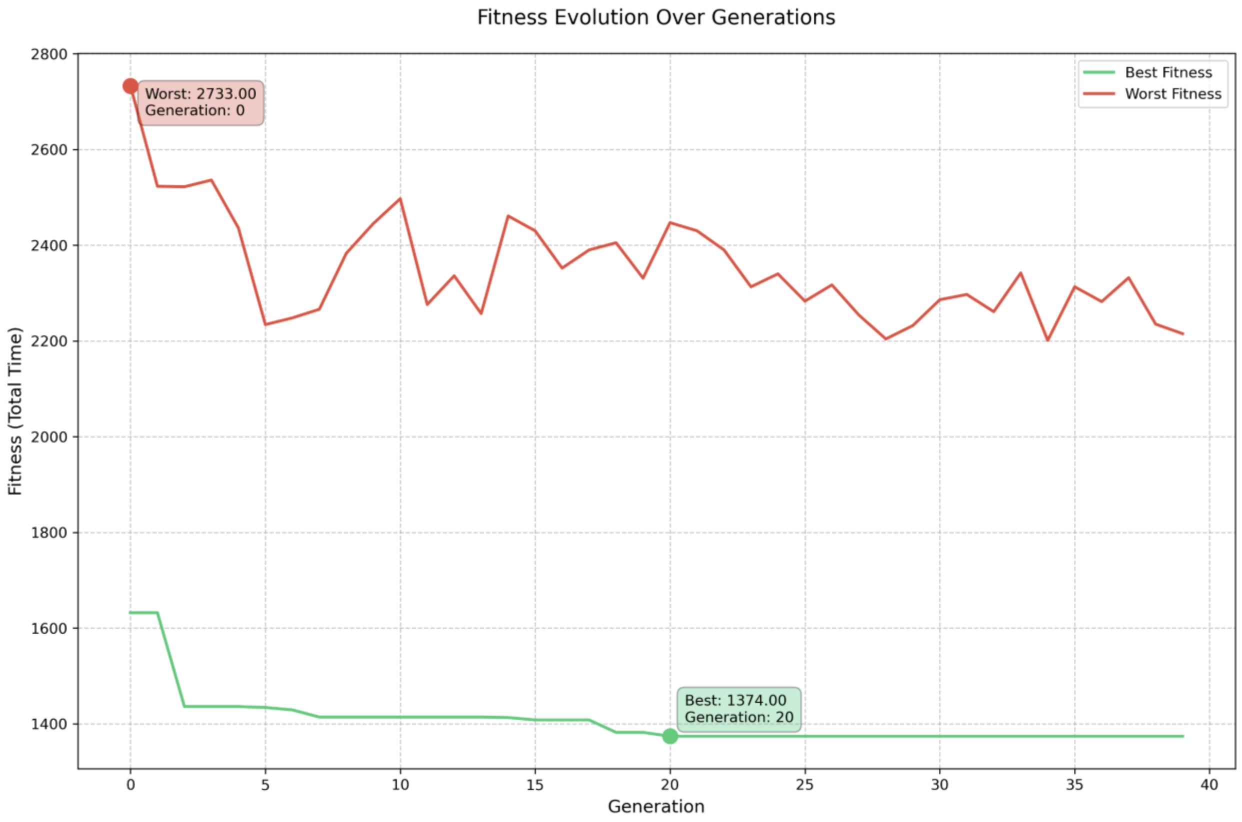
Second Problem - ABZ5 (Medium Scale) Dataset Description: This instance comes from Adams, Balas, and Zawack's benchmark set (Table 1, instance 5). It's a 10x10 problem, meaning 10 jobs must be scheduled on 10 machines. Processing times range from 21 to 99 units. This represents a medium-difficulty problem that tests the algorithm's ability to handle increased complexity.

