In The Name of God

Shiraz University

Navid Zare

Evolutionary Computing

The N-Queens Documentation

N-Queens Genetic Algorithm

Technical Documentation
Course: Evolutionary Computing (2025)
Institution: Shiraz University, Department of Computer Science
Language: Python 3.14.0
Last Updated: Feb 2025
Dependencies: NumPy, Matplotlib, Pandas

Table of Contents

---

1. Introduction

1.1 Problem Definition

> The N-Queens problem requires placing N chess queens on an N×N chessboard such that no two queens attack each other (same row, column, or diagonal).

Solution Representation: Permutation encoding [0, 1, 2, ..., N-1] where:

- Index = column position
- Value = row position
- This inherently prevents row and column conflicts

Example (N=4): [1, 3, 0, 2] means:

- Column 0: Queen at row 1
- Column 1: Queen at row 3
- Column 2: Queen at row 0
- Column 3: Queen at row 2

1.2 Fitness Function

Fitness is calculated as the **negative count of diagonal conflicts**:

```
fitness = 0
for each queen i at position (i, chromosome[i]):
   for each subsequent queen j at position (j, chromosome[j]):
      if |i - j| == |chromosome[i] - chromosome[j]|:  # Diagonal check
         fitness -= 1
```

- **Optimal fitness:** 0 (no conflicts)
- **Range:** [-N(N-1)/2, 0]
- **Objective:** Maximization (higher is better)

Time Complexity: $O(N^2)$ per evaluation

---

2. System Architecture

2.1 File Structure

```
project/
├── main.py                    # Entry point
├── queens_ga.py               # Core GA engine
├── individual.py              # Candidate solution class
├── crossover_functions.py     # Recombination operators
├── mutation_functions.py      # Mutation operators
├── survival_functions.py      # Selection strategies
├── utility_functions.py       # Helper functions
```

```
├── plotting_functions.py          # Visualization toolkit
├── all_experiments_Runner.py      # Experiment orchestration
└── Results/                       # Output directory (auto-created)
    ├── *.png                      # Generated plots
    └── *.csv                      # Statistical tables
```

2.2 Design Principles

- **Modular Design:** Operators are interchangeable via strategy pattern
- **Lazy Evaluation:** Fitness cached to avoid redundant calculations
- **Repair Mechanism:** Invalid chromosomes automatically fixed
- **Comprehensive Logging:** Tracks convergence history and statistics

---

3. Core Components

3.1 Individual Class (individual.py)

Encapsulates a candidate solution with metadata.

Key Attributes:

```
_chromosome      # NumPy array: queen positions
_fitness         # Cached fitness value
_fitness_calculated  # Boolean flag for lazy evaluation
_age             # Generations survived
_generationBirth # Creation generation
_id              # Unique identifier (auto-increment)
```

Essential Methods:

- get_chromosome() / set_chromosome(value) - Invalidates fitness cache on update
- get_fitness() / set_fitness(fitness) - Manages cached evaluation
- is_fitness_calculated() - Checks if fitness needs recomputation

Design Rationale:

- Unique IDs enable tracking across generations

- Lazy evaluation reduces redundant $O(N^2)$ fitness calculations
- Age tracking supports adaptive strategies (not currently used)

3.2 QueensGA Class (queens_ga.py)

Main genetic algorithm engine orchestrating the evolutionary process.

Configuration Parameters:

Parameter	Default	Description
n	8	Board size (N×N)
pop_size	100	Population size
mutation_prob	0.5	Mutation probability per individual
recombination_rate	1.0	Crossover application probability
max_evaluations	10000	Evaluation budget (termination criterion)
crossover_type	'cut_and_fill'	Crossover operator selection
mutation_type	'swap'	Mutation operator selection
survival_strategy	'fitness_based'	Survivor selection method

Core Algorithm:

```
def run_ga():
    initialize_population()  # Random permutations

    while not termination_criteria_met():
        # Evaluation
        evaluate_population()
        record_statistics()

        # Selection
        parent1, parent2 = select_parents()  # Tournament (size=5)

        # Variation
```

```
        child1, child2 = recombine(parent1, parent2)
        child1 = mutate(child1)
        child2 = mutate(child2)

        # Replacement
        population = select_survivors([child1, child2])

        generation += 1

    return best_individual, convergence_history
```

Termination Criteria:

1. Optimal solution found (fitness = 0), **OR**
2. Evaluation budget exhausted (evaluations ≥ max_evaluations)

---

4. Genetic Operators

4.1 Crossover Operators (crossover_functions.py)

All crossover operators maintain permutation validity.

