Matthew Poncini
J.B. Speed School of Engineering
University of Louisville

11 November 2025

—

CSE 545 Final Project

Genetics and Wisdom of Crowds Hybrid Algorithm for Open Shop Scheduling

Abstract— In the previous project, a genetic and wisdom of crowds hybrid algorithm (GA-WoC) was designed to solve the Traveling Salesperson problem (TSP). The hybrid implementation showed better tours, produced in shorter time when compared to the Genetic Algorithm created in Project 4. While this implementation usually provided higher quality tours, it was significantly slower than other methods used to solve TSP. In this project, a genetic and wisdom of crowds hybrid algorithm will be used to solve a series of open shop scheduling problems (OSSP). Applying a wisdom of crowds strategy in creating schedules is not as straight forward as it was when creating tours to solve TSP. Edge frequencies in expert TSP tours give a clear insight to what sub-tours lead to optimal solutions. In OSSP, the "edges" are no longer simple city-to-city links along a single tour.

I. Introduction

This project covers an implementation of a genetic and wisdom of crowds hybrid algorithm to the open-shop scheduling program (OSSP). In an OSSP, multiple jobs have multiple operations and each operation must be completed on a different machine. A machine can only work on one job at a time, and cannot work on an operation belonging to a job that is currently being worked on by another machine. That is, a job cannot be simultaneously processed by two different machines. The time required to complete an operation will be given; each operation has various completion times. The objective to solving OSSP is to create a schedule for each machine that completes each operation of each job in a way that minimizes the time it takes the machine that takes the most amount of time to complete all jobs. In other words, the goal is to create a schedule that minimizes the time for all jobs to be completed. This time amount is referred to as the makespan.

An example of a (near) optimal solution to OSSP is seen in Figure 1. In this example, there are 5 machines annotated on the y-axis. Each rectangle represents an operation associated with a job. The x-axis shows the time that each machine completes a job and it's respective schedule. Notice that a vertical line will not cross a block of the same color twice and that each machine completes each job once. This shows that the requirements of an OSSP solution have been met. The makespan of this schedule can be identified by the time that M1 takes to complete it's schedule. Notice the time M1 spent not completing any job after it complete J4. Perhaps, a more optimal solution places J3 first and J4 last on M1. Of course, this swap has a ripple effecting the schedules on each machine.

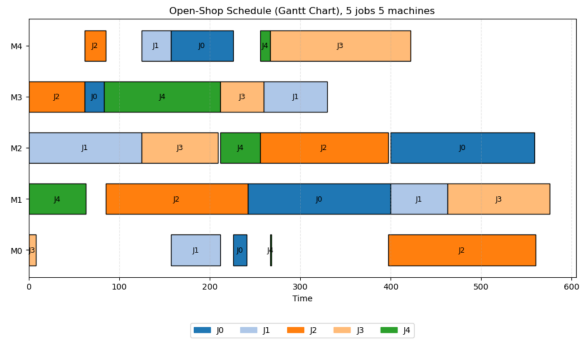

Figure 1: Example of an OSSP Solution. This representation is known as Gantt Chart. This schedule had a makespan of 576.

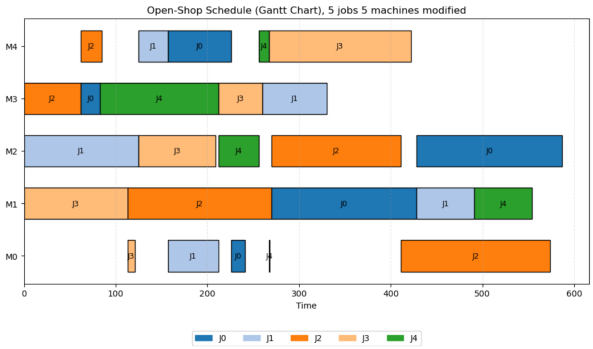

Figure 2: Schedule from Figure 1, but with J3 and J4 on M1 in swapped positions. This schedule yielded a larger makespan of 587, versus the original of 576.

