Solving the N-Queens Problem using two SAT solvers

TimmoTaffy

June, 2025

Abstract

The N-Queens problem is a classic constraint satisfaction task. We encode the problem into CNF and implement two solvers for it: DPLL (a complete backtracking algorithm) and WALKSAT (an incomplete local search method). Through Python-based experiments on boards of various sizes, we compare their correctness, efficiency, and scalability. Results show that DPLL performs reliably on small boards, while WALKSAT scales better on larger instances despite variability. These findings highlight the trade-off between completeness and efficiency in SAT solvers, and their broader relevance in structured combinatorial problems.

1 Introducing the N-Queens Problem

The N-Queens problem asks how to place N queens on an N-by-N chessboard such that no two queens attack each other. That is, no two queens can be in the same row, column, or diagonal.

Depending on the goal, solving the N-Queens problem may involve finding a single valid solution, enumerating all possible solutions, or determining whether at least one solution exists for a given N. These variations lead to different algorithmic strategies, ranging from exhaustive search to more efficient heuristic approaches.

Solutions exist for all $n \in N$ except when $N = 2, 3$, although $N = 27$ is the highest-order version that has been completely enumerated. There is no known formula for the exact number of solutions for N-Queens problems, In 2021, Michael Simkin proved that for large numbers n, the number of solutions of the n queens problem is approximately $(0.143n)^n$. [1]

2 Solving the N-Queens Problem as an SAT problem: Modeling and Two Solvers

2.1 Preparations: SAT and CNF

The Satisfiability Problem (SAT) is a fundamental problem in computer science and mathematical logic. It asks whether the variables of a given Boolean formula can be assigned truth values in such a way that the formula evaluates to true. If such an assignment exists, the formula is said to be satisfiable; otherwise, it is unsatisfiable.

SAT was the first problem proven to be NP-complete, a landmark result in complexity theory, and its intractability in the general case has significant implications for computational limits. Despite its complexity, SAT solvers have become highly sophisticated tools, capable of solving large and complex instances arising from various real-world applications.

Most modern SAT solvers operate on Boolean formulas expressed in Conjunctive Normal Form (CNF), a standardized format where the formula is a conjunction (AND) of clauses, and each clause is a disjunction (OR) of literals (variables or their negations).

All Boolean formulas can be converted to CNF using De Morgan's laws and distribution. SAT solvers use CNF as their standard inputs, so in the following subsection we model the Eight Queens Problem using CNF.

2.2 Problem Modeling in Propositional Logic [2]

From the problem description we see the following three constraints:

- At least one queen per row.
  (This automatically guarantees that there is at most one queen per row, since num(rows) = num(queens)).
- At most one queen per column.
  (This automatically guarantees that there is at least one queen per row, for a similar reason).
- At most one queen per diagonal.
  (We don't have enough queens for each diagonal).

Now we rewrite the three constraints in CNF. Let $p(i,j)$ denote "there is a queen on the square in the i-th row and j-th column".

- At least one queen per row:

$$Q_1 = \bigwedge_{i=1}^{N} \left(\bigvee_{j=1}^{N} p(i,j) \right)$$

- At most one queen per column:

$$Q_2 = \bigwedge_{j=1}^{N} \bigwedge_{i=1}^{N-1} \bigwedge_{k=i+1}^{N} (\neg p(i,j) \vee \neg p(k,j))$$

- At most one queen per diagonal (we need to):

$$Q_3 = \bigwedge_{i=2}^{N} \bigwedge_{j=1}^{N-1} \bigwedge_{k=1}^{\min(i-1,N-j)} (\neg p(i,j) \vee \neg p(i-k, k+j))$$

$$Q_4 = \bigwedge_{i=1}^{N-1} \bigwedge_{j=1}^{N-1} \bigwedge_{k=1}^{\min(N-i,N-j)} (\neg p(i,j) \vee \neg p(i+k, j+k))$$

Putting together, we find that the solutions of the n-queens problem are given by the assignments of truth values to the variables $p(i,j)$, i = 1, 2, ... , n and j = 1, 2, ... , n that make

$$Q = Q_1 \wedge Q_2 \wedge Q_3 \wedge Q_4 \wedge Q_5$$

true.

