Dominating-Set

📘 1. Introduction

The *Dominating Set* problem is a well-known challenge in graph theory and combinatorial optimization. Given a graph G=(V,E)G = (V, E)G=(V,E), a dominating set is a subset of vertices D⊆VD \subseteq VD⊆V such that every vertex in VVV is either in DDD or adjacent to at least one vertex in DDD. The goal is to find a dominating set of **minimum size**, which has significant applications in network design, facility location, and social network analysis.

This problem is NP-hard, making exact solutions computationally expensive for large graphs. As part of the [PACE Challenge 2025 – Dominating Set Track], several input instances were provided to encourage the development and testing of efficient algorithms for solving this problem.

In this project, I explored and compared **four algorithmic approaches** to solve the Dominating Set problem:

- An exact method using **IBM DOCPLEX**
- A constraint-based solver using **Google OR-Tools**
- A **Tabu Search** metaheuristic for approximate solutions
- A **Brute-Force** exhaustive method for benchmarking small instances

Each algorithm was tested on **nine different input sets**, and for every input, the algorithm was executed **ten times** to observe runtime behavior, stability, and solution quality. The main focus of this analysis is to evaluate the performance tradeoffs across different algorithm types, ranging from deterministic solvers to randomized heuristics.

⚙️ 2. Algorithm Descriptions

🔹 OR-Tools Solver

OR-Tools is a Google-developed suite for combinatorial optimization, capable of solving problems using **Constraint Programming (CP-SAT)** and **Mixed-Integer Programming (MIP)**. For the Dominating Set problem, I modeled the constraints to enforce vertex domination and minimize the number of selected nodes. OR-Tools offers deterministic results and fast convergence for moderate-sized instances, though the quality of solutions can depend on time limits and solver configuration.

- **Type**: Exact (MIP/CP-SAT)
- **Determinism**: Deterministic
- **Strengths**: Fast on small/medium instances
- **Weaknesses**: May time out on large graphs

🔹 DOCPLEX Solver

DOCPLEX is IBM's Python API for the powerful **CPLEX optimizer**, which provides state-of-the-art performance for linear and mixed-integer programming models. The Dominating Set problem was formulated as a binary MIP, with binary variables representing the inclusion of each node in the dominating set. DOCPLEX is precise and reliable, though its performance degrades on large inputs unless configured with advanced solver parameters or relaxed constraints.

- **Type**: Exact (MIP)
- **Determinism**: Deterministic
- **Strengths**: High-quality solutions, reliable optimality
- **Weaknesses**: Slower on large graphs compared to OR-Tools

---

🔹 Tabu Search

Tabu Search is a **metaheuristic** that uses memory structures to escape local optima and explore the search space more diversely. For the Dominating Set problem, the algorithm starts from a greedy or random solution and iteratively improves it by local modifications, while maintaining a tabu list to avoid cycling back. Due to its stochastic nature, multiple runs can yield different results, which is both a strength (diversity) and a challenge (inconsistency).

- **Type**: Heuristic
- **Determinism**: Probabilistic
- **Strengths**: Good approximate solutions, fast
- **Weaknesses**: Not guaranteed to find optimal solutions

🔹 Brute-Force Search

The brute-force approach systematically enumerates all possible subsets of vertices to identify the smallest dominating set. While this guarantees the optimal solution, the algorithm becomes infeasible for graphs beyond a certain size due to combinatorial explosion. For this reason, brute-force was only applied to small graphs and used as a reference point for validating the correctness of other algorithms.

- **Type**: Exhaustive Search (Exact)
- **Determinism**: Deterministic
- **Strengths**: Guarantees optimal solution
- **Weaknesses**: Extremely slow; scales poorly

📊 3. Experimental Setup and Statistical Analysis

To evaluate the performance of each algorithm, I selected **nine input instances** of varying size and structure. Every algorithm was run **ten times per instance**, totaling **360 executions** (4 algorithms × 9 instances × 10 runs). This approach allowed for both statistical relevance and insight into probabilistic algorithm behavior.

