Metaheuristic approach to the Hamiltonian Path

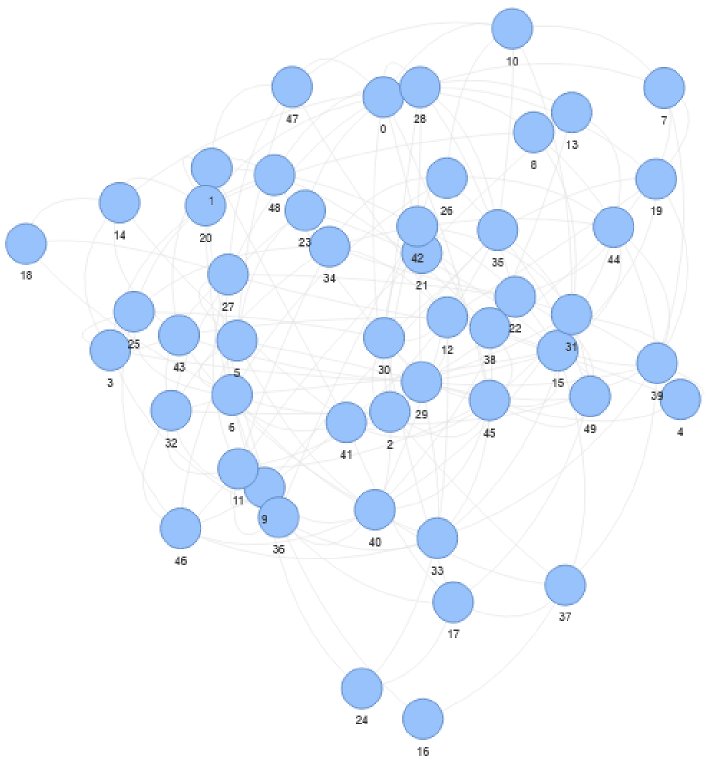

Figure 1: *Visualised with Pyvis, depicts the best Hamiltonian path that could be solved for. The GA found using the 2-OPT (inversion) operator in the Erdős-Rényi random graph. Broken edges are depicted in red.*

Module: Algorithms and Combinatorial Thinking 2025-2026

Written by:
Christos Christophoros Mitsakopoulos

Supervised by: Dr. Franck Delaplace

January 23, 2026

Contents

1 Formal Definition and Introduction

This report evaluates three metaheuristic approaches: *Simulated Annealing*, *Tabu Search*, and a *Genetic Algorithm*.

To start with, the Hamiltonian Path problem asks if a graph $G = (V, E)$ contains a path that visits every vertex / node $v \in V$ exactly once.

Given that the Hamiltonian Path problem is NP-Complete, as the number of vertices of the vertex set $n = |V|$ increases, the computational complexity required to solve the problem via brute force grows factorially ($O(n!)$). Consequently, exact algorithms become computationally intractable for large n, necessitating the use of metaheuristics – like the ones tested in this report. For this exact reason no evaluation of a brute force approach has been considered in the experiments of this report, as the computational and time complexity required is too high to practically implement.

In the context of bioinformatics, specifically *de novo* genome assembly, this problem is interesting. Arranging DNA reads in an acyclic graph can be modelled as finding a Hamiltonian path that maximises read overlap to create a contiguous sequence. Since this is a bioinformatics course, a genetic algorithm appropriated from the course material (much like all other algorithms implemented) was tested to match the theme of genome assembly heuristics.

Find all relevant results / as well as reproduce the experiment using the `main.py` at the following address: https://github.com/cmitsakopoulos/Delaplace_coursework. The Python script automates all tests demonstrated in this report and moreover, accepts user arguments through the command line interface; intended for tweaking parameters regarding base graph generation. The `-mode` argument allows running specific experiments:

```
# Default: runs all standard experiments
python main.py --nodes 50 --prob 0.15

# Specific modes:
python main.py --mode batch    # Batch testing only
python main.py --mode stats    # Statistical tests (Kruskal-Wallis,
    Wilcoxon)
python main.py --mode adaptive # Compare standard vs adaptive mutation
    GA
```

Listing 1: *Available experiment modes. The default mode "all" runs batch, optimisation, phase transition, and visualisation experiments.*

2 Random Generation of the Problem: Erdős-Rényi Random Graph

To benchmark the metaheuristic algorithms, an Erdős-Rényi (ER) $G(n, p)$ model was used. In which:

- n: The number of vertices in the graph.
- p: The probability that an edge exists between any two distinct vertices.

A larger parameter p will by effect increase the likelihood of finding a Hamiltonian Path, given that the probability of any two nodes having a connecting edge is larger. This was evident when first trying out the `main.py` Python script, where preliminary tests with a p of ≈ 0.3 and $n = 50$ showed that a Hamiltonian Path was indeed mathematically possible to find. It was so much so that all algorithms would converge to a zero cost solution (no broken edges).

```
# n default = 50, p default = 0.1, seed hardcoded to 42
g_exp = nx.erdos_renyi_graph(n=num_nodes, p=prob, seed=SEED)
```

Listing 2: *Small snippet of code in which NetworkX in Python can generate an ER graph of chosen parameters, identify that the default options are intended for a "challenging" benchmark for the algorithms. Hardcoding the seed was not a deliberate analytical choice, just to ensure reproducibility of the graph itself – the algorithms are stochastic too...*

3 Test Environment: Experimental Choices and Reasoning

To ensure significance of the results, each metaheuristic algorithm was executed for $N = 30$ runs. Relying on the Central Limit Theorem, the sampling distribution of the mean approximates a normal distribution as $N \geq 30$, even if the underlying population distribution is non-Gaussian. While increasing N reduces the Standard Error of the Mean (SEM), the precision improves only with the square root of N (i.e., $\text{SEM} = \sigma/\sqrt{N}$).

Additionally, in order to observe the impact on phase transition across all metaheuristics tested, the Komlós Szemerédi theorem/threshold was used to compute the theoretical–p–limit of the problem graph, specifically using the following equation:

$$p_{\text{limit}} = \frac{\ln(n) + \ln(\ln(n))}{n} \tag{1}$$

While $p < p_{\text{limit}}$, the probability of a Hamiltonian path existing approaches zero; thus any algorithm claiming a valid solution is likely "hallucinating", being overconfident (due to verification errors), or has encountered a statistical anomaly (highly unlikely in my testing case). Conversely, for $p > p_{\text{limit}}$, a path almost surely exists: consequently, if an algorithm fails to converge to a zero-cost solution in this region (zero broken edges), it shows that the algorithm is underperforming rather than the problem being impossible.

Another important consideration, was to examine the impact of switching from a "Swap" operator to an **"Inversion" (2-OPT) operator** between testing cases. This was also tested with the GA to demonstrate how 2-OPT can be a positive addition to an already accurate algorithm. Additionally, it was tested with SA and TS to demonstrate that the impact of this change is not algorithm-specific. This phenomenon should become evident when looking at the **"Phase Transition"** experiment of this report, where the 2-OPT operator was used instead of the Swap operator.

To move beyond visual comparisons and quantify whether observed performance differences are statistically meaningful, non-parametric statistical tests were employed. The Kruskal-Wallis H-test determines whether at least one algorithm performs significantly differently from the others, while pairwise Wilcoxon signed-rank tests identify which specific pairs differ. These non-parametric tests are appropriate because solution quality scores do not follow a normal distribution—algorithms either find optimal solutions (cost = 0) or get stuck at various local optima. Additionally, Cohen's d effect size quantifies the practical magnitude of differences, distinguishing between statistically significant results and practically meaningful ones. An effect size $|d| \geq 0.8$ indicates a large, practically important difference between algorithms.

