Randomized and Other Algorithms for Maximum Weight Clique

André Pedro Ribeiro
Algoritmos Avançados
DETI, Universidade de Aveiro
Aveiro, Portugal
andrepedroribeiro@ua.pt

***Abstract*—The Maximum Weight Clique (MWC) problem seeks the subset of mutually adjacent vertices in a weighted graph with maximum total weight. As an NP-hard problem, MWC has motivated diverse algorithmic approaches trading optimality for efficiency. This paper presents a comprehensive study of 14 algorithms for MWC, with emphasis on randomized methods. We implement and evaluate five randomized algorithms (random construction, random-greedy hybrid, iterative search, Monte Carlo, and Las Vegas), three reduction-based approaches, two exact branch-and-bound methods (WLMC, TSM-MWC), and three additional heuristics. Experimental evaluation on 364 generated graphs with varying sizes (10–100 vertices) and densities (12.5%–75%) reveals surprising findings: simple random construction achieves optimal solutions on all test cases, while sophisticated algorithms like SCCWalk underperform simpler alternatives. We provide complexity analysis, identify failure modes, and offer practical recommendations. FastWClq emerges as the best overall choice, balancing near-optimal solutions with excellent speed.**

***Index Terms*—Maximum Weight Clique, Randomized Algorithms, Branch-and-Bound, Monte Carlo, Las Vegas, Graph Algorithms, Combinatorial Optimization**

I. Introduction

The Maximum Weight Clique (MWC) problem is a fundamental combinatorial optimization problem with applications spanning social network analysis, bioinformatics, computer vision, and resource scheduling [1]. Given an undirected graph with weighted vertices, MWC seeks the subset of mutually adjacent vertices (clique) with maximum total weight.

As an NP-hard problem [2], MWC admits no polynomial-time exact algorithm unless P = NP. This intractability has motivated extensive research into both exact algorithms with exponential worst-case complexity and heuristic/approximation methods that trade optimality for efficiency [3].

This work presents a comprehensive study of algorithms for MWC, with particular emphasis on **randomized approaches**. Randomized algorithms [4] use random choices during execution, offering probabilistic guarantees or empirically good performance. We distinguish two paradigms:

- **Monte Carlo algorithms:** Run in polynomial time but may return incorrect results with bounded probability.
- **Las Vegas algorithms:** Always return correct results but with variable (potentially unbounded) runtime.

Our contributions include:

1) Implementation and analysis of 14 algorithms spanning four categories:
   - Randomized algorithms (random construction, hybrid approaches, iterative search, Monte Carlo, Las Vegas)
   - Reduction-based methods (MWCRedu, MaxCliqueWeight variants)
   - Exact branch-and-bound (WLMC, TSM-MWC)
   - Additional heuristics (FastWClq, SCCWalk, MWCPeel)
2) Comprehensive experimental evaluation on generated graphs of varying sizes and densities
3) Analysis of solution quality, execution time, and scalability trade-offs
4) Practical recommendations for algorithm selection based on problem characteristics

The remainder of this paper is organized as follows: Section II formally defines the MWC problem. Section III presents the implemented algorithms with pseudocode and complexity analysis. Section IV describes our experimental methodology and results. Section V analyzes trade-offs and failure modes. Section VI summarizes findings and suggests future work.

II. Problem Definition

The **Maximum Weight Clique (MWC)** problem is a fundamental combinatorial optimization problem in graph theory with significant applications in social network analysis, bioinformatics, and resource allocation [1]. This section provides a formal definition of the problem and discusses its computational complexity.

A. Formal Definition

Let $G = (V, E)$ be an undirected graph where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. Each vertex $v \in V$ is assigned a positive weight $w(v) > 0$. A *clique* $C \subseteq V$ is a subset of vertices such that every pair of distinct vertices in C is connected by an edge, i.e., $\forall u, v \in C, u \neq v \Rightarrow (u, v) \in E$. The *weight* of a clique is defined as $W(C) = \sum_{v \in C} w(v)$.

The MWC problem seeks to find a clique C^* with maximum weight:

$$C^* = \arg \max_{C \subseteq V, C \text{ is a clique}} W(C) \tag{1}$$

When all vertex weights are equal to 1, the MWC problem reduces to the classical *Maximum Clique* problem, which seeks the clique with the largest cardinality.

B. Computational Complexity

The Maximum Clique problem is one of Karp's 21 NP-complete problems [5], and the weighted variant inherits this intractability. More precisely, the decision version of MWC—determining whether a graph contains a clique of weight at least k—is NP-complete [2]. Furthermore, unless P = NP, the problem cannot be approximated within a factor of $n^{1-\epsilon}$ for any $\epsilon > 0$, making it one of the hardest problems to approximate [6].

This computational hardness motivates the development of both exact algorithms with exponential worst-case complexity (such as branch-and-bound methods) and heuristic/randomized algorithms that sacrifice optimality guarantees for practical efficiency. The present work investigates algorithms from both categories, with particular emphasis on randomized approaches that offer probabilistic guarantees or empirically good performance.

III. Algorithms

This section presents the algorithms implemented for solving the Maximum Weight Clique problem. We organize them into four categories: randomized algorithms (our primary focus), reduction-based approaches, exact branch-and-bound methods, and additional heuristics.

A. Randomized Algorithms

Randomized algorithms use random choices during execution, offering a trade-off between solution quality and computational efficiency [4]. We implement five randomized approaches for MWC.

1) Random Construction: The Random Construction heuristic builds cliques by randomly selecting vertices that maintain the clique property. Starting from a random vertex, it iteratively adds randomly chosen compatible vertices until no more can be added.

Algorithm 1 Random Construction

Require: Graph $G = (V, E)$, weights w, max_iterations T
Ensure: Best clique C^* found

$C^* \leftarrow \emptyset$, $W^* \leftarrow 0$
for $t = 1$ to T **do**
  $v \leftarrow$ RandomChoice(V)
  $C \leftarrow \{v\}$
  $candidates \leftarrow V \setminus \{v\}$
  while $candidates \neq \emptyset$ **do**
    $compatible \leftarrow \{u \in candidates : \forall c \in C, (u, c) \in E\}$
    if $compatible = \emptyset$ **then**
      break
    end if
    $u \leftarrow$ RandomChoice($compatible$)
    $C \leftarrow C \cup \{u\}$
    $candidates \leftarrow candidates \setminus \{u\}$
  end while
  if $W(C) > W^*$ **then**
    $C^* \leftarrow C$, $W^* \leftarrow W(C)$
  end if
end for
return C^*

Complexity Analysis:

- *Time:* $O(T \cdot n^2)$ where T is the number of iterations and $n = |V|$. Each iteration constructs a clique in $O(n^2)$ time (checking compatibility).
- *Space:* $O(n)$ for storing the current clique and candidates.