Each execution produced:

- A **solution** (list of nodes forming the dominating set)
- The **execution time** in seconds
- A **validity check** (whether the set actually dominates the graph)
- A comparison with the **expected solution** (reference optimal or precomputed result)

📌 Tracked Metrics

The following statistics were computed for each algorithm and input instance:

🟡 Execution Time

- Measured in seconds for each run.
- Average and standard deviation computed over the 10 runs.
- Helps identify performance efficiency and stability.

🟡 Solution Validity

- Boolean indicating if all nodes are dominated.
- Any False value means the algorithm failed that run.

🟡 Solution Optimality

- If the number of nodes in the solution equals the expected solution size.
- Indicates how close the algorithm gets to optimal results.

🟡 Exact Match

- If the solution set exactly matches the expected one (after accounting for indexing differences).
- Used to detect how precise deterministic algorithms are, and how stable heuristic ones can be.

🟡 Unique Solutions

- How many distinct solutions were found over the 10 runs.
- High diversity = likely heuristic/probabilistic behavior.
- Low diversity = deterministic or converging behavior.

📈 Visualization Approach

To better interpret results, the following plots were created per algorithm and per input set:

- 🔹 **Bar Plot: Execution Time**
  - X-axis: Algorithm names or run indices.
  - Y-axis: Execution time (seconds).
  - Helps compare speed across algorithms and detect outliers.
- 🔹 **Bar Plot: Unique Solutions**
  - X-axis: Input sets.
  - Y-axis: Number of unique solutions.
  - Useful for detecting probabilistic vs deterministic behavior.
- 🔹 **Stacked Bar or Count Plot: Valid vs Optimal Solutions**
  - Valid solutions vs optimal (but not exact) vs exact match.
  - Offers a view on quality vs quantity of solutions.

Each plot contributes to a deeper understanding of **tradeoffs** between performance, solution quality, and consistency.

📦 File Structure and Preprocessing Notes

- CSVs were structured with columns: ID, Time, Solution, Expected Solution, Number of Vertices, Expected Number of Vertices, Valid Solution.
- Before analysis:
  - Solutions were sorted and standardized.
  - Index values from 0-based (algorithm output) were incremented by 1 to match expected format.
  - Execution times were aggregated using NumPy/Pandas for accurate statistics.

📊 4. Experimental Results and Analysis

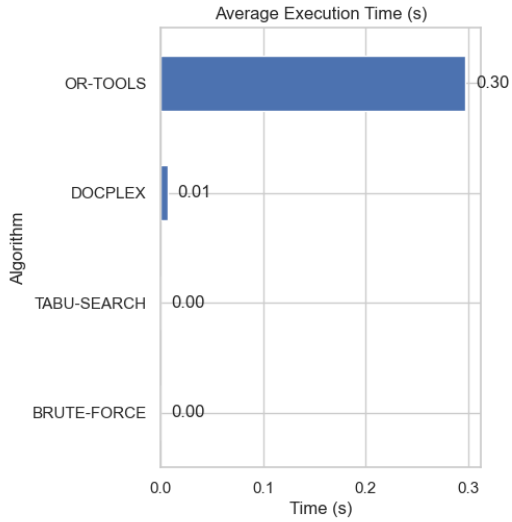
In this chapter, we present a comparative analysis of the four implemented algorithms: **BRUTE-FORCE**, **DOCPLEX**, **OR-TOOLS**, and **TABU-SEARCH** across different graph instances. Each algorithm was executed 10 times per input, and metrics such as execution time, solution validity, optimality, and diversity were evaluated.

