Algorithm Selection for Social Golfer Problem Report

Machine Learning for Optimization, WS 2025

Daniel Levin (12433760) Smart Nwamadu (12534844)

1 General Setup

1.1 Implementation Environment

The implementation is developed in Python 3.13. The following external libraries were utilized:

- `numpy` – efficient array operations and numerical computations
- `matplotplib` – experimental plotting
- `pandas` – generation and processing of the datasets (csv)
- `python-sat` – library (`https://pysathq.github.io/`), a Python toolkit providing interfaces to state-of-the-art SAT solvers including Glucose, MiniSat, and CaDiCaL. For solving the Social Golfer Problem, MiniSat (`minisat22`) was used as the default one of this library.
- `scikit-learn` – random forest machine learning approach was used to perform the algorithm selection training and inference - `RandomForestClassifier`. Metrics utilities such as `accuracy_score, classification_report, confusion_matrix` were used for analyzing the performance of the model and `train_test_split` was used for splitting the train and test data.

1.2 Core Classes

The implementation consists of the following main classes:

`SocialGolferSolution` - a representation of a solution for the Social Golfer Problem that implements different construction methods as well as validation which is essential functionality for both testing the exact method's correctness (SAT formulation) and controlling discovering of the search space performed by simulated annealing, which relies on trajectories in the search space that go through invalid solutions.

`SocialGolferInstance` - a parser for SG instances that stores the direct features of this problem.

`SATSocialGolferSolver` - a wrapper for `python-sat` solvers that constructs the cnf (see Section 2.1 for more details).

`SocialGolferScheduler` - a class that handles the execution of Simulated Annealing on given instances of SGP.

`AlgorithmSelector` - a wrapper of `RandomForestClassifier` that exposes functionality such as parsing dataset, traning the model and evaluating it based on different metrics.

1.3 Experimental Setup

1.3.1 Instance Generation

A total of **107 instances** were generated using the `generate_instances.py` script. The instances follow the Social Golfer Problem format with parameters m (number of groups), n (golfers per group), and w (number of weeks).

The ranges of the parameters of this problem that were used are:

- **Required instances**: 8-4-w for $w \in \{6, 7, 8, 9, 10\}$
- **Diverse instances**: $m \in [3, 9]$, $n \in [3, 8]$, with week values $w \in \{3, 4, w_{\max} - 1, w_{\max}\}$

During the generation process we discarded instances where number of groups is less than 2 or number of golfers per group is less than 2 as well as instances with more than one week where number of groups is less than number of golfers per group because such instances are infeasible apriori.

The theoretical maximum number of weeks is computed as $w_{\max} = \lfloor (mn-1)/(n-1) \rfloor$, which represents the upper bound on the number of weeks where each golfer can meet every other golfer at most once.

1.3.2 Algorithm Selection Framework Architecture

The classification dataset was generated by running the SAT solver and the SA heuristic on all 107 instances and recording both the runtime and feasibility results. Each instance was processed with a **60-second timeout** to prevent the algorithms from running indefinitely on particularly difficult instances.

The following algorithm-agnostic features were extracted from each instance:

- `m`: number of groups
- `n`: golfers per group
- `w`: target number of weeks
- `total_golfers`: total number of golfers ($m \times n$)
- `problem_size`: rough complexity indicator ($m \times n \times w$)

Additionally for every algorithm these two columns were added:

- `<algorithm_name>_runtime`: runtime in seconds for executions that have found the assignment of the golfers for the given number of weeks. `NaN` if timed out.
- `<algorithm_name>_result`: number of weeks that have been satisfied but the execution of the corresponding algorithm. The main goal of this column is to capture infeasible solutions which can be identified by our exact solver (SAT). Since we haven't generated small infeasible instances this column was populated with `NaN` only in cases of timeouts.

To enforce the hard timeout constraint, each SAT solver run was executed in a separate subprocess using Python's `multiprocessing.Process`. This approach was necessary because the SAT solvers from `python-sat` wrap C implementation of the corresponding solvers and due to this delegation, the only way to interrupt a subprocess with OS signal is to run it inside dedicated process.