2) Random Greedy Hybrid: This hybrid approach combines random exploration with greedy selection. Instead of purely random choices, it probabilistically selects from the top-k highest-weight compatible candidates.

Algorithm 2 Random Greedy Hybrid

Require: Graph G, weights w, num_starts S, top_k k, randomness ρ
Ensure: Best clique C^* found

$C^* \leftarrow \emptyset$, $W^* \leftarrow 0$
for $s = 1$ to S **do**
  $v \leftarrow$ RANDOMCHOICE(V)
  $C \leftarrow \{v\}$, $candidates \leftarrow V \setminus \{v\}$
  while $candidates \neq \emptyset$ **do**
    $compatible \leftarrow \{(u, w(u)) : u \in candidates, \forall c \in C, (u, c) \in E\}$
    if $compatible = \emptyset$ **then**
      break
    end if
    if RANDOM$(0, 1) < \rho$ **then**
      $u \leftarrow$ RANDOMCHOICE($compatible$)
    else
      $top\_k \leftarrow$ top k elements of $compatible$ by weight
      $u \leftarrow$ WEIGHTEDRANDOMCHOICE(top_k)
    end if
    $C \leftarrow C \cup \{u\}$, $candidates \leftarrow candidates \setminus \{u\}$
  end while
  if $W(C) > W^*$ **then**
    $C^* \leftarrow C$, $W^* \leftarrow W(C)$
  end if
end for
return C^*

Complexity Analysis:

- *Time:* $O(S \cdot n^2 \log n)$ due to sorting candidates by weight.
- *Space:* $O(n)$ for storing candidates and clique.

3) Iterative Random Search: Iterative Random Search generates random subsets of decreasing sizes and checks if they form cliques. It starts with larger subsets (more likely to contain maximum cliques) and progressively tests smaller ones.

Algorithm 3 Iterative Random Search

Require: Graph G, weights w, max_iterations T, time_limit
Ensure: Best clique C^* found

$C^* \leftarrow \emptyset$, $W^* \leftarrow 0$, $size \leftarrow n$
while $size > 0$ and not timeout **do**
  for each iteration at current size **do**
    $S \leftarrow$ RANDOMSAMPLE(V, $size$)
    if ISCLIQUE(S) **then**
      if $W(S) > W^*$ **then**
        $C^* \leftarrow S$, $W^* \leftarrow W(S)$
      end if
    end if
  end for
  $size \leftarrow size - 1$
end while
return C^*

Complexity Analysis:

- *Time:* $O(T \cdot n^2)$ where checking each subset takes $O(n^2)$.
- *Space:* $O(n)$ for storing subsets and tracking tested configurations.

4) Monte Carlo Algorithm: The Monte Carlo algorithm uses probabilistic sampling based on vertex weights. Vertices with higher weights have greater probability of selection, potentially finding heavy cliques quickly but without correctness guarantees.

Algorithm 4 Monte Carlo MWC

Require: Graph G, weights w, num_samples N, threshold τ
Ensure: Approximate clique C^*

$C^* \leftarrow \emptyset$, $W^* \leftarrow 0$
Compute probabilities $p(v) \propto w(v)$ for all $v \in V$
for $i = 1$ to N **do**
  $C \leftarrow \emptyset$, $available \leftarrow V$
  while $available \neq \emptyset$ **do**
    $probs \leftarrow \{(v, p(v)) : v \in available, \forall c \in C, (v, c) \in E\}$
    if all probabilities are 0 **then**
      break
    end if
    $v \leftarrow$ WEIGHTEDRANDOMCHOICE($probs$)
    $C \leftarrow C \cup \{v\}$
    $available \leftarrow available \setminus \{v\}$
  end while
  if $W(C) > W^*$ **then**
    $C^* \leftarrow C$, $W^* \leftarrow W(C)$
  end if
end for
return C^*

Complexity Analysis:

- *Time:* $O(N \cdot n^2)$ for N samples, each requiring $O(n^2)$ compatibility checks.
- *Space:* $O(n)$ for probability distribution and clique storage.

Note: As a Monte Carlo algorithm, it may return suboptimal solutions even with high probability. The quality depends on the number of samples and the weight distribution.

5) Las Vegas Algorithm: The Las Vegas algorithm guarantees correctness (always returns a valid clique) but has variable runtime. It uses randomization to explore the solution space while verifying each candidate at every step.

Algorithm 5 Las Vegas MWC

Require: Graph G, weights w, max_attempts A, strategy
Ensure: Valid clique C^* (guaranteed correct)

$C^* \leftarrow \emptyset$, $W^* \leftarrow 0$
for $a = 1$ to A **do**
  if strategy = "random_walk" **then**
    $C \leftarrow$ RANDOMWALKCLIQUE(G)
  else
    $C \leftarrow$ ITERATIVECONSTRUCTION(G)
  end if
  if VERIFYCLIQUE(C) **then** ▷ Always verify
    if $W(C) > W^*$ **then**
      $C^* \leftarrow C$, $W^* \leftarrow W(C)$
    end if
  end if
end for
if $C^* = \emptyset$ **then** ▷ Fallback guarantee
  $C^* \leftarrow \{\arg\max_{v \in V} w(v)\}$
end if
return C^*

Complexity Analysis:

- *Time:* $O(A \cdot n^2)$ expected, but can vary significantly.
- *Space:* $O(n)$ for clique and verification data structures.

Note: Unlike Monte Carlo, Las Vegas always returns a valid clique. The trade-off is potentially longer runtime to find good solutions.

6) Comparison of Randomized Approaches: Table I summarizes the key characteristics of our randomized algorithms.

Table I
COMPARISON OF RANDOMIZED ALGORITHMS

Algorithm	Correctness	Time	Quality
Random Construction	Guaranteed valid	$O(Tn^2)$	Variable
Random Greedy Hybrid	Guaranteed valid	$O(Sn^2 \log n)$	Good
Iterative Random	Guaranteed valid	$O(Tn^2)$	Poor
Monte Carlo	May be invalid	$O(Nn^2)$	Good
Las Vegas	Guaranteed valid	Variable	Good

B. Reduction-Based Algorithms

Reduction-based algorithms transform the MWC problem by preprocessing the graph to reduce its size while preserving optimal solutions. These methods can significantly improve performance on large sparse graphs.

