

Metaheuristic approach to the Hamiltonian Path

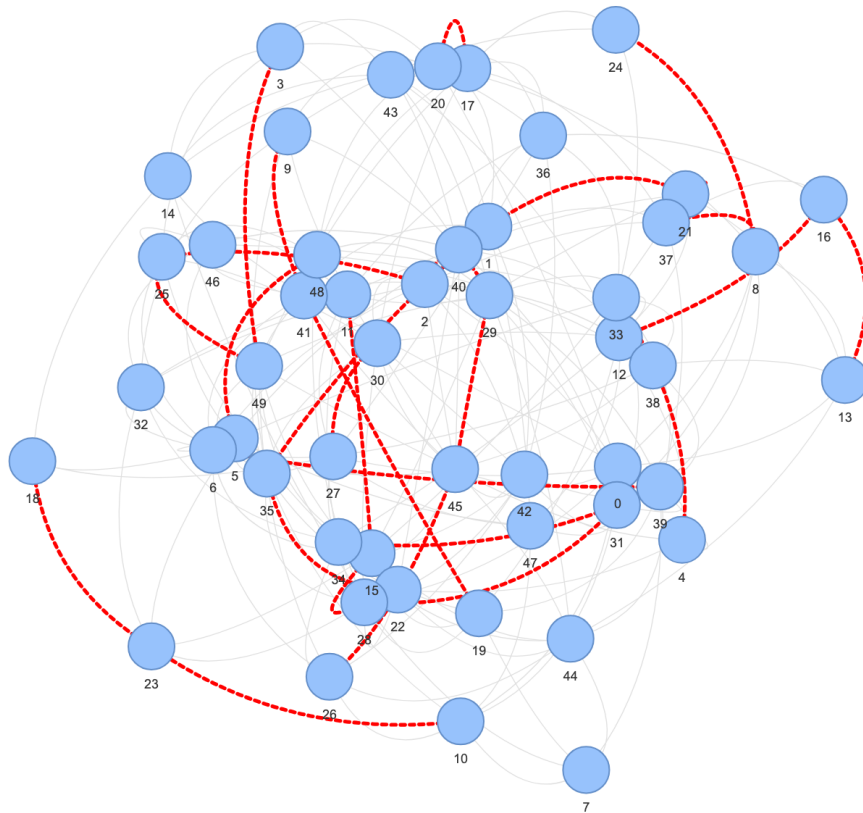


Figure 1: *Visualised with Pyvis, depicts the incomplete solution – Hamiltonian path, which the GA found using the 2-OPT (inversion) operator in the Erdős-Rényi random graph. Broken edges are depicted in red.*

Module: Algorithms and Combinatorial Thinking 2025-2026

Written by:

Christos Christophoros Mitsakopoulos

Supervised by: Dr. Franck Delaplace

December 29, 2025

Contents

1	Formal Definition and Introduction	2
2	Random Generation of the Problem: Erdős-Rényi Random Graph	2
3	Test Environment: Experimental Choices and Reasoning	3
4	Metaheuristic Algorithms: Mode of Action and Code Snippets	3
4.1	Objective Function	3
4.2	Simulated Annealing (SA)	4
4.3	Tabu Search (TS)	4
4.4	Genetic Algorithm (GA)	5
5	Experiment: Comparing the Metaheuristic Algorithms	5
5.1	Batch Testing	5
5.2	Phase Transition	6
6	GA: Comparing Swap Operator to 2-Opt	7
7	Annex	8

1 Formal Definition and Introduction

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

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.

```
1 (project) chrismitsakopoulos@192 Delaplace_coursework % python3 main.py  
--nodes 50 --prob 0.15
```

Listing 1: Usage scenario where the number of nodes for the problem graph are set to 50, with the probability of any two nodes having a connecting edge is 0.15. (This was ran in a Python environment – ensure you install the dependencies listed in the annex.)

