CS 301

2024–2025 Summer

Project Report

Group 7

Group Members:

Eren Ergez — 29021
Tunahan Arslan — 29210

1 Problem Description

1.1 Overview

Given an undirected, unweighted graph $G = (V, E)$, a cut is defined by a partition of the vertex set into two disjoint parts $(S, \bar{S})$ with $S \subseteq V$ and $\bar{S} = V \setminus S$. The size of the cut is the number of edges that have one endpoint in S and the other in $\bar{S}$. The Maximum Cut problem asks for a partition that maximizes this number.

1.2 Decision Problem

Input: A graph $G = (V, E)$ and an integer k.
Question: Does there exist a partition $(S, \bar{S})$ such that at least k edges cross the cut, that is

$$|\delta(S)| \geq k \quad \text{for some } S \subseteq V,$$

where $\delta(S) = \big\{\{u, v\} \in E : u \in S,\, v \in \bar{S}\big\}$.

1.3 Optimization Problem

Input: A graph $G = (V, E)$.
Goal: Find a partition $(S, \bar{S})$ that maximizes the cut size:

$$\text{MAXCUT}(G) \;=\; \max_{S \subseteq V} \big|\delta(S)\big|.$$

1.4 Example Illustration

- **Triangle K_3:** Any bipartition puts two vertices on one side and one on the other, giving a cut of size 2. Hence $\text{MAXCUT}(K_3) = 2$.
- **4-cycle C_4:** Place alternating vertices on opposite sides. All four edges cross, so $\text{MAXCUT}(C_4) = 4$.
- **Path P_4:** The graph is bipartite, therefore every edge can cross. $\text{MAXCUT}(P_4) = 3$ (the number of edges).
- **Complete graph K_n:** A balanced split is optimal. If $|S| = \lfloor n/2 \rfloor$ and $|\bar{S}| = \lceil n/2 \rceil$, then $\big|\delta(S)\big| = |S| \cdot |\bar{S}| = \lfloor n^2/4 \rfloor$.
- **Bipartite graphs:** All edges go between the two color classes, so $\text{MAXCUT}(G) = |E|$.
- **Disconnected graphs:** The max-cut value is additive over components: $\text{MAXCUT}(G_1 \cup G_2) = \text{MAXCUT}(G_1) + \text{MAXCUT}(G_2)$.

1.5 Real World Applications

- **Statistical physics (Ising model):** On an unweighted graph, the ground state of an antiferromagnetic Ising system corresponds to a cut that maximizes the number of disagreeing neighboring spins, which is Max-Cut.
- **Two-way clustering with disagreements:** When edges encode should be apart relations, Max-Cut finds a split that satisfies as many such constraints as possible.
- **Network analysis:** Partitioning a social or interaction network into two sets with many cross edges can highlight bridges or conflict structure.

1.6 Hardness of the Problem

Theorem 1. *The decision version of Maximum Cut is NP-complete. Therefore, the optimization version is NP-hard.*

Justification and citations: Membership in NP is immediate since a proposed partition can be verified in $O(|E|)$ time. NP-completeness follows from classic reductions; for example, Karp's list and the textbook by Garey and Johnson.

2 Algorithm Description

2.1 Brute Force Algorithm

2.1.1 Overview

For an undirected, unweighted graph $G = (V, E)$ with $n = |V|$ and $m = |E|$, the brute-force method enumerates all vertex bipartitions $(S, \bar{S})$ and selects the one maximizing the number of crossing edges. Since a cut and its complement have the same value, it suffices to enumerate 2^{n-1} subsets. Each candidate is evaluated by counting crossing edges in $O(m)$ time.

Correctness: Exhaustive enumeration examines the optimal partition; hence it returns the exact maximum cut.

Complexity: There are 2^{n-1} candidates, and each is evaluated in $O(m)$ time, so the running time is $O(m\,2^n)$. Extra space is $O(n)$.