Now that we have transformed the problem into the standard input form for SAT solvers, it's time to introduce the solver algorithms.

2.3 DPLL Algorithm [3]

2.3.1 Algorithm Principles

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is a complete, recursive method for determining the satisfiability of propositional logic formulas in Conjunctive Normal Form (CNF). It forms the theoretical foundation for most modern SAT solvers.

The core idea of DPLL is to recursively simplify the CNF formula using logical rules and systematic search with backtracking, until either a satisfying assignment is found or all possibilities are exhausted, proving the formula unsatisfiable.

2.3.2 Key Steps of DPLL

The DPLL algorithm proceeds through four major steps: *unit propagation, pure literal elimination, splitting, backtracking.* The first two steps simplify the CNF formula before decision-making by narrowing the search space. The following steps perform a recursive search over possible truth assignments. In essence, the algorithm first simplifies what it can with certainty, and only then begins to explore different possibilities through systematic trial and error. Now we discuss each steps in detail.

1.Unit Propagation

Def. A *unit clause* is a clause that contains only a single literal (e.g.,(A) or $(\neg B)$).
Rule If a formula contains a unit clause, then its literal must be assigned true to satisfy that clause.
Eg. Given the formula $(A) \wedge (\neg B \vee C)$, the unit clause (A) requires the literal A = True so as to be True. Substituting this reduces the formula to $(\neg B \vee C)$.

2.Pure Literal Elimination

Def. A *pure literal* is a variable that appears either always positive or always negative in the entire formula.
Rule A pure literal can safely and should be assigned True.
Eg. In the formula $(A \vee B) \wedge (\neg B \vee C)$, both A and C are pure literals, so we set them to True. The formula then simplifies to $(\neg B)$.

3.Splitting (Decision Making)

Rule Choose an unassigned variable and tentatively assign it T or F, then recursively solve the simplified formula. If that fails, backtrack and try the other assignment.
Strategy Can first work on the most frequent variable.

4.Backtracking

Rule If a contradiction is encountered (i.e., an empty clause is derived), the algorithm backtracks to the most recent split point and tries the alternative value.

Termination condition of DPLL:

- A satisfying assignment is found (formula satisfiable); or
- All branches lead to contradictions (formula unsatisfiable).

2.3.3 Pseudocode Summary of DPLL

```
def DPLL(formula, assignment):
    if contains_empty_clause(formula):
        return False
    if formula is fully satisfied:
        return assignment
    formula = unit_propagation(formula)
    formula = pure_literal_elimination(formula)
    variable = choose_variable(formula)
    return DPLL(formula + [variable=True]) or DPLL(formula + [variable=False])
```

2.4 WALKSAT Algorithm [3]

2.4.1 Algorithm Principles

WALKSAT is a local search SAT solver. Starting from a random assignment, it iteratively flips variables to reduce the number of unsatisfied clauses. It combines:

- Greedy selection: Flip the variable that satisfies the most clauses
- Random walk: With probability p, randomly flip a variable in a conflicting clause

2.4.2 Pseudocode Description of DPLL

```
def WALKSAT(formula, max_flips, p):
    assignment = random_assignment()
    for i in range(max_flips):
        if assignment satisfies formula:
            return assignment
        clause = random_unsatisfied_clause(assignment)
        if random() < p:
            flip a random variable in clause
        else:
            flip the variable that minimizes unsatisfied clauses
    return failure
```

3 Implementation and Experiments

3.1 Implementation of the Algorithms

I tried to implement the algorithms in a QueenSAT class using Python. The core modules include:

- `queens_cnf()`: encodes the N-Queens problem into CNF.
- `dpll()` and `walksat()`: implement the exact and heuristic solvers, respectively.
- `simplify()`, `is_clause_satisfied()`, `num_satisfied_after_flip()`: helper functions supporting clause evaluation and simplification.