4 Metaheuristic Algorithms: Mode of Action and Code Snippets

4.1 Objective Function

The aim of these algorithms is to identify a permutation S of vertices that minimises the number of broken edges in the path. For a graph $G = (V, E)$ and a candidate path $S = [v_1, v_2, \ldots, v_n]$, the cost function $C(S)$ is defined as:

$$C(S) = (n-1) - \sum_{i=1}^{n-1} \mathbb{I}((v_i, v_{i+1}) \in E) \tag{2}$$

Where $\mathbb{I}$ is an indicator function that equals 1 if the edge exists and 0 otherwise. A global optimum is reached when $C(S) = 0$ – a Hamiltonian Path.

4.2 Simulated Annealing (SA)

Simulated Annealing explores solutions by accepting both better or even worse solutions, based on a probability that decreases over time (referred to as parameter T). This logic prevents the algorithm from arriving at a final solution before reaching a truly optimal or near optimal solution. The probability P of accepting a new solution S' with cost difference $\Delta C = C(S') - C(S)$, is governed by the Metropolis criterion (see also the Python implementation below):

$$P(\text{accept}) = \begin{cases} 1 & \text{if } \Delta C < 0 \\ e^{-\frac{\Delta C}{T}} & \text{if } \Delta C \geq 0 \end{cases} \tag{3}$$

```python
# From main.py: Calculate cost difference
delta = neighbor_cost - current_cost

# Accept if better (delta < 0) OR with probability exp(-delta/T)
if delta < 0 or random.random() < exp(-delta / temp):
    current = neighbor
    current_cost = neighbor_cost
```

Listing 3: *Metropolis Criterion: As the loop progresses, the temperature T decays geometrically ($T_{k+1} = 0.985 \cdot T_k$), gradually turning the search into a simple greedy descent (hill climbing)*

4.3 Tabu Search (TS)

Tabu Search differs from SA by using a deterministic, memory-based approach. TS explores the immediate neighbourhood of a current solution, then moves to the best available neighbouring solution, even if that neighbouring solution is worse than the current solution.

To prevent cycling (revisiting the same solutions again and again), the algorithm maintains a *Tabu List* – a short-term memory that prevents recent solutions for a specific duration, called tenure.

```python
# From main.py: Moving to the best candidate not in the Tabu list
if cand not in tabu_list:
    current = cand
    found_move = True

# Update memory
tabu_list.append(current)
if len(tabu_list) > tenure:
    tabu_list.pop(0) # Remove oldest entry
```

Listing 4: Tabu Search Memory Logic

4.4 Genetic Algorithm (GA)

The Genetic Algorithm attempts to mimic natural selection. Unlike SA and TS, which improve a single solution, GA evolves a population of solutions. The operator for permutation is *Ordered Crossover* (OX1), which is important because standard single-point crossover would result in duplicate or missing vertices / nodes.

The code uses OX1 to preserve the ordering of a sub-segment from one "parent" while filling the remaining slots with genes from the second "parent".

```python
# From main.py: Preserves sub-segment from parent 1
child[start:end] = p1[start:end]

# Fills remaining slots from parent 2, skipping duplicates
for i in range(size):
    if child[i] is None:
        while p2[current_p2_idx] in child:
            current_p2_idx += 1
        child[i] = p2[current_p2_idx]
```

Listing 5: Ordered Crossover (OX1) Implementation

Using the standard `swap` operator (exchanging two indices) disrupts the adjacency of the path significantly. In comparison, the `inversion` (2-Opt) operator reverses a segment of the path. This is mathematically better for path search problems because it preserves the internal adjacency of the reversed segment, only breaking the two edges at the endpoints of the segment. This distinction is implemented via the 'op_inversion' function in 'main.py' and is the primary driver for convergence in denser graphs. A graph is produced at the end during testing to demonstrate the differences of both applications.

$$\text{Swap}(S) \rightarrow \text{High disruption of edges} \tag{4}$$

$$\text{Inversion}(S) \rightarrow \text{Minimal disruption (2-Opt)} \tag{5}$$

4.4.1 Adaptive Mutation Rate

Another improvement over that of the mutation operator (Swap vs 2-Opt), which determines how to mutate a solution in order to develop it, is improving on how often to mutate. With a high fixed mutation rate, the algorithm promotes exploration but disrupts good solutions, while low mutation rates enable exploitation but risk converging early; thereby leading to suboptimal solutions. An adaptive approach addresses this by adjusting the mutation probability μ based on population diversity D:

$$D = \frac{|\mathcal{F}_{\text{unique}}|}{P}, \quad \mu_{\text{adaptive}} = \mu_{\text{base}} \cdot (1 + (0.5 - D)) \tag{6}$$

When $D < 0.5$ (population becoming homogeneous), mutation increases to reintroduce variation. When $D > 0.5$ (healthy diversity), mutation decreases to exploit promising solutions. The rate is clamped to $[0.1, 0.6]$. This self-regulating mechanism helps prevent premature convergence without manual tuning.

```python
def calculate_diversity(fitnesses):
    return len(set(fitnesses)) / len(fitnesses)

# Inside GA loop, per generation:
if adaptive:
    diversity = calculate_diversity(fitnesses)
    current_mut = base_mut_rate * (1 + (0.5 - diversity))
    current_mut = max(0.1, min(0.6, current_mut))
```

Listing 6: *Adaptive mutation rate logic from main.py. Diversity is computed per generation, and mutation rate is adjusted accordingly.*

For a fixed mutation rate μ_0, the change in diversity that is expected per generation matches a drift-selection balance. When diversity is low ($D \to 0$), the population converges early; such that all individuals occupy similar regions of the search space. The adaptive scheme $\mu_{\text{adaptive}} = \mu_0 \cdot (1.5 - D)$ gives a self-correcting feedback loop: when $D < 0.5$, mutation increases to escape local optima; when $D > 0.5$, mutation decreases to enable for more promising solutions. The equilibrium $D^* = 0.5$ represents an optimal exploration-exploitation balance.

5 Experiment: Comparing the Metaheuristic Algorithms

5.1 Batch Testing

The experiment code in `main.py` has been hardcoded to repeat the stochastic run of each metaheuristic algorithm 30 times, with the user chosen parameters upon initialization through the command line. The results depicted in Figure 2 and Figure 3, were computed with an ER problem graph of $n = 50$ and $p = 0.15$.

Briefly, with reference to Figure 2, the lowest solution quality (highest number of broken edges) is demonstrated by SA, then TS and with the GA, being the best performer. Inversely, the highest execution time per batch run, was demonstrated by GA, followed by TS and lastly SA. These results confirm the expected correlation between computational cost and solution quality (indicating the code works as intended). SA is the fastest because it performs a single $O(1)$ evaluation per step, but its single-trajectory stochasticity struggles to escape bad solutions (local optima) in a solution landscape. TS evaluates a neighbourhood of size $k = 50$ per step ($O(k)$) (focus on local searches), which linearly increases runtime. Lastly, the GA achieves a near-optimal solution albeit, at the highest computational cost. It maintains diversity through a population-based search ($O(P \cdot N)$ per generation), allowing it to traverse the complex fitness landscape more effectively than the trajectory-based methods.

Nevertheless, the results in Figure 2 clearly show that all algorithms – apart from the GA – are greatly underperforming in this testing case. For an ER graph with $n = 50$, the Komlós Szemerédi limit is $p = 0.106$, considering that the chosen – testing – $p = 0.15$, is greater than the theoretical limit, a solution should be mathematically infeasible; albeit, difficult to obtain.

