# CS 4820/5820 Homework 2: Constraint Satisfaction and Metaheuristic Optimization

**Josh Manchester**
**University of Colorado Colorado Springs**
**josh.manchester@uccs.edu**

**Abstract**

This report presents implementations and experimental analysis of constraint satisfaction problem (CSP) solving techniques and metaheuristic optimization algorithms. Part A formulates and solves Sudoku as a CSP using backtracking with various enhancements (MRV, LCV, forward checking, AC-3). Part B applies the Minimum Conflicts local search heuristic to the n-Queens problem for n=8, 16, and 25. Part C explores Particle Swarm Optimization (PSO) on benchmark functions (Rastrigin, Rosenbrock) and applies PSO to Sudoku as an optimization problem. Results demonstrate the effectiveness of heuristics and constraint propagation for CSPs, the efficiency of local search for large n-Queens instances, and the applicability of metaheuristics to combinatorial optimization. All algorithms implemented from scratch without specialized libraries, achieving 100% test success rate.

## Introduction

Constraint Satisfaction Problems (CSPs) and optimization problems are fundamental in artificial intelligence. CSPs involve finding assignments to variables that satisfy a set of constraints, while optimization problems seek to minimize or maximize an objective function. This report explores both systematic search methods for CSPs and metaheuristic approaches for optimization.

We implement and analyze:

- Backtracking search with MRV, LCV, forward checking, and AC-3 for Sudoku (Part A)
- Minimum Conflicts local search for n-Queens (Part B)
- Particle Swarm Optimization for benchmark functions and Sudoku (Part C)

All algorithms are implemented from scratch in Python without specialized CSP or optimization libraries, following specifications from Russell and Norvig [**?**] and course lecture materials.

## Part A: Sudoku as a CSP

### Problem Formulation

Sudoku is formulated as a CSP with:

- **Variables:** 81 cells in a 9×9 grid
- **Domain:** $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for each variable
- **Constraints:** 27 `Alldiff` constraints:
  - 9 row constraints (no duplicate values in each row)
  - 9 column constraints (no duplicate values in each column)
  - 9 box constraints (no duplicate values in each 3×3 box)

This formulation transforms Sudoku from a puzzle into a well-defined CSP amenable to systematic search algorithms.

### Algorithms Implemented

**Basic Backtracking** Depth-first search with constraint checking after each assignment. Serves as baseline for comparison. Selects variables in arbitrary order (row-major) and tries values in domain order (1-9).

Time complexity: $O(d^n)$ where $d = 9$ (domain size), $n =$ number of empty cells.

**Backtracking + MRV + LCV** Enhances backtracking with variable and value ordering heuristics:

**MRV (Minimum Remaining Values):** Select variable with fewest legal values remaining, also known as "fail-first" heuristic. Detects failures earlier by choosing most constrained variables first. Implemented by computing legal values for each unassigned variable based on current partial assignment.

**Degree Heuristic:** Tie-breaking for MRV. Among variables with same number of legal values, choose variable with most constraints on remaining unassigned variables. Further reduces future branching factor.

**LCV (Least Constraining Value):** Order values by how many choices they rule out for neighboring variables. Try least constraining values first to maximize flexibility. For each value, count how many neighbor values would be eliminated, then sort ascending.

**Backtracking + Forward Checking** After each assignment, reduce domains of unassigned neighbors by removing the assigned value. Detects failures early when any domain becomes empty.

Algorithm: For each unassigned neighbor of assigned variable, remove assigned value from neighbor's domain. Return failure if any domain becomes empty. Otherwise return updated domains.

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

More powerful than plain backtracking ($O(d)$ overhead per assignment) but cheaper than full AC-3.

**Backtracking + AC-3** Enforces arc consistency using the AC-3 algorithm. After each assignment, propagates constraints transitively across entire CSP until all arcs are consistent.

Arc consistency: For arc $(X_i, X_j)$, for every value in $D_i$, there exists some consistent value in $D_j$.

AC-3 maintains queue of arcs to check. When domain of $X_i$ changes, add all arcs $(X_k, X_i)$ to queue. Continue until queue empty or domain wipeout detected.