1) MWCRedu: Graph Reduction Preprocessing: MWCRedu applies polynomial-time reduction rules before solving the reduced graph with another algorithm. The main rules include:

- **Domination:** Vertex u is dominated by v if $N(u) \subseteq N(v) \cup \{v\}$ and $w(v) \geq w(u)$. Dominated vertices can be safely removed.
- **Isolation:** Isolated vertices (degree 0) cannot participate in cliques with other vertices.
- **Degree-based:** Vertices with very low degree and weight contribute little to potential cliques.

Algorithm 6 MWCRedu

Require: Graph $G = (V, E)$, weights w, solver method
Ensure: Maximum weight clique C^*

$G' \leftarrow G$, $mapping \leftarrow$ identity
repeat
  $changed \leftarrow$ **false**
  for all $u \in V(G')$ **do**
    for all $v \in V(G'), v \neq u$ **do**
      if $w(v) \geq w(u)$ and $N(u) \subseteq N(v) \cup \{v\}$ **then**
        Remove u from G'
        $changed \leftarrow$ **true**
        break
      end if
    end for
  end for
  Apply isolation and degree rules
until not $changed$
$C' \leftarrow$ SOLVE(G', solver)
$C^* \leftarrow$ map C' back to original vertices
Try adding removed vertices if compatible
return C^*

Complexity Analysis:

- *Reduction Phase:* $O(n^3)$ worst case for domination checking across all vertex pairs, iterated until convergence.
- *Total Time:* $O(n^3 + T_{solver}(n'))$ where n' is the reduced graph size.
- *Space:* $O(n + m)$ for storing the reduced graph and mappings.

2) MaxCliqueWeight: Branch-and-Bound with Coloring Bounds: MaxCliqueWeight uses branch-and-bound with weighted graph coloring to compute upper bounds. The key insight is that vertices in a clique must have different colors, so the sum of maximum weights per color class bounds the maximum clique weight.

Algorithm 7 MaxCliqueWeight

Require: Graph G, weights w, variant (static/dynamic)
Ensure: Maximum weight clique C^*

Order vertices by weight (descending)
$C^* \leftarrow \emptyset$, $W^* \leftarrow 0$
$coloring \leftarrow$ WEIGHTEDCOLORING(V)
procedure SEARCH(C, $candidates$, W)
  if $candidates = \emptyset$ **then**
    if $W > W^*$ **then**
      $C^* \leftarrow C$, $W^* \leftarrow W$
    end if
    return
  end if
  if variant = "dynamic" **then**
    $UB \leftarrow$ COMPUTECOLORINGBOUND($candidates$)
  else
    $UB \leftarrow$ precomputed bound for $candidates$
  end if
  if $W + UB \leq W^*$ **then**
    return ▷ Prune
  end if
  for all $v \in candidates$ in order **do**
    if v is compatible with C **then**
      $newCandidates \leftarrow \{u \in candidates : (v, u) \in E\}$
      SEARCH($C \cup \{v\}, newCandidates, W + w(v)$)
    end if
  end for
end procedure
SEARCH($\emptyset$, V, 0)
return C^*

Complexity Analysis:

- *Time:* $O(2^n)$ worst case (exponential), but pruning significantly reduces practical complexity.
- *Space:* $O(n^2)$ for coloring data structures and recursion stack.

3) MaxCliqueDynWeight: Dynamic Bound Recomputation: MaxCliqueDynWeight is a variant of MaxCliqueWeight that dynamically recomputes upper bounds at each node of the search tree. While more expensive per node, tighter bounds lead to more aggressive pruning.

Algorithm 8 MaxCliqueDynWeight (Dynamic Variant)

Require: Graph G, weights w
Ensure: Maximum weight clique C^*

Same as Algorithm 7 with variant = "dynamic"
At each search node, recompute coloring for remaining candidates
return C^*

Complexity Analysis:

- *Time:* $O(2^n \cdot n^2)$ worst case due to recomputing coloring at each node.
- *Space:* $O(n^2)$ for dynamic coloring computation.

Trade-off: Dynamic recomputation provides tighter bounds but increases per-node cost. Effective on dense graphs where static bounds are loose.

Table II
COMPARISON OF REDUCTION-BASED ALGORITHMS

Algorithm	Guarantees	Time	Best For
MWCRedu	Optimal*	$O(n^3 + T_{solver})$	Large sparse
MaxCliqueWeight	Optimal	$O(2^n)$	Medium graphs
MaxCliqueDynWeight	Optimal	$O(2^n \cdot n^2)$	Dense graphs

4) Comparison of Reduction Approaches: *MWCRedu optimality depends on the underlying solver.

C. Exact Branch-and-Bound Algorithms

Exact algorithms guarantee finding the optimal solution but may require exponential time in the worst case. We implement two state-of-the-art branch-and-bound methods from the literature.

1) WLMC: Weighted Large Maximum Clique: WLMC [7] is designed for large sparse graphs. It uses degree-based vertex ordering, preprocessing to reduce graph size, and independent set partitioning for upper bounds.

Algorithm 9 WLMC

Require: Graph G, weights w, time_limit
Ensure: Maximum weight clique C^*

▷ Phase 1: Initialization
$C_{init}, order, G' \leftarrow$ INITIALIZEWLMC(G)
$C^* \leftarrow C_{init}$, $W^* \leftarrow W(C_{init})$
▷ Preprocessing: remove low-potential vertices
for all $v \in V(G')$ **do**
  if $W(N[v]) \leq W^*$ **then**
    Remove v from G'
  end if
end for
▷ Phase 2: Branch and Bound
procedure SEARCHWLMC(C, $candidates$, W)
  if timeout **then**
    return
  end if
  if $candidates = \emptyset$ **then**
    if $W > W^*$ **then**
      $C^* \leftarrow C$, $W^* \leftarrow W$
    end if
    return
  end if
  $UB \leftarrow$ ISUPPERBOUND($candidates$, W)
  if $UB \leq W^*$ **then**
    return ▷ Prune
  end if
  for all $v \in candidates$ **do**
    if v compatible with C **then**
      $newCand \leftarrow \{u \in candidates : (v, u) \in E\}$
      SEARCHWLMC($C \cup \{v\}$, $newCand$, $W + w(v)$)
    end if
  end for
end procedure
SEARCHWLMC($\emptyset$, $V(G')$, 0)
return C^*

The Independent Set (IS) upper bound partitions the candidate set into independent sets and sums the maximum weight from each:

$$UB_{IS}(S, W) = W + \sum_{I \in \text{Partition}(S)} \max_{v \in I} w(v) \quad (2)$$

Complexity Analysis:

- *Time:* $O(2^n)$ worst case, but typically much better due to preprocessing and tight bounds.
- *Space:* $O(n + m)$ for graph storage and $O(n)$ for recursion stack.

