Randomized Algorithms for the Minimum Weight Vertex Cover Problem: Simulated Annealing and Genetic Algorithms

Eduardo Lopes
Master's degree in Computer Engineering

Abstract **– Given an undirected graph G(V, E) with weighted vertices, randomized algorithms are employed to find minimum weight vertex covers. Two metaheuristic approaches are implemented and analyzed: Simulated Annealing and Genetic Algorithm. Comprehensive analysis of computational complexity, execution time, and basic operations demonstrates polynomial-time behavior, enabling processing of graphs up to 950 vertices compared to the exhaustive search limit of 25 vertices from prior work. These randomized algorithms achieve significant scalability improvements while trading optimality guarantees for practical execution times on large-scale instances.**

Keywords **– Minimum Weight Vertex Cover, Randomized Algorithms, Simulated Annealing, Genetic Algorithm, Metaheuristics, NP-hard Problems, Combinatorial Optimization**

I. Introduction

The Minimum Weight Vertex Cover problem represents a fundamental challenge in graph theory with numerous practical applications in network design, resource allocation, and computational biology. Given an undirected graph $G(V, E)$ where each vertex carries a positive weight, the objective is to find a subset C of vertices such that every edge in the graph is incident to at least one vertex in C, while minimizing the total weight of the selected vertices. This problem is known to be NP-hard [1], meaning that no polynomial-time algorithm is known to solve it optimally for all instances.

A previous investigation of this problem explored two contrasting algorithmic paradigms: exhaustive enumeration and greedy heuristics. The exhaustive search approach systematically tests all possible vertex subsets to guarantee optimal solutions, but exhibits exponential time complexity that limits practical applicability to small graphs. In contrast, greedy heuristics employ polynomial-time strategies that trade optimality guarantees for computational efficiency, making deterministic, irreversible decisions at each step. While greedy algorithms demonstrate substantial computational savings, they commit to a single solution path without mechanisms to escape local optima, resulting in solution quality variability and no theoretical guarantees on optimality.

This prior work established a fundamental algorithmic trade-off: exhaustive search guarantees optimality but scales exponentially, while greedy heuristics scale polynomially but sacrifice solution quality and provide no optimality guarantees. Randomized algorithms offer a promising middle ground, combining polynomial-time scalability with the ability to escape local optima through probabilistic exploration of the solution space. Unlike deterministic greedy heuristics that commit to a single trajectory through the search space, randomized metaheuristics such as Simulated Annealing and Genetic Algorithms employ stochastic mechanisms to explore diverse solution regions. Simulated Annealing uses temperature-based probabilistic acceptance of worse solutions to avoid premature convergence to local optima [2], [3], while Genetic Algorithms maintain a population of candidate solutions that evolve through selection, crossover, and mutation operators, with recent advances in adaptive repair mechanisms improving solution quality [4], [5]. These approaches have demonstrated success on various NP-hard combinatorial optimization problems and may potentially achieve higher quality solutions than the greedy baseline established in prior work, while remaining practical for instances beyond the exhaustive search limit.

To accomplish this comparative analysis of randomized metaheuristics, the primary objectives encompass several interconnected goals. First, characterizing the formal time and space complexity of both Simulated Annealing and Genetic Algorithm approaches, establishing theoretical bounds that predict their behavior. Second, conducting extensive experimental measurements across diverse graph instances to observe actual performance, tracking basic operations, execution times, and solution quality metrics. Third, evaluating how effectively these randomized approaches scale to large problem instances that exceed the practical limits of exhaustive search. Finally, determining algorithm selection guidelines based on instance characteristics and time constraints, providing practical recommendations for real-world applications.

II. Problem Definition

A. Formal Statement

Consider an undirected graph G = (V, E) where V represents the set of n vertices and E represents the set of m edges. Each vertex v ∈ V carries an associated positive weight w(v) > 0. The objective is to identify a vertex cover C ⊆ V that minimizes the total weight

W(C) = $\sum_{v \in C}$ w(v).

This formulation naturally leads to a constrained optimization problem that requires balancing two competing objectives: ensuring complete edge coverage while minimizing the cumulative weight of selected vertices. The challenge stems from the exponential number of possible vertex subsets, each of which must be evaluated for validity before its weight can be considered.

B. Vertex Cover Definition

A subset C ⊆ V qualifies as a vertex cover if and only if every edge in the graph has at least one endpoint contained in C. Formally, this requirement can be expressed as: ∀(u, v) ∈ E: u ∈ C ∨ v ∈ C. Intuitively, a vertex cover must "cover" all edges, meaning that no edge can exist with both endpoints outside the cover.

This definition has important implications for algorithm design. Any vertex not included in the cover effectively restricts which other vertices can be excluded, creating complex dependencies that make greedy approaches challenging. The cover property must be satisfied completely, as partial coverage where some edges remain uncovered does not constitute a valid solution.