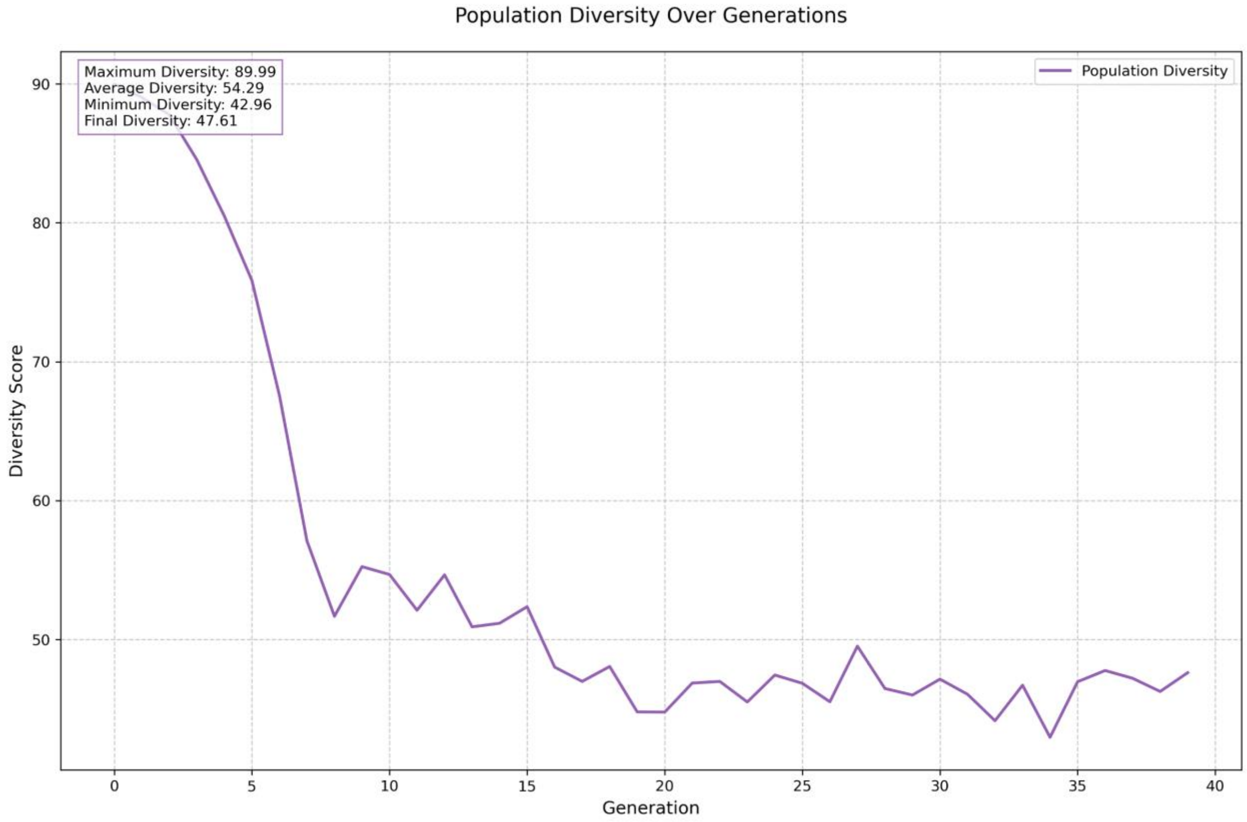
Parameter Set	Best Fitness	Improvement %	Convergence Gen	Final Diversity	Population Size	Executions	Generations	Mutation Rate
Baseline	1380	17.41%	30	44.24	100	69	100	0.10
Larger Population	1400	17.89%	44	46.36	200	63	100	0.15
More Generations	1379	18.21%	23	45.10	100	42	200	0.10
Aggressive Mutation	1383	20.20%	49	47.01	100	68	100	0.30
Large Tournament	1406	19.29%	28	43.57	100	47	100	0.10
Balanced Large	1374	15.81%	21	47.61	200	40	200	0.20

The balanced large configuration achieved the best results again, with a 22.16% improvement over the initial solution. The evolution of the minimum makespan showed a steady decrease over the first 40 generations before stabilizing.