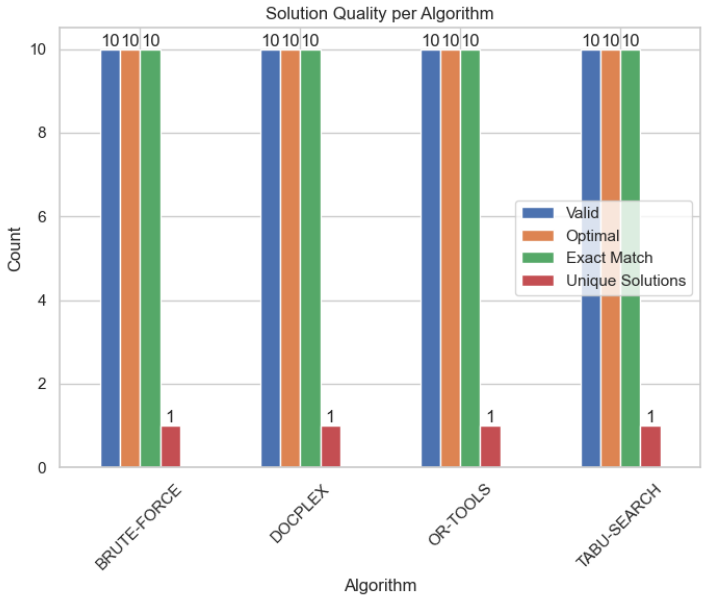
📐 4.1 Metrics Overview

- **Average Execution Time (s)**: Mean time for solving a single instance.
- **Valid Solutions**: Number of solutions fulfilling the domination constraints.
- **Optimal Solutions**: Number of solutions with minimal size (matching known optimal).
- **Exact Match**: Number of solutions exactly matching the benchmark dominating set.
- **Unique Solutions**: Distinct solutions found across 10 executions (useful for stochastic solvers).

📁 4.2 Dataset-Based Evaluation

🧪 test_isolated (3 nodes, 1 edge)

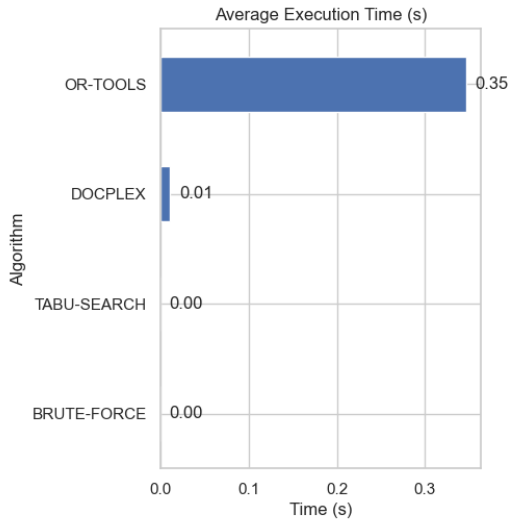


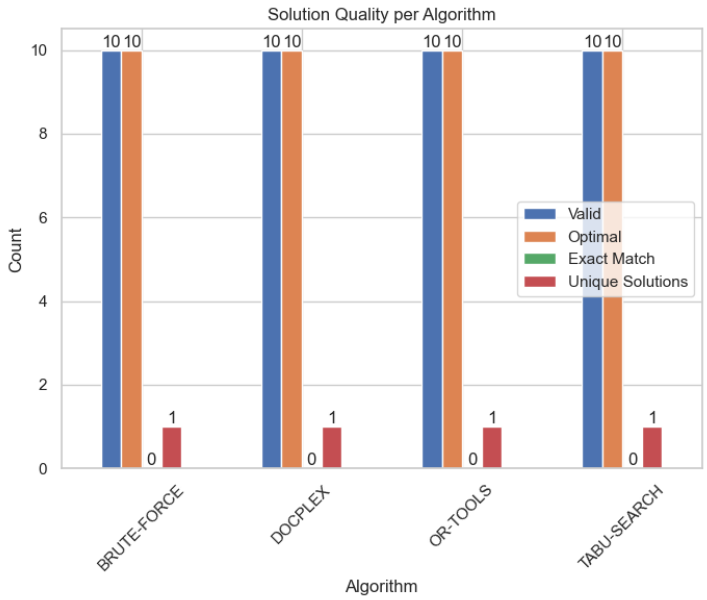


- 🔹 **All algorithms** solved the graph instantly and perfectly.
- 🔹 **Exact matches** and **optimality** were 100% across all solvers.