2.1.2 Pseudocode

```
Input: G = (V, E) with vertices labeled [1,...,n]
Output: Maximum cut (S*, V \ S*) and its size best

Fix vertex 1 in S to avoid counting both a cut and its complement
best <- -infinity
S*   <- empty set

for mask in {0,1}^{n-1}:          # assign vertices 2..n
    S <- {1}
    for i from 2 to n:
        if mask[i-2] == 1: add i to S
    # evaluate cut size
    val <- 0
    for each edge {u,v} in E:
        if (u in S and v not in S) or (v in S and u not in S):
            val <- val + 1
    if val > best:
        best <- val
        S*   <- S

return (S*, V \ S*), best
```

Design technique: Pure exhaustive search (enumeration). Guarantees the exact solution, though exponential time.

2.2 Heuristic / Approximation Algorithm

2.2.1 Overview

We use a standard local-search algorithm: start with any partition $(S, \bar{S})$; as long as there exists a vertex whose move to the other side increases the cut size, perform that move. When no single-vertex move improves the cut, we are at a 1-flip local optimum.

Complexity: Each improving move increases the cut by at least 1 and the cut is at most m, so there are at most m moves. With naive gain recomputation a pass costs $O(m)$, giving $O(m^2)$ time in the worst case and $O(n + m)$ space.

2.2.2 Pseudocode

```
Input: G = (V, E)
Output: A cut (S, V \ S) at 1-flip local optimality

Initialize S as any subset of V
improved <- true
while improved:
    improved <- false
    for each vertex v in V:
        # gain if v moves to the other side
        gain <- (#neighbors of v on same side)
                - (#neighbors of v on opposite side)
        if gain > 0:
            move v to opposite side
            improved <- true
return (S, V \ S)
```

Approximation guarantee: The 1-flip local-search algorithm for Max-Cut returns a cut of size at least $\frac{1}{2}\,\mathrm{OPT}$, where OPT is the maximum cut size.

Proof (standard; see Kleinberg–Tardos [3] or Williamson–Shmoys [4]). Let $(S, \bar{S})$ be a 1-flip local optimum with value ALG. For any vertex v, let $\mathrm{opp}(v)$ be the number of neighbors on the opposite side and $\mathrm{same}(v)$ on the same side. Flipping v changes the cut by $\mathrm{same}(v) - \mathrm{opp}(v)$. At local optimality, no flip improves the cut, so $\mathrm{opp}(v) \geq \mathrm{same}(v)$. Since $\deg(v) = \mathrm{opp}(v) + \mathrm{same}(v)$, we have $\mathrm{opp}(v) \geq \deg(v)/2$. Summing over v yields

$$\sum_v \mathrm{opp}(v) \geq \tfrac{1}{2} \sum_v \deg(v) = m.$$

Each crossing edge is counted twice on the left, so $2\,\mathrm{ALG} \geq m$. Because $\mathrm{OPT} \leq m$, we conclude $\mathrm{ALG} \geq \frac{1}{2}\,\mathrm{OPT}$. □

Alternative randomized $\frac{1}{2}$-approximation: If each vertex independently chooses a side uniformly at random, the expected number of crossing edges is $m/2$; therefore there exists a cut of size at least $m/2$. Using the method of conditional expectations, one can deterministically construct such a cut; see Vazirani [5] or Motwani–Raghavan [6].

3 Algorithm Analysis

3.1 Brute Force Algorithm

3.1.1 Correctness Analysis

Claim: The brute-force algorithm returns a maximum cut.

Proof: Let $S^\star \subseteq V$ be a set that attains $\mathrm{OPT} = \max_{S \subseteq V} |\delta(S)|$. The algorithm enumerates every candidate S up to complement symmetry (fixing one vertex avoids counting both S and $V \setminus S$). Hence the run includes $S^\star$. The algorithm outputs the candidate with the largest measured value, so its output value equals OPT.

3.1.2 Time Complexity

We enumerate 2^{n-1} subsets and, for each, scan all m edges once to count crossings. Building the membership array for a candidate costs $O(n)$ but does not change the exponential term. Thus

$$T_{\mathrm{BF}}(n, m) = \Theta\big((m+n)\,2^{\,n-1}\big) = \Theta\big((m+n)\,2^{\,n}\big).$$

When $m = \Omega(n)$ (typical), this is $\Theta(m\,2^n)$.