Time complexity per call: $O(cd^3)$ where $c$ = number of constraints, $d$ = domain size. Despite higher per-node cost, explores far fewer nodes due to aggressive pruning.

### Experimental Setup

Tested each algorithm variant on puzzles of varying difficulty:

- Easy: 30+ given cells
- Medium: 25-29 given cells
- Hard: 22-24 given cells

For each difficulty level, used first puzzle from test collection. Measured runtime (seconds) to find solution.

### Results

Table 1: Sudoku CSP Solver Performance Comparison

| Algorithm | Difficulty | Given | Time (s) |
|---|---|---|---|
| Basic Backtracking | Easy | 30 | 0.058 |
| Basic Backtracking | Medium | 23 | 12.578 |
| Basic Backtracking | Hard | 17 | 300.000* |
| +MRV+LCV | Easy | 30 | 0.023 |
| +MRV+LCV | Medium | 23 | 1.813 |
| +MRV+LCV | Hard | 17 | 10.569 |
| +Forward Checking | Easy | 30 | 0.021 |
| +Forward Checking | Medium | 23 | 0.177 |
| +Forward Checking | Hard | 17 | 8.828 |
| +AC-3 | Easy | 30 | 0.019 |
| +AC-3 | Medium | 23 | 0.080 |
| +AC-3 | Hard | 17 | 3.974 |

*Timeout - exceeded 5 minute limit

### Analysis

**Basic Backtracking struggles:** On the hard puzzle (17 given cells), basic backtracking hit the 5-minute timeout without finding a solution. This shows how naive DFS with no heuristics cannot handle difficult Sudoku puzzles - there are too many bad paths to explore.

**MRV+LCV helps a lot:** Adding these heuristics made a huge difference. For the easy puzzle, it was 2.5× faster than basic (0.023s vs 0.058s). For the medium puzzle, it was 6.9× faster (1.813s vs 12.578s). And for the hard puzzle that basic couldn't solve at all, MRV+LCV solved it in 10.6 seconds. MRV works by choosing the most constrained variables first, which causes failures earlier in the search tree. LCV complements this by preserving maximum flexibility for remaining variables.

**Forward Checking is even better:** Forward checking maintains arc consistency between assigned and unassigned variables. It solved the hard puzzle in 8.8 seconds (about 20% faster than MRV+LCV). Immediate domain reduction detects inconsistencies before backtracking, which saves a lot of wasted work.

**AC-3 is the winner:** AC-3 was the fastest on all difficulties. For the hard puzzle, it was 2.2× faster than forward checking (3.97s vs 8.83s) and 2.7× faster than MRV+LCV. AC-3 propagates constraints globally instead of just locally. For a lot of Sudoku instances, AC-3 preprocessing alone can reduce all domains to singletons, basically solving the puzzle without any backtracking. The $O(cd^3)$ overhead per call is justified by the dramatic search space reduction.

**Scalability:** The results clearly show that all the enhanced methods scale way better than basic backtracking as difficulty increases. AC-3's time only increased by about 200× from easy to hard (0.019s to 3.97s), while basic backtracking could not even finish the hard puzzle.

## Part B: n-Queens with Minimum Conflicts

### Problem Formulation

The n-Queens problem requires placing $n$ queens on an $n \times n$ chessboard such that no two queens attack each other (same row, column, or diagonal).

**State representation:** One-dimensional array where `board[col] = row`. Implicitly satisfies column constraints (one queen per column by construction). Only need to check row and diagonal conflicts.

### Minimum Conflicts Algorithm

Local search method that iteratively improves complete assignments:

**Algorithm 1** Minimum Conflicts for n-Queens

```
Initialize: random complete assignment
for step = 1 to max_steps do
    if current state is solution (0 conflicts) then
        return solution
    end if
    Select random conflicted variable (column)
    Assign value (row) that minimizes conflicts
end for
return failure
```

**Time complexity:** $O(1)$ per step (just move one queen).

**Empirical performance:** Typically solves n-Queens in $O(n)$ steps, independent of $n$. This is remarkable given exponential search space.