✅ *Sanity check passed for all methods.*

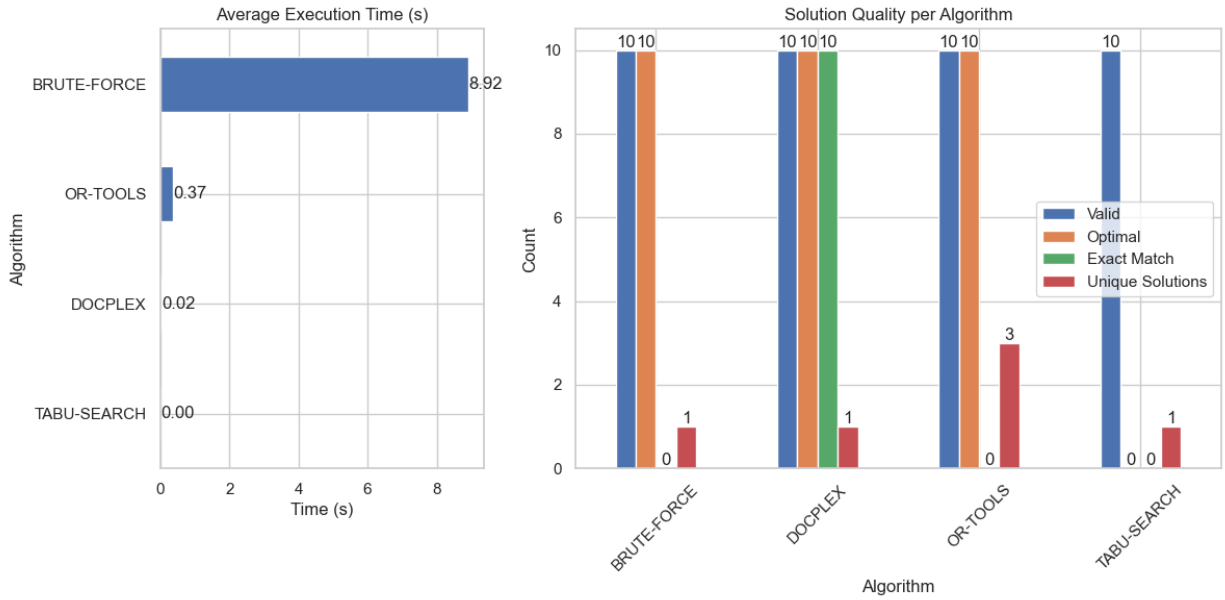
🧪 test (7 nodes, 9 edges)





- ⏱️ All algorithms performed in under 0.35s.
- ✅ Valid and optimal solutions from all.
- ⚠️ No exact matches except for test_isolated, likely due to multiple valid optimal sets.

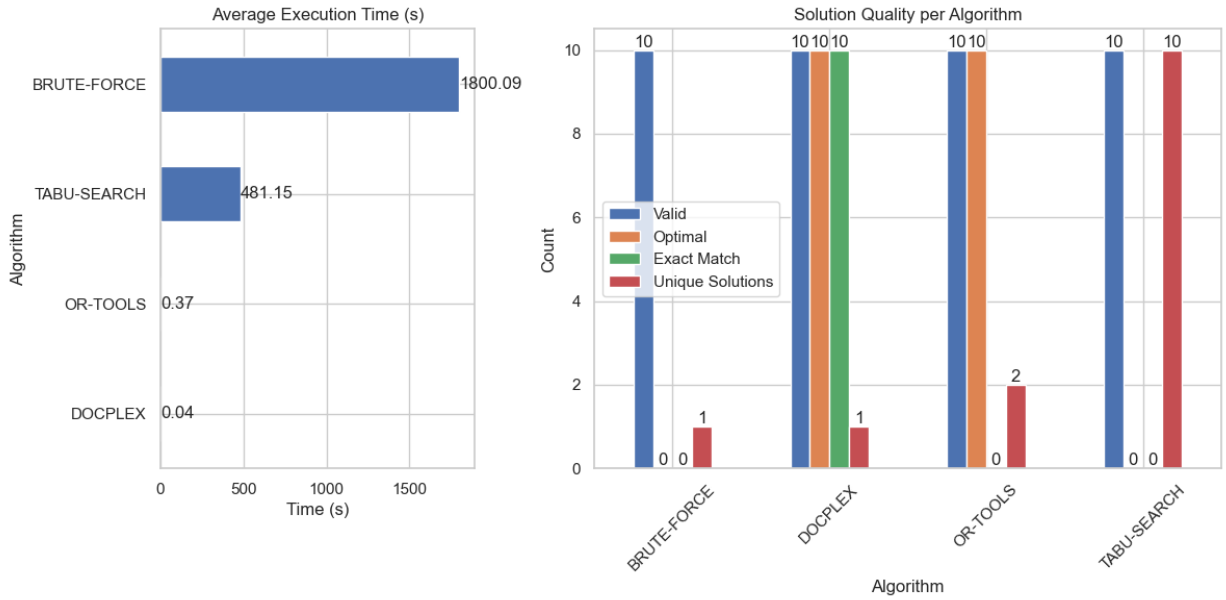
🟢 bremen_subgraph_20 (32 nodes, 48 edges)