Full implementation code is provided in Appendix C.

3.2 Experimental Setup

In order to evaluate the correctness and performance of the implemented DPLL and WalkSAT algorithms, we designed a series of experiments based on the classic N-Queens problem. The experiments were run using Python 3.13 on a machine with an Apple M2 SoC and 16GB RAM.

Key aspects of the setup include:

- Problem Instances: N-Queens problems with sizes $N = 4, 8, 10, 12, 14, 16, 20, 24$ were tested.
- Both the exact solver `dpll()` and the heuristic solver `walksat()` were tested on each instance.
- WalkSAT Parameters: For WalkSAT, we used a maximum of 10,000 iterations per run and a noise parameter p=0.5.
- Experimental Steps:
  1. For each instance, encode the problem into CNF using `queens_cnf()`.
  2. Test using `dpll()` and `walksat()` respectively.
  3. Repeat each test for 10 times to obtain statistically significant results.
  4. Record the corresponding metrics.
- Metrics Collected:
  - Solution found (boolean)
  - Time to solution (in milliseconds)
  - Number of recursive calls (for DPLL)
  - Number of flips (for WalkSAT)

3.3 Experimental Results (Analysis)

3.3.1 Correctness

All instances (from N=4 to N=24) were successfully solved by both DPLL and WalkSAT across all 10 runs per instance. This confirms that both solvers are functionally correct for the N-Queens problem up to moderate scales.

3.3.2 Data Processing and Visualization

Figure 1 shows the average running time (in milliseconds) of the DPLL and WalkSAT algorithms on N-Queens instances of increasing size. Each data point represents the average of 10 runs, and the error bars indicate the standard deviation. We observe that DPLL outperforms WalkSAT on small instances, but scales poorly as N increases beyond 16. WalkSAT, while less consistent, becomes significantly faster on larger boards.

Note: For full details of the experimental runs, including individual runtime records and summary statistics, please refer to Appendix D and Appendix E, respectively.