A Wisdom of Crowds (WoC) approach, as presented by Sheng Kung, Michael Yi, Mark Steyvers, and Michael D. Lee in their publication "Wisdom of the Crowds in Traveling Salesman Problems", the authors demonstrate that from suboptimal individual solutions, a collective "crowd" solution could be constructed that often matches or even exceeds the performance of the best individual. This is achieved not through heuristics, but by aggregating structural regularities that emerge across many good, but imperfect solutions. By identifying features (edges in TSP) that consistently appear in high-quality solutions, the WoC method infers which features are most likely to contribute to an optimal solution. New solutions use the crowd-derived fre-

quencies as guidance.

Two close to optimal schedules may appear very different than each other when viewing the Gantt charts. Ultimately, this was the largest obstacle in implementing a GA-WoC hybrid algorithm to solve OSSP. Because the structure of OSSP solutions is highly flexible, collective information from a population need to come from multiple aspects of a schedule. Instead of relying on one aspect of multiple schedules, the hybrid algorithm in this project searched for multiple patterns that recur across good schedules, such as preferred machine-ordering tendencies, consistent early or late placement of certain operations, or temporal clusters that reduce overall congestion. Identifying and encoding these patterns allows the algorithm to bias offspring generation toward regions of the search space associated with strong performance, even when individual schedules look structurally unrelated.

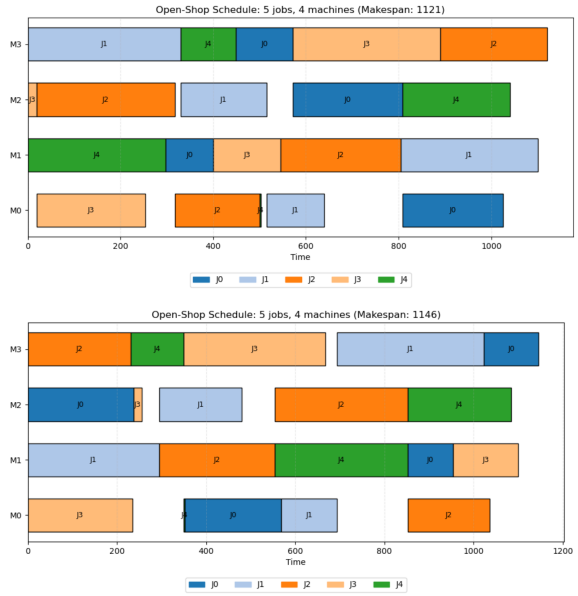

Figure 3: Two solutions to an OSSP with 5 jobs and 4 machines. The two solutions have similar makespans, but don't have many structural similarities in job sequences. This presents the main challenge in using WoC to solve OSSP.

In a traditional GA, each generation is populated entirely by solutions produced through crossover or by clones carried over from the previous generation. Solutions act as parents; they pair off, and pass down structural traits to their offspring using a crossover function. This is where a heuristic is implemented, ensuring that positive traits are passed, allowing the population to gradually drift toward better schedules over time. Clones serve as a stabilizing force, preserving strong individuals so their genetic patterns are not immediately lost.

In the GA–WoC hybrid, the role of clones is expanded. On top of copying high-quality schedules into the next generation, the top solutions act as "experts" whose structure influences a collective model of scheduling tendencies. These expert schedules' machine-order patterns are aggregated into probability distributions that guide the construction of a new, statistically informed offspring. In this way, the hybrid algorithm replaces direct genetic inheritance with a population-level consensus signal. Rather than an offspring receiving traits from two parents of the previous generation, the offspring receives traits from the whole entire expert class. The traits that an offspring receives is probabilistically driven, which in turn allows diversity in the population.

In the heart of WoC, this project does not use any heuristics unique to OSSP when creating new solutions. It does not apply rules, priorities, or domain knowledge to shape schedules based off previous generation scheduling. Instead, it acts as a copier of patterns that already exist in strong solutions. By measuring how often certain job–machine orderings appear among the best individuals, the WoC step builds new schedules that reflect this empirical structure. It does not propose new strategies or explore unseen ordering choices. That exploratory role belongs to the genetic, WoC operators. WoC simply amplifies what the population has already shown to work, serving as a memory mechanism rather than a rule-based heuristic.