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; so much so that all algorithms would converge to a zero cost solution (no broken edges).

```

1 # n default = 50, p default = 0.1, seed hardcoded to 42
2 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:

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

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 not only tested in the case of GA, to demonstrate how it can be a positive addition on an already accurate algorithm. However, it was also tested in the case of SA and GA to show that the impact of this change is not algorithm-specific; this 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.

4 Metaheuristic Algorithms: Mode of Action and Code Snippets

4.1 Objective Function

All algorithms used aim 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 $S = [v_1, v_2, \dots, v_n]$, the cost function $C(S)$ is defined as:

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

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 it has found the true solution inside the solution space. 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):

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

```

1 # From main.py: Calculate cost difference
2 delta = neighbor_cost - current_cost
3
4 # Accept if better (delta < 0) OR with probability exp(-delta/T)
5 if delta < 0 or random.random() < exp(-delta / temp):
6     current = neighbor
7     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 an immediate neighbourhood of a current solution it has found and moves to the best available neighbour (=solution), even if that neighbour 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.

```

1 # From main.py: Moving to the best candidate not in the Tabu list
2 if cand not in tabu_list:
3     current = cand
4     found_move = True
5
6 # Update memory
7 tabu_list.append(current)
8 if len(tabu_list) > tenure:
9     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".

```
1 # From main.py: Preserves sub-segment from parent 1
2 child[start:end] = p1[start:end]
3
4 # Fills remaining slots from parent 2, skipping duplicates
5 for i in range(size):
6     if child[i] is None:
7         while p2[current_p2_idx] in child:
8             current_p2_idx += 1
9         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.

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

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

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 improves solving quality by improving local search, by evaluating a neighbourhood of size $k = 50$ per step ($O(k)$), which linearly increases runtime. Lastly, the GA achieves a

near-optimal solution 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.

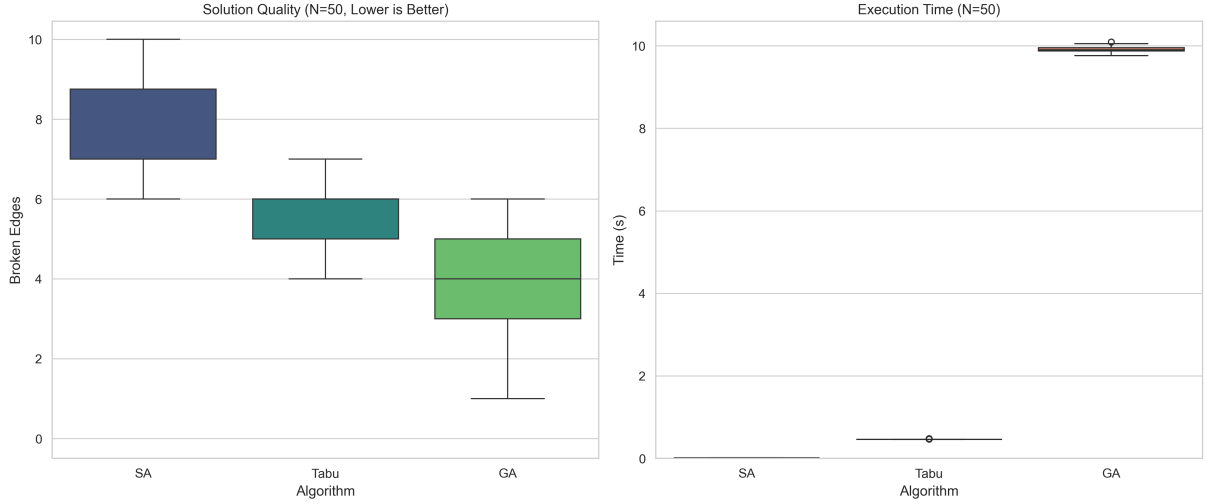


Figure 2: In the box whisker plots depicted, "Solution Quality" (Y-axis: Broken Edges) is shown on the left and "Execution Time" (Y-axis: Time(seconds)) on the right. As discussed prior, due to the low probability of edges between any two nodes, the graph is sparse and the metaheuristic algorithms – 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 occurred 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).