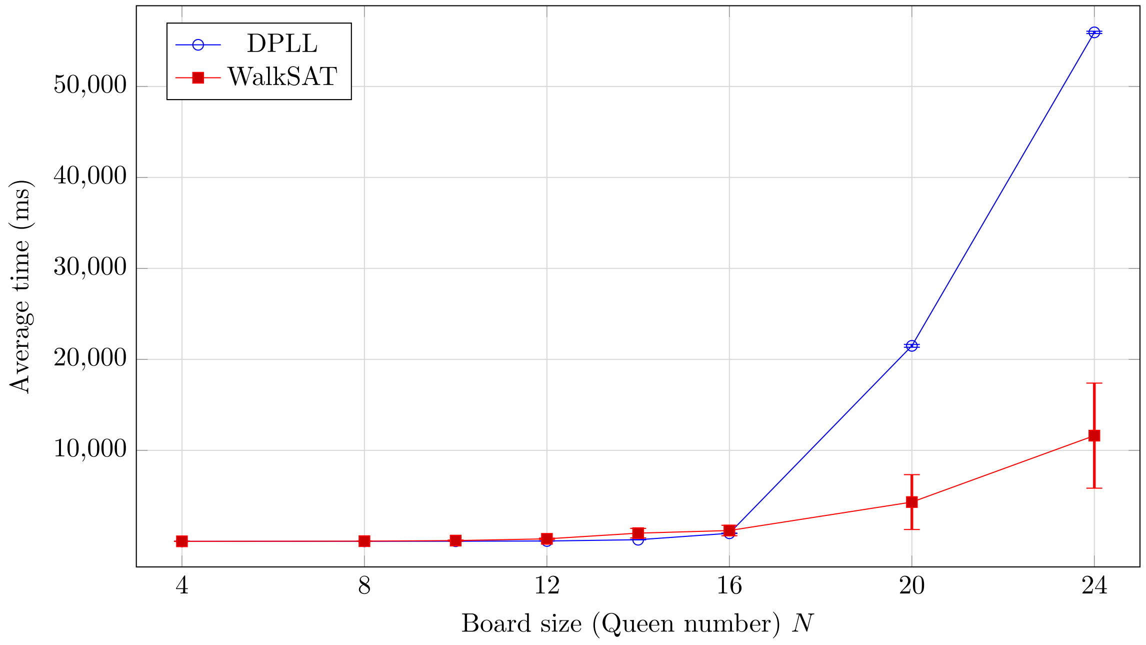

Figure 1: Average running time of DPLL vs WalkSAT with error bars (std dev).

3.3.3 Key Observations

1. Growth Trend and Crossover Point
   - DPLL exhibits exponential growth in time as N increases. The time increases dramatically from sub-millisecond at $N = 4$ to over 55 seconds at $N = 24$.
   - WalkSAT is initially slower than DPLL due to its randomized nature and restart strategy. However, it scales better starting at $N = 20$; at $N = 24$, it is roughly 5× faster than DPLL.
2. Variability
   - DPLL consistently exhibits extremely low standard deviation across all cases. This is attributed to its deterministic nature, resulting in an identical number of recursive calls for any fixed N across multiple runs.
   - WalkSAT shows high standard deviation, especially at larger N (e.g., over 5000 ms std at $N = 24$). This reflects the randomized nature of the algorithm and sensitivity to initial guesses and flips.

3. Implications

   - The performance data confirms DPLL as an exact solver and WalkSAT as a heuristic algorithm.
   - For small to medium N-Queens problems, DPLL is the preferred method due to its deterministic nature and faster runtime.
   - Conversely, for large-scale problems, particularly when execution time is critical, WalkSAT proves to be more practical, notwithstanding its inherent variability.
   - In a broader context, DPLL can be effectively utilized for proving unsatisfiability by exhaustively searching the entire solution space, whereas WalkSAT is suitable for rapidly identifying a satisfying model when satisfiability is the primary concern.

4 Features and Applications

The N-Queens problem has several interesting mathematical and logical features, making it a worthwhile case study which can shed light on many other fields.

4.1 Features of the Problem

First, the problem is computationally nontrivial although the constraints are easy to understand. As the board size N increases, the number of possible queen placements grows exponentially, but valid solutions become increasingly rare. This makes the problem a good example of a combinatorial explosion and an ideal candidate for exploring search algorithms and satisfiability solvers.

From a logical perspective, the constraints of the N-Queens problem can be cleanly formalized using propositional logic, and translated into Conjunctive Normal Form. This makes it a good testbed for SAT solvers, e.g. for comparing exact and heuristic methods, as we have done in this project.

Due to these features, the N-Queens problem has multiple analogies in other disciplines, especially when conflicting elements (queens) must be positioned to avoid interference. Such examples include particle placement problems in physics, molecular configuration in chemistry, and gene mapping or neural network layout in biology.

4.2 Applications of the SAT Solvers

In this project, we applied SAT solvers to the N-Queens problem, while the techniques can also be widely used in other fields where clear constraints need to be satisfied.

DPLL, as a complete and systematic search method, is suitable for situations where a guaranteed solution or proof of impossibility is required. WalkSAT, on the other hand, trades completeness for speed, making it helpful for problems where approximate solutions are acceptable and fast results are more important. Together, these two approaches show how different SAT solvers can be used depending on the needs of different problems.

5 Conclusion

This report focused on a particular solution of the N-Queens problem which reformulates it as a Boolean satisfiability (SAT) problem. We encoded the board constraints into Conjunctive Normal Form (CNF) and implemented two SAT solvers: DPLL (a complete, backtracking-based method) and WalkSAT (an incomplete, randomized local search method). Both were implemented in Python and tested on various board sizes.