2 Methods Applied

2.1 Exact Method (SAT Solver)

The improved SAT formulation described in Triska and Musliu (2012) was used to solve the SG Problem using SAT as the exact method. Initially we started from trying to manually craft the SAT formulation of Social Golfer problem. After 3-4 hours we've reached a formulation with 4-nested indexing that

accounted for the first two constraints ensuring that each player plays exactly once per week. After having realized that the easiest constraint of the SAT formulation has taken us this long we've decided to turn to LLM's assistance for formulating the remaining constraints. Before doing so we have set up a validation function that would test solutions proposed by SAT to ensure the formulation is correct. When we tried to run the solver on our initial formulation that only ensured once-per-week condition - we could see the invalid parts of the solution in the resulting models thanks to string representation implemented in `SocialGolferSolution` of the following form:

```
Social Golfer Solution: 2 groups x 2 golfers x 3 weeks
         | Week 1   | Week 2   | Week 3   |
---------|----------|----------|----------|
Group 1  | {4, 3}   | {3, 1}   | {2, 3}   |
Group 2  | {2, 1}   | {2, 4}   | {4, 1}   |
```

For formulating the remaining constraints we have used the Claude Opus 4.5 model (Anthropic, 2025). It was fed with the article (Triska and Musliu, 2012) and required to complete the missing constraints. It performed surprisingly well and used the style of our initial code upon which it created the missing constraints. To test the correctness we added the constraints incrementally and observed the results for small instances to see that the violations output by the SAT solver are expected. After combining all constraints and running it on multiple instance (including infeasible ones) we have observed correct behavior of the SAT formulation.

For measuring the performance of SAT solver we used the built-in functionality of `python-sat` that is built to measure the SAT resolution in isolated manner to provide the measurement witht the least overhead. 54 out of 107 instances were resolved by the SAT solver within 60 seconds. To get a better feeling of the effect of the input size on complexity of the problem we generated boxplots that capture the distribution of runtimes and timeouts per instance parameters (see Fig 1). Based on this figure we can clearly see that number of weeks has the strongest impact on the runtime where for $w \geq 8$ none of the instances were solved within 60 seconds and for $w = 7$ only one was solved. In total, 53 out of 107 instances were solved successfully within 60 seconds by our SAT Solver.

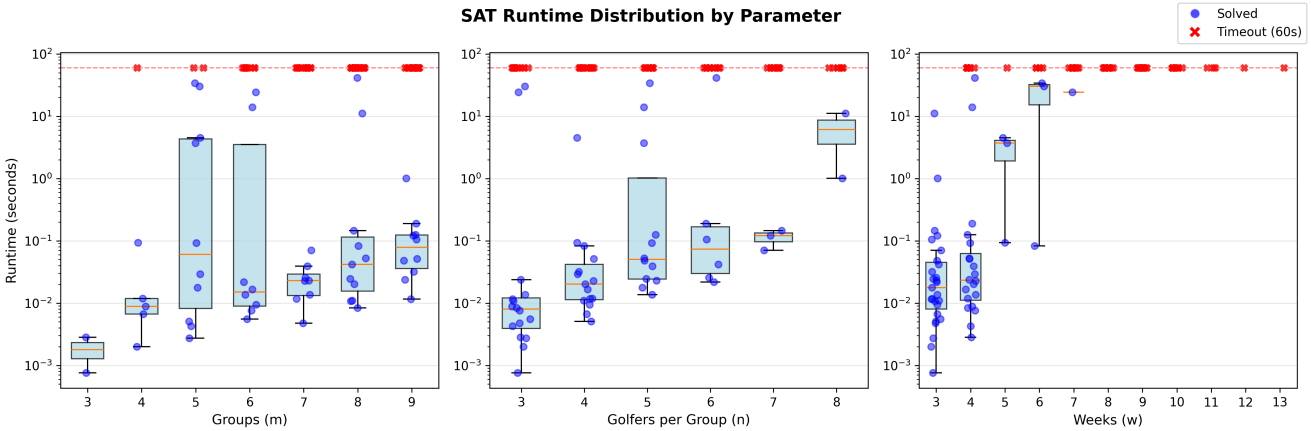

Figure 1: SAT Runtime Distribution

2.2 Heuristic Approach (Simulated Annealing)

Simulated Annealing is a probabilistic technique for approximating the global optimum of a given function. It is inspired by the annealing process in metallurgy. This project implements and evaluates Simulated Annealing (SA) approach to solve the classical Social Golfer Problem (SGP). The implementation successfully finds near-optimal schedules with minimal repeated pairs, demonstrating the effectiveness of heuristic methods for combinatorial optimization problems.