III. Simulated Annealing Algorithm

Simulated Annealing employs probabilistic local search inspired by the metallurgical annealing process to find approximate solutions through temperature-controlled acceptance of both improving and worsening moves. The key innovation lies in its ability to escape local optima: unlike greedy heuristics that commit to single solution paths and get trapped in local minima, SA explores the solution space by accepting worse solutions with probability $P_{accept} = e^{-\Delta W/T}$, where ΔW is the weight increase and T is the current temperature. This probabilistic acceptance provides a controlled mechanism for exploration that decreases over time.

A. Algorithm Description

The algorithm begins by constructing an initial solution using a greedy heuristic that iteratively selects vertices with favorable degree-to-weight ratios, providing a reasonable starting point for the search. The temperature parameter is initialized to $T_{initial}$, set sufficiently high to allow broad exploration of the solution space in early iterations.

At each iteration, a neighbor solution is generated by applying one of four random vertex operations to the current solution. If the neighbor has lower weight ($\Delta W < 0$), it is immediately accepted as an improvement. Crucially, if the neighbor has higher weight ($\Delta W > 0$), it may still be accepted with probability determined by the Metropolis criterion, enabling uphill moves that can lead to better solutions in different regions of the search space. After each iteration, temperature decreases geometrically according to $T_{k+1} = \alpha \times T_k$, gradually reducing the acceptance probability of worse solutions and transitioning from exploration to exploitation of promising regions.

Algorithm 1 Simulated Annealing for Minimum Weight Vertex Cover

```
1:  $C_{current} \leftarrow GreedyInitialSolution(G)$
2:  $W_{current} \leftarrow \sum_{v \in C_{current}} w(v)$
3:  $C_{best} \leftarrow C_{current}$
4:  $W_{best} \leftarrow W_{current}$
5:  $T \leftarrow T_{initial}$
6:
7:  while $T > T_{min}$ and iterations $< max\_iterations$ do
8:      $C_{neighbor} \leftarrow GenerateNeighbor(C_{current})$
9:
10:     if $IsVertexCover(G, C_{neighbor})$ then
11:         $W_{neighbor} \leftarrow \sum_{v \in C_{neighbor}} w(v)$
12:         $\Delta W \leftarrow W_{neighbor} - W_{current}$
13:
14:         if $\Delta W < 0$ then
15:             $C_{current} \leftarrow C_{neighbor}$        ▷ Accept better solution
16:             $W_{current} \leftarrow W_{neighbor}$
17:
18:             if $W_{current} < W_{best}$ then
19:                 $C_{best} \leftarrow C_{current}$
20:                 $W_{best} \leftarrow W_{current}$
21:             end if
22:         else
23:             $P \leftarrow e^{-\Delta W/T}$        ▷ Metropolis criterion
24:
25:             if $random() < P$ then
26:                 $C_{current} \leftarrow C_{neighbor}$        ▷ Accept worse solution
27:                 $W_{current} \leftarrow W_{neighbor}$
28:             end if
29:         end if
30:     end if
31:
32:     $T \leftarrow T \times \alpha$        ▷ Cool down
33: end while
34:
35: return $(C_{best}, W_{best})$
```

B. Neighborhood Generation

The effectiveness of SA depends critically on the neighborhood structure. Four operators generate candidate solutions: (1) Add selects a random vertex not in the current cover and includes it, always maintaining validity; (2) Remove attempts to eliminate a vertex while preserving coverage, or removes a random vertex if no safe removal exists; (3) Swap exchanges one vertex for another, exploring solutions with similar cardinality; (4) Repair identifies uncovered edges and adds vertices to restore validity. Random operator selection at each iteration ensures diverse search trajectories, enabling both fine-grained local exploitation through small modifications and exploration of distant solution regions through more disruptive changes.

C. Temperature Schedule

The cooling schedule controls the exploration-exploitation balance. Geometric cooling $T_{k+1} = \alpha \times T_k$ with $\alpha \in [0.90, 0.99]$ provides exponential temperature reduction, creating a smooth transition from exploration to exploitation. At high initial temperature $T_{initial}$, most worse solutions are accepted, enabling broad exploration. As temperature decreases, acceptance probability drops, gradually focusing the search on high-quality regions. The process terminates when temperature reaches minimum threshold T_{min} or the maximum iteration limit is exceeded.

D. Complexity Analysis

The time complexity analysis follows the algorithm structure. Each iteration of the main loop performs several operations: neighbor generation, cover validation, weight calculation, and acceptance decision. The neighbor generation through random operations requires $O(n)$ time to select and modify vertices. Cover validation examines all edges to verify coverage, contributing $O(m)$ time. Weight calculation sums vertex weights, requiring $O(n)$ time in the worst case. The acceptance probability calculation and random number generation require $O(1)$ time.