Through experimentation, we found that DPLL is reliable and efficient for small board sizes but suffers from performance degradation as the size increases. WalkSAT, on the other hand, is more suitable for larger boards due to its scalability, although it introduces more variability in runtime. This confirmed that different solver strategies offer trade-offs between completeness, speed, and robustness.

Overall, this project demonstrates that SAT techniques can effectively solve structured combinatorial problems like N-Queens, and highlights how solver design influences practical performance.

Appendix A: Implementation Details of DPLL and WALKSAT

```python
class QueenSAT:

    def __init__(self, n):
        """
        :param n: The size of the chessboard (and the number of queens).
        """
        self.n = n
        self.clauses = self.queens_cnf()

    def queens_cnf(self):
        """
        Generate CNF clauses for the n-queens problem based on our modeling in 2.3:
        Returns a list of clauses; each clause is a list of integers.
        Positive integer: queen is present; negative: queen is absent.
        """
        n = self.n
        clauses = []

        # Q1: At least one queen per row
        for i in range(n):
            clauses.append([i * n + j + 1 for j in range(n)])

        # Q2: At most one queen per column
        for j in range(n):
            for i in range(n - 1):
                for k in range(i + 1, n):
                    clauses.append([-(i * n + j + 1), -(k * n + j + 1)])

        # Q3: At most one queen per \ direction diagonal (i-k, k+j)
        for i in range(1, n):
            for j in range(n):
                for k in range(1, min(i, n - j)):
                    a = i * n + j + 1
                    b = (i - k) * n + (k + j) + 1
                    clauses.append([-a, -b])

        # Q4: At most one queen per / direction diagonal (i+k, j+k)
        for i in range(n - 1):
            for j in range(n - 1):
                for k in range(1, min(n - i, n - j)):
                    a = i * n + j + 1
                    b = (i + k) * n + (j + k) + 1
                    clauses.append([-a, -b])
        return clauses

    def dpll(self, clauses=None, assignment=None):
        """
        DPLL algorithm for solving SAT problems.
        :param clauses: List of clauses (CNF), each clause is a list of literals.
        :param assignment: Current variable assignment as a dictionary.
        :return: A satisfying assignment (dict) or None if unsatisfiable.
        """
        if clauses is None:
            clauses = self.clauses
        if assignment is None:
            assignment = {}
        clauses = self.simplify(clauses, assignment)
        if [] in clauses:
            return None # [] implies the clause is unsatisfiable
        if not clauses:
            return assignment # All clauses satisfied

        # Unit Propagation: assign variables that must be true/false
        unit_clauses = [c[0] for c in clauses if len(c) == 1]
        if unit_clauses:
            lit = unit_clauses[0]
            return self.dpll(clauses, {**assignment, abs(lit): lit > 0})

        # Pure Literal Elimination: assign variables that appear with only one polarity
        literals = [lit for clause in clauses for lit in clause]
        for lit in set(literals):
            if -lit not in literals:
                return self.dpll(clauses, {**assignment, abs(lit): lit > 0})

        # Splitting: try assigning True/False to an unassigned variable
        for clause in clauses:
            for lit in clause:
                variable = abs(lit)
                if variable not in assignment:
                    for val in [True, False]:
                        new_assignment = {**assignment, variable: val}
                        result = self.dpll(clauses, new_assignment)
                        if result is not None:
                            return result
                    return None
        return None

    def simplify(self, clauses, assignment):
        """
        Simplify the CNF clauses based on the current assignment.
        Remove satisfied clauses and remove assigned literals from others.
        :param clauses: List of clauses (CNF).
        :param assignment: Current variable assignment as a dictionary.
        :return: Simplified list of clauses.
        """
        simplified = []
        for clause in clauses:
            new_clause = []
            satisfied = False
            for lit in clause:
                variable = abs(lit)
                if variable in assignment:
                    val = assignment[variable]
                    if (lit > 0 and val) or (lit < 0 and not val):
                        satisfied = True # Clause satisfied
                        break
                    else:
                        continue # Assigned literal is False, skip it
                else:
                    new_clause.append(lit)
            if not satisfied:
                simplified.append(new_clause)
        return simplified

    def walksat(self, max_flips=10000, p=0.5):
        """
        WalkSAT algorithm for solving SAT problems (local search).
        :param max_flips: Maximum number of variable flips allowed.
```

```python
        :param p: Probability of choosing a random variable to flip.
        :return: A satisfying assignment (dict) or None if unsatisfiable.
        """
        import random
        clauses = self.clauses
        variables = set(abs(lit) for clause in clauses for lit in clause)
        model = {v: random.choice([True, False]) for v in variables}
        for _ in range(max_flips):
            unsat = [c for c in clauses if not self.is_clause_satisfied(c, model)]
            if not unsat:
                return model # Found a satisfying assignment
            clause = random.choice(unsat)
            if random.random() < p:
                variable = abs(random.choice(clause)) # Randomly pick a variable to flip
            else:
                # Flip the variable that maximizes the number of satisfied clauses
                variable = max(clause, key=lambda l: self.num_satisfied_after_flip(model, abs(l),
                    clauses))
                variable = abs(variable)
            model[variable] = not model[variable]
        return None # No solution found within max_flips

    def is_clause_satisfied(self, clause, model):
        """
        Check if a clause is satisfied by the current model.
        :param clause: List of literals in the clause.
        :param model: Current variable assignment as a dictionary.
        :return: True if the clause is satisfied, False otherwise.
        """
        for lit in clause:
            val = model[abs(lit)]
            if (lit > 0 and val) or (lit < 0 and not val):
                return True
        return False

    def num_satisfied_after_flip(self, model, variable, clauses):
        """
        Calculate the number of clauses satisfied after flipping a variable,
        so we can choose the best variable to flip.
        :param model: Current variable assignment as a dictionary.
        :param variable: Variable to flip.
        :param clauses: List of clauses (CNF).
        :return: Number of clauses satisfied after flipping the variable.
        """
        flipped = model.copy()
        flipped[variable] = not flipped[variable]
        return sum(self.is_clause_satisfied(c, flipped) for c in clauses)
```