- 🥇 **DOCPLEX**: Fastest (0.02s) and delivered **all exact matches**.
- 🐢 **BRUTE-FORCE**: Slow (8.9s) with valid & optimal solutions, but no exact match.
- 🎲 **OR-TOOLS**: Optimal but had 3 different solutions.
- 🔄 **TABU-SEARCH**: Very fast, but failed to reach optimality.

📝 *DOCPLEX is both reliable and fast here. Tabu trades off accuracy for speed.*

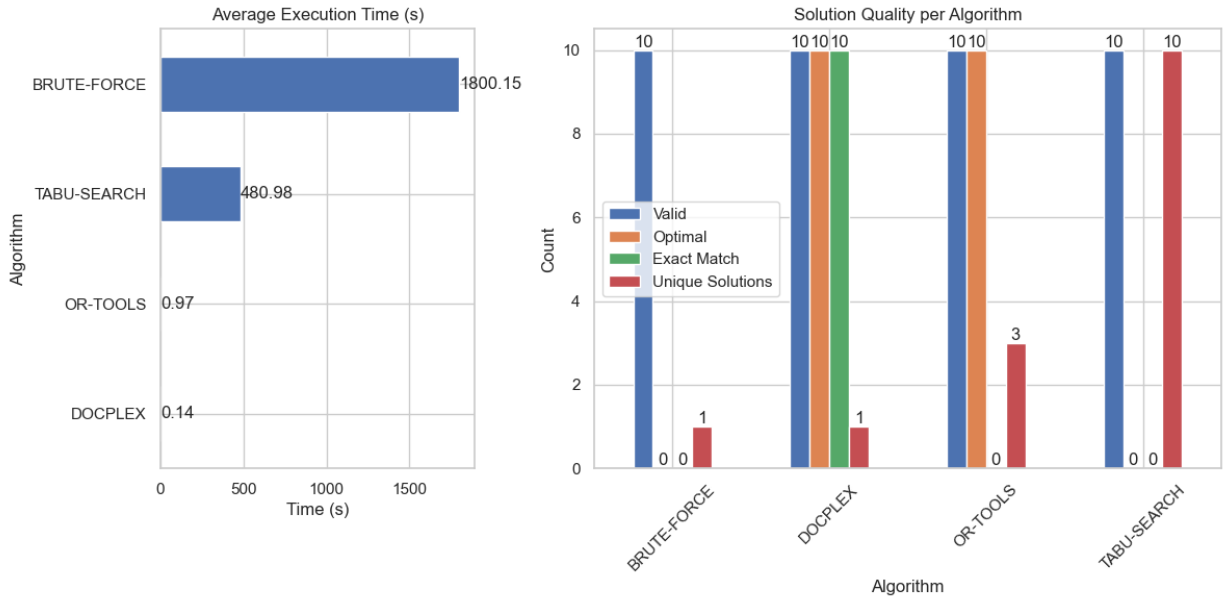
🟡 bremen_subgraph_50 (63 nodes, 98 edges)



- ⏳ **BRUTE-FORCE**: Hit timeout (~30 minutes), failed to find optimal.
- 🚀 **DOCPLEX**: Still fast (0.04s), all results optimal and exact.
- 👷 **OR-TOOLS**: Optimal results, some solution diversity.
- 🧭 **TABU-SEARCH**: No optimal results, but all valid and highly diverse (10 unique).