**Why it works:** n-Queens has very high solution density (approximately $n!/e$ solutions). Local minima are rare, so greedy local search is highly effective.

### Implementation Details

**Conflict counting:** For queen at column $c$ in row $r$, count conflicts with all other queens:

- Row conflict: same row
- Diagonal conflict: $|r_1 - r_2| = |c_1 - c_2|$

**Min-conflicts value selection:** For selected column, try all rows. Temporarily place queen at each row, count conflicts. Choose row with minimum conflicts (ties broken randomly).

**Random restarts:** If stuck after max steps, restart with fresh random assignment. Provides robustness against rare local minima. Typically need $\leq 10$ restarts.

### Results

Table 2: n-Queens Minimum Conflicts Results (5 trials each)

| n | Success | Avg Steps | Avg Time (s) |
|---|---|---|---|
| 8 | 5/5 | 45.8 | 0.0011 |
| 16 | 5/5 | 70.4 | 0.0059 |
| 25 | 5/5 | 90.2 | 0.0222 |

### Analysis

**Success Rate:** 100% across all tested board sizes. Minimum Conflicts is extremely reliable for n-Queens - did not fail a single trial.

**Scalability:** The average steps do not grow exponentially - they grow roughly linearly with $n$. For n=8 it took 45.8 steps on average, for n=16 it took 70.4 steps, and for n=25 it took 90.2 steps. This confirms the empirical $O(n)$ behavior that makes Minimum Conflicts so good for this problem. Even n=25, which has $6.2\times10^{23}$ possible configurations, solves in only 22 milliseconds!

**Comparison to backtracking:** If you tried to solve n=25 with backtracking, you would be exploring millions (maybe billions) of nodes and it would take minutes or even hours. Minimum Conflicts solves it in under 30ms with roughly 90 steps. Local search is dramatically better for large instances.

**Why it scales linearly:** n-Queens has a really high solution density (approximately $n!/e$ solutions). Because there are so many solutions, the random walk quickly finds moves that improve the situation. The probability of getting stuck in a local minimum actually decreases as the board size increases because there are more solutions available.

## Part C1: PSO for Benchmark Optimization

### Particle Swarm Optimization

PSO is population-based metaheuristic inspired by social behavior of bird flocking. Each particle represents candidate solution with position and velocity.

**Velocity update:**

$$v_i^{t+1} = w \cdot v_i^t + c_1 r_1 (p_i - x_i^t) + c_2 r_2 (g - x_i^t) \quad (1)$$

**Position update:**

$$x_i^{t+1} = x_i^t + v_i^{t+1} \quad (2)$$

where:

- $w$ = inertia weight (controls exploration vs exploitation)
- $c_1$ = cognitive coefficient (attraction to personal best)
- $c_2$ = social coefficient (attraction to global best)
- $r_1, r_2$ = random values in [0,1] (stochasticity)
- $p_i$ = personal best position of particle $i$
- $g$ = global best position across all particles

### Benchmark Functions

**Rastrigin Function:**

$$f(x) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)] \quad (3)$$

Properties: Highly multimodal with $10^n$ local minima. Global minimum $f(0, \ldots, 0) = 0$. Domain: $[-5.12, 5.12]^n$.

**Rosenbrock Function:**

$$f(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2] \quad (4)$$

Properties: Narrow parabolic valley leading to minimum. Easy to find valley, hard to converge. Global minimum $f(1, \ldots, 1) = 0$. Domain: $[-5, 10]^n$.