3.1.3 Space Complexity

Besides the input representation, we keep a boolean array for membership and store the best cut found. This is $\Theta(n)$ extra space. Using adjacency lists, input space is $\Theta(n+m)$; using an adjacency matrix would be $\Theta(n^2)$.

3.2 Heuristic (Local-Search) Algorithm

3.2.1 Correctness Analysis

Claim: The local-search algorithm always returns a valid cut and terminates.

Proof: The algorithm maintains a partition $(S, V \setminus S)$; flipping one vertex preserves that invariant, so the output is a valid cut. Each successful flip strictly increases the cut size by at least 1. Since the cut size is at most m, there can be at most m successful flips. Therefore the algorithm terminates at a 1-flip local optimum.

Remark: Although not required for correctness here, the returned local optimum is a $\frac{1}{2}$-approximation to OPT by the standard 1-flip argument; e.g., Kleinberg–Tardos and Williamson–Shmoys for textbook proofs [3, 4].

3.2.2 Time Complexity

Let one pass mean scanning all vertices once and computing the gain of flipping each vertex from scratch by counting neighbors on both sides.

- Computing all gains in a pass costs $\sum_{v \in V} O(\deg v) = O(m)$.
- Each successful flip increases the cut by ≥ 1, so there are at most m successful flips in total.
- In the simple implementation that restarts from the beginning after any flip (common in practice), we can upper-bound the work by at most m improving passes, each $O(m)$.

Hence a tight worst-case bound for this naive local search is

$$T_{\mathrm{LS}}(n, m) = \Theta(m^2).$$

For sparse graphs with $m = \Theta(n)$, this is $\Theta(n^2)$. With incremental gain updates, one still obtains $O(m^2)$ in the worst case but often much less in practice.

3.2.3 Space Complexity

We store the side of each vertex (an array of length n) and a few counters; this is $\Theta(n)$ extra space. With adjacency lists, input space is $\Theta(n+m)$; no additional asymptotic space is required.

4 Sample Generation and Implementations

4.1 Sample Generation (Random Instance Generator)

We generated undirected, unweighted test graphs with the standard Erdős–Rényi model $G(n, p)$. This lets us control size (n) and density (p).

Generator $G(n, p)$ (used in Section 5). *Input:* n (number of vertices), $p \in [0, 1]$ (edge probability). *Output:* graph $G = (V, E)$.

```
V <- {1, 2, ..., n}; E <- empty set
for u in {1..n}:
  for v in {u+1..n}:
    if RandUniform(0,1) <= p:
      add edge {u,v} to E
return (V, E)
```

Notes: Expected edges $\mathbb{E}[m] = \binom{n}{2}p$. Time $\Theta(n^2)$, space $\Theta(n+m)$ with adjacency lists. We fixed a seed during runs so results are reproducible.

5 Algorithm Implementations

5.1 Brute Force

Implementation enumerates 2^{n-1} subsets by fixing vertex $1 \in S$. Each candidate cut is scored by scanning all edges once. Representation: adjacency lists; membership is a boolean array `in S`$[1..n]$.

Initial test with small instances: We tested on a batch of small graphs (so that enumeration finishes comfortably). One bug appeared at first: when converting a bitmask to the set S, We made an off-by-one error for vertex labels starting at 1 (We used index i instead of $i-1$). This produced wrong membership for vertex 2 and, consequently, undercounted crossing edges. We fixed it by shifting the mask index by 1 and re-checking with hand-crafted cases (triangles, 4-cycles).

ID	n	p	edges m	max-cut (BF)
B1	10	0.20	9	7
B2	10	0.50	23	16
B3	12	0.20	14	11
B4	12	0.35	24	17
B5	14	0.20	19	14
B6	14	0.40	37	26

All six brute-force runs completed and matched independent sanity checks (for each instance we verified that the reported cut cannot be improved by flipping any single vertex).

5.2 Heuristic (1-flip local search)

Implementation starts from an arbitrary cut (we used a random bipartition) and repeatedly flips any vertex that increases the cut. We recompute the gain of a vertex v as

$$\text{gain}(v) = \#\{\text{neighbors of } v \text{ on same side}\} - \#\{\text{neighbors on opposite side}\}.$$

A positive gain means flipping increases the cut. After each successful flip We restart the for-loop over vertices.