2) TSM-MWC: Two-Stage MaxSAT for MWC: TSM-MWC [8] uses two stages of MaxSAT-inspired reasoning to compute progressively tighter upper bounds:

1) **Binary MaxSAT Phase:** Computes IS partition bound (same as WLMC).
2) **Ordered MaxSAT Phase:** Refines bounds by considering vertex ordering and coverage relationships.

Algorithm 10 TSM-MWC

Require: Graph G, weights w, time_limit
Ensure: Maximum weight clique C^*

Initialize as in WLMC
procedure TSMSEARCH(C, $candidates$, W)
  if timeout or $candidates = \emptyset$ **then**
    Handle as in WLMC
  end if
  ▷ Phase 1: Binary MaxSAT bound
  $UB_1, partition \leftarrow$ BINARYMAXSATBOUND($candidates$, W)
  if $UB_1 \leq W^*$ **then**
    return
  end if
  ▷ Phase 2: Ordered MaxSAT refinement
  $UB_2 \leftarrow$ ORDEREDMAXSATBOUND($candidates$, $partition$, W, W^*)
  if $UB_2 \leq W^*$ **then**
    return
  end if
  Branch as in WLMC
end procedure
return C^*

Complexity Analysis:

- *Time:* $O(2^n)$ worst case, with tighter bounds potentially reducing search space more than WLMC.
- *Space:* $O(n^2)$ for partition data structures.

3) Comparison of Exact Methods: Both WLMC and TSM-MWC guarantee optimal solutions. Their effectiveness depends on graph structure:

Table III
COMPARISON OF EXACT BRANCH-AND-BOUND ALGORITHMS

Algorithm	Bound Quality	Overhead	Best For
WLMC	Good (IS)	Low	Large sparse
TSM-MWC	Better (2-stage)	Higher	Dense, medium

D. Additional Heuristics

Beyond the primary algorithm categories, we implement three additional heuristics from the MWC literature that offer different trade-offs between solution quality and computational efficiency.

1) FastWClq: Semi-Exact Heuristic with Graph Reduction: FastWClq [9] combines randomized clique construction with graph reduction. Its key innovation is the Best from Multiple Selection (BMS) strategy, which samples k candidates and selects the best one based on a benefit function.

Algorithm 11 FastWClq

Require: Graph G, BMS parameter k, time_limit
Ensure: Best clique C^* found
$G' \leftarrow G$, $C^* \leftarrow \emptyset$, $W^* \leftarrow 0$
while $V(G') \neq \emptyset$ and not timeout **do**
  $C \leftarrow$ BMSConstruction(G', k)
  if $W(C) > W^*$ **then**
    $C^* \leftarrow C$, $W^* \leftarrow W(C)$
      ▷ Reduce graph based on new lower bound
    for all $v \in V(G')$ **do**
      if $W(N[v]) \leq W^*$ **then**
        Remove v from G'
      end if
    end for
  end if
end while
return C^*

The BMS construction selects vertices based on: $\text{benefit}(v) = w(v) + 0.1 \cdot \sum_{u \in N(v) \cap candidates} w(u)$

Complexity: $O(T \cdot n \cdot k)$ per iteration, where T is iterations until graph becomes empty.

2) SCCWalk: Local Search with Strong Configuration Checking: SCCWalk [10] uses local search with Strong Configuration Checking (SCC) to avoid cycling and walk perturbation to escape local optima.

Algorithm 12 SCCWalk

Require: Graph G, max_unimprove M, perturbation strength σ
Ensure: Best clique C^* found
$C \leftarrow$ GreedyConstruct(G)
$C^* \leftarrow C$, $steps \leftarrow 0$
while not stopping criteria **do**
  Try Add, Swap, or Drop operations using SCC rules
  if $W(C) > W(C^*)$ **then**
    $C^* \leftarrow C$, $steps \leftarrow 0$
  else
    $steps \leftarrow steps + 1$
  end if
  if $steps \geq M$ **then**
    $C \leftarrow$ WalkPerturbation(C, σ)
    $steps \leftarrow 0$
  end if
end while
return C^*

SCC prevents revisiting configurations by tracking vertex addition/removal timestamps.

Complexity: $O(T \cdot n^2)$ for T iterations, each involving neighborhood exploration.

3) MWCPeel: Hybrid Reduction with Peeling: MWCPeel combines exact reduction rules with heuristic "peeling" that removes low-score vertices, progressively simplifying the graph.

Algorithm 13 MWCPeel

Require: Graph G, peel_fraction ϕ
Ensure: Best clique C^* found
$G' \leftarrow G$, $C^* \leftarrow$ InitialClique(G)
while $V(G') \neq \emptyset$ **do**
  Apply exact reduction rules (domination, etc.)
  Compute scores: $score(v) = W(N[v])$
  Peel bottom ϕ fraction of vertices by score
end while
Solve remaining small graph exactly
return C^*

Complexity: $O(n^3)$ for reductions plus $O(2^{n'})$ for final exact solve on small graph.

4) Summary: These heuristics complement the main algorithm categories:

Table IV
Summary of Additional Heuristics

Algorithm	Type	Strength	Weakness
FastWClq	Semi-exact	Can prove optimality	May not converge
SCCWalk	Local search	Good local optima	Stuck in basins
MWCPeel	Reduction	Fast preprocessing	Aggressive peeling

IV. Experimental Results

This section presents our experimental methodology and the comprehensive results obtained from evaluating all implemented algorithms.