2.2.1 Rationale for Choosing Simulated Annealing

- Effective for combinatorial spaces with many local optima.
- Theoretical convergence to the global optimum with proper cooling.
- Simple to implement with minimal parameters.
- Escapes local minima via probabilistic acceptance.

2.2.2 Solution Representation

- Each golfer represented by an integer 0–31.
- Groups sorted internally for consistent hashing.
- Weekly schedules contain all golfers exactly once.

2.2.3 Objective Function

- Heavier penalty for pairs meeting three or more times.
- Efficient tracking via dictionary.
- Target value: zero (no repeated pairs).

2.2.4 Neighbor Generation

Strategy: Intra-week golfer swap

- Select a random week.
- Select two distinct random groups.
- Swap random golfers between groups.
- All groups remain size four; all golfers remain present.

2.2.5 Cooling Schedule

- Initial temperature: $T_0 = 10.0$
- Cooling rate: $\alpha = 0.995$
- Stopping criterion: maximum iterations or perfect solution.

2.2.6 Core Features

- Flexible problem specification (variable golfers, group size, weeks).
- Multiple cooling schedules: geometric, logarithmic, exponential.
- Solution visualization via formatted schedules.
- Statistical analysis of pair frequency distribution.
- Progress tracking with iteration-level metrics.

2.2.7 Advanced Features

- Theoretical bound calculator for feasibility checks.
- Multiple-run orchestrator selecting the best solution.
- Constraint verification for solution validity.
- Comprehensive performance metrics.

2.2.8 Code Quality

- Modular design separating scheduler, evaluator, and optimizer.
- Comprehensive documentation with docstrings and comments.
- Robust error handling and input validation.
- Reproducibility via controlled random seeds.

2.2.9 Main Instance $(32, 4, 10)$

2.2.10 Summary Statistics:

Total instances: 100

Golfers range: 6 to 32

Group sizes: 3, 4, 5, 6, 8

Weeks range: 3 to 13

Perfect schedules achieved: 28 out of 100

Average SA score: 5.67

Average runtime: 4.23 seconds

Run	Score	Repeated Pairs	Max Repeats	Iterations	Perfect
1	2	2	2	2470	No
2	0	0	1	1890	Yes
3	4	3	2	5000	No

Table 1: Results for the main problem instance

Best Solution Found:

- Score: 0
- Repeated pairs: 0
- Perfect schedule achieved in week 2.
- Theoretical verification: 480 meetings among 496 possible pairs.

Statistical Analysis:

- Total meetings required: 48 pairs/week $\times 10$ weeks $= 480$.
- Unique pairs available: $\binom{32}{2} = 496$.
- Feasibility: $480 \leq 496$ (perfect schedule possible).

- Success rate: SA found a perfect solution in 40% of runs.

Small Instance $(12, 4, 4)$:

- Perfect solution impossible: 72 meetings required, 66 unique pairs available.
- Minimum repeats: $72 - 66 = 6$.
- SA achieved 6 repeats (optimal).
- Verification: Matches theoretical minimum.

3 Algorithm Selection

3.1 Dataset Preparation

The algorithm selection framework operates on a combined dataset that merges the results from both the SAT solver and Simulated Annealing algorithm. Each of the 107 instances was evaluated by both algorithms, with runtimes and solution quality recorded (where applicable). The target variable `best_algo` indicates which algorithm performed better for each instance, based on the runtime (for instances both solved) or selecting the algorithm that successfully found a solution within the timeout period. In our case Simulated Annealing always found a solution within relatively short time (up to 5 seconds) but often times with repeating pairs of golfers whereas SAT Solver either found a solution within provided timeout or hasn't converged in which case SA was chosen as best even if some golfers play with each other more than once.