Appendix B: Raw Experimental Data

Raw Performance Data for the DPLL Algorithm on N-Queens Problems

All rows in this section have **Method = DPLL**, **Satisfied = True**.

N	No.	Runtime (ms)	Recursive Calls
4	0	0.21	44
	1	0.19	
	2	0.18	
	3	0.18	
	4	0.17	
	5	0.17	
	6	0.17	
	7	0.17	
	8	0.17	
	9	0.17	
8	0	7.14	426
	1	6.12	
	2	6.48	
	3	5.99	
	4	6.07	
	5	6.19	
	6	6.13	
	7	6.17	
	8	5.97	
	9	6.0	
10	0	13.41	454
	1	13.48	
	2	13.86	
	3	13.07	
	4	13.19	
	5	13.32	
	6	13.83	
	7	12.98	
	8	14.3	
	9	13.41	
12	0	38.85	1020
	1	40.83	
	2	37.92	
	3	40.95	
	4	38.15	
	5	40.92	
	6	38.37	
	7	40.96	
	8	39.02	
	9	40.88	
14	0	182.88	6890
	1	179.27	
	2	176.8	
	3	177.09	
	4	176.88	
	5	176.98	
	6	175.32	
	7	177.82	
	8	183.82	
	9	178.56	
16	0	878.33	35326
	1	879.3	
	2	876.11	
	3	879.54	
	4	878.79	
	5	889.09	
	6	883.5	
	7	877.84	
	8	885.45	
	9	945.09	
20	0	21368.45	732755
	1	21433.81	
	2	21705.65	
	3	21860.83	
	4	21450.25	
	5	21444.33	
	6	21393.06	
	7	21441.1	
	8	21441.02	
	9	21370.59	
24	0	55719.64	1460646
	1	55944.79	
	2	55915.45	
	3	56007.95	
	4	55875.51	
	5	55936.71	
	6	55951.05	
	7	55928.85	
	8	56256.33	
	9	55909.93	