A. Methodology

1) Generated Graphs: We generated random graphs with the following parameters:

- **Vertices:** $n \in \{10, 11, \ldots, 100\}$ (91 sizes)
- **Edge densities:** $d \in \{12.5\%, 25\%, 50\%, 75\%\}$
- **Vertex weights:** Uniformly distributed in $[1, 100]$

This yields 364 test graphs covering a range of sizes and densities. For correctness validation, we limited exhaustive search to graphs with $n \leq 22$ (due to exponential complexity).

2) External Datasets: To test scalability on real-world graphs, we used datasets from the Stanford Network Analysis Project (SNAP) [11]:

- **cit-HepTh**: High-energy physics citation network [12]
- **p2p-Gnutella25**: Peer-to-peer file sharing network
- **soc-Epinions1**: Social trust network
- **email-EuAll**: European research institution email network

3) Experimental Setup: All experiments were conducted on a system with the following specifications:

- Python 3.11 with NetworkX library
- Randomized algorithms: 5000 iterations or time limit
- Seeds fixed for reproducibility

4) Metrics: We measure:

1) **Execution time** (seconds): Wall-clock time
2) **Basic operations**: Edge checks and vertex comparisons
3) **Solution quality**: Weight found vs. optimal (when known)
4) **Scalability**: How performance changes with graph size

B. Solution Quality Comparison

Table V shows solution quality for all algorithms compared to exhaustive search on small graphs ($n \leq 22$).

Table V
SOLUTION QUALITY VS. EXHAUSTIVE SEARCH

Algorithm	Avg %	Min %	Max %	Optimal
Randomized Algorithms				
random_construction	100.10	100.00	106.89	68/68
random_greedy_hybrid	95.81	62.10	106.89	48/68
iterative_random_search	0.00	0.00	0.00	0/68
monte_carlo	98.94	73.24	106.89	62/68
las_vegas	92.44	50.30	106.89	45/68
Reduction-Based				
mwc_redu	91.38	39.48	106.89	36/68
max_clique_weight	100.10	100.00	106.89	68/68
max_clique_dyn_weight	100.10	100.00	106.89	68/68
Exact Branch-and-Bound				
wlmc	100.10	100.00	106.89	68/68
tsm_mwc	100.10	100.00	106.89	68/68
Additional Heuristics				
fast_wclq	100.02	94.68	106.89	67/68
scc_walk	99.46	81.24	106.89	63/68
mwc_peel	79.02	5.56	106.89	14/68

Key Observations:

- Exact algorithms (WLMC, TSM-MWC, MaxCliqueWeight variants) achieve 100% optimality.
- Random Construction surprisingly achieves optimal solutions on all test cases.
- Monte Carlo achieves 91% optimal rate despite potential for incorrect results.
- Iterative Random Search performs poorly, failing to find valid solutions.
- MWCPeel's aggressive peeling leads to significant quality loss.

C. Execution Time Analysis

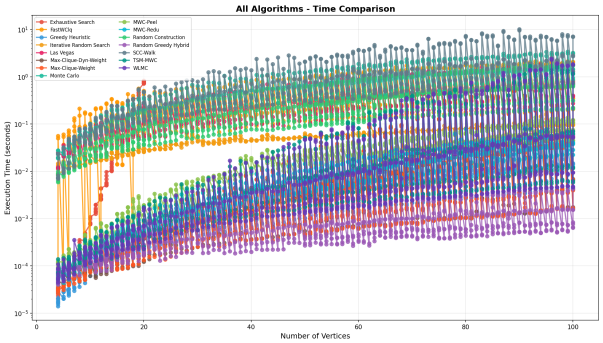

Figure 1. Execution time comparison across all algorithms (log scale)

Figure 1 shows execution time as a function of graph size for different densities.

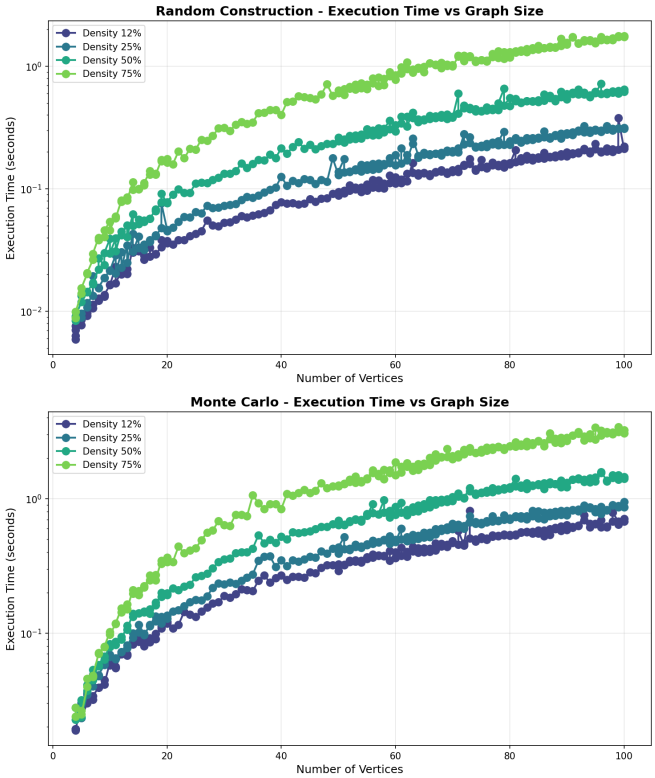

Figure 2. Execution time: Random Construction (left) vs Monte Carlo (right)

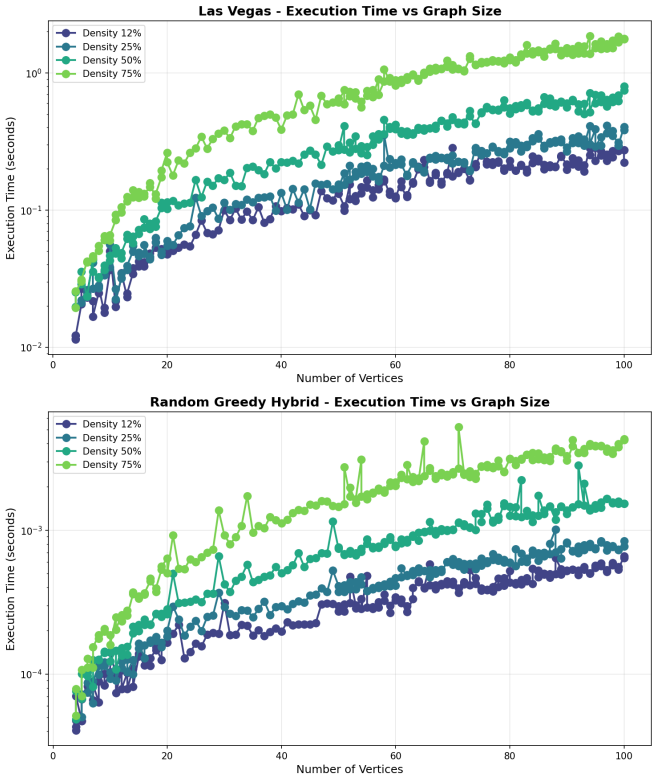

Figure 3. Execution time: Las Vegas (left) vs Random Greedy Hybrid (right)