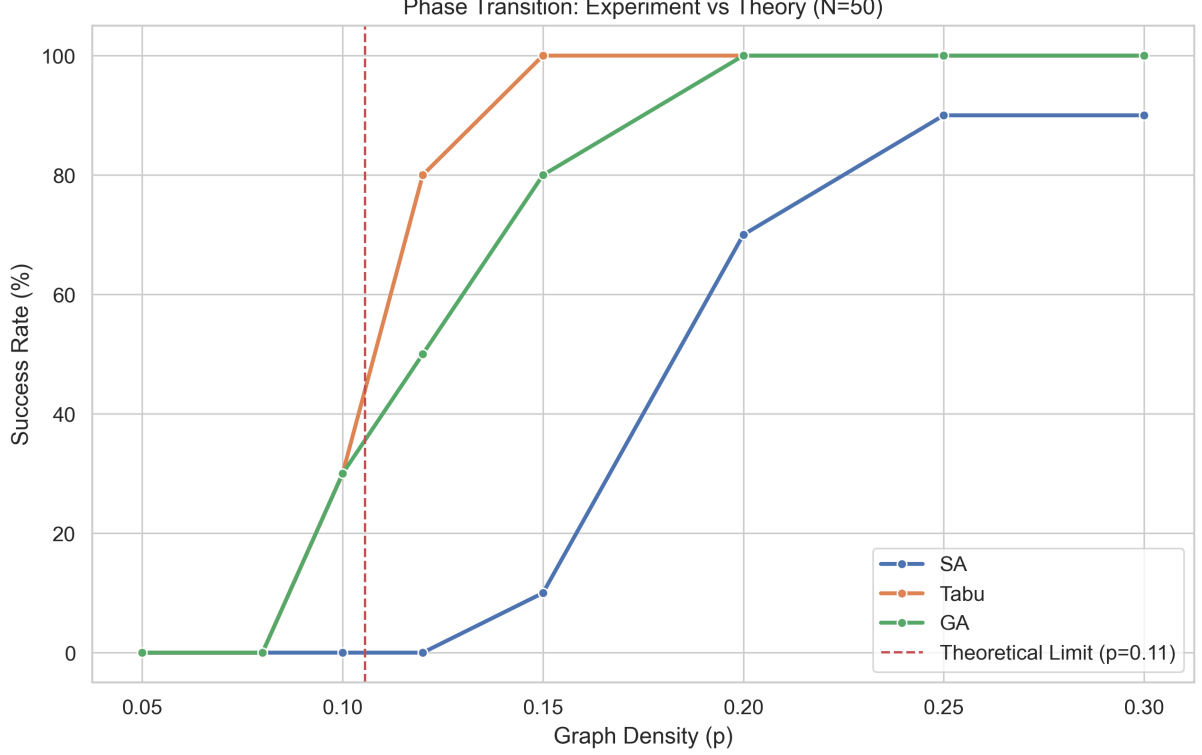


Figure 3: This line chart depicts the phase transition of the proposed problem Erdos-Renyi graph, at a cycling p rate between 0.05 and 0.3. The theoretical limit is computed automatically based on the input problem graph, given the Komlós Szemerédi formula. Observe overconfidence from the TS and GA algorithms, 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 to the algorithm making more destructive changes at each iteration; causing the search

to stagnate (maintain one solution over multiple iterations) until a more rare, but more beneficial solution is accidentally found by the destructive search.