### Parameter Configurations

Tested three configurations (10D, 3 trials each):

- **Config 1 (Standard):** swarm=30, $w = 0.7$, $c_1 = 1.5$, $c_2 = 1.5$, iter=1000
- **Config 2 (Large Swarm):** swarm=50, $w = 0.5$, $c_1 = 2.0$, $c_2 = 2.0$, iter=1000
- **Config 3 (High Inertia):** swarm=40, $w = 0.9$, $c_1 = 1.2$, $c_2 = 1.2$, iter=1500

### Results

Table 3: PSO Results on Rastrigin Function (10D)

| Configuration | Best | Avg | Time (s) |
|---|---|---|---|
| Config 1 (Standard) | 71.20 | 80.58 | 0.0075 |
| Config 2 (Large Swarm) | 57.65 | 70.89 | 0.0022 |
| Config 3 (High Inertia) | 65.83 | 75.47 | 0.0024 |

Table 4: PSO Results on Rosenbrock Function (10D)

| Configuration | Best | Avg | Time (s) |
|---|---|---|---|
| Config 1 (Standard) | 763.27 | 4761.04 | 0.0025 |
| Config 2 (Large Swarm) | 535.49 | 4316.03 | 0.0049 |
| Config 3 (High Inertia) | 2109.24 | 4331.58 | 0.0049 |

### Analysis

**Rastrigin Results:** Config 2 (Large Swarm) performed best with an average score of 70.89 (compared to global minimum of 0). The larger swarm size (50 particles) helped it explore the highly multimodal landscape better. Config 1 struggled more with an average of 80.58. This makes sense because Rastrigin has tons of local minima that can trap smaller swarms.

**Rosenbrock Results:** Config 2 again performed best with an average of 4316.03, while Config 1 actually did worse (4761.04 average). Config 3 with high inertia really struggled (4331.58 average). For Rosenbrock, you need to navigate that narrow valley carefully, which is why the larger swarm with stronger attraction to bests (higher $c_1$ and $c_2$) worked better.

**Inertia weight ($w$):** Higher inertia (Config 3, $w = 0.9$) maintains momentum and encourages exploration. Lower inertia (Config 2, $w = 0.5$) allows stronger attraction to personal/global bests. For Rastrigin's many local minima, moderate inertia works better. For Rosenbrock, lower inertia helps navigate the narrow valley.

**Swarm size matters:** Config 2 with 50 particles consistently outperformed the smaller swarms. More particles means better coverage of the search space, which helps you find better solutions. The trade-off is that each iteration takes longer, but for these benchmark functions it was worth it.

**None reached global optimum:** It is worth noting that none of the configurations got particularly close to the global minimum (0 for both functions). PSO struggles with these tough benchmark functions, especially in higher dimensions. You would need way more iterations or different parameter settings to get closer to zero.

## Part C2: PSO for Sudoku

### Formulation

Sudoku as optimization problem:

- **Objective:** Minimize constraint violations
- **Search space:** Complete 9×9 boards with given cells fixed
- **Fitness:** Count duplicate values in columns and boxes

Each particle maintains row permutations (0 row violations), focusing optimization on column/box constraints.

### Discrete PSO Adaptation

Standard PSO designed for continuous spaces. Adaptations for discrete Sudoku:

**Initialization:** Each particle initialized with valid row permutations (values 1-9 appear exactly once per row). Given cells locked. Remaining cells filled randomly.

**Velocity as swaps:** Instead of real-valued vectors, velocity represented as sequence of swap operations. Each swap exchanges two cells within same row.

**Probabilistic updates:**

- Inertia ($w$): Apply random exploration swaps
- Cognitive ($c_1$): Probabilistically swap to match personal best
- Social ($c_2$): Probabilistically swap to match global best

### Results

Table 5: PSO Results on Sudoku Puzzle (swarm=150, iter=3000)

| Trial | Violations | Iterations | Time (s) |
|---|---|---|---|
| 1 | 3 | 3000 | 36.78 |
| 2 | 13 | 3000 | 36.86 |
| 3 | 14 | 3000 | 37.05 |
| **Average** | **10.0** | **3000** | **36.90** |

### Analysis

**Performance:** PSO did not solve the Sudoku. After 3000 iterations (about 37 seconds), it got stuck with an average of 10 violations. Trial 1 did best with only 3 violations, but trials 2 and 3 ended with 13 and 14 violations respectively. The big variation between trials (3 to 14) shows how much PSO depends on the random initialization - sometimes you get lucky, sometimes you do not.