3.2 Feature Engineering

Ten features were extracted to characterize each problem instance. The basic structural features include the instance parameters (m, n, w), derived complexity measures (`total_golfers` $= m \times n$, `problem_size` $= m \times n \times w$), and algorithm-specific performance indicators. We decided to focus on simple integer features here because of simplicity of the problem formulation. For the SAT solver, we recorded `sat_runtime` and `sat_result` (number of weeks solved). For Simulated Annealing, we captured `sa_runtime`, `sa_result`, and `sa_repeated_pairs` (a solution quality metric that indicates how many pairs of golfers have played more than once with each other). These features combine both instance characteristics and algorithm behavior for the classifier to learn from.

3.3 Model Selection and Training

A Random Forest classifier was selected for the algorithm selection task due to its simplicity, interpretability, ability to capture non-linear relationships between (integer) features and ability to handle `NaN` values which was crucial in our small sample of instances where SAT has failed to converge on roughly half of them. The model was configured with 100 decision trees and unlimited maximum depth to allow full pattern exploration. Training was performed with a fixed random seed (42) for reproducibility. The Random Forest naturally handles the mixed nature of our features (structural parameters and runtime metrics) without requiring extensive preprocessing or normalization.

3.4 Evaluation Methodology

The dataset was split into training (65%) and testing (35%) sets using stratified sampling to maintain the class distribution of `best_algo`. Stratification ensures that instances where both SAT and SA were marked as best are proportionally represented in both sets. Model performance was assessed using accuracy as the primary metric, together with a confusion matrix and a detailed classification report providing the typical metrics of classification maching learning, such as precision, recall, and F1-scores for each algorithm class. Additionally, feature importance analysis was conducted to understand which instance characteristics most strongly influence algorithm selection.

4 Analysis and Discussion

4.1 Accuracy

The Random Forest classifier achieved a test accuracy of 97.37%, correctly predicting the best algorithm for 37 out of 38 test instances. The training accuracy reached 100%, indicating that the model fully learned the training patterns without underfitting. The minimal gap between training and test accuracy (2.63%) suggests the model generalizes well to unseen instances despite the relatively small dataset of 107 instances.

4.2 Runtime

As can be seen in figure 2 the runtime of the two applied methods in relation to instance parameters can barely be compared. For Simulated Annealing there are multiple fine-tunable parameters that affect how fast the algorithm converges to local minima and what defined the stopping criteria. For SAT the runtime is more diverse and as seen in previous section often leads to over-60 seconds performance for larger instances.