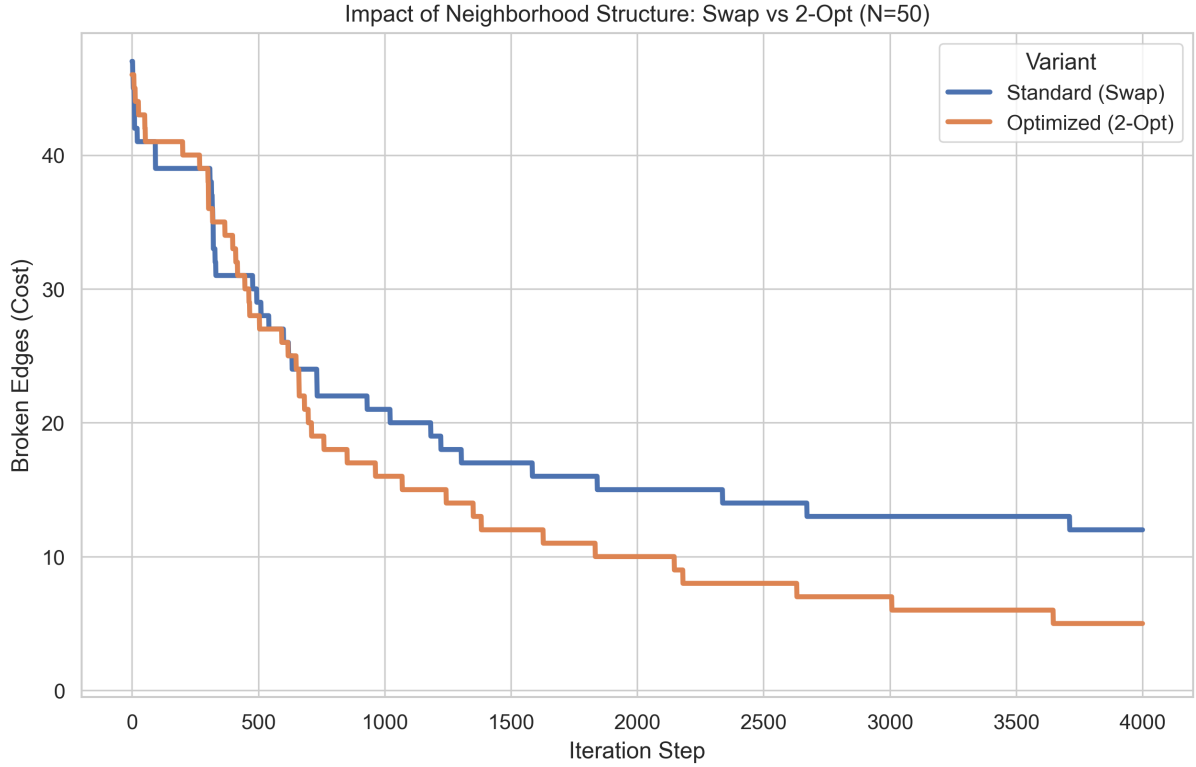


Figure 4: In the depicted line chart, Broken Edges (Y-axis) for which the objective function seeks to minimise (Cost) by maximising the number of edges in the path, is plotted against the number of generations – repeated runs (X-axis). The GA with the Swap operator is shown in blue, while the GA with 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 Annex

```

1 import random
2 import time
3 import math
4 import itertools
5 import argparse
6 import numpy as np
7 import pandas as pd
8 import networkx as nx
9 import matplotlib.pyplot as plt
10 import seaborn as sns
11 from math import exp, log
12
13 # --- CONFIGURATION ---
14 SNS_THEME = "whitegrid"
15 SNS_CONTEXT = "paper"
16 SEED = 42