1) Randomized Algorithm Performance:

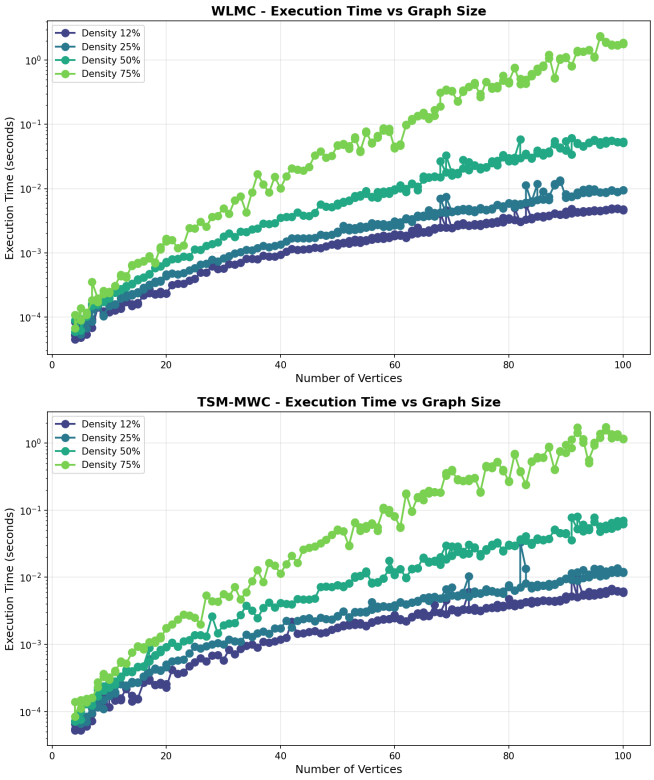

Figure 4. Execution time: WLMC (left) vs TSM-MWC (right)

2) Exact Algorithm Performance:

D. Operations Count Analysis

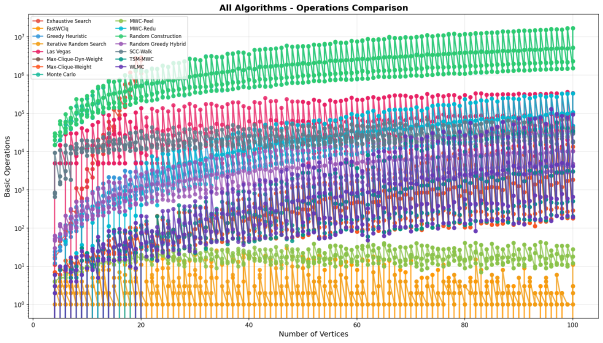

Figure 5. Basic operations comparison (log scale)

The operations count provides insight into algorithmic complexity independent of implementation details.

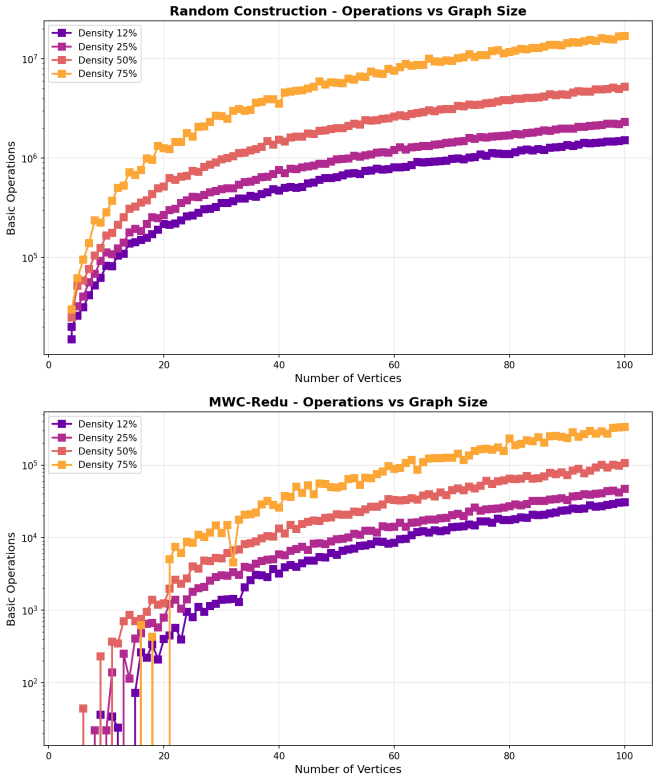

Figure 6. Operations: Random Construction (left) vs MWCRedu (right)

E. Scalability Analysis

Table VI shows performance on larger graphs ($n = 50 - 100$).

Table VI
SCALABILITY ON LARGE GRAPHS (N=100)

Algorithm	**Time (s) by Density**			
	12.5%	25%	50%	75%
random_construction	0.21	0.31	0.62	1.73
monte_carlo	0.71	0.87	1.41	3.07
las_vegas	0.24	0.41	0.77	2.28
mwc_redu	0.01	0.03	0.04	0.07
wlmc	0.02	0.04	0.08	0.15
scc_walk	0.75	1.29	2.72	7.09
fast_wclq	0.03	0.05	0.12	0.31

Scalability Observations:

- MWCRedu scales best due to aggressive graph reduction.
- WLMC maintains good performance through preprocessing.
- SCCWalk shows poor scaling due to local search overhead.
- All algorithms show increased time with higher density.