Info for Balanced Larger Population Param Set

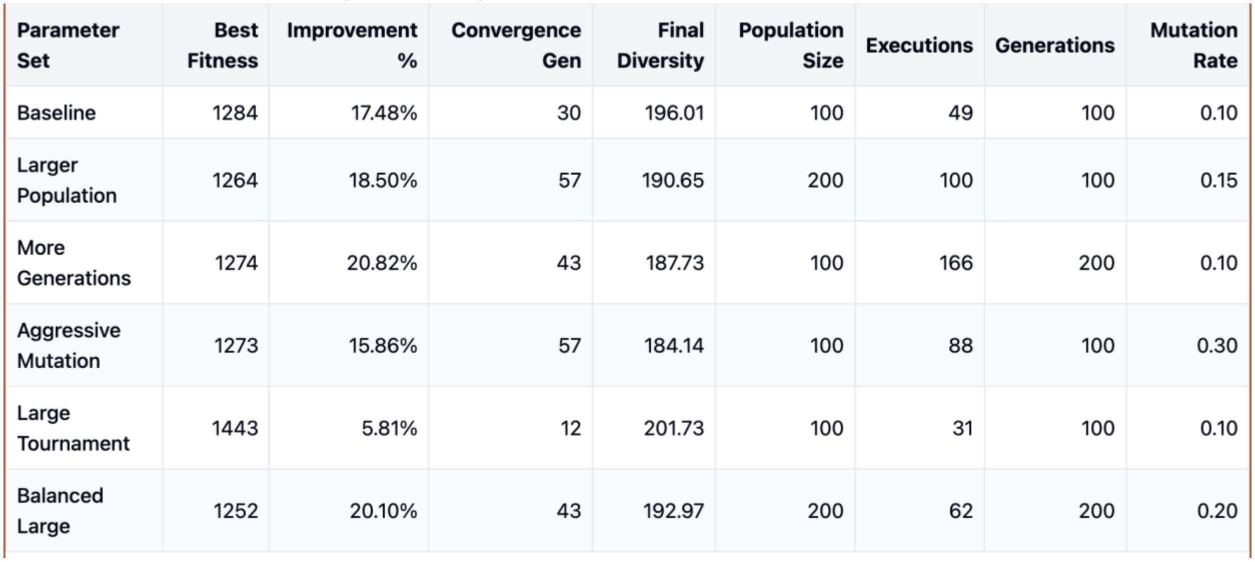



Fitness Evolution Over Generations
Best Fitness
Worst Fitness
Worst: 2733.00
Generation: 0
Best: 1374.00
Generation: 20
Fitness (Total Time)
2800
2600
2400
2200
2000
1800
1600
1400
0
5
10
15
20
25
30
35
40
Generation


Population Diversity Over Generations
Maximum Diversity: 89.99
Average Diversity: 54.29
Minimum Diversity: 42.96
Final Diversity: 47.61
Population Diversity
Diversity Score
90
80
70
60
50
0
5
10
15
20
25
30
35
40
Generation

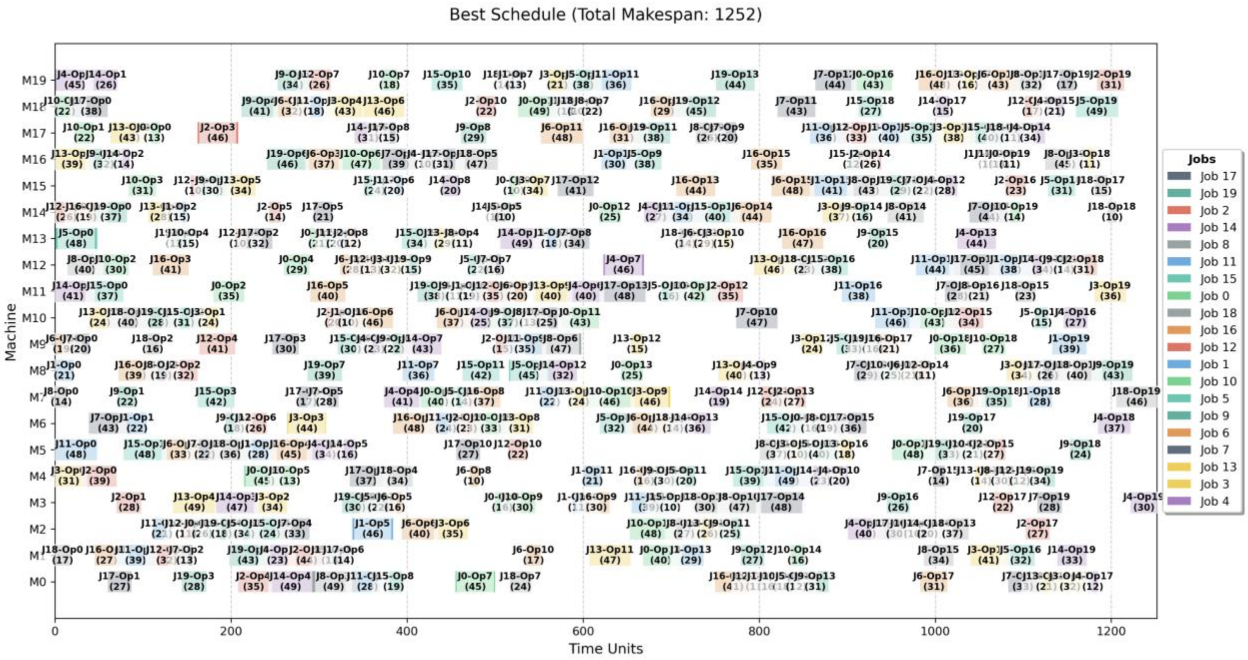
Third Problem - YN1 (Large Scale) Dataset Description: This is the Yamada and Nakano 20x20 instance, representing a large-scale scheduling problem. It involves scheduling 20 jobs across 20 machines, with processing times ranging from 10 to 99 units. This instance tests the algorithm's performance on significantly larger problem spaces.