```

```

17
18 # Apply settings
19 random.seed(SEED)
20 np.random.seed(SEED)
21 sns.set_theme(style=SNS_THEME, context=SNS_CONTEXT, font_scale=1.2)
22
23 # =====
24 # 1. CORE HELPER & FITNESS FUNCTIONS
25 # =====
26
27 def get_initial_solution(nodes):
28     """Returns a random permutation of nodes."""
29     p = list(nodes).copy()
30     random.shuffle(p)
31     return p
32
33 def count_edges(path, graph):
34     """Counts valid edges in the path for a specific graph."""
35     edges = 0
36     # Optimization: Iterate via index to avoid creating new lists
37     for i in range(len(path) - 1):
38         if graph.has_edge(path[i], path[i+1]):
39             edges += 1
40     return edges
41
42 def fitness_min(path, graph):
43     """Minimisation: Returns number of broken edges. Target = 0."""
44     target = len(path) - 1
45     valid = count_edges(path, graph)
46     return float(target - valid)
47
48 def fitness_max(path, graph):
49     """Maximisation: Returns number of valid edges. Target = N-1."""
50     return float(count_edges(path, graph))
51
52 # =====
53 # 2. OPERATORS (THE OPTIMISATIONS)
54 # =====
55
56 def op_swap(genome):
57     """Standard Swap: Exchanges two random nodes."""
58     n = len(genome)
59     i, j = random.sample(range(n), 2)
60     genome[i], genome[j] = genome[j], genome[i]
61     return genome
62
63 def op_inversion(genome):
64     """
65     2-Opt Inversion: Reverses a random sub-segment.
66     OPTIMIZATION: Preserves adjacency better than swap.
67     """
68     n = len(genome)
69     i, j = sorted(random.sample(range(n), 2))
70     # Reverse the segment between i and j
71     genome[i:j+1] = genome[i:j+1][::-1]
72     return genome
73
74 # =====

```

```

75 # 3. METAHEURISTIC ALGORITHMS
76 # =====
77
78 def run_simulated_annealing(graph, max_steps=3000, temp0=100.0, operator
="swap"):
79     """Simulated Annealing with selectable operator."""
80     nodes = list(graph.nodes)
81     current = get_initial_solution(nodes)
82     best = current.copy()
83
84     current_cost = fitness_min(current, graph)
85     best_cost = current_cost
86     trace = [best_cost]
87     temp = temp0
88
89     mutate_func = op_inversion if operator == "inversion" else op_swap
90
91     for step in range(max_steps):
92         if best_cost == 0: break
93
94         neighbor = current.copy()
95         neighbor = mutate_func(neighbor)
96
97         neighbor_cost = fitness_min(neighbor, graph)
98         delta = neighbor_cost - current_cost
99
100        if delta < 0 or random.random() < exp(-delta / temp):
101            current = neighbor
102            current_cost = neighbor_cost
103            if current_cost < best_cost:
104                best = current.copy()
105                best_cost = current_cost
106
107        trace.append(best_cost)
108        temp *= 0.985
109
110    return best, best_cost, trace
111
112 def run_tabu_search(graph, max_steps=1000, tenure=20, operator="swap"):
113     """Tabu Search with selectable operator."""
114     nodes = list(graph.nodes)
115     current = get_initial_solution(nodes)
116     best = current.copy()
117     best_cost = fitness_min(best, graph)
118
119     tabu_list = []
120     trace = [best_cost]
121     mutate_func = op_inversion if operator == "inversion" else op_swap
122
123     for step in range(max_steps):
124         if best_cost == 0: break
125
126         candidates = []
127         for _ in range(50):
128             cand = current.copy()
129             cand = mutate_func(cand)
130             candidates.append(cand)
131

```

```

132     candidates.sort(key=lambda p: fitness_min(p, graph))
133
134     found_move = False
135     for cand in candidates:
136         cand_cost = fitness_min(cand, graph)
137         if cand_cost < best_cost:
138             current = cand
139             best = cand
140             best_cost = cand_cost
141             found_move = True
142             break
143         if cand not in tabu_list:
144             current = cand
145             found_move = True
146             break
147
148     trace.append(best_cost)
149     if found_move:
150         tabu_list.append(current)
151         if len(tabu_list) > tenure:
152             tabu_list.pop(0)
153
154     return best, best_cost, trace
155
156 def run_genetic_algorithm(graph, pop_size=200, generations=1000,
157 cross_rate=0.8, mut_rate=0.3, operator="swap"):
158     """Genetic Algorithm with selectable operator."""
159     nodes = list(graph.nodes)
160     target = len(nodes) - 1
161
162     mutate_func = op_inversion if operator == "inversion" else op_swap
163
164     def ordered_crossover(p1, p2):
165         size = len(p1)
166         start, end = sorted(random.sample(range(size), 2))
167         child = [None] * size
168         child[start:end] = p1[start:end]
169         current_p2_idx = 0
170         for i in range(size):
171             if child[i] is None:
172                 while p2[current_p2_idx] in child:
173                     current_p2_idx += 1
174                 child[i] = p2[current_p2_idx]
175         return child
176
177     population = [get_initial_solution(nodes) for _ in range(pop_size)]
178     best_sol = None
179     best_broken = float('inf')
180     trace = []
181
182     for gen in range(generations):
183         fitnesses = [fitness_max(p, graph) for p in population]
184         best_val = max(fitnesses)
185         current_broken = target - best_val
186
187         if current_broken < best_broken:
188             best_broken = current_broken
189             best_sol = population[fitnesses.index(best_val)]