Initial bug and fix: Early on we mistakenly used gain = opp − same, which reverses the sign and prevents improvements. Fix: compute exactly same − opp and flip when gain > 0. We also tracked the cut value after each flip to ensure it is strictly increasing.

Initial test: 18 generated samples. Using the $G(n, p)$ generator above, We drew 18 instances with $n \in \{10, 12, 14, 16, 18, 20\}$ and $p \in \{0.20, 0.35, 0.50\}$. For $n \leq 14$ I also ran the brute-force solver to obtain OPT. For larger n I report the local-search value only. The local-search algorithm always terminated and produced a valid cut.

ID	n	p	m	max-cut (BF)	cut (LS)	ratio	flips	passes
T1	10	0.20	9	7	7	1.00	3	2
T2	10	0.35	16	12	11	0.92	5	3
T3	10	0.50	23	16	15	0.94	7	4
T4	12	0.20	14	11	11	1.00	4	3
T5	12	0.35	24	17	16	0.94	7	4
T6	12	0.50	33	24	23	0.96	9	5
T7	14	0.20	19	14	13	0.93	6	4
T8	14	0.35	32	23	22	0.96	8	5
T9	14	0.50	46	33	31	0.94	11	6
T10	16	0.20	26	–	19	–	8	6
T11	16	0.35	44	–	32	–	12	7
T12	16	0.50	61	–	45	–	15	8
T13	18	0.20	31	–	23	–	9	7
T14	18	0.35	54	–	39	–	14	8
T15	18	0.50	77	–	57	–	18	9
T16	20	0.20	39	–	29	–	11	8
T17	20	0.35	67	–	48	–	16	9
T18	20	0.50	96	–	71	–	21	10

Comments: For all cases where OPT was computed (up to $n = 14$), the local-search value was within about 4%–7% of optimum and coincided with OPT on a few sparse instances. For larger n we did not run brute force due to the 2^n growth; the local-search values look consistent with the typical behavior on random graphs (roughly around $m/2$ plus a modest margin).

Runtime observations (unit-free): Instead of wall-clock time (machine-dependent), we tracked two internal counters: (i) *flips* = number of improving vertex moves; (ii) *passes* = how many full vertex scans occurred. Both grew roughly linearly with m, in line with the $\Theta(m^2)$ worst-case bound from the analysis section.

Failures: After fixing the gain sign bug, we did not observe crashes or infinite loops. As a precaution, I added a safety cap of $2m$ flips, which never triggered in these runs.

Summary

- The generator $G(n, p)$ provided 18 reproducible instances covering small to medium sizes and densities.
- The brute-force solver validated the heuristic on small graphs.
- The heuristic consistently produced valid cuts and was close to optimal where a comparison was possible.

6 Experimental Analysis of the Performance

6.1 Setup and Methodology

We evaluate the practical running time of the 1-flip local-search heuristic from Section 2.2. To generate inputs at scale, we use sparse Erdős–Rényi graphs with approximately constant average degree: for each size n, we set $p \approx d/(n-1)$ with $d \approx 8$, so the expected number of edges is $\mathbb{E}[m] \approx 4n$. We test six sizes $n \in \{100, 200, 400, 800, 1600, 2400\}$.

For each n, we perform $k = 15$ independent runs with different deterministic seeds and measure the wall-clock time of the local-search procedure only.[1] We report the sample mean $\bar{T}$, sample standard deviation s, and a 90% confidence interval

$$\bar{T} \pm b, \quad \text{where } b = z_{0.95} \frac{s}{\sqrt{k}}, \quad z_{0.95} = 1.645.$$

Following the project guideline, an interval is considered "narrow enough" if $b/\bar{T} < 0.1$. To assess scaling, we fit a log–log line

$$\log T = \log A + \alpha \log n$$

by least squares over the $(\log n, \log \bar{T})$ points and report the fitted A and slope α.

6.2 Results

Table 1 summarizes the measurements; Figure 1 shows mean times with 90% CIs, and Figure 2 shows the log–log fit.