Cut-and-Fill Crossover (Default)

Algorithm:

1. Select random cut point
2. Copy prefix [0:cut_point] from parent to child
3. Fill remaining positions from other parent, skipping duplicates
4. Wrap around if necessary

Time Complexity: $O(N^2)$
Pros: Preserves large contiguous segments; good exploitation
Cons: Sequential fill may introduce bias

Best For: N-Queens (preserves diagonal-free segments)

Partially Mapped Crossover (PMX)

Algorithm:

1. Select two random cut points
2. Copy segment [point1:point2] from parent1 to child1
3. Map conflicting values from parent2 using position relationships
4. Fill remaining positions from parent2

Time Complexity: $O(N^2)$
Pros: Preserves relative ordering from both parents
Cons: More complex; higher computational overhead

N-Cut Crossover (2-cut, 3-cut)

Algorithm:

1. Generate N random cut points
2. Alternate between direct copy and fill-from-other-parent
3. Maintain permutation validity during fill

Time Complexity: $O(N^2 \times n\_cuts)$
Observation: Increasing cuts (2→3) shows **diminishing returns** (see Task 3 results)

4.2 Mutation Operators (mutation_functions.py)

Swap Mutation (Recommended)

Algorithm:

```
if random() < mutation_prob:
    i, j = select_two_random_indices()
    chromosome[i], chromosome[j] = chromosome[j], chromosome[i]
```

Time Complexity: O(N)
Properties:

- Naturally preserves permutation validity
- Low disruption (single transposition)
- Ideal for exploitation phase

Best For: N-Queens (no repair needed)

Bitwise Mutation

Algorithm:

for each gene in chromosome:

if random() < mutation_prob:

gene = random_value(0, N-1)

Time Complexity: O(N)
Critical Issue: Violates permutation constraint

- Introduces duplicate values
- Requires costly repair operation
- Adds significant computational overhead
- Disrupts evolutionary process

Recommendation: Not suitable for N-Queens (see Task 2 analysis)

4.3 Survival Strategies (survival_functions.py)

Fitness-Based Replacement (Default)

Replaces worst individuals only if offspring are better.

Algorithm:

1. Sort population by fitness (descending)
2. If child1 > worst_individual: replace
3. If child2 > next_worst_individual: replace

Selection Pressure: Moderate
Diversity: High (weak individuals persist longer)
Convergence Speed: Slow (~94 generations for N=8)

Generational Replacement

Replaces entire population each generation.

Algorithm:

1. Generate pop_size/2 offspring pairs
2. Replace all parents with offspring

Selection Pressure: Low
Diversity: Maximum (complete turnover)

Convergence Speed: Fast (~3 generations for N=8)
Best For: Early exploration; avoiding premature convergence

Elitism Replacement

Preserves top N individuals, replaces rest.

Algorithm:

1. Identify top n_elite individuals (default: 2)
2. Generate offspring to fill remaining slots
3. Keep elite + best offspring

Selection Pressure: High
Diversity: Moderate (elite preserved)
Convergence Speed: Fastest (~2.3 generations for N=8)
Best For: Exploitation; maintaining best solutions

---

5. Experimental Framework

5.1 Task 2: Parameter Sensitivity Analysis

Tests impact of key parameters on algorithm performance.

Experiments:

Parameter	**Values Tested**	**Runs**	**Key Finding**
Mutation Probability	0.2, 0.5, 1.0	30	**Higher is better:** 1.0 achieves ~59 gen (±65)
Recombination Rate	0.5, 1.0	30	**1.0 recommended:** ~85 gen vs 145 gen at 0.5
Mutation Type	swap, bitwise	30	**Swap only:** Bitwise breaks permutations

Output Files:

- parameter_sensitivity_mutation_probability.png
- parameter_sensitivity_recombination_rate.png
- mutation_probability_comparison.csv

5.2 Task 3: Crossover Strategy Exploration

Compares effectiveness of different recombination operators.

Tested Operators:

- Cut-and-Fill: ~81 gen (±93) - **Best balance**
- PMX: ~57 gen (±78) - High variance (inconsistent)
- 2-Cut: ~86 gen (±107)
- 3-Cut: ~82 gen (±76) - **No improvement over 2-cut**

Key Insight: Cut-and-Fill provides reliable performance for N-Queens by preserving large conflict-free segments.

Output Files:

- crossover_comparison.png
- crossover_type_comparison.csv

5.3 Task 4: Survival Strategy Comparison

Evaluates impact of selection pressure on convergence.