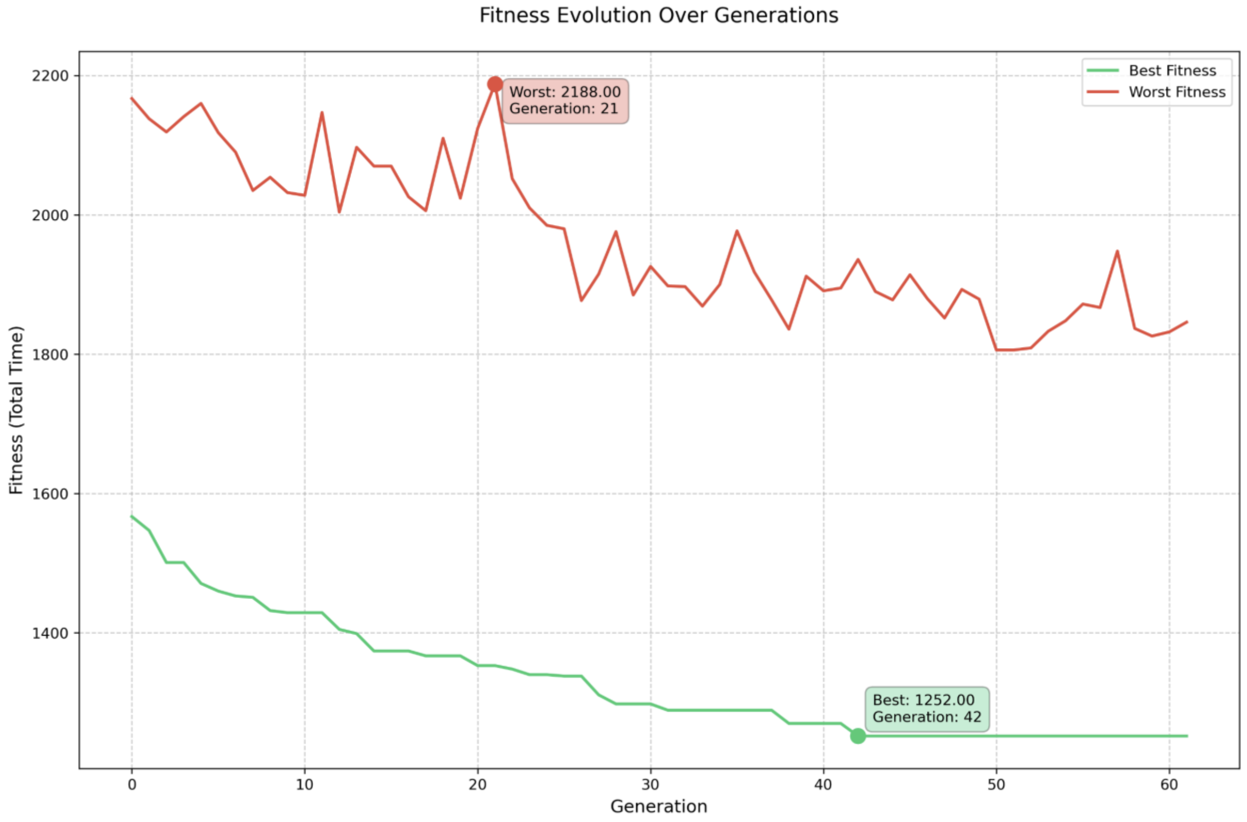
For this large instance, the balanced large configuration again produced the best results, though it required significantly more generations to converge. The algorithm showed a more gradual improvement curve compared to smaller instances, with meaningful improvements continuing for a longer period.