The number of iterations depends on the cooling schedule parameters. For a geometric schedule with cooling rate α and temperatures ranging from $T_{initial}$ to T_{min}, the number of iterations is:

$$k = \log_\alpha \left(\frac{T_{min}}{T_{initial}} \right) = \frac{\ln(T_{min}/T_{initial})}{\ln(\alpha)} \tag{1}$$

For typical parameter values ($T_{initial} = 1000$, $T_{min} = 0.01$, $\alpha = 0.95$), this yields approximately $k = 226$ iterations. With the maximum iteration limit also bounding execution, the total number of iterations is $k = \min(k_{temp}, max\_iterations)$.

Each iteration requires $O(n+m)$ time, yielding overall time complexity:

$$T(n, m) = O(k \times (n + m)) \tag{2}$$

Since k is typically configured as a constant or grows slowly with problem size (often set based on time limits rather than problem dimensions), the practical complexity is $O(n + m)$ per iteration with a configurable number of iterations. This represents polynomial-time execution per iteration, contrasting sharply with the exponential $O(2^n \times m)$ complexity of exhaustive search.

The space complexity is $O(n + m)$. The graph representation using adjacency lists requires $O(n + m)$ space. The algorithm maintains the current solution, best solution, and neighbor solution, each requiring at most $O(n)$ space. Additional tracking structures for ensuring unique solutions tested contribute $O(solutions\_tested)$ space, bounded by the number of iterations. Since all these components are dominated by the graph representation, the overall space complexity remains $O(n + m)$.

IV. Genetic Algorithm

Genetic Algorithm employs population-based evolutionary computation inspired by biological evolution to explore multiple solution regions simultaneously. The fundamental advantage over single-trajectory methods lies in population diversity: while SA explores one solution neighborhood at a time, GA maintains a population of diverse candidate solutions that evolve through selection, crossover, and mutation operators. This parallel exploration reduces susceptibility to local optima by simultaneously investigating different regions of the search space and enabling information exchange between high-quality solutions through recombination.

A. Algorithm Description

The algorithm initializes a population of size P by combining solutions generated through greedy construction with randomly generated alternatives, ensuring initial diversity. Each individual represents a vertex cover encoded as a set of vertices. Solution quality is evaluated using fitness function $f(C) = W(C) + \lambda \times |E_{uncovered}(C)|$, where $W(C)$ is the total vertex weight, $|E_{uncovered}(C)|$ counts uncovered edges, and penalty coefficient λ heavily penalizes invalid solutions. This formulation allows the algorithm to work with both valid and invalid covers, gradually driving the population toward feasibility.

The evolutionary cycle proceeds through four distinct phases at each generation: (1) Population sorting by fitness, with lower values indicating better solutions; (2) Parent selection using tournament selection, where random subsets of size 3-5 compete and the fittest individual is chosen for reproduction; (3) Genetic operators applied to produce offspring through crossover (with probability 0.8) and mutation (with probability 0.1); (4) Elitist replacement where the new generation replaces the old population while preserving the top 2-5 individuals, ensuring monotonic improvement of the best solution found.

B. Genetic Operators

The evolutionary process relies on three primary operators that introduce variation while preserving solution quality characteristics.

Tournament selection balances selection pressure with diversity maintenance. By sampling 3-5 random individuals and selecting the fittest, it provides moderate selection pressure that favors high-quality solutions while allowing weaker individuals occasional selection opportunities, maintaining genetic diversity within the population.

Uniform crossover combines genetic material from two parents to create offspring. For each vertex $v \in V$, the offspring independently inherits the inclusion decision from either parent with equal probability 0.5. This operator enables exploration of solution combinations not

Algorithm 2 Genetic Algorithm for Minimum Weight Vertex Cover

```
1:  $Population \leftarrow InitializePopulation(populationSize)$
2:  $C_{best} \leftarrow null$
3:  $f_{best} \leftarrow \infty$
4:
5:  for $generation = 1$ to $maxGenerations$ do
6:      $Sort(Population)$ by fitness    ▷ Lower is better
7:
8:      if $Population[0].fitness < f_{best}$ then
9:          $C_{best} \leftarrow Population[0].vertices$
10:         $f_{best} \leftarrow Population[0].fitness$
11:     end if
12:
13:     $NewPopulation \leftarrow \emptyset$
14:
15:     for $i = 1$ to $elitismCount$ do    ▷ Elitism
16:         $NewPopulation \leftarrow NewPopulation \cup \{Population[i]\}$
17:     end for
18:
19:     while $|NewPopulation| < populationSize$ do
20:         $parent_1 \leftarrow TournamentSelection(Population)$
21:         $parent_2 \leftarrow TournamentSelection(Population)$
22:
23:         if $random() < crossoverRate$ then
24:             $(child_1, child_2) \leftarrow Crossover(parent_1, parent_2)$
25:         else
26:             $(child_1, child_2) \leftarrow (parent_1, parent_2)$
27:         end if
28:
29:         if $random() < mutationRate$ then
30:             $child_1 \leftarrow Mutate(child_1)$
31:         end if
32:
33:         if $random() < mutationRate$ then
34:             $child_2 \leftarrow Mutate(child_2)$
35:         end if
36:
37:         $child_1 \leftarrow Repair(child_1)$    ▷ Ensure validity
38:         $child_2 \leftarrow Repair(child_2)$
39:
40:         $NewPopulation \leftarrow NewPopulation \cup \{child_1, child_2\}$
41:     end while
42:
43:     $Population \leftarrow NewPopulation$
44: end for
45:
46: return $(C_{best}, W(C_{best}))$
```