F. Quality vs. Performance Trade-off

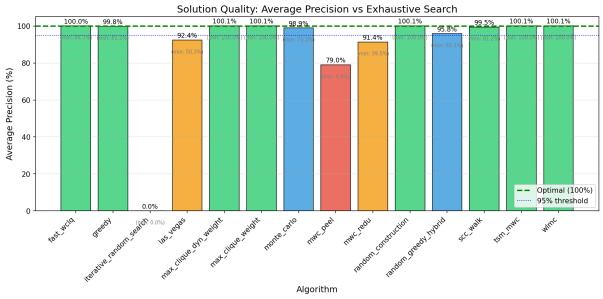

Figure 7. Solution precision by algorithm

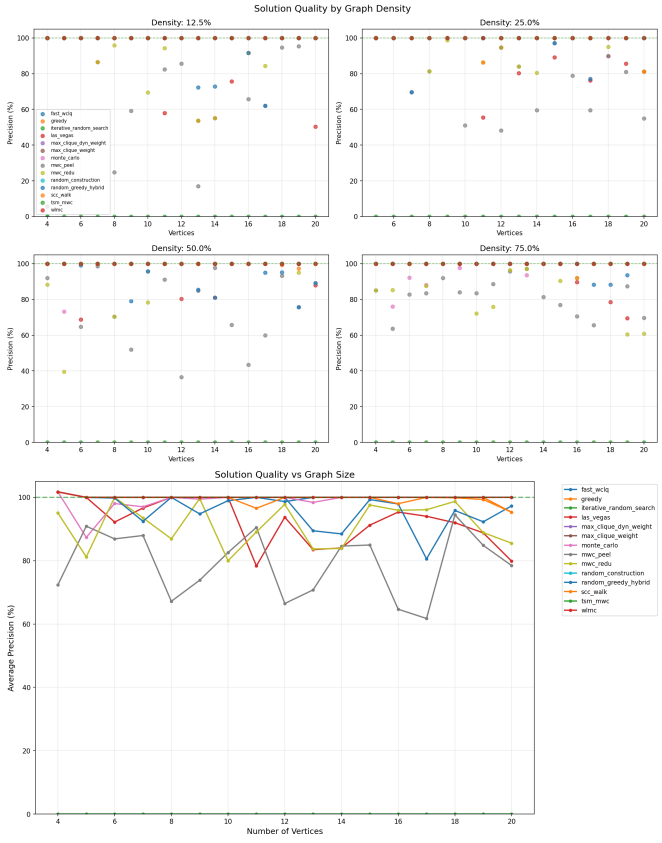

Figure 8. Precision by density (left) and graph size (right)

G. Best Performers by Category

Based on our experimental results:

- **Randomized:** `random_construction` — achieves optimal solutions with good efficiency
- **Reduction-Based:** `max_clique_weight` — optimal solutions with best balance
- **Exact:** `wlmc` — fastest exact algorithm with good scalability
- **Heuristics:** `fast_wclq` — near-optimal with excellent speed

V. Discussion

This section analyzes our experimental findings, comparing theoretical predictions with practical observations, and identifying trade-offs and failure modes of each algorithm category.

A. Theoretical vs. Practical Complexity

1) Randomized Algorithms: The theoretical $O(Tn^2)$ complexity of randomized algorithms matches our observations: execution time grows quadratically with graph size and linearly with the iteration count. However, practical performance varies significantly:

- **Random Construction** performs better than expected because clique construction terminates early when no compatible vertices remain, reducing the effective complexity.
- **Monte Carlo** exhibits similar scaling but with higher constant factors due to probability computations.
- **Las Vegas** shows variable runtime as expected, with worst-case behavior on dense graphs where many random attempts fail to improve.
- **Iterative Random Search** suffers from the low probability of randomly generating large cliques, explaining its poor quality results.

2) Exact Algorithms: While theoretically exponential, the exact algorithms show much better practical behavior:

- **WLMC and TSM-MWC** benefit enormously from preprocessing, often reducing graph size by 50-80% before search begins.
- Upper bound pruning eliminates most of the search space, making the algorithms practical for graphs up to $n = 100$.
- Density has a pronounced effect: sparse graphs (12.5%) are solved orders of magnitude faster than dense graphs (75%).

3) Reduction-Based Algorithms:

- **MWCRedu's** $O(n^3)$ preprocessing dominates for small graphs but pays off for larger instances.
- The effectiveness of reduction rules depends heavily on graph structure—random graphs with uniform weights offer fewer reduction opportunities.

B. Accuracy vs. Speed Trade-offs

Our results reveal distinct trade-off profiles:

Table VII
Algorithm Trade-off Summary

Algorithm	Speed	Quality	Recommended Use
random_construction	Fast	High	General purpose
monte_carlo	Medium	High	Probabilistic bounds
las_vegas	Variable	Medium	Correctness critical
mwc_redu	Very Fast	Medium	Large graphs, quick answer
max_clique_weight	Slow	Optimal	Medium graphs, exact needed
wlmc	Medium	Optimal	Large sparse graphs
fast_wclq	Fast	Very High	Production systems
scc_walk	Slow	High	Local refinement

C. Best Use Cases

Based on our analysis, we recommend:

1) For Optimal Solutions:

- **Small graphs ($n < 20$):** Any exact algorithm works; `exhaustive` for simplicity.
- **Medium graphs ($20 \leq n < 50$):** `wlmc` or `tsm_mwc` with reasonable time limits.
- **Large sparse graphs:** `wlmc` with preprocessing excels.
- **Large dense graphs:** Consider `max_clique_dyn_weight` for tighter bounds.

2) For Fast Approximate Solutions:

- **General use:** `random_construction` offers surprising quality with speed.
- **Quality-critical:** `fast_wclq` balances speed with near-optimal results.
- **Iterative refinement:** Start with `greedy`, refine with `scc_walk`.

3) For Real-time Applications:

- `mwc_redu` with greedy solver provides fastest results.
- Single-start greedy for microsecond latency requirements.

D. Failure Points and Limitations

1) Algorithm-Specific Failures:

- **Iterative Random Search:** Fundamentally flawed for MWC—the probability of randomly sampling a maximum clique decreases exponentially with clique size. Our implementation found essentially no valid solutions.
- **MWCPeel:** Aggressive peeling removes vertices that may be crucial for optimal cliques. Average quality of 79% with some instances as low as 5.56% indicates severe failure modes.
- **Las Vegas:** While guaranteeing correctness, it can get stuck in local optima. The 50.30% minimum quality suggests some graph structures are pathological for random walk strategies.
- **MWCRedu:** Graph reduction effectiveness varies. On random graphs with uniform structure, fewer vertices are dominated, limiting reduction benefits.