II. Methodology

The GA-WoC hybrid is structured liked an ordinary GA. In our experiments, we kept the population size to 40 for the first 3 tests. Ten of the 40 members were clones, the experts from a previous generation. The other 30 members were created using three different WoC driven crossover functions. For the larger OSSP problem sets, all these values doubled. Each generation had 80 members coming from four sources in groups of 20.

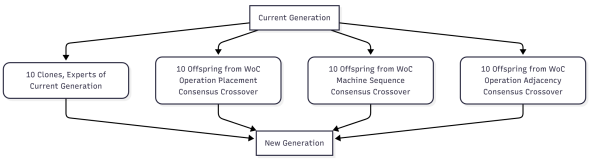

Figure 4: GA Structure, with Clones and WoC Crossover Functions

Initial Population

WoC algorithms rely on expert classes. To create the expert class for the initial population, 100 random solutions to the OSSP instance were generated for smaller OSSP instances, and 200 were generated for the larger OSSP instances. Each solution was represented as an $m \times n$ schedule matrix, where each of the m rows corresponds to a machine and lists the n jobs in the order they are to be processed. Random permutations of these rows produced a diverse set of initial schedules.

For each schedule matrix, the makespan was computed by simulating machine behavior. Machines begin processing jobs in their assigned order as soon as they become available, subject only to the restriction that a job cannot be processed on multiple machines at the same time. When two machines were ready to process the same job simultaneously, priority was given to the machine with the larger remaining workload.

After evaluating all 100 schedules, the 40 with the lowest makespans formed the initial population, and the top 10 were designated as the expert class for constructing the next generation.

WoC Operation Placement Consensus Crossover

The Operation Placement Consensus Crossover is the first of three wisdom-of-crowds mechanisms used to generate offspring schedules. It identified where strong schedules tended to place each job on each machine, then built a new schedule that reflects this collective preference. The method works by analyzing the expert class (the top ten schedules of the previous generation). For every machine and every position in its job sequence, the algorithm counts how often each job appears in that slot across the experts. These frequencies are then converted into probabilities, giving a distribution that reflects the population's collective determination about where each job should go.

Using these probability tables, a new schedule is sampled one machine-row at a time. For each position in a row, the algorithm draws a job according to its consensus probability for that machine and position. This produces a draft schedule that expresses the average tendencies of the experts rather than any single individual. Because sampling can introduce duplicates and in turn omit certain jobs, a repair step follows: duplicates in each machine's row are replaced with missing jobs, choosing replacements that best match the consensus probabilities for that position. This ensures every machine processes all jobs exactly once while maintaining WoC driven schedules.

WoC Machine Sequence Consensus Crossover

Each machine has its own row of job operations, and these rows can differ dramatically even among near-optimal schedules. The WoC Machine Sequence Consensus Crossover focuses on these full machine rows. For every expert schedule, the algorithm looks at each machine and records the complete sequence of jobs assigned to it. Each unique machine sequence is tallied across the expert class. Machines whose rows appear frequently across experts are considered reliable structural components of good schedules.

To produce a child, the algorithm samples a row for each machine with probability proportional to how often that row occurred in the expert class. In other words, if a particular ordering of jobs on Machine 7 appeared five times among the top ten experts, it has five times the chance of being selected compared to a row that appeared only once. After sampling one sequence per machine, these rows are assembled into a full schedule matrix.

Because the sampled rows come directly from expert solutions, the child schedule is already valid and needs little or no repair. The GA then simulates the schedule to compute its makespan. This crossover does not generate structure heuristically; it simply elevates entire machine sequences that repeatedly appear among high-performing schedules. It is a form of pattern amplification rather than pattern invention. It leans on the assumption that strong schedules share machine-specific building blocks worth reusing.

WoC Operation Adjacency Consensus Crossover

This crossover focuses on local job-to-job transitions within each machine's schedule. Instead of copying entire machine rows, this version captures patterns within those rows. It keeps track of which job tends to follow which job when strong schedules are examined. To build this information, the expert class is scanned machine by machine. For each machine, the algorithm tallies how often each job appears in the first position and it tallies every observed adjacency pair within that machine's operation sequence. For example, if a machine has the pattern

$$[3, 1, 2, 4],$$

then it would tally a point for job 3 being in the first position, a point for the transition between job 3 and job 1, a point for the transition between job 1 and job 2, and another for the transition between job 2 and job 4.