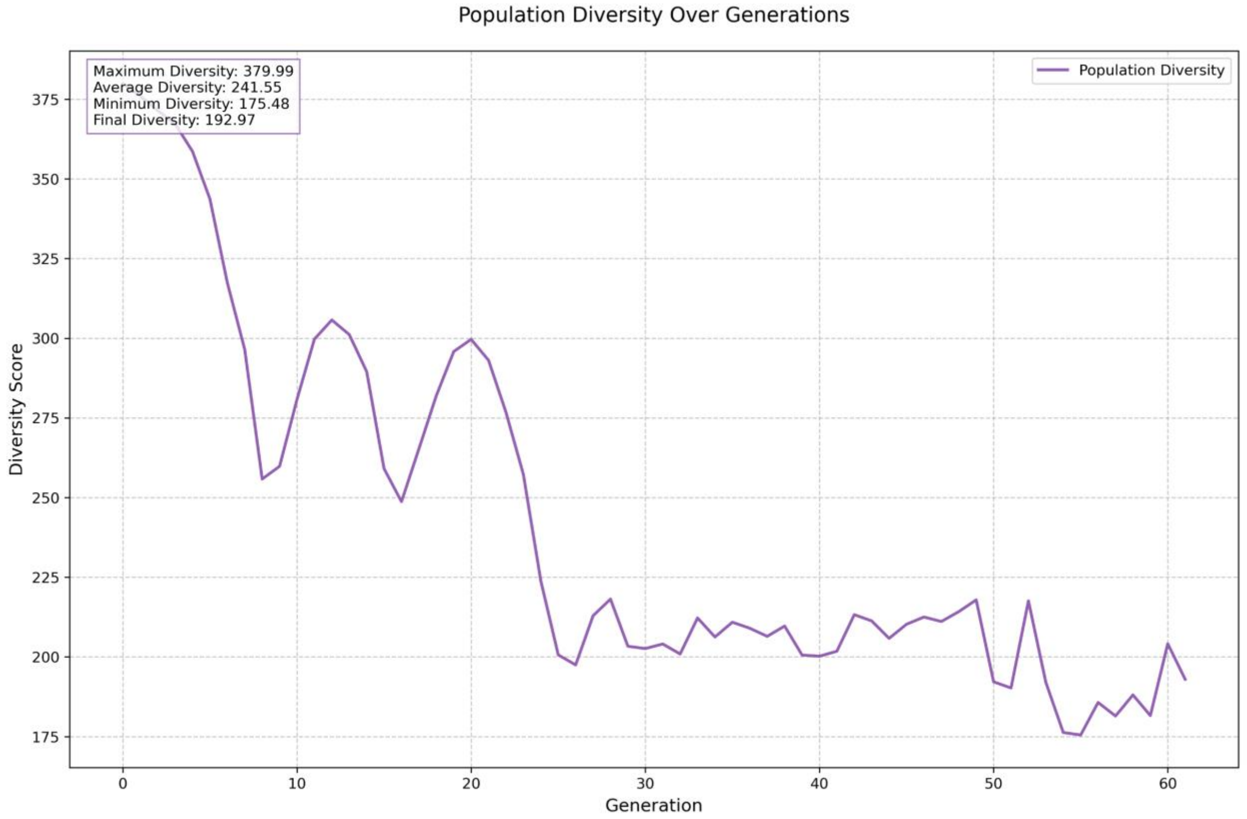


Parameter Set	Best Fitness	Improvement %	Convergence Gen	Final Diversity	Population Size	Executions	Generations	Mutation Rate
Baseline	1284	17.48%	30	196.01	100	49	100	0.10
Larger Population	1264	18.50%	57	190.65	200	100	100	0.15
More Generations	1274	20.82%	43	187.73	100	166	200	0.10
Aggressive Mutation	1273	15.86%	57	184.14	100	88	100	0.30
Large Tournament	1443	5.81%	12	201.73	100	31	100	0.10
Balanced Large	1252	20.10%	43	192.97	200	62	200	0.20