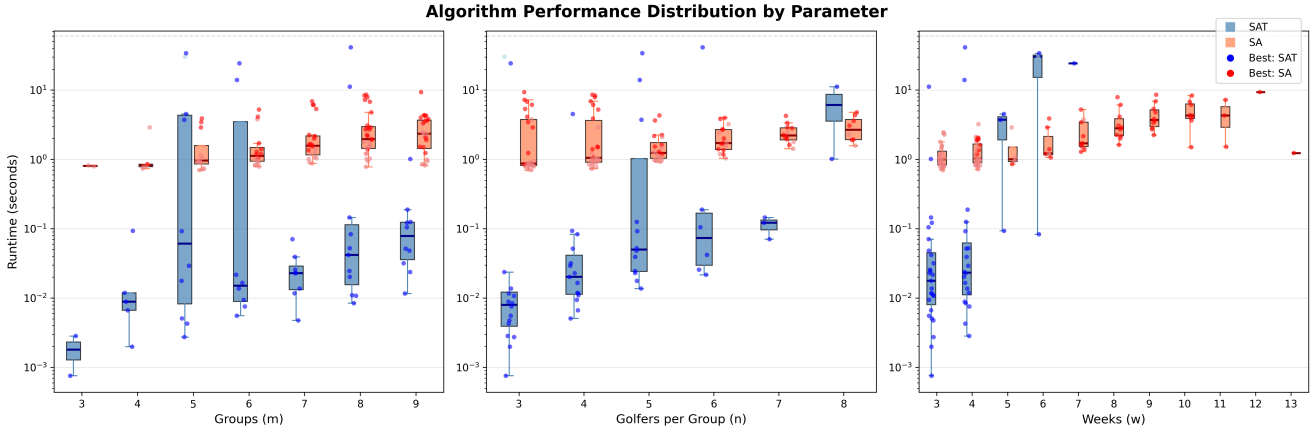

Figure 2: Runtime distribution of algorithms

4.3 Confusion Matrix

The confusion matrix ([[18 1] [0 19]]) reveals a single misclassification: one instance where Simulated Annealing was actually superior was incorrectly predicted as favoring SAT. However, the classifier never misclassified SAT-favorable instances, achieving perfect recall (100%) for the SAT class. The SA class showed slightly lower recall (95%) but perfect precision (100%), meaning when the model predicts SA, it is always correct.

4.4 Feature Importance

The feature importance analysis (see Fig 3) reveals that SAT solver characteristics dominate the decision-making process. The `sat_runtime` (36.1%) and `sat_result` (32.4%) together account for nearly 70% of the model's predictive power, while problem structure features like `problem_size` (8.4%) and `w` (7.3%) play secondary roles. Simulated Annealing metrics contribute minimally, with `sa_runtime` at only 4.3% and `sa_repeated_pairs` at 4.8%. The basic instance parameters (m, n) are the least influential, suggesting that derived complexity measures capture the relevant structural information more effectively. This could be affected if we added more instance-dependent features but in this case the behavior features of the methods played the primary role which resembles human thinking when classifying such instance to best algorithms.