Raw Performance Data for the WalkSAT Algorithm on N-Queens Problems

All rows in this section have **Method = WalkSAT**, **Satisfied = True**.

N	No.	Runtime (ms)	Flips
4	0	35.75	9
	1	2.33	155
	2	0.16	10
	3	0.16	10
	4	0.74	45
	5	0.26	18
	6	0.28	13
	7	0.11	6
	8	1.07	63
	9	0.18	11
8	0	39.28	270
	1	10.7	80
	2	35.46	238
	3	33.57	218
	4	17.08	119
	5	15.18	96
	6	11.09	93
	7	22.25	156
	8	14.52	111
	9	29.19	177
10	0	238.49	713
	1	28.97	93
	2	28.98	97
	3	37.13	133
	4	56.91	172
	5	65.34	207
	6	67.68	210
	7	141.19	391
	8	121.47	412
	9	50.3	151
12	0	261.6	391
	1	334.11	499
	2	241.19	379
	3	245.96	391
	4	94.82	99
	5	360.01	530
	6	227.98	381
	7	407.74	497
	8	300.26	345
	9	289.77	449
14	0	647.0	556
	1	614.1	532
	2	1217.83	1114
	3	1277.29	1153
	4	790.38	688
	5	635.29	584
	6	208.73	221
	7	300.04	289
	8	2013.66	1720
	9	1323.59	1135
16	0	838.3	503
	1	1065.94	582
	2	1455.61	743
	3	1152.95	647
	4	1272.3	676
	5	2696.76	1367
	6	727.32	405
	7	1393.62	761
	8	619.5	349
	9	688.11	363
20	0	2936.44	705
	1	1536.38	406
	2	11312.83	2300
	3	3362.52	740
	4	1866.98	526
	5	3630.77	864
	6	2134.52	541
	7	7882.08	1729
	8	6195.54	1314
	9	2320.94	579
24	0	22222.18	2318
	1	5426.15	693
	2	6027.51	767
	3	15461.12	1717
	4	8994.35	1132
	5	19157.59	2028
	6	4007.65	563
	7	14575.13	1686
	8	11210.85	1249
	9	9128.78	1085

Appendix C: Summary Statistics

The table below summarizes the average and standard deviation of time costs collected over 10 independent runs for each solver and board size.

N	Method	Avg Time (ms)	Std Dev (ms)	Runs
4	DPLL	0.18	0.01	10
8	DPLL	6.23	0.33	10
10	DPLL	13.48	0.38	10
12	DPLL	39.69	1.26	10
14	DPLL	178.54	2.62	10
16	DPLL	887.30	19.63	10
20	DPLL	21490.91	152.99	10
24	DPLL	55944.62	126.08	10
4	WalkSAT	4.10	10.57	10
8	WalkSAT	22.83	10.16	10
10	WalkSAT	83.65	62.67	10
12	WalkSAT	276.34	81.25	10
14	WalkSAT	902.79	522.36	10
16	WalkSAT	1191.04	576.28	10
20	WalkSAT	4317.90	3015.79	10
24	WalkSAT	11621.13	5775.50	10

References

[1] Michael Simkin, *The Number of N-Queens Configurations*, Advances in Mathematics vol. 427, 109127, 2023,

[2] Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, Eighth edition, McGraw-Hill, 2019.

[3] Stuart J. Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, Fourth edition, Pearson, 2021.