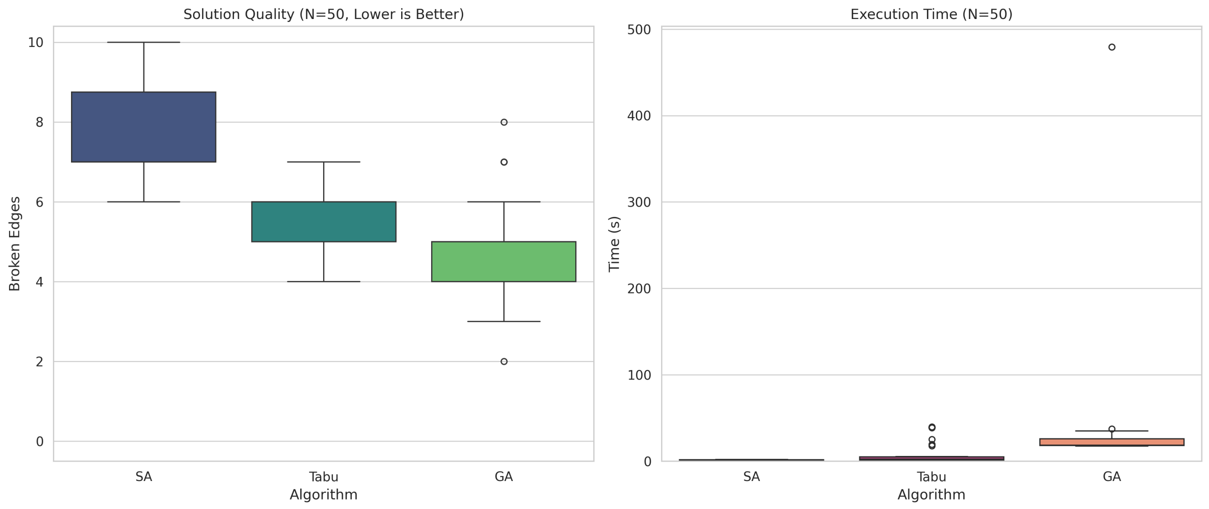

Figure 2: *In the box whisker plots depicted, "Solution Quality" (Y-axis: Broken Edges) is shown on the left and "Execution Time" (Y-axis: Time(seconds)) on the right. As discussed prior, due to the low probability of edges between any two nodes, the graph is sparse and the metaheuristic algorithms – while performing differently from one another – are not succesful in this testing case. As such, the amount of broken edges seen between algorithms is not unlikely, and the inverse relationship between the number of broken edges and execution time is a clear demonstration of algorithmic complexity – which leads to positive outcomes in problem solving.*

5.2 Phase Transition

Briefly, with reference to Figure 3, there appears to be overconfidence of the TS and GA algorithms, which demonstrate a non-zero level of success at a mathematically improbable rate of $p = 0.1$; where $p = 0.1 < p_{\text{limit}} = 0.106$. Given that the difference between p and p_{limit} is $\Delta p = 0.006$, this could be believed to be a statistical anomaly that occured in this new 30-run test – compared to the previous batch testing scenario. After introducing a new `argparse` argument to the `main.py` file to exclusively re-run the phase transition experiment, the results – over 5 re-runs – led to the same conclusions as in Figure 3. One could argue that due to a **switch from the "Swap" operator to the more optimised "Inversion" (2-Opt) operator**, the GA and TS algorithms are able to find a global optimum at a mathematically improbable rate of $p = 0.1$, as well as be considerably succesful at higher graph densities ($p > 0.1$); all the while SA is equally performing better than before (than in Figure 2).

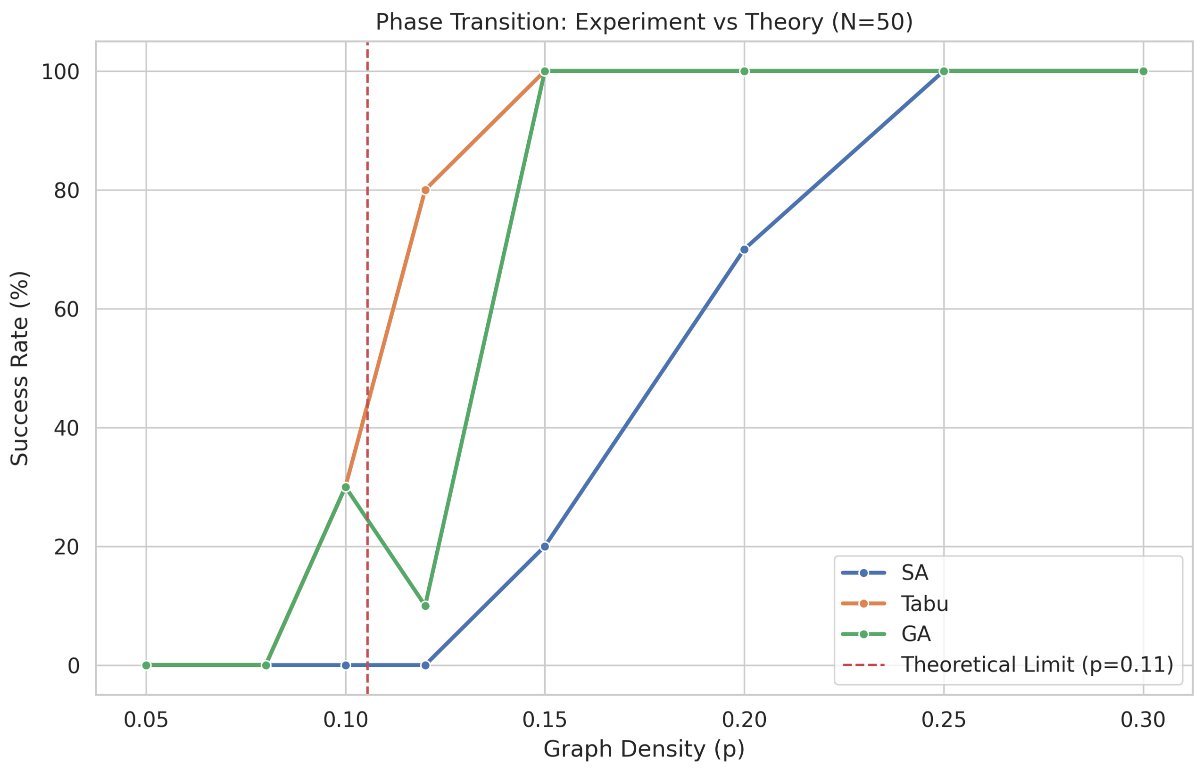

Figure 3: *This line chart depicts the phase transition of the proposed problem Erdos-Renyi graph, at a cycling p rate between 0.05 and 0.3. The theoretical limit is computed automatically based on the input problem graph, given the Komlós Szemerédi formula. Observe overconfidence from the TS and GA algorithms, which demonstrate success at a mathematically improbable rate of p. The success rate of all algorithms is congruent with previous results.*

6 GA: Comparing Swap Operator to 2-Opt

Congruent with the phase transition results, it is evident that the switch from the Swap operator (Figure 2) to the Inversion (2-Opt) operator (Figure 3), improves the performance of the GA (see Figure 4). Importantly, looking at the number of broken edges at each iteration, it is clear that the 2-OPT based GA converges at a faster and smoother, or more consistent pace, than the Swap based GA. To briefly reiterate on what was said prior on the report, this phenomenon can be due to two complementary reasons:

Firstly, the `2-OPT` (or inversion) operator is less destructive at each iteration. It reverses a segment between indices i and j at each iteration, removing two edges (v_i, v_{i+1}) and (v_j, v_{j+1}), and replacing them with (v_i, v_j) and (v_{i+1}, v_{j+1}), whilst preserving the adjacency of all nodes within the segment. As such, the 2-OPT operator enables the algorithm to make small changes which are beneficial at each iteration. Therefore, as can be seen in Figure 4, the GA with the 2-OPT operator improves its global solution at each iteration, whereas the Swap operator will repeatedly plateau/stagnate due to destructive solutions between iterations. Secondly, as mentioned before, the `Swap` operator exchanges two nodes v_i and v_j arbitrarily, which can affect up to four edges: $(v_{i-1}, v_i), (v_i, v_{i+1}), (v_{j-1}, v_j)$, and (v_j, v_{j+1}). Instead of making minor improvements locally, the Swap operator will lead the algorithm to more destructive changes at each iteration. In turn, this will cause the search to stagnate (maintain one solution over multiple iterations) over a certain amount of iterations. While a more beneficial solution can be found accidentaly, the algorithm risks gettingn stuck in local minima; minima which can be avoided by the 2-OPT operator.