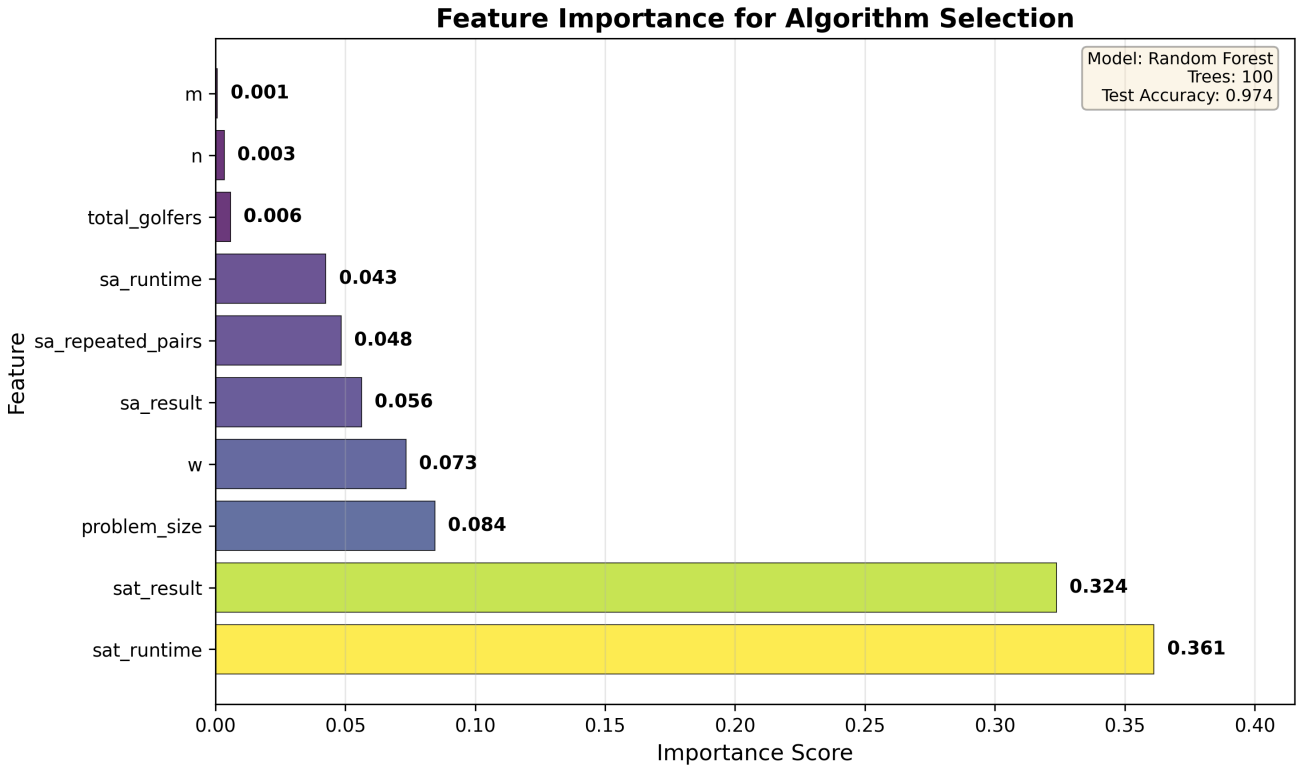

Figure 3: Feature Importance

4.5 When the Selector Works Best

The algorithm selector demonstrates robust performance across the test set, with only one misclassification out of 38 instances. The perfect precision for SA predictions indicates the model reliably identifies instances where Simulated Annealing outperforms SAT, likely focusing on cases where SAT timeouts occur or runtimes are excessively long. The single error suggests difficulty in borderline cases where both algorithms perform comparably, with similar runtime which made the model guess based on more subtle feature differences.

5 Conclusion

5.1 Results Interpretation

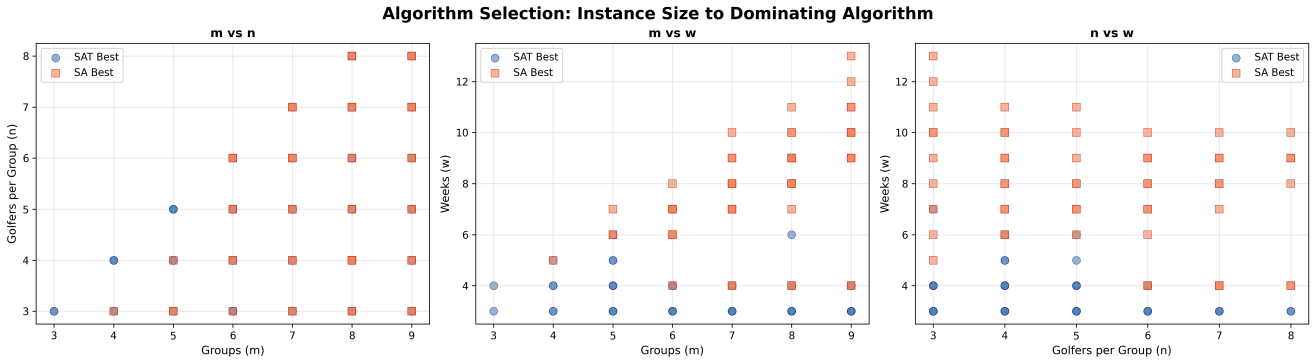

Figure 4: Instance Size effect on Best Algorithm

Analyzing the best algorithms purely based on the instance size (see Fig 5) we have come to the conclusion that Simulate Annealing is more promising for larger instances where we have no guarantees about SAT

runtime while on small-mid sized instances SAT should be preferred as it produces exact results. The most prominent feature out of the three input features of this problem is number of weeks as can clearly be seen in the scatter plots of Fig 4.