These frequencies create a simple probabilistic model for each machine. A job with a higher "start" frequency is more likely to be selected as the first operation. For each later position, the next job is sampled according to the observed adjacency frequencies after the job that was just placed for the respective machine. A child schedule is constructed row by row. For each machine, the first job is sampled from the start-frequency distribution. The second job and following jobs are sampled from the adjacency distribution of the first job and consequent jobs.

Because sampling is based on adjacency counts rather than entire rows, this method can mix and combine local patterns from multiple experts, potentially producing sequences that never appeared entirely in any previous schedule but still reflect the statistical structure of the expert set. After building all machine rows, the child schedule is repaired if needed to ensure that each job appears exactly once per machine, resolving duplicates and filling in missing jobs. Once the schedule is valid, the makespan is calculated.

Stopping Criteria

For smaller OSSP, GA-WoC hybrid algorithm will continue to create new generations until it has completed 100 generations, or 20 generations have passed without an improved solution. For larger OSSP, the stop criteria will be set to 500 max generations and a 50 patience limit.

Test Data

The problem instances used in this study were drawn from the publicly-available Starjob dataset hosted on the Hugging Face Datasets repository. The data set provides nearly 130,000 job-shop scheduling instances, meant to be used to train LLM in job shop scheduling problems. In this project, only 10 problems were used with following job to machine ratios.

number of jobs	number of machines
5	4
5	5
10	10
15	10
15	15
20	20
30	30
30	15
40	40
40	20

Table 1: OSSP Instance Sizes Included in the Dataset

III. Results

Each OSSP instance was attempted to be solved 5 times. The results are compared to a solver that creates n randomly generated solutions and returns the best performing one. For each problem set, the size of n was set so that each the runtime of the random solver would be close to the average run time of the GA-WoC hybrid solver. In each evaluation, there was a chance that the initial population provided a near optimal or optimal solution, in which the GA-WoC hybrid algorithm was unable to improve upon. These cases were ignored in this section.

Test 1: 5 Jobs, 4 Machines

max_generations=100, patience=20,
generation_size = 40

GA-WoC		
Run	**Makespan**	**Runtime**
1	1121	0.445711
2	1146	0.298255
3	1121	0.499967
4	1121	0.343933
5	1132	0.340571
Average	1128.2	0.3856874

Random Solver (3500 samples)		
Run	**Makespan**	**Runtime**
1	1138	0.315915
2	1156	0.397986
3	1121	0.346070
4	1126	0.338912
5	1136	0.563628
Average	1135.4	0.3925022

For the 5-job, 4-machine instance, the GA-WoC hybrid outperformed the Random Solver in average solution quality. Across five runs, GA-WoC achieved the most optimal solution 3 times, where the Random Solver only found the most optimal solution once.

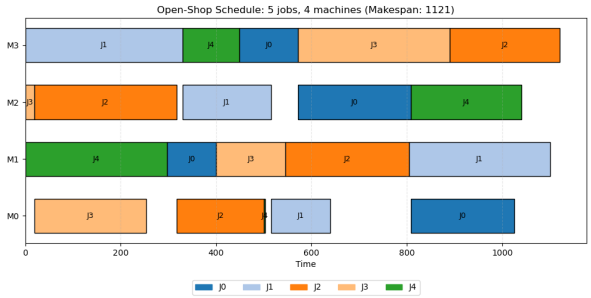

Figure 5: Test 1 Optimal Solution: Makespan 1121

Test 2: 5 Jobs, 5 Machines

max_generations=100, patience=20, generation_size = 40

GA-WoC		
Run	**Makespan**	**Runtime**
1	263	0.276341
2	267	0.403403
3	263	0.342749
4	263	0.711652
5	263	0.302420
Average	263.8	0.407313

Random Solver (2000 samples)		
Run	**Makespan**	**Runtime**
1	263	0.472280
2	264	0.381733
3	264	0.370398
4	264	0.376329
5	263	0.371052
Average	263.6	0.3943584

For the 5-job, 5-machine instance, both methods consistently located near-optimal schedules, but the GA-WoC hybrid showed stronger reliability. GA-WoC reached the most optimal makespan (263) in four out of five runs, while the Random Solver found it only twice.