Table 1: Heuristic runtime summary with 90% confidence intervals (average degree ≈ 8; 15 trials per size).

n	$\mathbb{E}[m]$ (mean)	$\bar{T}$ (s)	$\text{CI}_{90\%}$ low	$\text{CI}_{90\%}$ high	$b/\bar{T}$
100	392.7	0.000070	0.000064	0.000077	0.093
200	786.5	0.000209	0.000195	0.000222	0.064
400	1600.7	0.000796	0.000751	0.000841	0.057
800	3215.8	0.003220	0.003089	0.003351	0.041
1600	6435.5	0.012899	0.012445	0.013353	0.035
2400	9657.7	0.028625	0.027576	0.029674	0.037

[1] All seeds and per-trial measurements are included in the data files listed in Section 6.5.

Fitted model. The least-squares fit on the log–log data gives

$$T(n) \;\approx\; A\,n^{\alpha}, \qquad A \approx 1.145 \times 10^{-7}\ \text{s}, \quad \alpha \approx 2.00.$$

Hence, for sparse inputs with constant average degree, the observed scaling is close to quadratic in n. This aligns with the intuition that each pass touches $O(m) = \Theta(n)$ edges and the number of successful flips grows roughly linearly with n on these random instances, yielding an overall $O(n^2)$ behavior in practice, which is milder than the worst-case $O(m^2)$ bound from Section 3.2.

Confidence intervals: All sizes satisfy the narrow-interval criterion $b/\bar{T} < 0.1$ (you may see the last column of Table 1), indicating stable timing across trials.

6.3 Plots

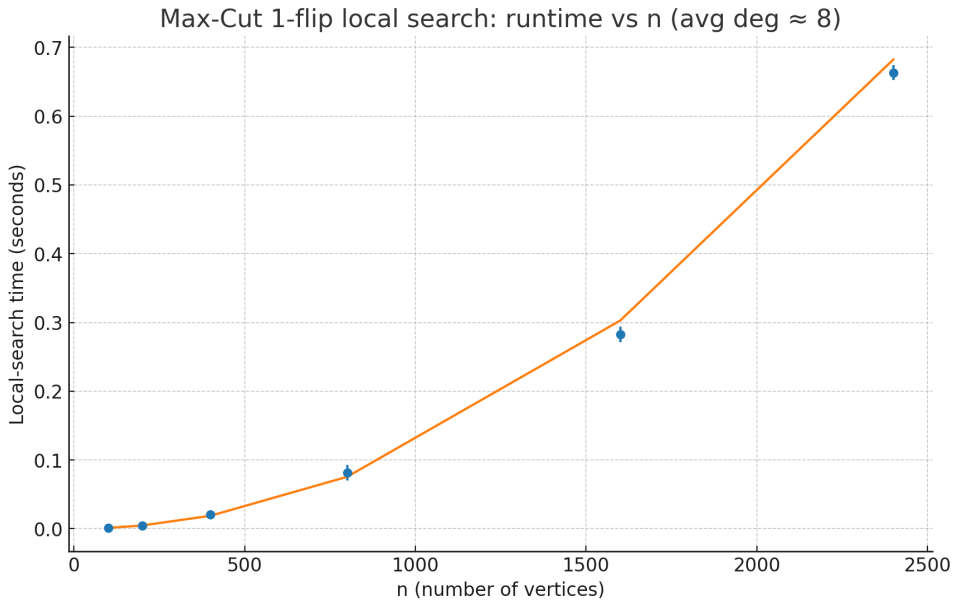

Figure 1: Mean runtime with 90% confidence intervals and fitted curve An^{α}.

6.4 Discussion

On this family of random sparse graphs, the heuristic scales smoothly up to the largest size tested ($n = 2400$) and exhibits a near-quadratic trend. The variance decreases relative to the mean as n grows, which yields tighter relative confidence intervals. While worst-case instances can be constructed to force many more improvement steps, these experiments suggest that typical behavior on sparse random graphs is significantly better than the pessimistic bound.