Results:

Strategy	**Avg Generations**	**Variance**	**Selection Pressure**
Elitism	~2.3	±2.0	**High** (fastest)
Generational	~3.0	±1.9	Low (diverse)
Fitness-based	~94	±136	Moderate (slow)

Recommendation:

- Use **Elitism** for fast convergence on well-defined problems
- Use **Generational** for exploration/early phases
- Avoid Fitness-based for simple problems (too slow)

Output Files:

- survival_strategy_comparison.png
- survival_strategy_comparison.csv

5.4 Task 5: Scalability Study

Analyzes algorithm behavior across problem sizes.

Results:

N	Success Rate	Avg Generations	Avg Evaluations	Complexity
8	100%	113 (±161)	325 (±322)	Low
10	100%	394 (±505)	888 (±1010)	Moderate
12	83.3%	592 (±458)	1283 (±916)	**Transition point**
20	86.7%	2625 (±1915)	5449 (±3830)	High

Degradation Factors:

1. **Search Space Explosion:** N! permutations (8! = 40,320 → 20! = 2.4×10^{18})
2. **Complexity Growth:** Near-quadratic (Generations/N^2)
3. **Fixed Resources:** pop_size=100-200 insufficient for large N
4. **Fitness Landscape:** Exponentially more potential conflicts

Observations:

- N≤10: Reliable solutions with standard settings
- N=12: Success rate drops (transition point)
- N=20: Requires 5× evaluation budget, still 13% failure rate

Output Files:

- scalability_study.png
- n_value_comparison.csv

---

6. Usage Guide

6.1 Quick Start

Single Run:

```
from queens_ga import QueensGA
from utility_functions import visualize_chessboard


# Initialize and run
ga = QueensGA(n=8, pop_size=100, mutation_prob=0.5)
```

```
ga.initialize_population()
best_ind, fitness, gens, history, _ = ga.run_ga()

# Display result
print(f'Fitness: {fitness}, Generations: {gens}")
visualize_chessboard(best_ind)
```

Output Example:

```
Fitness: 0, Generations: 73
Chessboard:
Q  .  .  .  .  .  .  .
.  .  .  .  Q  .  .  .
.  Q  .  .  .  .  .  .
...
```

6.2 Running Experiments

All Experiments (Complete Suite):

```
from all_experiments_Runner import run_all_experiments

run_all_experiments()  # ~30-60 minutes for 540 total runs
```

Individual Tasks:

```
from all_experiments_Runner import (
   run_task_2_only,  # Parameter sensitivity
   run_task_3_only,  # Crossover comparison
   run_task_4_only,  # Survival strategies
   run_task_5_only   # Scalability study
)

run_task_2_only()  # Run only Task 2
```

6.3 Custom Configuration

```
from all_experiments_Runner import run_experiment
from plotting_functions import plot_single_experiment

# Custom experiment
results = run_experiment(
    num_runs=30,
    n=12,
    pop_size=150,
    mutation_prob=0.7,
    recombination_rate=1.0,
    max_evaluations=15000,
    crossover_type='cut_and_fill',
    mutation_type='swap',
    survival_strategy='elitism'
)

# Visualize
plot_single_experiment(results, "Custom 12-Queens")
```

6.4 Recommended Configurations

For Quick Testing (N=8):

n=8, pop_size=50, max_evaluations=5000

mutation_prob=0.5, recombination_rate=1.0

crossover_type='cut_and_fill', survival_strategy='elitism'

For Standard Performance (N=8-12):

n=8-12, pop_size=100, max_evaluations=10000

mutation_prob=1.0, recombination_rate=1.0

crossover_type='cut_and_fill', survival_strategy='elitism'

For Large-Scale Problems (N≥20):

n=20, pop_size=200, max_evaluations=50000

mutation_prob=0.3, recombination_rate=1.0

crossover_type='cut_and_fill', survival_strategy='elitism'

---

7. Results Interpretation

7.1 Output Files

All results saved in Results/ directory.

Plots:

- Parameter sensitivity analysis (3 files)
- Crossover comparison
- Survival strategy comparison
- Scalability study
- Baseline experiment results

Data Tables (CSV):

- Statistical summaries for each experiment
- Columns: Success Rate, Avg Generations, Avg Evaluations, Fitness

7.2 Visualization Components

Convergence Curves:

- X-axis: Generation number
- Y-axis: Best fitness (higher is better)
- Red dashed line: Optimal fitness (y=0)
- Shaded regions: Standard deviation bands
- Interpretation: Steeper descent = faster convergence

Success Rate Charts:

- Percentage of runs achieving fitness=0
- Higher is better

- Indicates algorithm reliability

Box Plots:

- Center line: Median
- Box: 25th-75th percentile (interquartile range)
- Whiskers: Min/Max (excluding outliers)
- Outliers: Individual points beyond whiskers
- Interpretation: Narrower box = more consistent performance

Scalability Analysis:

- Log-scale plots: Show exponential growth
- Normalized complexity: Reveals algorithmic scaling behavior

7.3 Key Metrics

Success Rate:

- **Definition:** Percentage achieving fitness=0
- **Good:** >90%
- **Acceptable:** 70-90%
- **Poor:** <70%

Average Generations (Successful Runs Only):

- **Fast:** <100 generations (N=8)
- **Moderate:** 100-500 generations
- **Slow:** >500 generations
- **Note:** Lower variance indicates reliability

Average Evaluations (All Runs):

- Measures computational cost
- Compare across configurations with similar success rates
- Lower is better (efficiency)

Final Fitness Distribution:

- Tight distribution around 0 = consistent
- Wide spread = unreliable

- Median closer to 0 = better average case

---

8. Performance Analysis

8.1 Optimal Configuration (N=8)

Based on experimental results:

```
n = 8
pop_size = 100
mutation_prob = 1.0          # Task 2.1: Best convergence
recombination_rate = 1.0     # Task 2.2: Lower variance
crossover_type = 'cut_and_fill'  # Task 3: Most reliable
mutation_type = 'swap'       # Task 2.3: Only viable option
survival_strategy = 'elitism'  # Task 4: Fastest (2.3 gen)
max_evaluations = 10000
```

Expected Performance:

- Success Rate: 100%
- Generations: ~50-80
- Evaluations: ~200-300

8.2 Complexity Analysis

Time Complexity per Generation:

```
Population evaluation: O(N × pop_size)        # Dominant factor
Parent selection: O(1)                        # Fixed tournament size
Crossover: O(N²)                              # Worst case
Mutation: O(N)                                # Single pass
Survivor selection: O(pop_size × log pop_size) # Sorting
```

Total per generation: $O(N \times pop\_size + N^2)$

Space Complexity:

Population storage: O(N × pop_size)

Fitness history: O(generations)

Total: O(N × pop_size)

8.3 Scalability Limits

Practical Limits with Standard Settings:

- **N≤10:** Excellent (100% success, fast convergence)
- **N=12:** Good (83% success, moderate speed)
- **N≤16:** Feasible (requires increased resources)
- **N≥20:** Challenging (high variance, 85-90% success)

For N≥20, consider:

- Increasing population size (200-500)
- Higher evaluation budgets (50000-100000)
- Hybrid approaches (GA + local search)
- Parallel island models

8.4 Common Issues & Solutions

Issue	Cause	Solution
Low success rate	Insufficient exploration	Increase pop_size or max_evaluations
Premature convergence	Too much exploitation	Use survival_strategy='generational'
Slow convergence	Weak selection pressure	Switch to survival_strategy='elitism'
High variance	Stochastic instability	Increase mutation_prob to 1.0
Invalid chromosomes	Crossover/mutation bugs	Automatic repair via repair_chromosome()

Appendix A: Experimental Results Summary

Task 2 Findings

- **Mutation Probability:** 1.0 optimal (59 gen vs 231 gen at 0.2)
- **Recombination Rate:** 1.0 recommended (85 gen vs 145 gen at 0.5)
- **Mutation Type:** Swap only (bitwise unsuitable for permutations)

Task 3 Findings

- **Best Crossover:** Cut-and-Fill (~81 gen, low variance)
- **PMX Performance:** Fast average but inconsistent (high variance)
- **N-Cut Analysis:** No benefit from increasing cuts (2-cut ≈ 3-cut)

Task 4 Findings

- **Fastest:** Elitism (2.3 gen, ±2.0)
- **Most Diverse:** Generational (3.0 gen, ±1.9)
- **Slowest:** Fitness-based (94 gen, ±136)

Task 5 Findings

- **Reliable Range:** N≤10 (100% success)
- **Transition Point:** N=12 (83% success)
- **Challenge Zone:** N≥20 (87% success, high variance)

Appendix B: Utility Functions

repair_chromosome(chromosome):

- Fixes permutation violations by replacing duplicates
- Used automatically after crossover/mutation
- Time: O(N)

is_valid_chromosome(chromosome, n):

- Validates permutation property
- Checks: length, data type, range, uniqueness

visualize_chessboard(individual):

- ASCII board representation
- 'Q' = queen, '.' = empty

print_summary_table(results):

- Statistical summary of experimental runs
- Success rate, avg generations, avg evaluations