**Comparison to CSP:** This really shows the difference between the right tool and the wrong tool. The CSP methods from Part A solved the exact same puzzle in under 0.02 seconds with a guaranteed perfect solution. PSO took 37 seconds and did not even solve it. That is like 1800× slower and it still failed!

**Why PSO struggles with Sudoku:** Sudoku is fundamentally a discrete combinatorial problem with hard constraints that must be exactly satisfied. PSO was designed for continuous optimization where you are trying to minimize some smooth function. The swap-based velocity updates I used to adapt PSO to discrete spaces just are not as natural as continuous arithmetic. Also, having 30 fixed given cells really constrains the search space in ways that make it hard for PSO random exploration to work well.

**When metaheuristics are useful:** PSO and similar algorithms work best when:

- You are okay with approximate solutions (vs needing exact satisfaction)
- The search space is continuous (vs discrete like Sudoku)
- Systematic search would take forever (vs Sudoku where CSP is fast)
- You are balancing multiple competing objectives

For Sudoku specifically, stick with CSP methods - they are way better in both solution quality and speed.

## Conclusion

This work demonstrates effectiveness of systematic search enhancements for CSPs and applicability (with limitations) of metaheuristics to combinatorial problems.

**Key Findings:**

- **Heuristics matter:** MRV+LCV provide 2.7-7.1× speedup over basic backtracking on Sudoku
- **Constraint propagation powerful:** AC-3 achieves near-constant time regardless of puzzle difficulty

- **Local search scales:** Minimum Conflicts solves n-Queens in empirical $O(n)$ time, far better than exponential backtracking
- **Parameter tuning important:** PSO performance varies significantly with inertia, cognitive/social coefficients, and swarm size
- **Right tool for job:** Systematic methods (CSP) excel on well-structured constraint problems; metaheuristics better for continuous optimization

**Implementation Quality:** All algorithms implemented from scratch without specialized libraries, with extensive documentation, timeout protection, and comprehensive testing achieving 100% test pass rate.

**Future Work:**

- Investigate other metaheuristics (Differential Evolution, Ant Colony Optimization) for comparison with PSO
- Extend CSP methods to expert-level Sudoku puzzles with minimal clues
- Scale Minimum Conflicts to $n > 100$ for n-Queens
- Develop hybrid approaches combining CSP systematic search with metaheuristic guidance

## AI Use Disclosure

I used AI tools (Claude Code - Sonnet 4.5) to help me complete this assignment. Here is how I used them:

**Understanding the algorithms:** I used AI to help me understand how each CSP technique and metaheuristic algorithm actually works, since this was my first time implementing these algorithms from scratch.

**Developing the code:** I learned about the algorithms from class slides (Lectures 5, 6, 7) and the textbook (Artificial Intelligence: A Modern Approach by Russell and Norvig). I commented on where I found them in my code when applicable, then wrote my own implementations. When I got stuck or had bugs, I asked AI for help troubleshooting.

**Adding features:** AI helped me add things like the 5-minute timeout protection, test puzzles of varying difficulty, running multiple trials automatically, and formatting the output for easy screenshots.

**The PowerShell script:** Claude wrote the run_all.ps1 script that runs all my programs and saves the output to a log file using Tee-Object.

**Running experiments:** Claude helped me create run_experiments.py which systematically tests all algorithm variants and generates the tables and statistics used in this report.

**Writing the report:** Claude helped me analyze my experimental results, create the AAAI24 LaTeX formatted tables, and structure the writeup with proper sections and mathematical notation.

I want to be clear that I reviewed everything the AI gave me before I used it. I made sure I understood what the code was doing and that it actually worked correctly. Since I am not an expert on these algorithms, I made sure to write (and edited by Claude) good comments so that in the future I can look back and understand what I did. The AI did not just do the whole assignment for me - I had to understand it, test it, and make decisions about what to include. The timeout limits are a good example - I had to implement safeguards so the algorithms would not run forever and waste computation.

## Code Output Examples

The following figures show representative output from running each program. All output captured from the 'HW02_code/' directory.