```

```

189     trace.append(best_broken)
190     if current_broken == 0: break
191
192     next_pop = [best_sol.copy()]
193     while len(next_pop) < pop_size:
194         competitors = random.sample(population, 3)
195         winner = max(competitors, key=lambda p: fitness_max(p, graph
196 ))
197         next_pop.append(winner.copy())
198     population = next_pop
199
200     for i in range(1, pop_size - 1, 2):
201         if random.random() < cross_rate:
202             population[i] = ordered_crossover(population[i],
203 population[i+1])
204
205     for i in range(1, pop_size):
206         if random.random() < mut_rate:
207             population[i] = mutate_func(population[i])
208
209     return best_sol, best_broken, trace
210 # =====
211 # 4. EXPERIMENT RUNNERS
212 # =====
213
214 def run_batch_experiments(num_nodes, prob):
215     """Runs each algorithm 30 times and plots results."""
216     # Baseline comparison using Standard Swap
217     OPERATOR = "swap"
218
219     g_exp = nx.erdos_renyi_graph(n=num_nodes, p=prob, seed=SEED)
220
221     print(f"\n--- 1. BATCH EXPERIMENT (Baseline 'Swap', N={num_nodes}, p
222 ={prob}, 30 Runs) ---")
223
224     def run_batch(algo_func, name):
225         scores = []
226         times = []
227         successes = 0
228         for _ in range(30):
229             start = time.time()
230             _, cost, _ = algo_func(g_exp)
231             dur = time.time() - start
232             scores.append(cost)
233             times.append(dur)
234             if cost == 0: successes += 1
235
236         rate = (successes/30)*100
237         print(f"{name}: Mean Cost={np.mean(scores):.2f}, Success={rate
238 :.1f}%, Mean Time={np.mean(times):.4f}s")
239         return scores, times
240
241     sa_scores, sa_times = run_batch(lambda g: run_simulated_annealing(g,
242 max_steps=3000, operator=OPERATOR), "SA")
243     tabu_scores, tabu_times = run_batch(lambda g: run_tabu_search(g,
244 max_steps=1500, operator=OPERATOR), "Tabu")