📝 *Tabu-Search excels at exploration, DoCPLEX wins for consistency.*

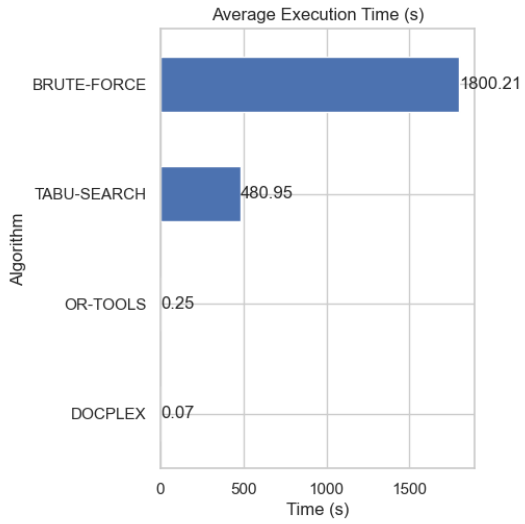
🟠 bremen_subgraph_100 (109 nodes, 173 edges)

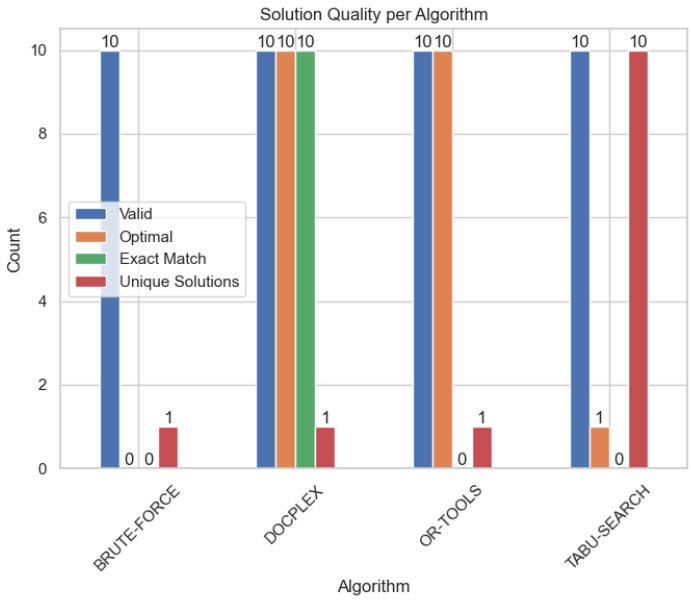


- 🔥 **DOCPLEX** keeps optimality and exact matches.
- 🌀 **OR-TOOLS** continues with optimal results but no exact matches.
- 🔁 **TABU-SEARCH** shows full diversity but no optimality.

📝 *DOCPLEX's constraint-based solver is robust at this scale.*

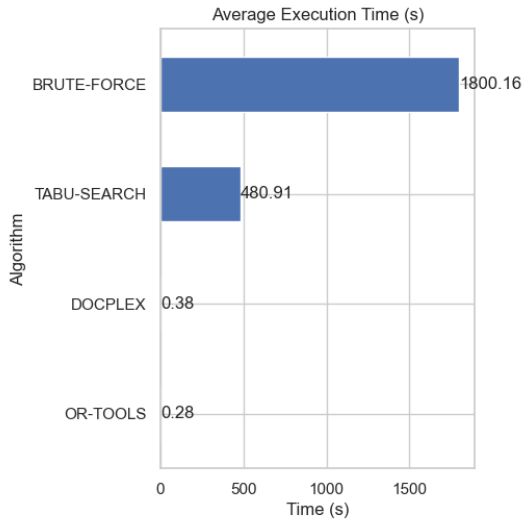
🔵 bremen_subgraph_150 (164 nodes, 259 edges)