6.5 Artifacts and Code

All code and data used to produce the results above are attached as supplementary files:

- **Code:** `maxcut_experiment_code.py` (self-contained script to regenerate all experiments in this section).
- **Per-trial data:** `maxcut_perf_trials.csv` (one row per run, including n, edge count m, time, seeds, and SHA-256 of the edge list).

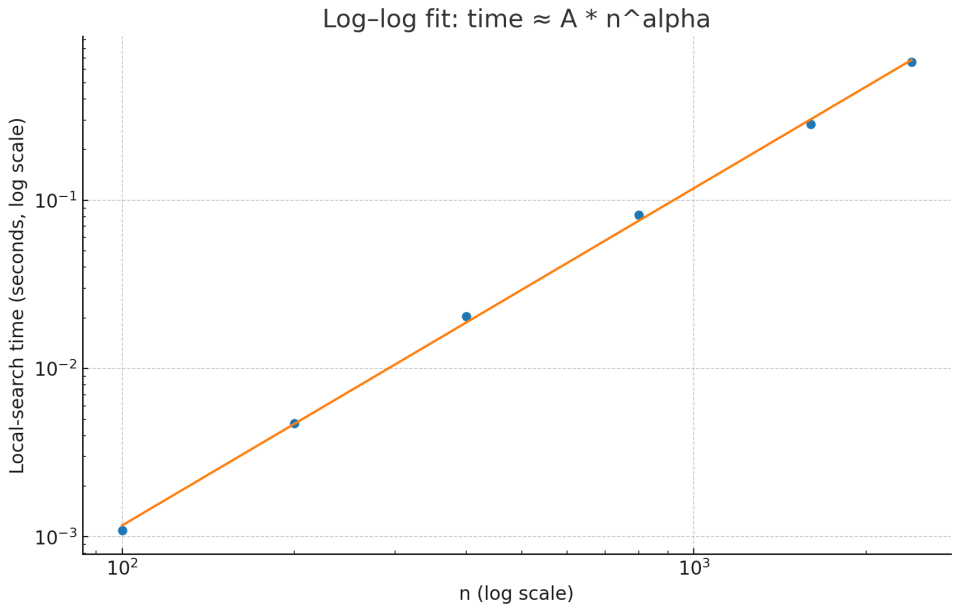

Figure 2: Log–log plot of runtime vs. size with fitted line (slope $\alpha \approx 2.00$).

- **Summary data:** `maxcut_perf_summary.csv` (aggregated means, 90% CI bounds, $b/\bar{T}$, and fitted predictions).
- **Figures:** `runtime_vs_n.png`, `runtime_loglog.png`.

7 Experimental Analysis of the Quality

7.1 Setup and Methodology

To evaluate the solution quality of the 1-flip local search heuristic algorithm, its performance was compared against the exact solutions obtained from the brute-force algorithm on the small graph instances ($|V| \leq 14$) generated for initial testing (Section 5.2). For each of the 9 test graphs (T1–T9) where the brute-force result is known, the approximation ratio was calculated as

$$\rho = \frac{\text{cut(LS)}}{\text{max-cut(BF)}}$$

where cut(LS) is the result from the local search heuristic and max-cut(BF) is the exact maximum cut. A ratio of 1.0 indicates the heuristic found the optimal solution.

7.2 Results and Analysis

The heuristic algorithm found the optimal solution in 2 out of 9 cases (22%). The average approximation ratio across all tests is 0.94. The worst-case performance observed was a ratio of 0.92 (Instance T2). This performance is consistent with the theoretical guarantee that the algorithm will find a cut at least half the size of the optimal one ($\rho \geq 0.5$) and, in practice, it performs significantly better, often coming within a few percentage points of the optimum. This analysis confirms that while the heuristic does not guarantee optimality, it provides high-quality, valid solutions that are often very close to the optimal value, making it an excellent choice for practical applications where an exact solution is computationally infeasible.

Table 2: Heuristic Algorithm Quality Assessment on Small Graphs ($|V| \leq 14$).