```

```

241 ga_scores, ga_times = run_batch(lambda g: run_genetic_algorithm(g,
generations=1500, operator=OPERATOR), "GA")
242
243 data = {
244     'Algorithm': ['SA']*30 + ['Tabu']*30 + ['GA']*30,
245     'Broken Edges': sa_scores + tabu_scores + ga_scores,
246     'Time (s)': sa_times + tabu_times + ga_times
247 }
248 df = pd.DataFrame(data)
249
250 fig, axes = plt.subplots(1, 2, figsize=(14, 6))
251
252 sns.boxplot(data=df, x='Algorithm', y='Broken Edges', hue='Algorithm',
253             legend=False, ax=axes[0], palette="viridis")
254 axes[0].set_title(f'Solution Quality (N={num_nodes}, Lower is Better)')
255 axes[0].set_ylim(bottom=-0.5)
256
257 sns.boxplot(data=df, x='Algorithm', y='Time (s)', hue='Algorithm',
258             legend=False, ax=axes[1], palette="magma")
259 axes[1].set_title(f'Execution Time (N={num_nodes})')
260 axes[1].set_ylim(bottom=0)
261
262 plt.tight_layout()
263 plt.savefig('experiment_1_batch_results.png', dpi=300, bbox_inches='
tight')
264 print("Saved 'experiment_1_batch_results.png'")
265 # plt.show() # Uncomment if running interactively
266
267 def run_optimization_comparison(num_nodes):
268     """Compares Standard 'Swap' vs Optimized 'Inversion' (2-Opt)."""
269     print(f"\n--- 2. OPTIMISATION EVALUATION (N={num_nodes}) ---")
270     g_opt = nx.erdos_renyi_graph(n=num_nodes, p=0.1, seed=SEED)
271
272     print("Running SA with Standard Swap...")
273     _, _, trace_swap = run_simulated_annealing(g_opt, max_steps=4000,
operator="swap")
274
275     print("Running SA with Optimized Inversion (2-Opt)...")
276     _, _, trace_inv = run_simulated_annealing(g_opt, max_steps=4000,
operator="inversion")
277
278     df_opt = pd.DataFrame({
279         'Step': list(range(len(trace_swap))) + list(range(len(trace_inv)
)),
280         'Broken Edges': trace_swap + trace_inv,
281         'Variant': ['Standard (Swap)'] * len(trace_swap) + ['Optimized
(2-Opt)'] * len(trace_inv)
282     })
283
284     plt.figure(figsize=(10, 6))
285     sns.lineplot(data=df_opt, x='Step', y='Broken Edges', hue='Variant',
linewidth=2.5)
286     plt.title(f'Impact of Neighborhood Structure: Swap vs 2-Opt (N={
num_nodes})')
287     plt.ylabel('Broken Edges (Cost)')
288     plt.xlabel('Iteration Step')

```

```

289     plt.ylim(bottom=-0.5)
290
291     plt.savefig('experiment_2_optimization_comparison.png', dpi=300,
292     bbox_inches='tight')
293     print("Saved 'experiment_2_optimization_comparison.png'")
294 def run_phase_transition(num_nodes):
295     """
296     Analyzes difficulty vs graph density across ALL algorithms.
297     This provides a comprehensive view of algorithmic limits.
298     """
299     print(f"\n--- 3. PHASE TRANSITION ANALYSIS (N={num_nodes}) ---")
300     # Densities to test
301     densities = [0.05, 0.08, 0.1, 0.12, 0.15, 0.2, 0.25, 0.3]
302
303     # Calculate Theoretical Threshold for THIS N
304     # Koml s & Szemer di theorem:  $p = (\ln(n) + \ln(\ln(n))) / n$ 
305     if num_nodes > 1:
306         math_threshold = (log(num_nodes) + log(log(num_nodes))) /
num_nodes
307         print(f"Theoretical Critical Threshold for N={num_nodes}:  $p \sim \{$ 
math_threshold:.3f}")
308     else:
309         math_threshold = 0
310
311     # Store results for plotting
312     results = {'Density': [], 'Success Rate': [], 'Algorithm': []}
313
314     # Test all algorithms to see which one handles the transition best
315     algorithms = {
316         'SA': lambda g: run_simulated_annealing(g, max_steps=2500,
operator="inversion"),
317         'Tabu': lambda g: run_tabu_search(g, max_steps=1500, operator="
inversion"),
318         'GA': lambda g: run_genetic_algorithm(g, generations=1000,
operator="inversion")
319     }
320
321     for p in densities:
322         # Create a graph for this density
323         # Note: Ideally we average over multiple graphs, but for speed
we use one seed per density
324         # or we generate a fresh one each run. Let's stick to one graph
instance per density
325         # but 10 runs per algorithm on it.
326         g = nx.erdos_renyi_graph(n=num_nodes, p=p, seed=999)
327
328         print(f"Testing Density p={p}...")
329
330         for name, algo_func in algorithms.items():
331             successes = 0
332             for _ in range(10): # 10 runs per density/algorithm pair
333                 _, cost, _ = algo_func(g)
334                 if cost == 0: successes += 1
335
336             rate = (successes/10)*100
337             results['Density'].append(p)
338             results['Success Rate'].append(rate)