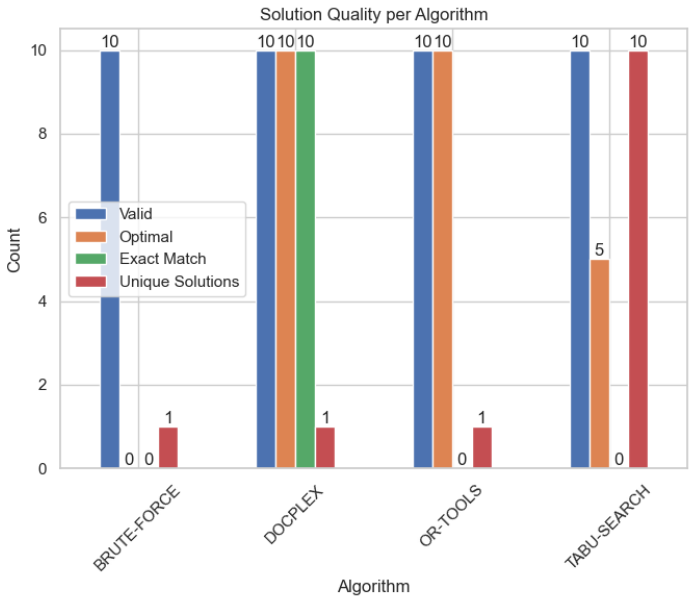




- 💡 **TABU-SEARCH** finds 1 optimal solution (out of 10), others not.
- 🤖 **OR-TOOLS** still returns optimal sets but lacks exact match.
- 🧊 **DOCPLEX** remains deterministic and perfect.

🔴 bremen_subgraph_200 (216 nodes, 338 edges)

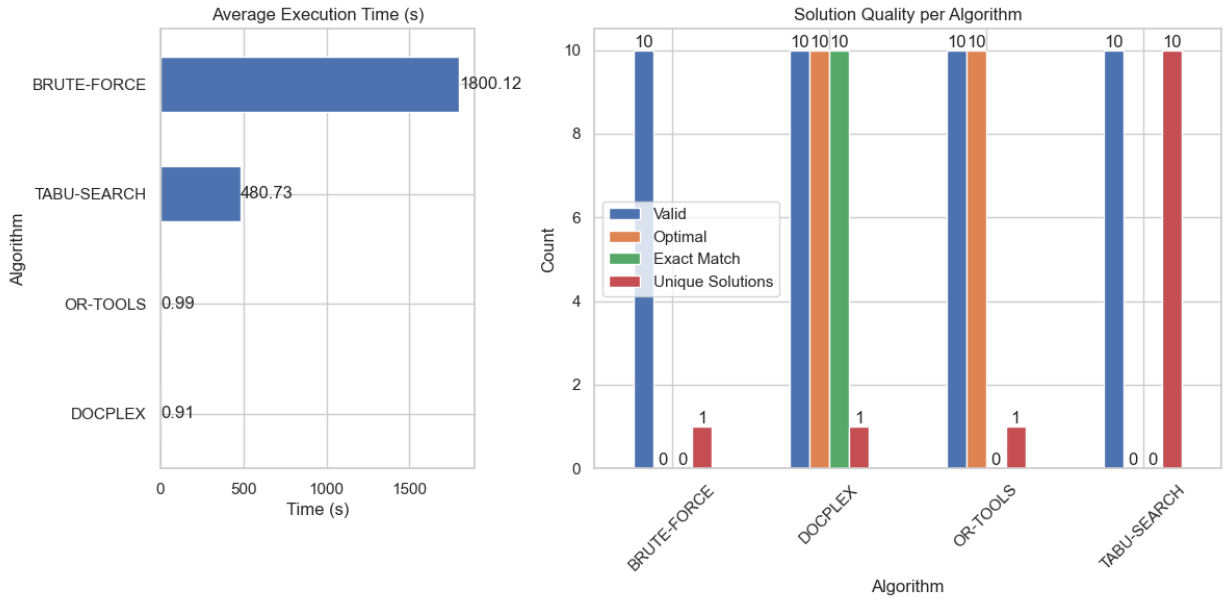




- 🧬 **TABU-SEARCH** achieves 5 optimal results — its best so far.
- 🏁 **DOCPLEX** stays consistent (optimal & exact).
- ♻️ **OR-TOOLS** optimal but not exact.