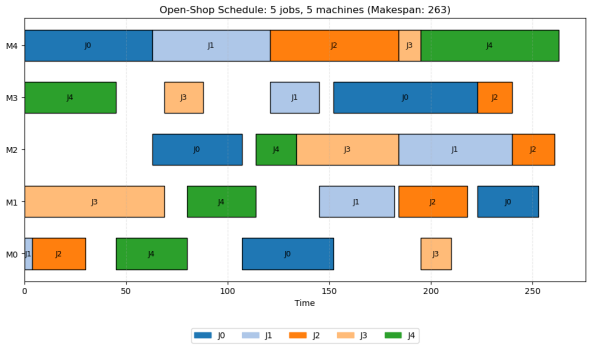

Figure 6: Test 2 Optimal Solution: Makespan 263

Test 3: 10 Jobs, 10 Machines

max_generations=100, patience=20, generation_size = 40

GA-WoC		
Run	**Makespan**	**Runtime**
1	389	3.723007
2	391	2.163649
3	386	1.980231
4	385	2.984135
5	370	4.859111
Average	384.2	3.1420266

Random Solver (3000 samples)		
Run	**Makespan**	**Runtime**
1	395	3.627862
2	389	3.515682
3	400	3.873833
4	402	3.660908
5	391	3.762032
Average	395.4	3.6880634

For the 10-job, 10-machine instance, the GA-WoC hybrid clearly outperformed the Random Solver. GA-WoC found the best overall makespan (370) in one run and produced four additional schedules near that optimum, leading to a substantially better average makespan (384.2 vs. 395.4). The Random Solver never reached the most optimal solution and consistently lagged behind GA-WoC in solution quality.

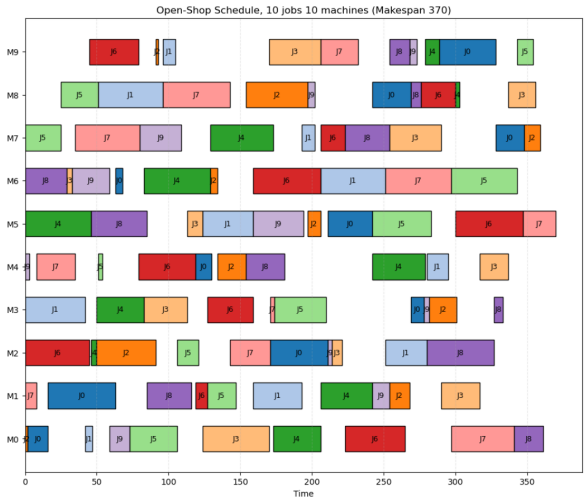

Figure 7: Test 3 Optimal Solution: Makespan 370

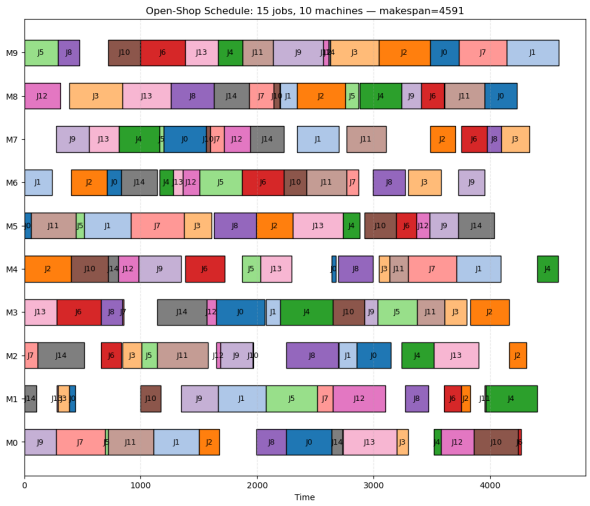

Figure 8: Test 4 Most Optimal Solution: Makespan 4591

Test 4: 15 Jobs, 10 Machines

max_generations=500, patience=50, generation_size = 80

GA-WoC

Run	Makespan	Runtime
1	4591	17.916676
2	5017	6.369663
3	5086	7.679743
4	4917	8.538400
5	4695	10.314707
Average	4861.2	10.1638378

Random Solver (5000 samples)

Run	Makespan	Runtime
1	5080	10.218113
2	4928	10.161585
3	4983	10.298895
4	4966	10.193262
5	4992	10.298457
Average	4989.8	10.2340624

For the 15-job, 10-machine instance, the GA-WoC hybrid showed improved schedule quality. The best GA-WoC run produced a makespan of 4591, while the remaining runs settled into the low-to-mid 5000s, giving an average of 4861.2. The GA-WoC produced lower makespans in most runs and demonstrated greater variability.