```

```

339         results['Algorithm'].append(name)
340
341     df_phase = pd.DataFrame(results)
342
343     plt.figure(figsize=(10, 6))
344     sns.lineplot(data=df_phase, x='Density', y='Success Rate', hue='
Algorithm', marker='o', linewidth=2)
345
346     # Plot the Theoretical Line
347     plt.axvline(x=math_threshold, color='r', linestyle='--', label=f'
Theoretical Limit (p={math_threshold:.2f})')
348
349     plt.title(f'Phase Transition: Experiment vs Theory (N={num_nodes})')
350     plt.xlabel('Graph Density (p)')
351     plt.ylabel('Success Rate (%)')
352     plt.ylim(-5, 105)
353     plt.legend()
354     plt.grid(True)
355
356     plt.savefig('experiment_3_phase_transition.png', dpi=300,
bbox_inches='tight')
357     print("Saved 'experiment_3_phase_transition.png'")
358
359 def save_best_graph_html(num_nodes, prob):
360     """Runs GA once and saves the result to HTML."""
361     print("\n--- 4. GENERATING VISUALISATION ---")
362     try:
363         from pyvis.network import Network
364     except ImportError:
365         print("Pyvis not installed. Skipping.")
366         return
367
368     # Use arguments, but if probability is too low, we might not find a
path
369     g = nx.erdos_renyi_graph(n=num_nodes, p=prob, seed=SEED)
370     path, cost, _ = run_genetic_algorithm(g, generations=2000, operator=
"inversion")
371
372     print(f"Final Path Cost: {cost}")
373
374     net = Network(height="600px", width="100%", cdn_resources='remote')
375
376     for n in g.nodes:
377         net.add_node(int(n), label=str(n), color='#97c2fc')
378
379     for u, v in g.edges:
380         net.add_edge(int(u), int(v), color='#e0e0e0', width=1)
381
382     for i in range(len(path) - 1):
383         u, v = int(path[i]), int(path[i+1])
384         if g.has_edge(u, v):
385             net.add_edge(u, v, color='red', width=4)
386         else:
387             net.add_edge(u, v, color='red', width=4, dashes=True)
388
389     net.show("hamiltonian_path.html", notebook=False)
390     print("Saved to 'hamiltonian_path.html'")
391

```



```

392 # =====
393 # MAIN EXECUTION
394 # =====
395
396 if __name__ == "__main__":
397     parser = argparse.ArgumentParser(description="Hamiltonian Path
398     Metaheuristic Analysis")
399     parser.add_argument("-N", "--nodes", type=int, default=50, help="
400     Number of nodes in the graph (default: 50)")
401     parser.add_argument("-p", "--prob", type=float, default=0.1, help="
402     Edge creation probability (default: 0.1)")
403     parser.add_argument("--mode", type=str, default="all", choices=["all
404     ", "batch", "opt", "phase", "visual"], help="Experiment mode to run
405     individually")
406
407     args = parser.parse_args()
408
409     print(f"Running experiments with N={args.nodes} and p={args.prob},
410     Mode={args.mode}")
411
412     # 1. Main Baseline (using 'Swap')
413     if args.mode in ["all", "batch"]:
414         run_batch_experiments(args.nodes, args.prob)
415
416     # 2. The Report Recommendation (Proving 2-Opt is better)
417     if args.mode in ["all", "opt"]:
418         run_optimization_comparison(args.nodes)
419
420     # 3. Physics/Difficulty Analysis (Runs ALL algorithms across
421     densities)
422     if args.mode in ["all", "phase"]:
423         run_phase_transition(args.nodes)
424
425     # 4. Visual Output
426     if args.mode in ["all", "visual"]:
427         save_best_graph_html(args.nodes, args.prob)

```

Listing 6: Python 3.14.2