

Metaheuristic approach to the Hamiltonian Path

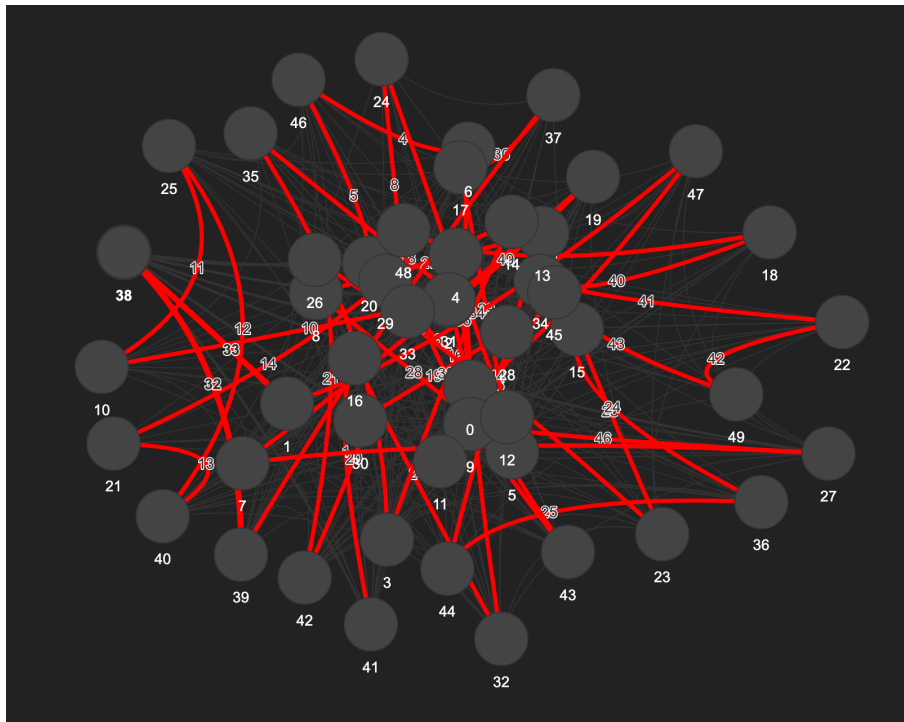


Figure 1: *Visualised with Pyvis, depicts the correct Hamiltonian path through the artificially generated, 50 node graph.*

Module: Algorithms and Combinatorial Thinking 2025-2026

Written by:

Christos Christophoros Mitsakopoulos

Supervised by: Dr. Franck Delaplace

December 24, 2025

Contents

1	Formal Definition and Introduction	2
2	Test Environment – Random Generation of the Problem: Erdős-Rényi Random Graph	2
3	Metaheuristic Algorithms: Mode of Action and their Implementation	3
3.1	Objective Function	3
3.2	Simulated Annealing (SA)	3
3.3	Tabu Search (TS)	4
3.4	Genetic Algorithm (GA)	4
3.5	Comparative Analysis: Neighbourhood Operators	4

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 Test Environment – 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...*

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., $\text{SEM} = \sigma/\sqrt{N}$).

3 Metaheuristic Algorithms: Mode of Action and their Implementation

3.1 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:

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

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.

3.2 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):

$$P(\text{accept}) = \begin{cases} 1 & \text{if } \Delta C < 0 \\ e^{-\frac{\Delta C}{T}} & \text{if } \Delta C \geq 0 \end{cases} \quad (2)$$

```

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)*

3.3 Tabu Search (TS)

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

To prevent cycling (revisiting the same solutions endlessly), the algorithm maintains a *Tabu List*—a short-term memory queue that forbids recently visited solutions for a specific duration (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

3.4 Genetic Algorithm (GA)

The Genetic Algorithm mimics natural selection. Unlike SA and TS, which improve a single solution, GA evolves a population of solutions. The core operator for permutation problems is *Ordered Crossover* (OX1), which is necessary because standard single-point crossover would result in invalid paths (duplicate or missing nodes).

The implementation uses OX1 to preserve the relative 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

3.5 Comparative Analysis: Neighbourhood Operators

A critical finding in the experimental setup was the impact of the mutation operator. The standard `swap` operator (exchanging two indices) disrupts the adjacency of the path significantly.

In contrast, the **inversion** (2-Opt) operator reverses a segment of the path. This is mathematically superior for path problems because it preserves the internal adjacency of the reversed segment, only breaking the two edges at the endpoints of the segment. The code allows for switching between these operators to demonstrate this efficiency gap.

$$\text{Swap}(S) \rightarrow \text{High disruption of edges} \quad (3)$$

$$\text{Inversion}(S) \rightarrow \text{Minimal disruption (2-Opt)} \quad (4)$$

This distinction is implemented via the ‘op_inversion’ function in ‘main.py’ and is the primary driver for convergence in denser graphs.