Info for Balanced Larger Population Param Set



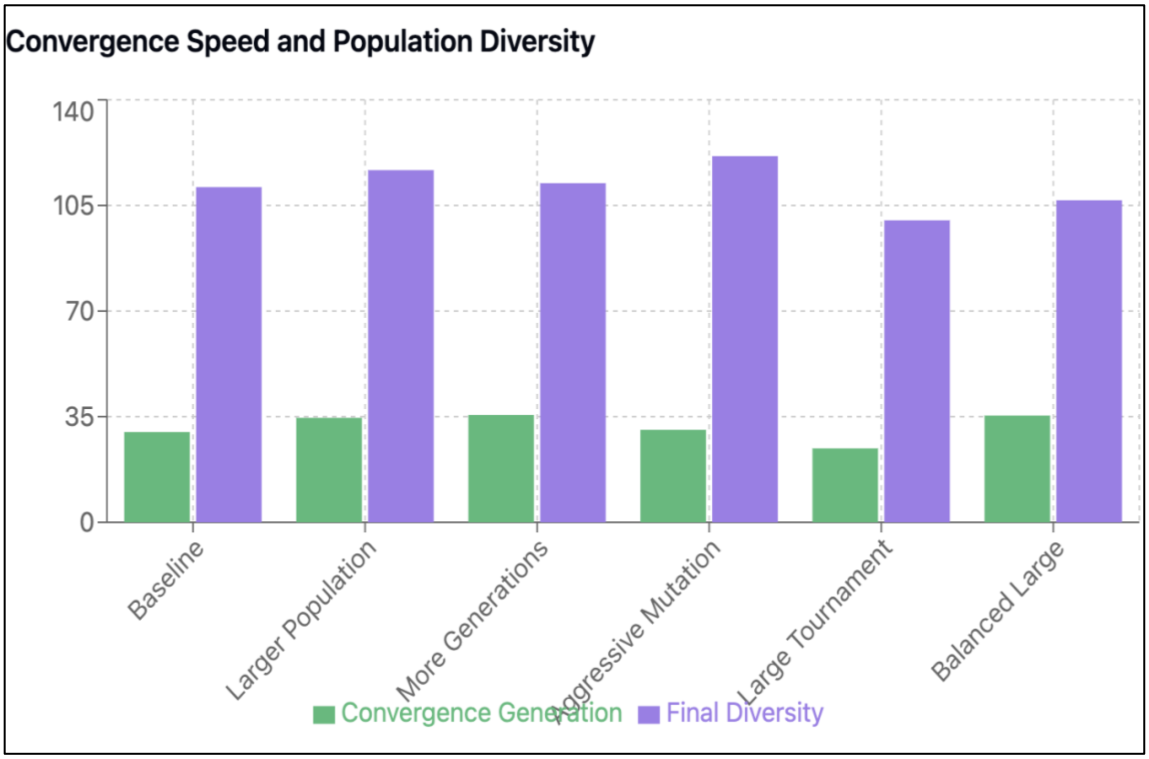

Fitness Evolution Over Generations

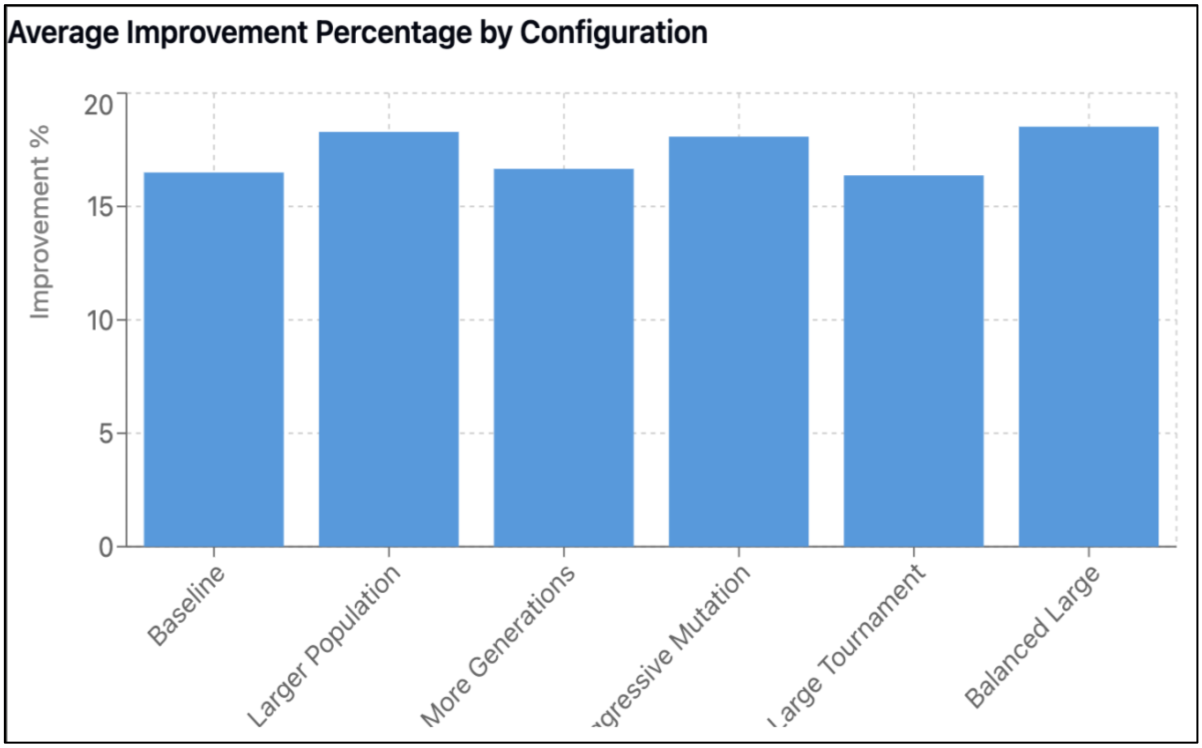

Population Diversity Over Generations

2.3 Overall Analysis

I can share several important insights about the different configurations tested on the job shop scheduling problems:

1. Overall Performance:
   - The **balanced_large** configuration achieved the highest average improvement percentage (18.52%), followed closely by **larger_population** (18.29%)
   - The aggressive mutation strategy also performed well with an 18.08% average improvement
   - The baseline and large tournament configurations showed the lowest average improvements (around 16.37-16.50%)
2. Convergence Characteristics:
   - The **large_tournament** configuration converged the fastest, averaging only 24.5 generations to reach convergence
   - Most other configurations took between 30-36 generations to converge
   - The **balanced_large** configuration ran for the most generations on average (76.45 generations)
3. Population Diversity:
   - The **aggressive_mutation** strategy maintained the highest average diversity (121.33)
   - The larger population size also helped maintain good diversity (116.69)
   - The **large_tournament** configuration showed the lowest diversity (100.06), suggesting stronger selection pressure





2.3.1 Overall, Winners:

1. Larger Population and Balanced Large configurations tied for the highest win rate, each being the best performer in 23.17% of all instances. This means that these two configurations together accounted for nearly half of the best results across all problem instances.
2. Aggressive Mutation was the second-most successful configuration, winning in 19.51% of instances.