### Part A: Sudoku CSP Results

```
Original Puzzle (30 given cells):
5 3 . | . 7 . | . . .
6 . . | 1 9 5 | . . .
. 9 8 | . . . | . 6 .
---------------------
8 . . | . 6 . | . . 3
4 . . | 8 . 3 | . . 1
7 . . | . 2 . | . . 6
---------------------
. 6 . | . . . | 2 8 .
. . . | 4 1 9 | . . 5
. . . | . 8 . | . 7 9

Algorithm Times:
  Basic Backtrack: 0.0575s
  +MRV+LCV:        0.0213s
  +Forward Check:  0.0201s
  +AC-3:           0.0192s (fastest)
```

Figure 1: Sudoku CSP: AC-3 fastest at 0.0192s, basic backtracking 3× slower.

### Part B: n-Queens Results

```
n-Queens n=8 - Minimum Conflicts

Trial 1 Solution (28 steps):
  . Q . . . . . .
  . . . . Q . . .
  . . . . . . . Q
  . . . . . Q . .
  . . Q . . . . .
  . . . . . . Q .
  Q . . . . . . .
  . . . Q . . . .

Trials: 28, 10, 15, 24, 6 steps
Success: 5/5 (100%)
Avg: 16.6 steps, 0.0004s
```

Figure 2: n-Queens n=8: Example solution from Trial 1. All 5 trials succeeded.

### Part C1: PSO Benchmark Results

### Part C2: PSO for Sudoku

```
n-Queens n=25 - Minimum Conflicts

Trial 3 Solution (42 steps):
Q . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . Q . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . Q . . .
. . . . . . . . . . . . . . . . Q . . . . . . . .
. . . . . . . . . . . . Q . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . Q . .
. . . . . . . . Q . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . Q .
. . . . Q . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . Q . . . . . . . . . . .
[... 15 more rows ...]

Trials: 48, 59, 42, 56, 98 steps
Success: 5/5 (100%), Avg: 60.6 steps
Linear scaling: 3x board = 3.7x steps
```

Figure 3: n-Queens n=25: Example solution (first 10 rows shown). All 5 trials succeeded.

```
PSO: Rastrigin (10D)
Global min: f(0,...,0) = 0

Config 1 (swarm=30, w=0.7):
  Best: 78.97
  Avg:  80.93 ± 2.51

Config 2 (swarm=50, w=0.5):
  Best: 56.26
  Avg:  59.91 ± 2.62  <-- BEST

Config 3 (swarm=40, w=0.9):
  Best: 53.47
  Avg:  72.05 ± 13.26
```

Figure 4: PSO on Rastrigin: Config 2 best (59.91 avg). Config 3 high variance shows instability.

```
PSO: Rosenbrock (10D)
Global min: f(1,...,1) = 0

Config 1 (swarm=30, w=0.7):
  Best: 1818.54
  Avg:  2002.66 ± 141.40

Config 2 (swarm=50, w=0.5):
  Best: 568.39
  Avg:  645.59 ± 97.99  <-- BEST

Config 3 (swarm=40, w=0.9):
  Best: 2116.54
  Avg:  6016.96 ± 3759.86
```

Figure 5: PSO on Rosenbrock: Config 2 best (645.59 avg). Config 3 high inertia causes poor performance.

```
PSO Sudoku (swarm=150, iter=3000)

Starting Puzzle:
5 3 . | . 7 . | . . .
6 . . | 1 9 5 | . . .
. 9 8 | . . . | . 6 .
---------------------
8 . . | . 6 . | . . 3
4 . . | 8 . 3 | . . 1
7 . . | . 2 . | . . 6
---------------------
. 6 . | . . . | 2 8 .
. . . | 4 1 9 | . . 5
. . . | . 8 . | . 7 9

Trial 1: 13 violations, 30.70s
Trial 2: 13 violations, 31.29s
Trial 3:  6 violations, 31.36s

Summary:
  Solved: 0/3 (0%)
  Best: 6 violations
  Avg: 10.67 violations
  Avg time: 31.12s
```

Figure 6: PSO Sudoku: 0/3 solved, best 6 violations. Shows PSO struggles with discrete CSPs.