present in either parent, potentially discovering high-quality vertex combinations through recombination of beneficial traits.

Mutation introduces random variation through three types: flip mutation toggles a random vertex (adding if absent, removing if present), add mutation includes a new random vertex, and remove mutation excludes a current vertex. Applied at controlled rates, mutation prevents premature convergence and maintains population diversity.

Since crossover and mutation may produce invalid solutions, an adaptive greedy repair mechanism restores validity by identifying uncovered edges and adding covering vertices with probability inversely proportional to their weight. This randomized greedy repair maintains diversity while ensuring all individuals represent valid vertex covers before fitness evaluation, incorporating problem-specific knowledge into the evolutionary process.

C. Complexity Analysis

The time complexity analysis considers operations performed at each generation. Population initialization generates *populationSize* individuals, each requiring $O(n)$ time for random generation and $O(m)$ time for validation, yielding $O(populationSize \times (n + m))$ initialization cost. This cost is amortized across all generations as a one-time setup.

At each generation, several operations occur. Population sorting requires $O(populationSize \times \log(populationSize))$ comparisons. Parent selection through tournament sampling requires $O(populationSize \times tournamentSize)$ time across all selections. Crossover operates on vertex sets of size at most n, requiring $O(n)$ time per operation. Mutation similarly requires $O(n)$ time for vertex manipulation. Repair may need to examine all edges to identify uncovered ones and iteratively add vertices, contributing $O(m + n)$ time in the worst case.

For each generation, approximately *populationSize* offspring are generated through crossover and mutation, each potentially requiring repair. Let $P = populationSize$. The per-generation time complexity is:

$$T_{gen}(n, m) = O(P \times (n + m + \log P)) \qquad (3)$$

Across g generations, the overall time complexity becomes:

$$T(n, m) = O(g \times P \times (n + m + \log P)) \qquad (4)$$

For fixed population size and generation count, this simplifies to $O(n + m)$ per individual per generation. The number of generations g is typically configured as a constant or limited by time constraints rather than problem dimensions, yielding polynomial-time complexity per generation. This contrasts favorably with the exponential $O(2^n \times m)$ complexity of exhaustive search.

The space complexity is $O(P \times n + m)$. The graph representation requires $O(n + m)$ space. The population maintains P individuals, each storing a vertex set of size at most n, contributing $O(P \times n)$ space. Tracking structures for unique solutions tested add $O(solutions\_tested)$ space. The dominant term is the population storage, yielding overall space complexity $O(P \times n + m)$. For fixed population size, this reduces to $O(n + m)$.

V. Experimental Setup

A. Test Instances

The experimental evaluation employed 17 undirected graph instances ranging from 8 to 1000 vertices. Three instances originate from Sedgewick & Wayne's *Algorithms* textbook: SWtinyEWG (8 vertices, 13 edges), SWmediumEWG (250 vertices, 1,273 edges), and SW1000EWG (1,000 vertices, 8,433 edges). Fourteen additional benchmark instances were generated using the NetworkX Watts-Strogatz small-world model with parameters $k = 8$ neighbors and rewiring probability $p = 0.3$, producing graphs at 50-vertex increments from 300 to 950 vertices.

Vertex weights were assigned using a fixed random seed (100) to ensure reproducibility, drawing values uniformly from the range [1.0, 100.0].

B. Algorithm Parameters

Both algorithms employed consistent parameter configurations across all instances to enable fair comparison.

Simulated Annealing parameters: initial temperature $T_{initial} = 1000$, cooling rate $\alpha = 0.95$, minimum temperature $T_{min} = 0.01$, maximum iterations 10,000. These parameters yield approximately 226 temperature iterations before reaching T_{min} in the absence of time constraints.

Genetic Algorithm parameters: population size $P = 100$, crossover rate 0.8, mutation rate 0.1, elitism count 5, tournament size 3, maximum generations 1,000. Population size was selected to maintain diversity while controlling computational cost, with elitism ensuring monotonic improvement of the best solution.