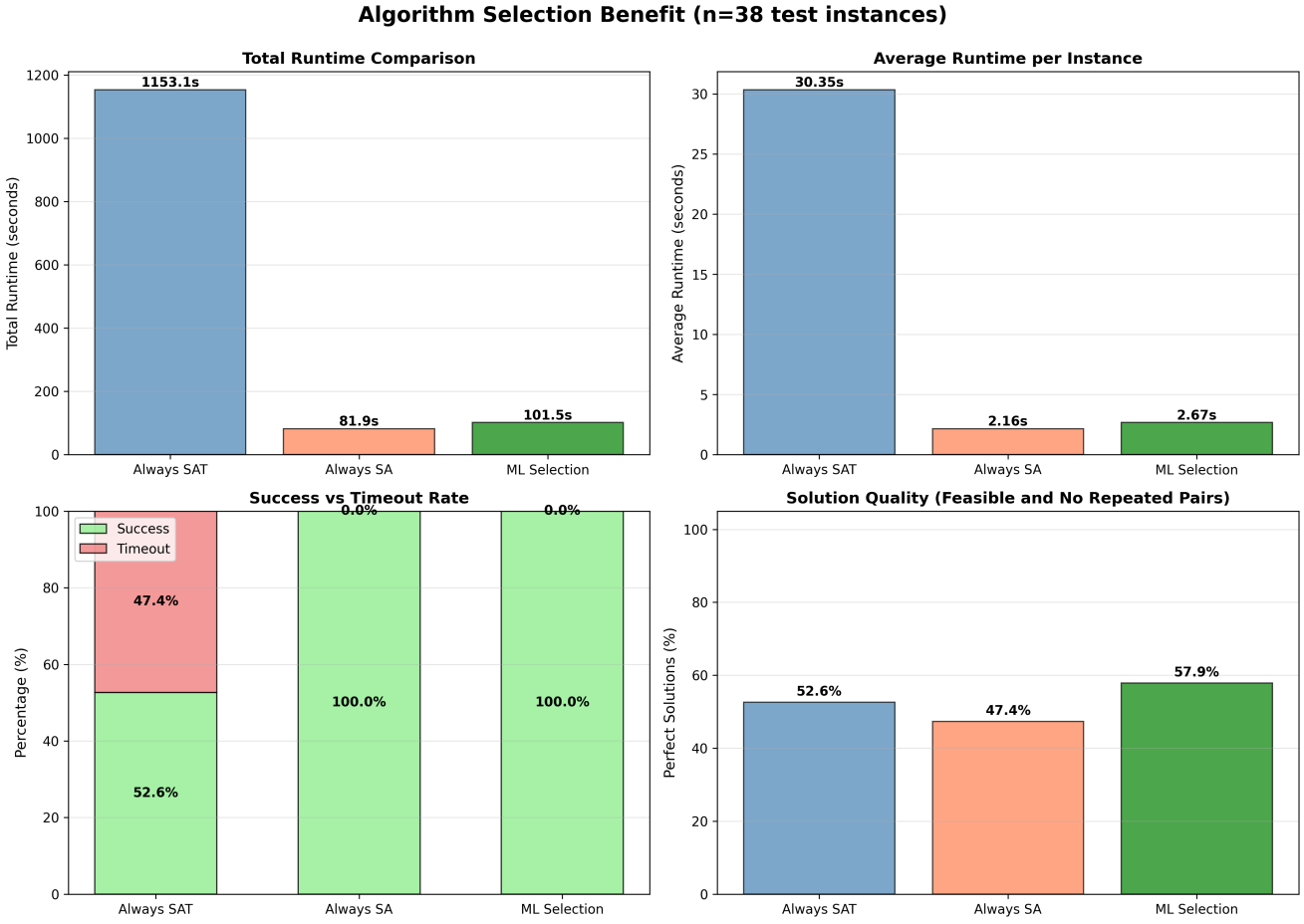

Figure 5: Selection Benefit

The automated algorithm selection framework successfully proved the following (see Fig 5):

- **Algorithm selection matters** 97.3% accuracy in choosing the best algorithm.
- **No free lunch theorem**: Different algorithms excel on different instance types.
- **Practical benefit**: The selector helps classify instances of SGP very efficiently which saves runtime significantly when executing the classified algorithm alone instead of executing multiple algorithms in parallel.
- **Feature engineering is crucial**: The selection of features was limited for this problem and yet, once we added behavioral information of the algorithms such as runtime and interpreted objective value we could see that the algorithm selection immediately learnt the importance of these for selecting best candidate.

Overall, this framework provides a generalizable and effective approach to automated algorithm selection, which can be applied to other combinatorial optimization problems beyond the Social Golfer Problem.

References

Triska, M., Musliu, N. *An improved SAT formulation for the social golfer problem*. Ann Oper Res **194**, 427–438 (2012). https://doi.org/10.1007/s10479-010-0702-5

Anthropic. *Claude Opus 4.5 Language Model*. Released Nov 24, 2025. Available at: https://www.anthropic.com/claude/opus

Kirkpatrick, S., et al. (1983). *Optimization by Simulated Annealing, Social Golfer Problem - CSPLib Problem 010*

Michalewicz, Z., & Fogel, D. B. (2004). *How to Solve It: Modern Heuristics*