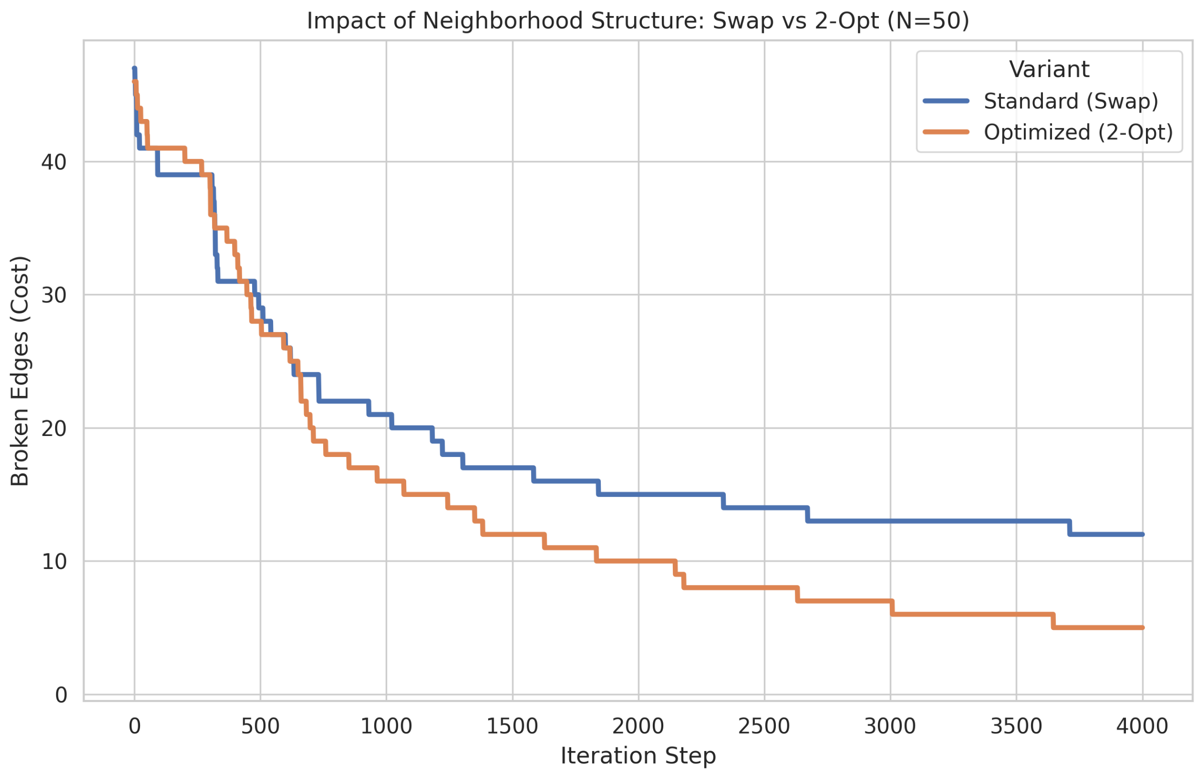

Figure 4: *In the depicted line chart, Broken Edges (Y-axis) for which the objective function seeks to minimise (Cost) by maximising the number of edges in the path, is plotted against the number of generations – repeated runs (X-axis). The GA with the Swap operator is shown in blue, while the GA with the Inversion (2-Opt) operator is in orange. Observe the clear disparity between the two operators, in which it's clear that the 2-OPT operator GA converges much faster to a near-global optimum (zero broken edges) than its Swap-GA counterpart.*

7 Statistical Validation

To quantify the statistical significance of the performance differences observed in the figures above, the Kruskal-Wallis H-test was applied:

$$H = \frac{12}{N(N+1)} \sum_{i=1}^{k} \frac{R_i^2}{n_i} - 3(N+1) \tag{7}$$

A significant result ($p < 0.05$) confirms that the algorithms perform differently. Pairwise Wilcoxon signed-rank tests further identify which specific pairs differ:

$$W = \min\left(\sum_{d_i>0} R_i, \sum_{d_i<0} R_i\right) \tag{8}$$

Finally, Cohen's d effect size quantifies the magnitude of these differences:

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s_{\text{pooled}}} \quad \text{where} \quad s_{\text{pooled}} = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}} \tag{9}$$

I visualise this statistical analysis through a three-panel figure containing:

- Box plot with Mann-Whitney significance annotations (star notation: $^*p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$).
- Mean broken edges with 95% confidence interval error bars.
- Effect size heatmap showing pairwise Cohen's d values.

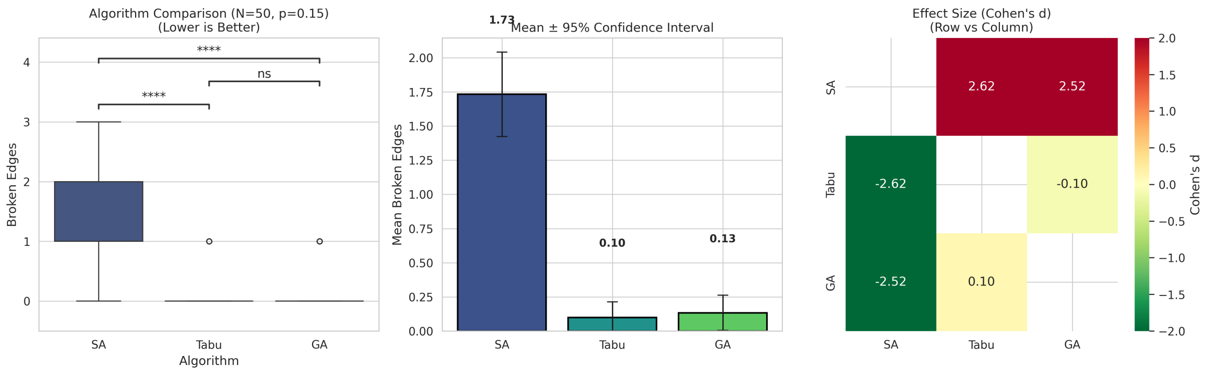

Figure 5: *Statistical comparison of SA, Tabu Search, and GA. Left: Box plot with Mann-Whitney significance annotations. Centre: Mean broken edges with 95% CI error bars. Right: Cohen's d effect size heatmap showing pairwise comparisons.*

With reference to Figure 5, SA exhibits a mean of $\bar{x}_{SA} = 1.73$ broken edges (95% CI: between 1.42 and 2.04), while Tabu Search ($\bar{x}_{Tabu} = 0.10$, CI: between -0.01 and 0.21) and GA ($\bar{x}_{GA} = 0.13$, CI: between 0.00 and 0.26) achieve near-optimal solutions. Importantly, but expectedly so, the Mann-Whitney tests confirm highly significant differences between SA and both other algorithms ($p < 0.0001$), whereas Tabu vs GA is non-significant. Equally, the effect size heatmap shows large differences between all: $d_{SA\leftrightarrow Tabu} = 2.62$ and $d_{SA\leftrightarrow GA} = 2.52$ (both $|d| > 0.8$), while $d_{Tabu\leftrightarrow GA} = -0.10$ indicates negligible difference. These results statistically confirm that the memory-based (Tabu) and population-based (GA) strategies substantially outperform the single-trajectory stochastic approach (SA) on this graph configuration. However, this strong distinction diminishes at higher graph densities ($p \geq 0.15$), where all algorithms achieve 100% success as shown in the phase transition analysis.