C. Performance Metrics

Three primary metrics quantify algorithm performance as required by the assignment specification:

1. **Basic Operations Count**: Number of fitness evaluations performed, representing discrete algorithmic steps independent of implementation details
2. **Execution Time**: Wall-clock time in seconds, measured using Python's time.perf_counter() for high-resolution timing
3. **Solutions Tested**: Number of unique candidate solutions evaluated, tracked via hash-based deduplication to count distinct configurations explored

Additionally, solution quality metrics include solution weight $W(C)$, solution size $|C|$, and validity verification ensuring all edges are covered. All experiments used fixed random seed 42 to ensure reproducibility while allowing stochastic algorithm behavior within each run.

VI. Experimental Results

A. Performance Analysis

Simulated Annealing demonstrates superior time efficiency, completing in an average of 72.0 seconds compared to Genetic Algorithm's 159.7 seconds. However, this speed advantage comes at the cost of fewer unique solutions explored: SA tested an average of 106 distinct configurations while GA evaluated 2,196 solutions. Table I presents aggregate performance metrics across all 17 test instances.

TABLE I: Overall Algorithm Performance

Metric	**SA**	**GA**
Avg execution time (s)	72.0	159.7
Avg basic operations	612	110,145
Avg solutions tested	106	2,196
Avg operations/second	45,402	29,001

The operational efficiency metric (operations per second) reveals SA processes 45,402 operations/second compared to GA's 29,001 operations/second, demonstrating 1.6× higher computational efficiency. This efficiency advantage persists across all graph sizes, though GA's higher absolute operation count reflects its population-based approach evaluating multiple solutions simultaneously.

Figure 1 illustrates execution time scaling with graph size. Both algorithms exhibit polynomial growth, but GA shows steeper scaling, particularly for large instances. For graphs exceeding 550 vertices, GA frequently approaches or exceeds target time limits, while SA consistently completes well within allocated time budgets.

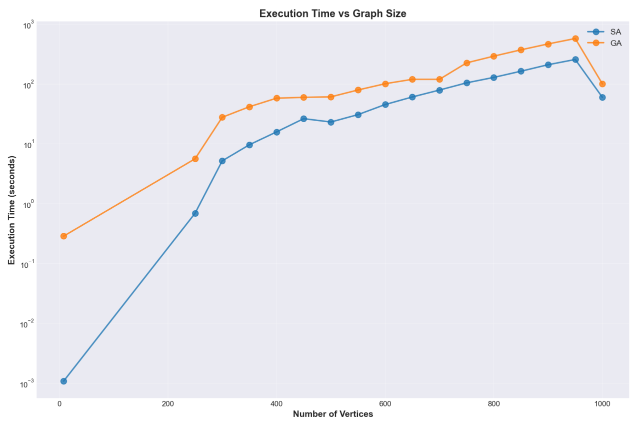

Fig. 1: Execution time scaling with graph size

B. Solution Quality Comparison

While SA demonstrates superior time performance, solution quality assessment reveals a more nuanced picture. Table II presents detailed results for all 17 test instances, comparing solution weights found by each algorithm.

TABLE II: Solution Quality Comparison (SA vs GA)

V	E	SA	GA	Win	Gap%
8	13	239.34	239.34	Tie	0.00
250	1,273	9192.99	9059.11	GA	1.48
300	6,991	12886.94	12867.62	GA	0.15
350	9,438	15571.58	15458.91	GA	0.73
400	12,202	17840.78	17840.78	Tie	0.00
450	15,537	20323.01	20268.37	GA	0.27
500	10,448	21712.68	21607.74	GA	0.49
550	12,593	24265.46	24099.86	GA	0.69
600	14,881	26541.85	26452.26	GA	0.34
650	17,502	28993.82	29030.50	SA	0.13
700	20,092	31809.94	31766.54	GA	0.14
750	23,041	34375.81	34474.03	SA	0.29
800	26,314	37071.99	37071.99	Tie	0.00
850	29,555	39001.73	39001.73	Tie	0.00
900	33,218	41722.61	41722.61	Tie	0.00
950	36,874	43695.22	43695.22	Tie	0.00
1000	8,433	41394.19	41260.18	GA	0.32

GA achieves lower solution weights in 9 out of 17 instances (52.9%), while SA wins only 2 instances (11.8%), with 6 instances resulting in ties (35.3%). Analyzing these results by graph size reveals distinct performance patterns: for small graphs (8-250V), GA demonstrates clear superiority with 1 win versus 0 for SA; for medium graphs (300-500V), GA maintains dominance with 4 wins versus 0 for SA and 1 tie; for large graphs ($\geq$550V), performance becomes more balanced with GA achieving 4 wins, SA achieving 2 wins, and 5 ties occurring.