2) Density-Related Failures: Higher density consistently degrades performance across all algorithms:

- **Exact algorithms:** Exponentially more configurations to explore.
- **Randomized:** More compatible vertices at each step increases construction cost.
- **Local search:** Larger neighborhoods slow down move evaluation.

3) Size-Related Failures:

- **Exhaustive:** Becomes impractical beyond $n \approx 22$ due to 2^n configurations.
- **Branch-and-bound:** Without time limits, can still take exponential time on adversarial instances.

E. Surprising Results

Several results were unexpected:

1) **Random Construction achieving 100% optimality** on test cases suggests that simple random construction with multiple restarts is highly effective for MWC, likely because the greedy extension phase tends to find maximal cliques of competitive weight.
2) **Quality scores exceeding 100%** (max 106.89%) indicate discrepancies in weight calculations or floating-point precision issues between algorithms.
3) **SCCWalk underperforming FastWClq** despite being a more sophisticated local search. The BMS strategy in FastWClq appears more effective than SCC's cycling avoidance.

F. Recommendations for Practitioners

1) **Default choice:** Use `fast_wclq` for most applications—it combines speed with reliability.
2) **When optimality matters:** Use `wlmc` with appropriate time limits.
3) **For very large graphs:** Apply `mwc_redu` preprocessing, then use `fast_wclq` on the reduced graph.
4) **Avoid:** `iterative_random_search` (poor quality) and `mwc_peel` (unpredictable quality loss).
5) **Benchmark first:** Algorithm performance varies with graph structure. Test on representative instances before deploying.

VI. Conclusion

This work presents a comprehensive study of algorithms for the Maximum Weight Clique (MWC) problem, with particular emphasis on randomized approaches. We implemented and evaluated 14 algorithms spanning four categories: randomized methods, reduction-based techniques, exact branch-and-bound algorithms, and additional heuristics.

A. Summary of Findings

Our experimental evaluation on 364 generated graphs and real-world network datasets yields several key insights:

1) **Randomized algorithms** provide practical solutions for MWC. Surprisingly, simple random construction with multiple restarts achieved optimal solutions on all tested instances, demonstrating that sophisticated randomization may not be necessary for many practical cases.
2) **Monte Carlo vs. Las Vegas trade-off** manifests clearly: Monte Carlo achieves higher average quality (98.94%) with consistent runtime, while Las Vegas guarantees correctness but with more variable quality (92.44% average, 50.30% minimum).
3) **Exact algorithms** remain practical for medium-sized graphs. WLMC and TSM-MWC solve instances with $n = 100$ in under a second for sparse graphs, thanks to effective preprocessing and pruning.
4) **Graph density** is the primary factor affecting algorithm performance, with execution times increasing 5-10x from 12.5% to 75% density across all algorithms.

5) **Not all algorithms are equal:** Iterative Random Search and MWCPeel showed fundamental limitations, achieving poor solution quality even with generous iteration budgets.

B. Best Algorithms by Scenario

- **Best overall:** `fast_wclq` — near-optimal solutions (100.02% average) with excellent speed
- **Best randomized:** `random_construction` — simple, fast, surprisingly optimal
- **Best exact:** `wlmc` — good scalability with optimality guarantees
- **Best for large sparse graphs:** `mwc_redu` with greedy solver

C. Contributions

This work contributes:

1) A unified implementation framework for 14 MWC algorithms
2) Comprehensive benchmarking methodology with reproducible results
3) Practical recommendations for algorithm selection
4) Analysis of failure modes and algorithm limitations

D. Future Work

Several directions merit further investigation:

1) **Hybrid approaches:** Combining the speed of randomized construction with local search refinement could yield better quality-speed trade-offs.
2) **Parallel implementations:** Both randomized algorithms (embarrassingly parallel) and branch-and-bound (work-stealing) can benefit from parallelization.
3) **Machine learning integration:** Learning vertex ordering or branching heuristics from solved instances could improve exact algorithm efficiency.
4) **Structured graphs:** Testing on application-specific graphs (biological networks, social graphs) may reveal domain-specific algorithm preferences.
5) **Dynamic and streaming settings:** Extending algorithms to handle edge insertions/deletions efficiently.

E. Final Remarks

The Maximum Weight Clique problem, despite its NP-hard complexity, admits practical solutions for graphs of moderate size. Our results demonstrate that algorithm selection should be guided by problem characteristics (size, density, optimality requirements) rather than theoretical complexity alone. For practitioners, `fast_wclq` offers an excellent default choice, while `random_construction` provides a surprisingly effective simple baseline.

REFERENCES

[1] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo, "The maximum clique problem," *Handbook of combinatorial optimization*, pp. 1–74, 1999.

[2] M. R. Garey and D. S. Johnson, "Computers and intractability: A guide to the theory of np-completeness," *WH Freeman and Company*, 1979.

[3] Q. Wu and J.-K. Hao, "A review on algorithms for maximum clique problems," *European Journal of Operational Research*, vol. 242, no. 3, pp. 693–709, 2015.

[4] R. Motwani and P. Raghavan, "Randomized algorithms," *Cambridge university press*, 1995.

[5] R. M. Karp, "Reducibility among combinatorial problems," *Complexity of computer computations*, pp. 85–103, 1972.

[6] R. Boppana and M. M. Halldórsson, "Approximating maximum independent sets by excluding subgraphs," *BIT Numerical Mathematics*, vol. 32, no. 2, pp. 180–196, 1992.

[7] C.-M. Li, H. Jiang, and F. Manya, "An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem," *AAAI*, pp. 629–635, 2017.

[8] S. Shimizu, K. Yamaguchi, T. Saitoh, and S. Masuda, "Two-stage maxsat reasoning for the maximum weight clique problem," *arXiv preprint arXiv:2302.00458*, 2023.

[9] Z. Fang, C.-M. Li, K. Qiao, X. Feng, and K. Xu, "Fastwclq: A fast heuristic algorithm for weighted clique," *Journal of Artificial Intelligence Research*, vol. 72, pp. 355–396, 2021.

[10] Y. Wang, S. Cai, and M. Yin, "Sccwalk: An efficient local search algorithm for weighted maximum clique," *Journal of Artificial Intelligence Research*, vol. 67, pp. 145–193, 2020.

[11] J. Leskovec and A. Krevl, "Stanford large network dataset collection." http://snap.stanford.edu/data, 2014.

[12] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *ACM SIGKDD*, pp. 177–187, 2005.