8 Annex

```python
import random
import time
import math
import itertools
import argparse
import numpy as np
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import seaborn as sns
from math import exp, log
from scipy import stats
from itertools import combinations

# Try to import statannotations for publication-ready p-value
    annotations
try:
    from statannotations.Annotator import Annotator
    HAS_STATANNOTATIONS = True
except ImportError:
    HAS_STATANNOTATIONS = False
    print("Note: Install 'statannotations' for p-value annotations on
    plots: pip install statannotations")

# --- CONFIGURATION ---
SNS_THEME = "whitegrid"
SNS_CONTEXT = "paper"
SEED = 42

# Apply settings
random.seed(SEED)
np.random.seed(SEED)
sns.set_theme(style=SNS_THEME, context=SNS_CONTEXT, font_scale=1.2)

# ==========================================
# 1. CORE HELPER & FITNESS FUNCTIONS
# ==========================================

def get_initial_solution(nodes):
    """Returns a random permutation of nodes."""
    p = list(nodes).copy()
    random.shuffle(p)
    return p

def count_edges(path, graph):
    """Counts valid edges in the path for a specific graph."""
    edges = 0
    # Optimization: Iterate via index to avoid creating new lists
    for i in range(len(path) - 1):
        if graph.has_edge(path[i], path[i+1]):
            edges += 1
    return edges

def fitness_min(path, graph):
    """Minimisation: Returns number of broken edges. Target = 0."""
    target = len(path) - 1
    valid = count_edges(path, graph)
    return float(target - valid)

def fitness_max(path, graph):
    """Maximisation: Returns number of valid edges. Target = N-1."""
    return float(count_edges(path, graph))

# ==========================================
# 1b. STATISTICAL ANALYSIS FUNCTIONS
# ========================================

def cohens_d(group1, group2):
    """Calculate Cohen's d effect size between two groups."""
    n1, n2 = len(group1), len(group2)
    var1, var2 = np.var(group1, ddof=1), np.var(group2, ddof=1)
    pooled_std = np.sqrt(((n1 - 1) * var1 + (n2 - 1) * var2) / (n1 + n2 - 2))
    if pooled_std == 0:
        return 0.0
    return (np.mean(group1) - np.mean(group2)) / pooled_std

def interpret_effect_size(d):
    """Interpret Cohen's d value."""
    d = abs(d)
    if d < 0.2:
        return "negligible"
    elif d < 0.5:
        return "small"
    elif d < 0.8:
        return "medium"
    else:
        return "large"

def compute_statistical_tests(scores_dict, alpha=0.05, output_file="statistical_results.txt"):
    """
    Perform formal statistical tests on algorithm comparison results.

    Args:
        scores_dict: Dictionary mapping algorithm names to lists of scores
        alpha: Significance level (default 0.05)
        output_file: Path to save results (default: statistical_results.txt)

    Returns:
        Dictionary with test results including p-values and effect sizes
    """
    results = {
        'kruskal_wallis': None,
        'pairwise_tests': [],
        'effect_sizes': [],
        'confidence_intervals': {}
    }

    lines = []  # Collect output for file writing

    # 1. Kruskal-Wallis H-test (non-parametric ANOVA)
    groups = list(scores_dict.values())
    if len(groups) >= 2:
        stat, p_value = stats.kruskal(*groups)
        results['kruskal_wallis'] = {
            'statistic': stat,
            'p_value': p_value,
            'significant': p_value < alpha
        }
        lines.append(f"\n--- STATISTICAL ANALYSIS (  ={alpha}) ---")
        lines.append(f"Kruskal-Wallis H-test: H={stat:.4f}, p={p_value:.4e}")
        lines.append(f"      {'Significant' if p_value < alpha else 'Not significant'} difference between groups")

    # 2. Pairwise Wilcoxon signed-rank tests
    algo_names = list(scores_dict.keys())
    lines.append("\nPairwise Wilcoxon signed-rank tests:")
    for (name1, name2) in combinations(algo_names, 2):
        scores1, scores2 = scores_dict[name1], scores_dict[name2]
        try:
            stat, p_value = stats.wilcoxon(scores1, scores2)
            significant = p_value < alpha
        except ValueError:
            # All differences are zero
            stat, p_value, significant = 0, 1.0, False

        results['pairwise_tests'].append({
            'pair': (name1, name2),
            'statistic': stat,
            'p_value': p_value,
            'significant': significant
        })
        lines.append(f"  {name1} vs {name2}: W={stat:.2f}, p={p_value:.4e} {'*' if significant else ''}")

    # 3. Effect sizes (Cohen's d)
    lines.append("\nEffect sizes (Cohen's d):")
    for (name1, name2) in combinations(algo_names, 2):
        d = cohens_d(scores_dict[name1], scores_dict[name2])
        interpretation = interpret_effect_size(d)
        results['effect_sizes'].append({
            'pair': (name1, name2),
            'cohens_d': d,
            'interpretation': interpretation
        })
        lines.append(f"  {name1} vs {name2}: d={d:.3f} ({interpretation})")

    # 4. Confidence intervals for each algorithm
    lines.append("\n95% Confidence Intervals:")
    for name, scores in scores_dict.items():
        mean = np.mean(scores)
        sem = stats.sem(scores)
        ci = stats.t.interval(0.95, len(scores)-1, loc=mean, scale=sem)
        results['confidence_intervals'][name] = {'mean': mean, 'ci_low': ci[0], 'ci_high': ci[1]}
        lines.append(f"  {name}: {mean:.2f} [{ci[0]:.2f}, {ci[1]:.2f}]")

    # Print to console
    for line in lines:
        print(line)

    # Write to file
    with open(output_file, 'a') as f:
        f.write(f"\n{'='*60}\n")
        f.write(f"Statistical Analysis Run\n")
        f.write(f"{'='*60}\n")
        for line in lines:
            f.write(line + '\n')
    print(f"\nResults appended to '{output_file}'")

    return results

def plot_statistical_comparison(scores_dict, output_prefix="statistical_comparison", title="Algorithm Comparison"):
    """
    Create publication-ready statistical comparison visualization.

    Generates:
    - Box plot with p-value annotations (if statannotations available)
    - Bar chart with 95% CI error bars
    - Effect size heatmap

    Args:
        scores_dict: Dictionary mapping algorithm names to lists of scores
        output_prefix: Prefix for output files
        title: Plot title
    """
    algo_names = list(scores_dict.keys())

    # Prepare data for plotting
    plot_data = []
    for name, scores in scores_dict.items():
        for score in scores:
            plot_data.append({'Algorithm': name, 'Broken Edges': score})
    df = pd.DataFrame(plot_data)

    # Create figure with multiple subplots
    fig, axes = plt.subplots(1, 3, figsize=(16, 5))

    # --- Panel 1: Box plot with significance annotations ---
    ax1 = axes[0]
    sns.boxplot(data=df, x='Algorithm', y='Broken Edges', hue='Algorithm',
                legend=False, ax=ax1, palette="viridis")
    ax1.set_title(f'{title}\n(Lower is Better)')
    ax1.set_ylim(bottom=-0.5)

    # Add p-value annotations if statannotations is available
    if HAS_STATANNOTATIONS and len(algo_names) >= 2:
        pairs = list(combinations(algo_names, 2))
        annotator = Annotator(ax1, pairs, data=df, x='Algorithm', y='Broken Edges')