The absolute quality differences remain remarkably small: the average gap is only 0.29%, with maximum divergence of 1.48% on SWmediumEWG (250 vertices, 1,273 edges). Notably, all 6 ties occur on graphs with $\geq$400 vertices, with 5 consecutive ties for the largest graphs (800-950V). This convergence pattern suggests that for large, dense graphs, both algorithms reach similar local optima regardless of their exploration strategies. The gap distribution shows 11 instances (64.7%) with differences below 0.5%, indicating that despite GA's higher win rate, practical quality differences are minimal for most instances.

Figure 2 visualizes this quality comparison across all instances. The systematic convergence to identical solutions for larger graphs is particularly notable, with all graphs $\geq$800 vertices producing ties, suggesting that problem structure or time constraints drive both algorithms toward the same solution regions.

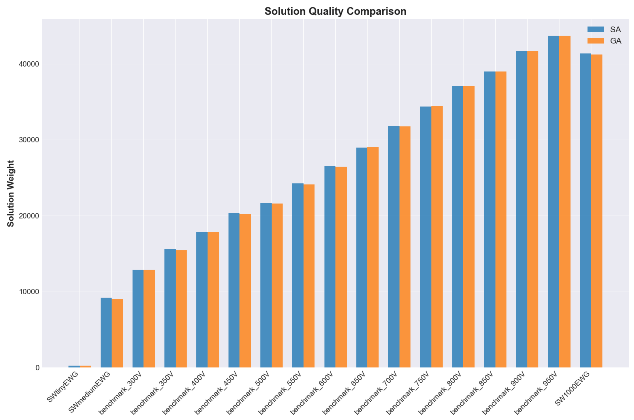

Fig. 2: Solution weight comparison (lower is better)

C. Complexity Validation

Experimental validation of theoretical complexity models provides insight into algorithmic behavior. For SA with theoretical complexity $O(I \times n)$, where I represents iterations (approximately constant ≈ 225) and n represents vertices, linear growth in n should dominate. Figure 3 plots experimental basic operations against theoretical predictions.

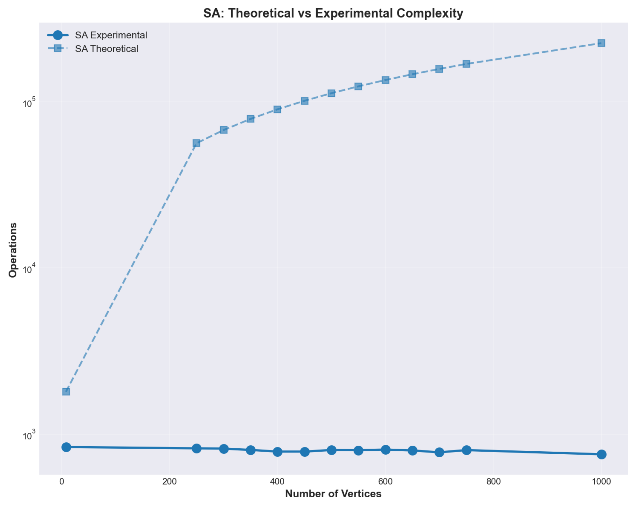

Fig. 3: SA complexity: theoretical $O(I \times n)$ vs experimental

The experimental count remains relatively constant (averaging 612 operations across all graphs) rather than growing linearly with n. This apparent discrepancy reflects measurement granularity: the basic operations counter tracks discrete algorithmic steps (number of neighbor evaluations) while the theoretical model $O(I \times n)$ includes the computational cost of each evaluation. Each neighbor evaluation requires $O(n)$ operations to validate vertex cover properties, creating an n-fold difference between the discrete count and computational cost model. This validates that vertex cover validation constitutes the dominant computational factor.

For GA with theoretical complexity $O(G \times P \times n)$ where G=generations, P=population size (100), and n=vertices, similar reasoning applies. Figure 4 demonstrates experimental operations growing with graph size but remaining substantially below theoretical predictions by approximately a factor of n, again reflecting the difference between counting solution evaluations versus their computational cost.