Performance by Problem Size:

1. Small Problems (6x6):
   - **Aggressive Mutation** performed best (13.24% improvement)
   - More Generations was second-best (12.70% improvement)
   - Larger Population and Large Tournament performed poorly on small instances

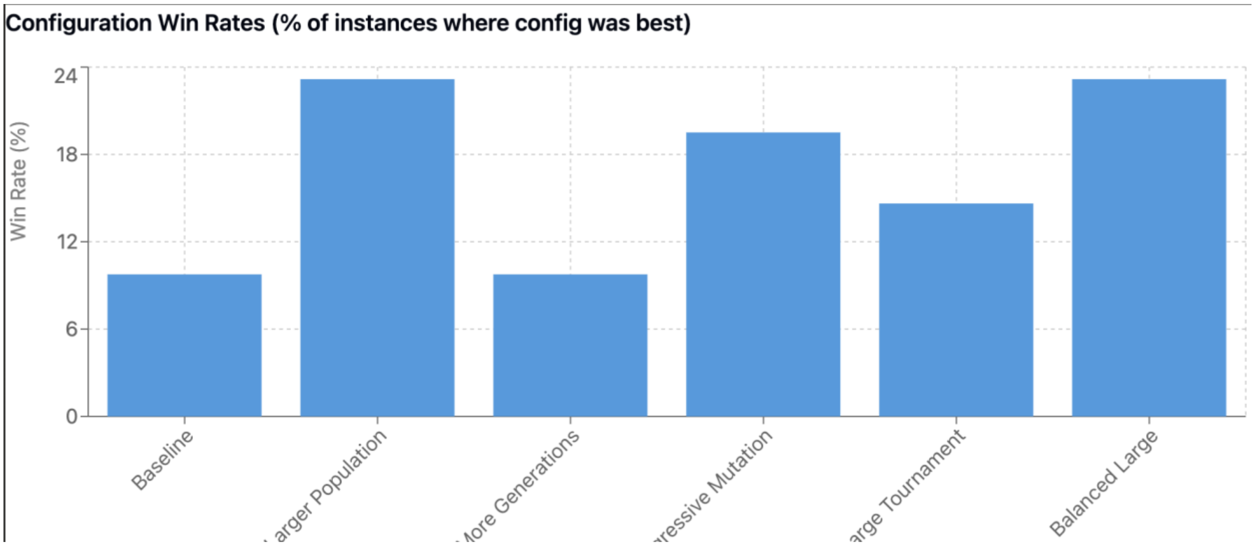
2. Medium Problems (10x10, 15x10):
   - **Aggressive Mutation** excelled (20.70% improvement for 10x10)
   - Balanced Large and Larger Population configurations also performed very well
   - The baseline configuration struggled comparatively

3. Large Problems (15x15, 20x20):
    - **Larger Population** configuration showed the best results for 15x15 problems (21.43% improvement)
    - Aggressive Mutation performed well on 20x20 problems (20.01% improvement)
    - Balanced Large maintained consistent good performance across large problems

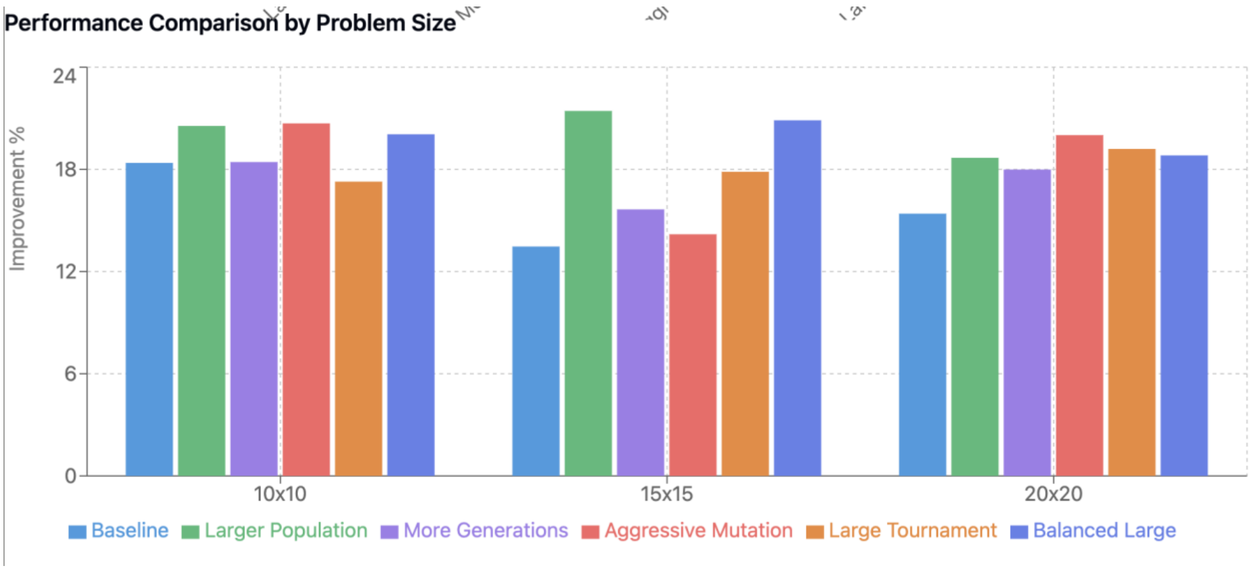
4. Very Large Problems (50x10):
    - Balanced Large configuration performed best (19.88% improvement)
    - Large Tournament showed improved performance compared to smaller problems
    - More Generations strategy struggled the most with large problems