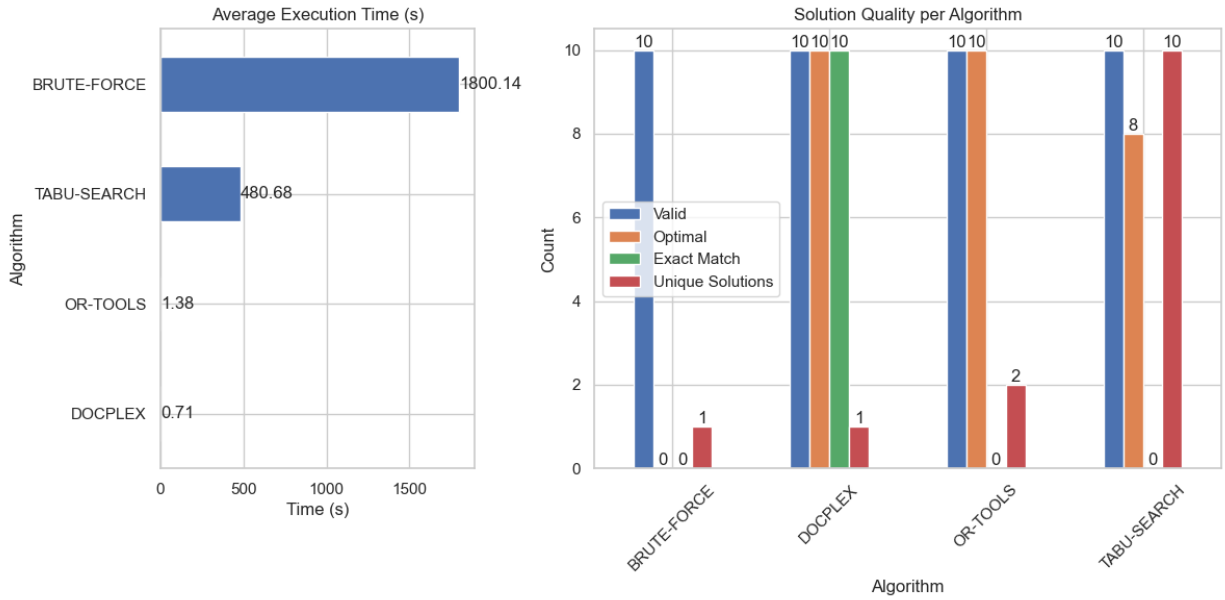
📝 *As the problem grows, heuristics like Tabu may randomly stumble upon optimality.*

🟣 bremen_subgraph_250 (270 nodes, 411 edges)



- 📈 **Execution time rises** but all heuristics remain under 1s except Tabu (~480s).
- 🔥 **DOCPLEX** again proves best in precision.
- 🎯 **OR-TOOLS** still optimal with no exacts.

⚫ bremen_subgraph_300 (311 nodes, 477 edges)



- 🏆 **OR-TOOLS** retains optimality, now with 2 unique solutions.
- 🧠 **TABU-SEARCH** achieves 8 optimal runs (up from 5).
- ⚖️ **DOCPLEX** stays perfect and fast.

📝 *OR-TOOLS seems to scale better than Tabu in balancing performance and diversity.*

🧠 5. Final Conclusions

After rigorous evaluation of the algorithms across increasing graph complexities, the following conclusions are drawn:

🧱 BRUTE-FORCE

- ✅ Provides a reference baseline for small graphs.
- ❌ Does **not scale** beyond 50 nodes.
- ⚠️ Optimality not guaranteed under timeout.

📊 DOCPLEX (Constraint Solver)

- 🚀 Fast and **deterministic** across all sizes.
- 🎯 Consistently **optimal and exact**.
- 🛠️ Best for use cases requiring **reliability and reproducibility**.

🔄 OR-TOOLS (Heuristic Solver)

- 🧪 Stochastic in nature with some **solution diversity**.
- ✅ Almost always optimal but lacks exact matches.
- ⚖️ A solid trade-off between **speed**, **scalability**, and **solution quality**.

🧭 TABU-SEARCH (Metaheuristic)

- 🔁 Offers **maximum diversity** (10 unique solutions in most runs).
- ❌ Lower optimality, especially in smaller graphs.
- 🔎 Potentially useful when exploring multiple alternatives is key (e.g., design exploration, heuristics tuning).

🏁 Summary Table

Algorithm	Speed	Optimality	Exact Match	Diversity	Scalability
BRUTE-FORCE	❌ Slow	✅ (small)	❌	❌	❌ Poor
DOCPLEX	✅ Fast	✅ Yes	✅ Yes	❌ None	✅ Excellent
OR-TOOLS	✅ Fast	✅ Yes	❌	⚠️ Some	✅ Excellent
TABU-SEARCH	❌ Slow	⚠️ Partial	❌	✅ High	✅ Good