Test 5: 15 Jobs, 15 Machines

max_generations=500, patience=50, generation_size = 80

GA-WoC

Run	Makespan	Runtime
1	2472	16.919525
2	2481	10.037199
3	2498	9.485925
4	2366	30.228359
5	2466	12.893874
Average	2456.6	15.9129764

Random Solver (4000 samples)

Run	Makesspan	Runtime
1	2504	16.683621
2	2479	16.628079
3	2505	16.558233
4	2479	16.605502
5	2500	16.605668
Average	2493.4	16.6162206

For the 15-job, 15-machine instance, GA-WoC produced schedules with an average makespan of 2456.6, slightly better than the Random Solver's time-matched average of 2493.4. GA-WoC also found the best individual schedule (2366), while the Random Solver's best was 2479.

Test 6: 20 Jobs, 20 Machines

max_generations=500, patience=50, generation_size = 80

GA-WoC

Run	Makespan	Runtime
1	1705	39.463820
2	1685	36.581432
3	1701	26.851126
4	1688	44.704213
5	1697	70.735887
Average	1695.2	43.6672956

Random Solver (5000 samples)

Run	Makespan	Runtime
1	1709	38.937707
2	1710	45.908644
3	1722	45.475912
4	1733	54.438954
5	1722	46.439773
Average	1719.2	46.240198

For the 20-job, 20-machine instance, the GA-WoC method outperformed the Random Solver in both average makespan and best-found solution. GA-WoC reached an average makespan of 1695.2, compared to the Random Solver's 1719.2, and produced the best individual result (1685). GA-WoC showed more variability due to convergence behavior at this scale.

Test 7: 30 Jobs, 30 Machines

max_generations=500, patience=50 generation_size = 80

GA-WoC

Run	Makespan	Runtime
1	2693	99.855197
2	2649	217.180249
3	2463	668.513197
Average	2601.67	328.5162143

Random Solver (30,000 samples)

Run	Makespan	Runtime
1	2629	2140.093019

For the 30-job, 30-machine instance, GA-WoC produced an average makespan of 2601.67 across three completed runs, with its best solution reaching 2463, substantially outperforming its other attempts. The Random Solver was evaluated once for this large instance due to runtime constraints and produced a makespan of 2629 after an extended runtime exceeding 30 minutes. Figure 9 shows a long, steady descent, with improvements spread broadly across more than 250 generations.

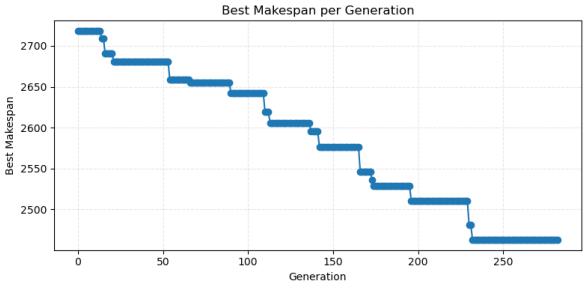

Figure 9: Makespan improvement in third test case

Test 8: 30 Jobs, 15 Machines

max_generations=500, patience=50 generation_size = 80

GA-WoC

Run	Makespan	Runtime
1	4642	170.896380
2	4847	154.603170
3	4803	89.198655
4	4823	79.782209
5	4847	117.395948
Average	4792.4	122.3752724

Random Solver (10,000 samples)

Run	Makespan	Runtime
1	4782	198.974601
2	4825	208.016239
Average	4803.5	203.49542

For the 30-job, 15-machine test, the GA-WoC approach produced stronger schedules than the Random Solver. However, the GA-WoC first trial provided the main difference between the two methods. The other four GA-WoC tries did not come close to this makespan.