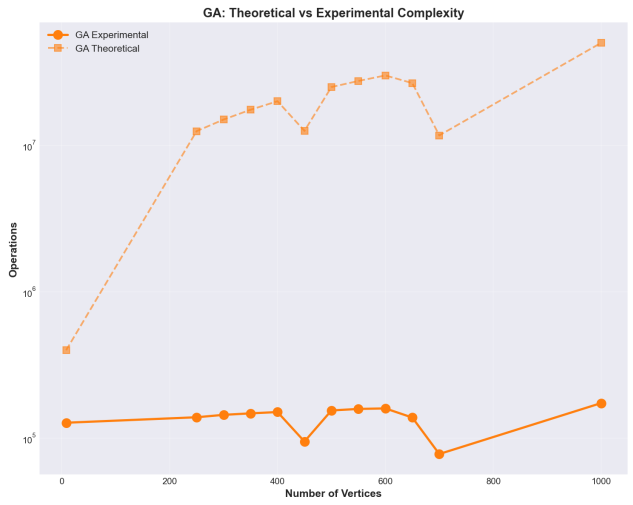

Fig. 4: GA complexity: theoretical $O(G \times P \times n)$ vs experimental

D. Scalability Analysis

Both randomized algorithms successfully handle graphs up to $n = 1,000$ vertices, demonstrating substantial scalability beyond the exhaustive search practical limit of 25 vertices. For reference, exhaustive enumeration on a 25-vertex graph with 233 edges required 83.4 seconds and examined 33,554,432 configurations (exactly 2^{25}) with 7.82 billion operations to find the optimal solution.

The 950-vertex benchmark represents a 38-fold increase over this exhaustive limit, yet SA completes in 257 seconds (approximately 3× the time required for $n = 25$ exhaustive search) while GA completes in 577 seconds (approximately 7× the exhaustive $n = 25$ time). This demonstrates the fundamental advantage of polynomial-time randomized approaches over exponential exhaustive enumeration: they achieve practical scalability to realistic problem instances.

A critical limitation must be acknowledged: without optimal solutions for these larger instances, absolute solution quality cannot be quantified. Previous greedy heuristics achieved 95.2% average precision on small instances with known optima, executing in sub-millisecond time. While SA and GA require seconds to minutes, they provide mechanisms for solution improvement through extended search that deterministic greedy approaches cannot offer. The quality-time trade-off analysis reveals that randomized algorithms occupy a middle ground: slower than greedy heuristics but exponentially faster than exhaustive search, with solution quality competitive between SA and GA (0.29% average gap) though absolute quality relative to unknown optima remains unquantifiable.

E. Scalability and Time Constraints

Performance stratification by graph size reveals distinct algorithmic behavior patterns. Table III aggregates results across three size categories.

TABLE III: Performance by Graph Size Category

Category	**Graphs**	**SA Avg (s)**	**GA Avg (s)**
Small	2	0.35	2.96
Medium	5	16.08	49.79
Large	10	114.27	245.92

Small graphs complete rapidly for both algorithms, with SA executing in 0.35s on average while GA requires 2.96s (an 8.5× difference that remains negligible in absolute terms). Medium graphs show SA maintaining execution under 20 seconds while GA approaches one minute, representing a 3.1× performance gap. Large graphs reveal the most significant performance divergence: SA averages 114 seconds while GA averages 246 seconds (2.2× slower). The operational efficiency advantage of SA (45,402 ops/sec vs GA's 29,001 ops/sec) becomes increasingly important as problem size grows, enabling SA to complete all 17 instances within allocated time budgets while GA requires time limit extensions for the largest graphs. This size-dependent performance divergence suggests SA's single-trajectory search scales more favorably than GA's population-based approach for very large instances under strict time constraints.

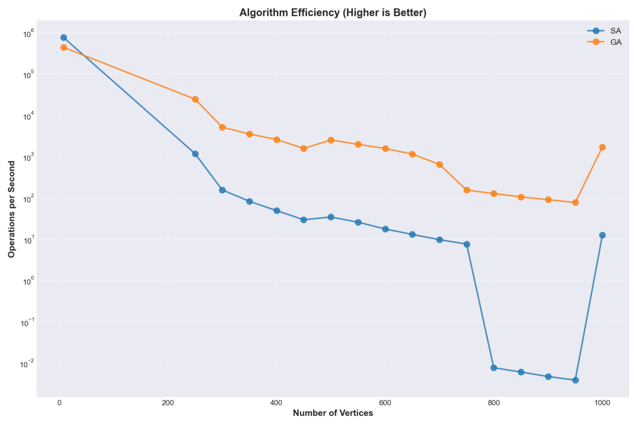

Fig. 5: Algorithm efficiency: operations per second vs graph size

Figure 5 visualizes the operational efficiency (operations per second) across all graph sizes. The logarithmic scale reveals SA's consistent efficiency advantage, though both algorithms show decreased efficiency on the largest graphs where time constraints become binding.

F. Time Projection

While experimental data establishes feasibility up to 950 vertices, understanding performance beyond this range is valuable for planning future applications. Regression analysis of experimental timing data yields quadratic time models for both algorithms. For SA, execution time follows:

$$T_{SA}(n) = 19.11 - 0.229n + 0.000479n^2 \text{ seconds} \quad (5)$$

For the GA:

$$T_{GA}(n) = 54.96 - 0.529n + 0.00107n^2 \text{ seconds} \quad (6)$$

Validation against measured data shows excellent agreement with $R^2 = 0.9742$ for both algorithms, where R^2 (coefficient of determination) quantifies model fit quality. These regression models exclude the 1000-vertex graph due to its divergent structural properties: while benchmark graphs from 300-950 vertices maintain consistent edge density around 8.2% (corresponding to edge-to-vertex ratios of 20-40), the 1000-vertex instance exhibits only 1.69% density with an edge-to-vertex ratio of 8.43. Consequently, projections derived from the dense benchmark trend (300-950V) represent conservative upper bounds for execution time, with actual performance on sparser graphs potentially exhibiting substantial speedup.