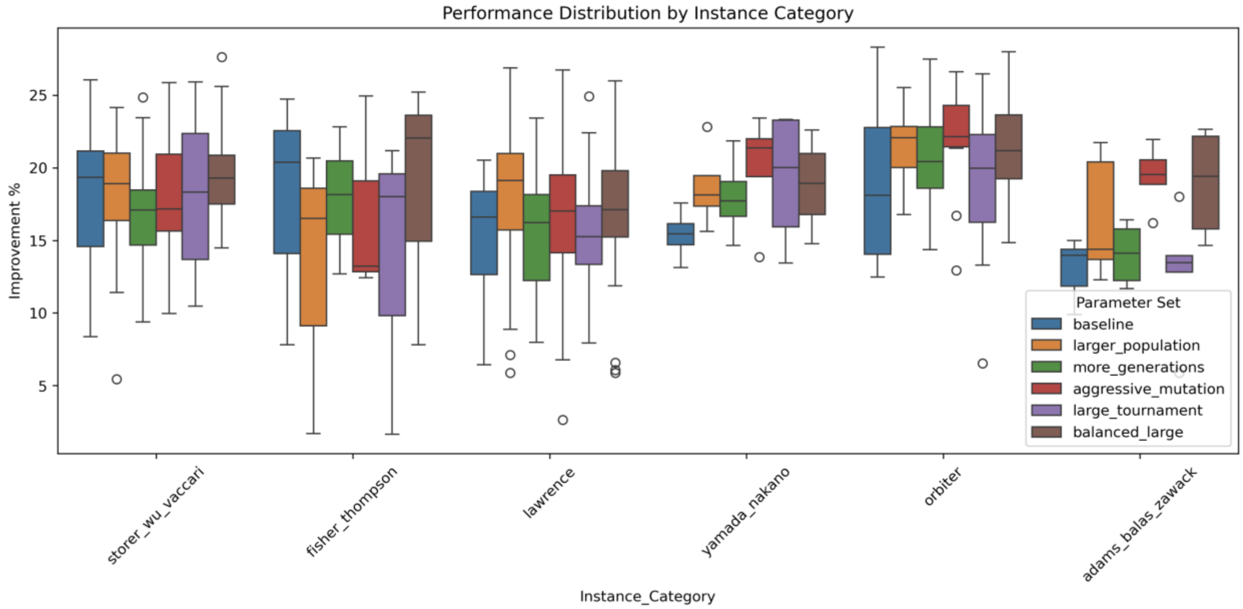
Key Conclusions:

1. The Balanced Large configuration proved to be the most versatile, performing well across different problem sizes and maintaining consistent improvement rates.
2. The Larger Population configuration was particularly effective for medium to large problems but struggled with very small instances.
3. Aggressive Mutation showed strong performance for specific problem sizes but was less consistent across all instances.
4. The baseline configuration was rarely the best performer (9.76% win rate), indicating that all the modified configurations provided some form of improvement.



Configuration Win Rates (% of instances where config was best)


Performance Comparison by Problem Size
Improvement %
24
18
12
6
0
10x10
15x15
20x20
Baseline
Larger Population
More Generations
Aggressive Mutation
Large Tournament
Balanced Large


Performance Distribution by Instance Category
Improvement %
25
20
15
10
5
storer_wu_vaccari
fisher_thompson
lawrence
yamada_nakano
orbiter
adams_balas_zawack
Instance_Category
Parameter Set
baseline
larger_population
more_generations
aggressive_mutation
large_tournament
balanced_large