Test 9: 40 Jobs, 40 Machines

max_generations=500, patience=50
generation_size = 80

GA-WoC

Run	Makespan	Runtime
1	9056	671.159948
2	9223	233.430000
3	9247	632.020974
Average	9175.33	512.2036407

Random Solver (1000 samples)

Run	Makespan	Runtime
1	9236	261.319000
2	9224	135.972129
3	9310	134.608198
Average	9256.67	177.2997757

In the 40-job, 40-machine test case, the GA-WoC method again outperformed the Random Solver in terms of average makespan. Even though only three runs completed, GA-WoC achieved a lower average makespan (9175 vs. 9257), showing a small but measurable improvement in schedule quality. The GA-WoC method produced stronger schedules, but became increasingly expensive as the job–machine matrix grew.

Test 10: 40 Jobs, 20 Machines

max_generations=500, patience=50
generation_size = 80

GA-WoC

Run	Makespan	Runtime
1	7235	259.761525
2	7272	112.867833
3	7284	220.346000
Average	7263.67	197.6584527

Random Solver (3000 samples)

Run	Makespan	Runtime
1	7322	217.583587
2	7203	209.316935
3	7357	190.313572
Average	7294.00	205.7380313

For the 40-job, 20-machine instance, GA-WoC produced slightly better schedules than the Random Solver, with an average makespan of 7263.7 compared to the Random Solver's 7294. All three completed GA-WoC runs were competitive, though none exceeded the best Random Solver result (7203).

Result Summary

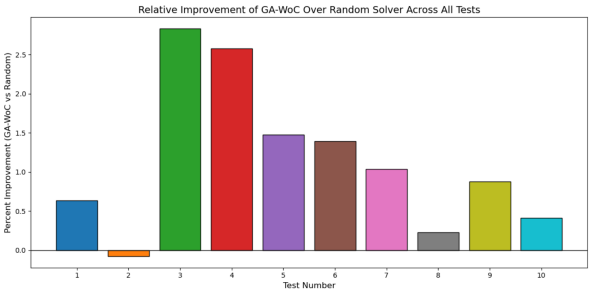

Figure 10: The relative-improvement chart shows how much better the GA-WoC hybrid performed compared to the Random Solver across all ten OSSP test cases. A positive value indicates that GA-WoC produced a lower (better) makespan; a negative value means the Random Solver did slightly better.

Across all ten OSSP benchmark tests, the GA-WoC hybrid consistently matched or outperformed the Random Solver, but its relative advantage depended strongly on the problem size. The largest improvements appeared in the mid-range tests with 10, 15, and 20 jobs, where the search space is complex enough for consensus patterns to form but still manageable for the GA-WoC sampling process. For the smallest instances with only 5 jobs, both methods performed nearly identical. Since runtimes for GA-WoC and the Random Solver were held close to each other, this suggests that GA-WoC is only a slight improvement to just randomly generating solutions in hopes in finding the optimal schedules. In contrast, for the large 30- and 40-job problems, GA-WoC still achieved slightly better average makespans, but the margin of improvement narrowed as the problem size grew and variance increased. Overall, the relative-improvement results indicate that GA-WoC is most effective in medium-scale OSSPs, while remaining competitive even as the scheduling space expands dramatically at the cost of increasing runtime.

WoC Operator Comparisons

Three WoC operators were used to build consensus from existing expert schedules. Each figure below shows the average makespan each operator produced in a generation in four OSSP instances.

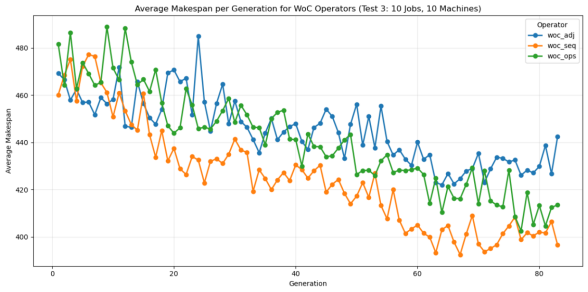

Figure 11: Test 3, 10 Jobs and 10 Machines)