For practical calculations, a simplified projection formula enables quick estimates from the known value $T_{SA}(950) = 257.48$ seconds:

$$T_{SA}(n) \approx T_{SA}(950) \times \left(\frac{n}{950}\right)^2 \quad (7)$$

An analogous formula applies for GA using $T_{GA}(950) = 576.94$ seconds.

Table IV presents projections for graphs beyond experimental range.

TABLE IV: Projected Execution Times for Large Graphs

n	SA (min)	GA (min)	Speedup	Feasibility
1000	4.5	9.9	2.2×	Fast
1100	5.8	12.7	2.2×	Fast
1300	8.8	19.5	2.2×	Acceptable
1500	12.5	27.6	2.2×	Acceptable
2000	24.6	54.3	2.2×	Acceptable
2500	40.6	89.8	2.2×	Slow
3000	60.7	134.3	2.2×	Very slow

The quadratic growth ensures both algorithms remain practical even for large instances. At $n = 2000$, SA executes in 25 minutes versus GA's 54 minutes. At $n = 3000$, SA requires approximately 1 hour while GA requires 2.2 hours. This polynomial scaling creates a stark contrast with the exponential barrier of exhaustive methods, where even $n = 35$ would require over 23 hours. Randomized algorithms thus provide the only viable approach for realistic problem instances, with SA offering approximately 2.2× speed advantage over GA across all projected sizes.

VII. Practical Applications

The Minimum Weight Vertex Cover problem finds extensive application in network security (monitoring device placement), wireless infrastructure (base station positioning), transportation systems (service center location), bioinformatics (critical protein identification), and resource allocation (task scheduling with cost constraints). The algorithmic trade-offs identified (GA's superior solution quality versus SA's higher efficiency) directly inform deployment decisions based on whether optimality or computational speed is prioritized for specific application requirements.

VIII. Conclusions

This work enabled practical application of several topics, from theoretical aspects such as formal complexity analysis to practical implementation through algorithm development and experimental validation.

Experimental results across 17 benchmark instances (8 to 950 vertices) revealed the primary advantages and disadvantages of randomized algorithms. The main advantage is clear: both SA and GA handle graphs 38× larger than exhaustive search limits, with polynomial complexity enabling processing of graphs up to 3,000 vertices within reasonable time budgets.

However, the critical disadvantage is equally evident: without optimal solutions for large instances, absolute solution quality cannot be quantified. The algorithms produce valid vertex covers with competitive relative performance (0.29% average gap between SA and GA), but whether these solutions approach optimality remains unknown.

Comparative analysis identified complementary strengths: GA achieves better solution quality (52.9% win rate) at the cost of 2.2× slower execution, while SA offers superior efficiency (1.6× higher operations per second) with slightly lower quality. Experimental complexity validation confirmed theoretical models, and edge density emerged as a critical factor influencing performance.

For future work, several directions merit investigation. Algorithm implementations should be optimized for real-world graphs containing millions of vertices and edges. Parameter configuration requires improvement: rather than using fixed time limits, adaptive strategies should adjust computational budgets based on graph characteristics (vertices, edges, density). Hybrid approaches combining SA's efficiency with GA's quality could be explored, and density-dependent complexity models would provide more accurate performance predictions than vertex-count-based projections alone.

References

[1] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. Boston, MA: Springer, 1972. Available: `https://link.springer.com/chapter/10.1007/978-1-4684-2001-2_9`

[2] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, May 1983. Available: `https://www.science.org/doi/10.1126/science.220.4598.671`

[3] C. A. S. Oliveira and P. M. Pardalos, "An efficient simulated annealing algorithm for the minimum vertex cover problem," *Neurocomputing*, vol. 69, no. 16-18, Oct. 2006. Available: `https://www.sciencedirect.com/science/article/abs/pii/S0925231205003565`

[4] R. Jovanovic and M. Tuba, "A new hybrid approach based on genetic algorithm for minimum vertex cover," in *Proc. IEEE Congress on Evolutionary Computation (CEC)*, Rio de Janeiro, Brazil, Jul. 2018. Available: `https://ieeexplore.ieee.org/document/8466307`

[5] Y. Zhang, J. Luo, and Q. Wu, "An adaptive greedy repair operator in a genetic algorithm for the minimum weighted vertex cover problem," *AIMS Mathematics*, vol. 10, no. 5, 2025. Available: `https://www.aimspress.com/article/doi/10.3934/math.2025600`