        annotator.configure(test='Mann-Whitney', text_format='star', loc='inside',
                            comparisons_correction=None)
        annotator.apply_and_annotate()

    # --- Panel 2: Mean with 95% CI error bars ---
    ax2 = axes[1]
    means = []
    ci_lows = []
    ci_highs = []
    for name in algo_names:
        scores = scores_dict[name]
        mean = np.mean(scores)
        sem = stats.sem(scores)
        ci = stats.t.interval(0.95, len(scores)-1, loc=mean, scale=sem)
        means.append(mean)
        ci_lows.append(mean - ci[0])
        ci_highs.append(ci[1] - mean)

    colors = sns.color_palette("viridis", len(algo_names))
    bars = ax2.bar(algo_names, means, yerr=[ci_lows, ci_highs], capsize=5,
                   color=colors, edgecolor='black', linewidth=1.5)
    ax2.set_ylabel('Mean Broken Edges')
    ax2.set_title('Mean    95% Confidence Interval')
    ax2.set_ylim(bottom=0)

    # Add value labels on bars
    for bar, mean in zip(bars, means):
        ax2.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.5,
                f'{mean:.2f}', ha='center', va='bottom', fontsize=10, fontweight='bold')

    # --- Panel 3: Effect size heatmap ---
    ax3 = axes[2]
    n = len(algo_names)
    effect_matrix = np.zeros((n, n))
    for i, name1 in enumerate(algo_names):
        for j, name2 in enumerate(algo_names):
            if i != j:
                d = cohens_d(scores_dict[name1], scores_dict[name2])
                effect_matrix[i, j] = d

    # Create annotated heatmap
    mask = np.eye(n, dtype=bool)  # Mask diagonal
    sns.heatmap(effect_matrix, annot=True, fmt='.2f', cmap='RdYlGn_r',
                xticklabels=algo_names, yticklabels=algo_names,
                mask=mask, ax=ax3, center=0, vmin=-2, vmax=2,
                cbar_kws={'label': "Cohen's d"})
    ax3.set_title("Effect Size (Cohen's d)\n(Row vs Column)")

    plt.tight_layout()

    # Save figure
    fig_path = f'{output_prefix}_plot.png'
    plt.savefig(fig_path, dpi=300, bbox_inches='tight')
    print(f"Saved '{fig_path}'")
    plt.close()

    # Save summary to CSV
    summary_data = {
        'Algorithm': algo_names,
        'Mean': means,
        'CI_Lower': [m - cl for m, cl in zip(means, ci_lows)],
        'CI_Upper': [m + ch for m, ch in zip(means, ci_highs)],
        'Std': [np.std(scores_dict[name]) for name in algo_names]
    }
    summary_df = pd.DataFrame(summary_data)
    csv_path = f'{output_prefix}_summary.csv'
    summary_df.to_csv(csv_path, index=False)
    print(f"Saved '{csv_path}'")

# ==========================================
# 2. OPERATORS (THE OPTIMISATIONS)
# ==========================================

def op_swap(genome):
    """Standard Swap: Exchanges two random nodes."""
    n = len(genome)
    i, j = random.sample(range(n), 2)
    genome[i], genome[j] = genome[j], genome[i]
    return genome

def op_inversion(genome):
    """
    2-Opt Inversion: Reverses a random sub-segment.
    OPTIMIZATION: Preserves adjacency better than swap.
    """
    n = len(genome)
    i, j = sorted(random.sample(range(n), 2))
    # Reverse the segment between i and j
    genome[i:j+1] = genome[i:j+1][::-1]
    return genome

# ==========================================
# 3. METAHEURISTIC ALGORITHMS
# ==========================================

def run_simulated_annealing(graph, max_steps=3000, temp0=100.0, operator
    ="swap"):
    """Simulated Annealing with selectable operator."""
    nodes = list(graph.nodes)
    current = get_initial_solution(nodes)
    best = current.copy()

    current_cost = fitness_min(current, graph)
    best_cost = current_cost
    trace = [best_cost]
    temp = temp0

    mutate_func = op_inversion if operator == "inversion" else op_swap

    for step in range(max_steps):
        if best_cost == 0: break

        neighbor = current.copy()
        neighbor = mutate_func(neighbor)

        neighbor_cost = fitness_min(neighbor, graph)
        delta = neighbor_cost - current_cost

        if delta < 0 or random.random() < exp(-delta / temp):
            current = neighbor
            current_cost = neighbor_cost
            if current_cost < best_cost:
                best = current.copy()
                best_cost = current_cost

        trace.append(best_cost)
        temp *= 0.985

    return best, best_cost, trace

def run_tabu_search(graph, max_steps=1000, tenure=20, operator="swap"):
    """Tabu Search with selectable operator."""
    nodes = list(graph.nodes)
    current = get_initial_solution(nodes)
    best = current.copy()
    best_cost = fitness_min(best, graph)

    tabu_list = []
    trace = [best_cost]
    mutate_func = op_inversion if operator == "inversion" else op_swap

    for step in range(max_steps):
        if best_cost == 0: break

        candidates = []
        for _ in range(50):
            cand = current.copy()
            cand = mutate_func(cand)
            candidates.append(cand)

        candidates.sort(key=lambda p: fitness_min(p, graph))

        found_move = False
        for cand in candidates:
            cand_cost = fitness_min(cand, graph)
            if cand_cost < best_cost:
                current = cand
                best = cand
                best_cost = cand_cost
                found_move = True
                break
            if cand not in tabu_list:
                current = cand
                found_move = True
                break

        trace.append(best_cost)
        if found_move:
            tabu_list.append(current)
            if len(tabu_list) > tenure:
                tabu_list.pop(0)

    return best, best_cost, trace

def run_genetic_algorithm(graph, pop_size=200, generations=1000,
   cross_rate=0.8, mut_rate=0.3, operator="swap", adaptive=False):
    """
    Genetic Algorithm with selectable operator and optional adaptive
   mutation.

    When adaptive=True:
```

```
    - Calculates population diversity (unique fitness values / pop_size)
    - Low diversity (< 0.3): INCREASE mutation to escape local optima
    - High diversity (> 0.7): DECREASE mutation to exploit good
solutions
    - Formula: current_mut = base_rate * (1 + (0.5 - diversity))
    - Clamped to [0.1, 0.6] range
    """
    nodes = list(graph.nodes)
    target = len(nodes) - 1

    mutate_func = op_inversion if operator == "inversion" else op_swap
    base_mut_rate = mut_rate

    def ordered_crossover(p1, p2):
        size = len(p1)
        start, end = sorted(random.sample(range(size), 2))
        child = [None] * size
        child[start:end] = p1[start:end]
        current_p2_idx = 0
        for i in range(size):
            if child[i] is None:
                while p2[current_p2_idx] in child:
                    current_p2_idx += 1
                child[i] = p2[current_p2_idx]
        return child

    def calculate_diversity(fitnesses):
        """Calculate diversity as ratio of unique fitness values."""
        unique = len(set(fitnesses))
        return unique / len(fitnesses)

    population = [get_initial_solution(nodes) for _ in range(pop_size)]
    best_sol = None
    best_broken = float('inf')
    trace = []
    diversity_trace = []  # Track diversity for analysis

    for gen in range(generations):