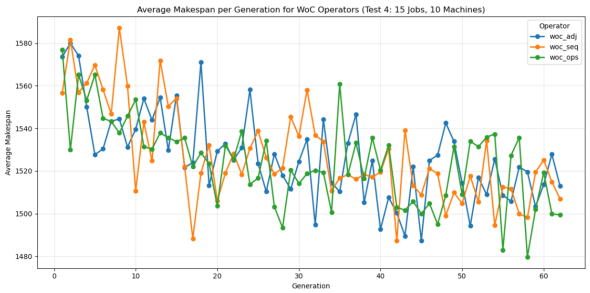

Figure 12: Test 4, 15 Jobs 10 Machines

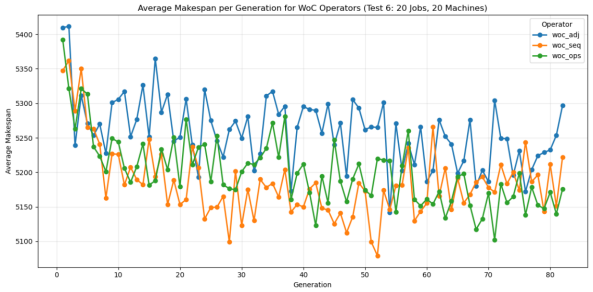

Figure 13: Test 6, 20 Jobs 20 Machines

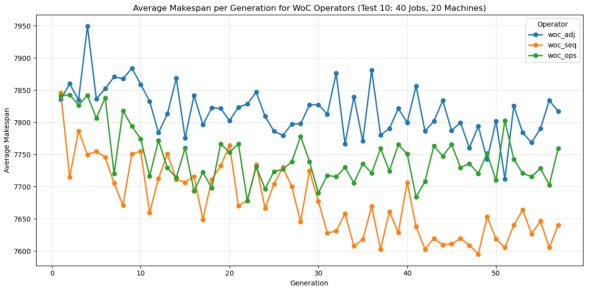

Figure 14: Test 10, 40 Jobs 40 Machines

The graphs give insight to the contributions of each WoC operator, and the overall effectiveness of hybridizing WoC to GA in OSSP. The expectation is to see downward trends for each operator. This is consistent in the woc_seq operator, that builds consensus on individual machine schedules and preserves individual machine schedules among expert shop schedules. The woc_adj operator, which was the most TSP WoC like in that it searched for frequent adjacency between operations in a machine schedule, performed the worse of the three WoC operators. The woc_ops operator was competitive with woc_seq in performance. The woc_ops operator built consensus on the position of an operation in machines schedule. This operators performance did at times produce the best schedule, but was inconsistent.

IV. Conclusion and Discussion

This work applied a GA–WoC hybrid to OSSP and evaluated it against a time-matched random baseline across ten instance sizes. Overall, the hybrid consistently matched or modestly outperformed the random solver, with its clearest advantages appearing on mid-scale problems (10–20 jobs). On very small instances (5 jobs), both approaches behaved similarly; on large instances (30–40 jobs), GA–WoC still edged out the baseline on average but at the cost of higher runtime variance and noticeably greater computational effort. Under equal runtime budgets, the GA–WoC hybrid was only slightly better than aimlessly sampling many schedules, and its advantage narrowed as instance size grew.

The operator-level analysis helps explain this behavior. The woc_seq operator (consensus over entire machine rows) showed the most reliable downward trend in makespan across generations, suggesting that machine-level building blocks are the most transferable structure in OSSP. The woc_ops (positional consensus) was competitive but less stable, and woc_adj (local adjacencies) was weakest.

A conclusion to this project is that WoC signals do exist in OSSP, but are not clear and therefore directly accessible. Strong schedules share machine-order patterns, but it is not in the adjacencies between two operations. Based on these results, a future WoC operator that could be implemented is the job - machine sequence (similar and complementary to woc_seq). This keeps track of the order of machines that each job goes to, rather than the order of jobs a machine runs. This pattern went undetected in this experiment, but could lead to more efficiency in future work.

Ant Colony Optimization and classic metaheuristics such as Tabu Search, Simulated Annealing and Iterated Local Search are baseline methods.

Submitted by Matthew Poncini on 11 November 2025.