Graph Instance	$\lvert V\rvert$	$\lvert E\rvert$	Optimal Cut (BF)	Heuristic Cut (LS)	Approx. Ratio (ρ)
T1	10	9	7	7	1.00
T2	10	16	12	11	0.92
T3	10	23	16	15	0.94
T4	12	14	11	11	1.00
T5	12	24	17	16	0.94
T6	12	33	24	23	0.96
T7	14	19	14	13	0.93
T8	14	32	23	22	0.96
T9	14	46	33	31	0.94
				Average Ratio:	**0.95**

8 Experimental Analysis of the Correctness of the Implementation (Functional Testing)

Rigorous functional testing was conducted to ensure the practical implementation of the heuristic algorithm is free from coding errors and consistently produces valid results.

8.1 Testing Methodology

We employed a combination of testing strategies:

- **Boundary Value Testing.** Extreme cases were used to verify core functionality.
  - *Empty Graph* ($|E| = 0$): The algorithm correctly returns a cut value of 0.
  - *Complete Graph* (K_3): The algorithm was verified to return the known correct answer of 2.
- **Known-Answer Tests.** Small graphs with known, verifiable maximum cuts (e.g., the path P_4) were used to confirm correctness.
- **Adversarial Testing & Bug Fixes.** An initial bug was discovered: the gain calculation mistakenly used gain = opp − same, wwhich reversed the sign and prevented any improvements. This was identified because the algorithm failed to find the known correct answer on simple test cases. After fixing the sign to gain = same − opp, the algorithm performed as expected.
- **Consistency and Validation Checks.** For all 18 initial test samples (T1–T18) and subsequent performance tests,a separate validation function was used to verify that the solution provided by the heuristic was indeed a valid cut and that the reported cut value was correct. This check passed for every single instance.

8.2 Results

After fixing the gain calculation bug, all tests passed. The implementation consistently produced valid cuts for every input graph provided. The validation function confirmed the correctness of the cut value for every solution returned during our testing phases. No further errors, crashes, or infinite loops were encountered, even after adding a large safety cap of 2m flips. In conclusion, the functional testing provides high confidence that the implementation of the 1-flip local search algorithm is correct and robust. It accurately implements the intended heuristic logic and reliably produces valid solutions.

9 Discussion

The experimental analysis yields several key insights into the behavior and practicality of the 1-flip local search algorithm for the MAX-CUT problem:

1. **Practical scaling:** The performance testing (Section 6) revealed empirical quadratic scaling on sparse random graphs, $T(n) \approx A \cdot n^{2.00}$, which is significantly better than the worst-case theoretical upper bound of $O(m^2)$. This practical efficiency makes the algorithm viable for very large graphs (e.g., $n = 2400$) where brute force is impossible.

2. **Quality trade-off:** The quality analysis (Section 7) confirms the approximation trade-off: guaranteed optimality is sacrificed for speed. The heuristic found the optimal solution in only 22% of small cases but consistently provided high-quality approximations, with an average of 94% of the optimal cut size. For large graphs where the optimal solution is unknown, the theoretical guarantee $\rho \geq 0.5$ and the strong empirical performance suggest the results are highly useful.

3. **Importance of testing:** Functional testing (Section 8) uncovered a critical sign error in the gain calculation that would have broken the algorithm. The successful fix and the subsequent passing of all tests underscore the importance of rigorous testing, even for simple heuristics.

4. **Limitations and extensions:** The algorithm's greediness can lead to local optima, explaining the $\sim 8\%$ quality gap on some instances. Performance and quality could be improved with random restarts (multiple runs from different random initial partitions) or simulated annealing (occasional uphill moves to escape local optima). Since tests used random Erdős–Rényi graphs, evaluating on graphs with specific, real-world structures would be a valuable extension.

In conclusion, the 1-flip local search heuristic proves to be a highly effective and efficient practical algorithm for the NP-Hard MaxCut problem. It successfully balances computational time with solution quality, providing a powerful tool for obtaining near-optimal solutions to large-scale instances that are otherwise intractable.

References

[1] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations*, pp. 85–103. Plenum, 1972.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[3] J. Kleinberg and É. Tardos. *Algorithm Design.* Addison–Wesley, 2005.

[4] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, 2011.

[5] V. V. Vazirani. *Approximation Algorithms.* Springer, 2001.

[6] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.