        fitnesses = [fitness_max(p, graph) for p in population]
        best_val = max(fitnesses)
        current_broken = target - best_val

        if current_broken < best_broken:
            best_broken = current_broken
            best_sol = population[fitnesses.index(best_val)]

        trace.append(best_broken)
        if current_broken == 0: break

        # Adaptive mutation rate adjustment
        if adaptive:
            diversity = calculate_diversity(fitnesses)
            diversity_trace.append(diversity)
            # Increase mutation when diversity is low, decrease when
high
            current_mut = base_mut_rate * (1 + (0.5 - diversity))
            current_mut = max(0.1, min(0.6, current_mut))  # Clamp to
[0.1, 0.6]
```

```python
        else:
            current_mut = mut_rate

        next_pop = [best_sol.copy()]
        while len(next_pop) < pop_size:
            competitors = random.sample(population, 3)
            winner = max(competitors, key=lambda p: fitness_max(p, graph))
            next_pop.append(winner.copy())
        population = next_pop

        for i in range(1, pop_size - 1, 2):
            if random.random() < cross_rate:
                population[i] = ordered_crossover(population[i], population[i+1])

        for i in range(1, pop_size):
            if random.random() < current_mut:
                population[i] = mutate_func(population[i])

    return best_sol, best_broken, trace

# ==========================================
# 4. EXPERIMENT RUNNERS
# ==========================================

def run_batch_experiments(num_nodes, prob):
    """Runs each algorithm 30 times and plots results."""
    # Baseline comparison using Standard Swap
    OPERATOR = "swap"

    g_exp = nx.erdos_renyi_graph(n=num_nodes, p=prob, seed=SEED)

    print(f"\n--- 1. BATCH EXPERIMENT (Baseline 'Swap', N={num_nodes}, p={prob}, 30 Runs) ---")

    def run_batch(algo_func, name):
        scores = []
        times = []
        successes = 0
        for _ in range(30):
            start = time.time()
            _, cost, _ = algo_func(g_exp)
            dur = time.time() - start
            scores.append(cost)
            times.append(dur)
            if cost == 0: successes += 1

        rate = (successes/30)*100
        print(f"{name}: Mean Cost={np.mean(scores):.2f}, Success={rate:.1f}%, Mean Time={np.mean(times):.4f}s")
        return scores, times

    sa_scores, sa_times = run_batch(lambda g: run_simulated_annealing(g, max_steps=3000, operator=OPERATOR), "SA")
    tabu_scores, tabu_times = run_batch(lambda g: run_tabu_search(g, max_steps=1500, operator=OPERATOR), "Tabu")
    ga_scores, ga_times = run_batch(lambda g: run_genetic_algorithm(g,
generations=1500, operator=OPERATOR), "GA")

    data = {
        'Algorithm': ['SA']*30 + ['Tabu']*30 + ['GA']*30,
        'Broken Edges': sa_scores + tabu_scores + ga_scores,
        'Time (s)': sa_times + tabu_times + ga_times
    }
    df = pd.DataFrame(data)

    fig, axes = plt.subplots(1, 2, figsize=(14, 6))

    sns.boxplot(data=df, x='Algorithm', y='Broken Edges', hue='Algorithm',
                legend=False, ax=axes[0], palette="viridis")
    axes[0].set_title(f'Solution Quality (N={num_nodes}, Lower is Better)')
    axes[0].set_ylim(bottom=-0.5)

    sns.boxplot(data=df, x='Algorithm', y='Time (s)', hue='Algorithm',
                legend=False, ax=axes[1], palette="magma")
    axes[1].set_title(f'Execution Time (N={num_nodes})')
    axes[1].set_ylim(bottom=0)

    plt.tight_layout()
    plt.savefig('experiment_1_batch_results.png', dpi=300, bbox_inches='tight')
    print("Saved 'experiment_1_batch_results.png'")
    # plt.show() # Uncomment if running interactively

def run_optimization_comparison(num_nodes):
    """Compares Standard 'Swap' vs Optimized 'Inversion' (2-Opt)."""
    print(f"\n--- 2. OPTIMISATION EVALUATION (N={num_nodes}) ---")
    g_opt = nx.erdos_renyi_graph(n=num_nodes, p=0.1, seed=SEED)

    print("Running SA with Standard Swap...")
    _, _, trace_swap = run_simulated_annealing(g_opt, max_steps=4000, operator="swap")

    print("Running SA with Optimized Inversion (2-Opt)...")
    _, _, trace_inv = run_simulated_annealing(g_opt, max_steps=4000, operator="inversion")

    df_opt = pd.DataFrame({
        'Step': list(range(len(trace_swap))) + list(range(len(trace_inv))),
        'Broken Edges': trace_swap + trace_inv,
        'Variant': ['Standard (Swap)'] * len(trace_swap) + ['Optimized (2-Opt)'] * len(trace_inv)
    })

    plt.figure(figsize=(10, 6))
    sns.lineplot(data=df_opt, x='Step', y='Broken Edges', hue='Variant', linewidth=2.5)
    plt.title(f'Impact of Neighborhood Structure: Swap vs 2-Opt (N={num_nodes})')
    plt.ylabel('Broken Edges (Cost)')
    plt.xlabel('Iteration Step')
    plt.ylim(bottom=-0.5)

    plt.savefig('experiment_2_optimization_comparison.png', dpi=300, bbox_inches='tight')
    print("Saved 'experiment_2_optimization_comparison.png'")

def run_phase_transition(num_nodes):
    """
    Analyzes difficulty vs graph density across ALL algorithms.
    This provides a comprehensive view of algorithmic limits.
    """
    print(f"\n--- 3. PHASE TRANSITION ANALYSIS (N={num_nodes}) ---")
    # Densities to test
    densities = [0.05, 0.08, 0.1, 0.12, 0.15, 0.2, 0.25, 0.3]

    # Calculate Theoretical Threshold for THIS N
    # Koml s & Szemer di theorem: p = (ln(n) + ln(ln(n))) / n
    if num_nodes > 1:
        math_threshold = (log(num_nodes) + log(log(num_nodes))) / num_nodes
        print(f"Theoretical Critical Threshold for N={num_nodes}: p ~ {math_threshold:.3f}")
    else:
        math_threshold = 0

    # Store results for plotting
    results = {'Density': [], 'Success Rate': [], 'Algorithm': []}

    # Test all algorithms to see which one handles the transition best
    algorithms = {
        'SA': lambda g: run_simulated_annealing(g, max_steps=2500, operator="inversion"),
        'Tabu': lambda g: run_tabu_search(g, max_steps=1500, operator="inversion"),
        'GA': lambda g: run_genetic_algorithm(g, generations=1000, operator="inversion")
    }

    for p in densities:
        # Create a graph for this density
        # Note: Ideally we average over multiple graphs, but for speed we use one seed per density
        # or we generate a fresh one each run. Let's stick to one graph instance per density
        # but 10 runs per algorithm on it.
        g = nx.erdos_renyi_graph(n=num_nodes, p=p, seed=999)

        print(f"Testing Density p={p}...")

        for name, algo_func in algorithms.items():
            successes = 0
            for _ in range(10): # 10 runs per density/algorithm pair
                _, cost, _ = algo_func(g)
                if cost == 0: successes += 1

            rate = (successes/10)*100
            results['Density'].append(p)
            results['Success Rate'].append(rate)
            results['Algorithm'].append(name)

    df_phase = pd.DataFrame(results)

    plt.figure(figsize=(10, 6))
    sns.lineplot(data=df_phase, x='Density', y='Success Rate', hue='
   Algorithm', marker='o', linewidth=2)

    # Plot the Theoretical Line
    plt.axvline(x=math_threshold, color='r', linestyle='--', label=f'
   Theoretical Limit (p={math_threshold:.2f})')

    plt.title(f'Phase Transition: Experiment vs Theory (N={num_nodes})')
    plt.xlabel('Graph Density (p)')
    plt.ylabel('Success Rate (%)')
    plt.ylim(-5, 105)
    plt.legend()
    plt.grid(True)

    plt.savefig('experiment_3_phase_transition.png', dpi=300,
   bbox_inches='tight')
    print("Saved 'experiment_3_phase_transition.png'")

def save_best_graph_html(num_nodes, prob):
    """Runs GA once and saves the result to HTML."""
    print("\n--- 4. GENERATING VISUALISATION ---")
    try:
        from pyvis.network import Network
    except ImportError:
        print("Pyvis not installed. Skipping.")
        return

    # Use arguments, but if probability is too low, we might not find a
   path
    g = nx.erdos_renyi_graph(n=num_nodes, p=prob, seed=SEED)
    path, cost, _ = run_genetic_algorithm(g, generations=2000, operator=
   "inversion")

    print(f"Final Path Cost: {cost}")

    net = Network(height="600px", width="100%", cdn_resources='remote')

    for n in g.nodes:
        net.add_node(int(n), label=str(n), color='#97c2fc')

    for u, v in g.edges:
        net.add_edge(int(u), int(v), color='#e0e0e0', width=1)

    for i in range(len(path) - 1):
        u, v = int(path[i]), int(path[i+1])
        if g.has_edge(u, v):
            net.add_edge(u, v, color='red', width=4)
        else:
            net.add_edge(u, v, color='red', width=4, dashes=True)

    net.show("hamiltonian_path.html", notebook=False)
    print("Saved to 'hamiltonian_path.html'")

def run_adaptive_comparison(num_nodes, prob=0.1, runs=20):
    """
    Compare standard GA vs adaptive mutation GA.

    Generates: experiment_5_adaptive_comparison.png
    """
    print(f"\n--- 6. ADAPTIVE MUTATION COMPARISON (N={num_nodes}, p={prob}) ---")

    g = nx.erdos_renyi_graph(n=num_nodes, p=prob, seed=SEED)

    # Run standard GA
    print("Running Standard GA (fixed mutation)...")
    standard_scores = []
    standard_traces = []
    for _ in range(runs):
        _, cost, trace = run_genetic_algorithm(g, generations=1000, operator="inversion", adaptive=False)
        standard_scores.append(cost)
        standard_traces.append(trace)

    # Run adaptive GA
    print("Running Adaptive GA (diversity-based mutation)...")
    adaptive_scores = []
    adaptive_traces = []
    for _ in range(runs):
        _, cost, trace = run_genetic_algorithm(g, generations=1000, operator="inversion", adaptive=True)
        adaptive_scores.append(cost)
        adaptive_traces.append(trace)

    # Print summary
    print(f"\nStandard GA: Mean={np.mean(standard_scores):.2f}, Success={(standard_scores.count(0)/runs)*100:.0f}%")
    print(f"Adaptive GA: Mean={np.mean(adaptive_scores):.2f}, Success={(adaptive_scores.count(0)/runs)*100:.0f}%")

    # Statistical test and visualization
    scores_dict = {'Standard GA': standard_scores, 'Adaptive GA': adaptive_scores}
    compute_statistical_tests(scores_dict, output_file="adaptive_comparison_stats.txt")
    plot_statistical_comparison(scores_dict,
                                output_prefix="adaptive_comparison_stats",
                                title=f"Standard vs Adaptive GA (N={num_nodes})")

    # Plot convergence comparison (using median trace)
    def get_median_trace(traces):
        max_len = max(len(t) for t in traces)
        padded = [t + [t[-1]] * (max_len - len(t)) for t in traces]
        return np.median(padded, axis=0)

    median_standard = get_median_trace(standard_traces)
    median_adaptive = get_median_trace(adaptive_traces)

    plt.figure(figsize=(10, 6))
    plt.plot(median_standard, label='Standard GA (Fixed Mutation)',
linewidth=2)
    plt.plot(median_adaptive, label='Adaptive GA (Diversity-Based)', linewidth=2, linestyle='--')
    plt.title(f'GA Convergence: Standard vs Adaptive Mutation (N={num_nodes})')
    plt.xlabel('Generation')
    plt.ylabel('Broken Edges (Cost)')
    plt.ylim(bottom=-0.5)
    plt.legend()
    plt.grid(True)

    plt.savefig('experiment_5_adaptive_comparison.png', dpi=300, bbox_inches='tight')
    print("Saved 'experiment_5_adaptive_comparison.png'")

    # Save detailed results to file
    with open('statistical_results.txt', 'a') as f:
        f.write(f"\n{'='*60}\n")
        f.write(f"Adaptive Mutation Comparison (N={num_nodes}, p={prob})\n")
        f.write(f"{'='*60}\n")
        f.write(f"Standard GA: Mean={np.mean(standard_scores):.2f}, Success={(standard_scores.count(0)/runs)*100:.0f}%\n")
        f.write(f"Adaptive GA: Mean={np.mean(adaptive_scores):.2f}, Success={(adaptive_scores.count(0)/runs)*100:.0f}%\n")
    print("Results appended to 'statistical_results.txt'")

# ==========================================
# MAIN EXECUTION
# ==========================================

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Hamiltonian Path Metaheuristic Analysis")
    parser.add_argument("-N", "--nodes", type=int, default=50, help="Number of nodes in the graph (default: 50)")
    parser.add_argument("-p", "--prob", type=float, default=0.1, help="Edge creation probability (default: 0.1)")
    parser.add_argument("--mode", type=str, default="all",
                        choices=["all", "batch", "opt", "phase", "visual", "stats", "adaptive"],
                        help="Experiment mode to run individually")


    args = parser.parse_args()

    print(f"Running experiments with N={args.nodes} and p={args.prob}, Mode={args.mode}")

    # 1. Main Baseline (using 'Swap')
    if args.mode in ["all", "batch"]:
        run_batch_experiments(args.nodes, args.prob)

    # 2. The Report Recommendation (Proving 2-Opt is better)
    if args.mode in ["all", "opt"]:
        run_optimization_comparison(args.nodes)

    # 3. Physics/Difficulty Analysis (Runs ALL algorithms across
densities)
    if args.mode in ["all", "phase"]:
        run_phase_transition(args.nodes)

    # 4. Visual Output
    if args.mode in ["all", "visual"]:
        save_best_graph_html(args.nodes, args.prob)

    # 5. Statistical Analysis Mode (batch + stats)
    if args.mode == "stats":
        print("\n--- RUNNING BATCH WITH STATISTICAL ANALYSIS ---")
        OPERATOR = "inversion"
        g_exp = nx.erdos_renyi_graph(n=args.nodes, p=args.prob, seed=SEED)

        def run_batch_for_stats(algo_func, name):
            scores = []
            for _ in range(30):
                _, cost, _ = algo_func(g_exp)
                scores.append(cost)
            return scores

        sa_scores = run_batch_for_stats(lambda g: run_simulated_annealing(g, max_steps=3000, operator=OPERATOR), "SA")
        tabu_scores = run_batch_for_stats(lambda g: run_tabu_search(g, max_steps=1500, operator=OPERATOR), "Tabu")
        ga_scores = run_batch_for_stats(lambda g: run_genetic_algorithm(g, generations=1500, operator=OPERATOR), "GA")

        scores_dict = {'SA': sa_scores, 'Tabu': tabu_scores, 'GA': ga_scores}
        compute_statistical_tests(scores_dict)
        plot_statistical_comparison(scores_dict,
                                    output_prefix=f"stats_N{args.nodes}_p{args.prob}",
                                    title=f"Algorithm Comparison (N={args.nodes}, p={args.prob})")

    # 6. Adaptive Mutation Comparison
    if args.mode == "adaptive":
        run_adaptive_comparison(args.nodes, prob=args.prob)
```
Listing 7: Python 3.14.2