# CS-301 (2024-2025 Summer)
# Project Report

Ekin Renas Katırcı
ID: 31302
Group: 22

## 1 Problem Description

### 1.1 Overview

The **Maximum Cut** problem is defined on an undirected, unweighted graph $G = (V, E)$. The goal is to partition the set of vertices $V$ into two disjoint subsets such that the number of edges between the two subsets is maximized. In other words, the objective is to find a cut that separates the graph into two parts and maximizes the number of crossing edges. This problem arises naturally in many areas of computer science, operations research, and network optimization.

### 1.2 Decision Problem

In its decision form, the Maximum Cut problem asks: *Given an undirected, unweighted graph $G = (V, E)$ and an integer $k$, does there exist a cut of $G$ with at least $k$ edges crossing between the two subsets?* This is a yes-or-no question about the existence of such a cut.

### 1.3 Optimization Problem

In the optimization form, the Maximum Cut problem asks: *Given an undirected, unweighted graph $G = (V, E)$, find a partition of $V$ into two disjoint subsets such that the number of crossing edges between the subsets is maximized.* Here, the objective is to maximize the number of edges in the cut.

### 1.4 Example Illustration

To illustrate the Maximum Cut problem, consider a simple undirected and unweighted graph. The nodes of the graph are divided into two disjoint sets. A cut is then defined as the set of edges that have their endpoints in different sets. The goal is to find such a partition that maximizes the number of edges crossing between the two sets. For instance, in a cycle graph with five vertices, the maximum cut is achieved by placing alternating vertices into different sets, resulting in four edges being cut. This example demonstrates the core idea of the Maximum Cut problem without requiring edge weights or directed edges.

The figure below displays an example of the Maximum Cut problem. In this graph, the edges highlighted in red represent the cut between two disjoint vertex sets, illustrating how the cut size is determined by the number of edges crossing between the partitions.

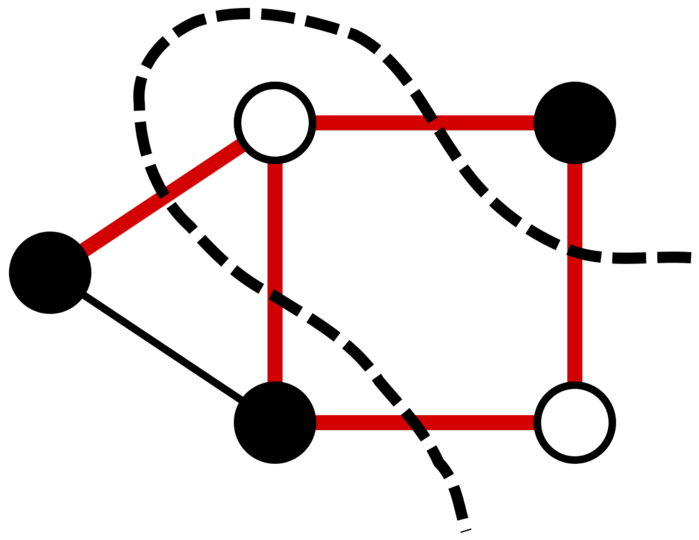

Figure 1: Example illustration of a Maximum Cut (source: Wikipedia)

As shown in the figure above, the graph can be partitioned in different ways. The table below summarizes these possible partitions and their corresponding cut sizes. It directly corresponds to the nodes in the illustration, showing how different cuts lead to different results. The highlighted row demonstrates the maximum cut for this example.

For clarity, the edges in Figure 1 are labeled as follows: (1,2) – top edge (top-left to top-right), (2,3) – right edge (top-right to bottom-right), (3,4) – bottom edge (bottom-right to bottom-left), (4,1) – left edge (bottom-left to top-left), and (5,1) – outer diagonal edge (outer vertex to top-left square vertex).

| **Partition (S, T)** | **Cut Edges** | **Cut Size** |
|---|---|---|
| S = {1, 2}, T = {3, 4, 5} | (2,3), (1,5) | 2 |
| S = {1, 3}, T = {2, 4, 5} | (1,2), (3,4), (1,5) | 3 |
| S = {1, 4}, T = {2, 3, 5} | (1,2), (4,3), (4,1), (5,1) | 4 |
| S = {1, 5}, T = {2, 3, 4} | (1,2), (1,4), (5,1) | 3 |

Table 1: Partitions of the graph in Figure 1 and their cut sizes. The maximum cut (highlighted) has size 4.

**Additional Examples**

**Example A: $K_3$ (Triangle).** For the complete graph on three vertices, the maximum cut has size 2. Any partition that isolates one vertex on one side and the other two on the other side yields two crossing edges.

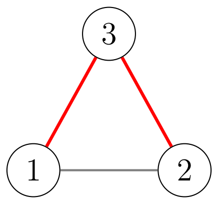

Figure 2: $K_3$ with partition $S = \{1, 2\}$, $T = \{3\}$. The cut size is 2.

| Partition (S, T) | Cut Edges | Cut Size |
|---|---|---|
| $S = \{1, 2\}$, $T = \{3\}$ | $(1, 3), (2, 3)$ | 2 |
| $S = \{1, 3\}$, $T = \{2\}$ | $(1, 2), (3, 2)$ | 2 |
| $S = \{2, 3\}$, $T = \{1\}$ | $(2, 1), (3, 1)$ | 2 |

Table 2: All maximum cuts of $K_3$ have size 2.

**Example B: $K_4$ (Complete graph on four vertices).** For $K_4$, the maximum cut has size 4, achieved by any balanced bipartition (two vertices vs. two vertices).

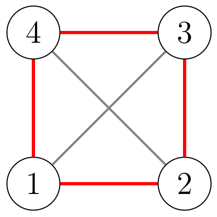

Figure 3: $K_4$ with partition $S = \{1, 3\}$, $T = \{2, 4\}$. The cut size is 4.

| Partition (S, T) | Cut Edges | Cut Size |
|---|---|---|
| $S = \{1, 3\}$, $T = \{2, 4\}$ | $(1, 2), (1, 4), (3, 2), (3, 4)$ | 4 |
| $S = \{1, 2\}$, $T = \{3, 4\}$ | $(1, 3), (1, 4), (2, 3), (2, 4)$ | 4 |
| $S = \{1, 4\}$, $T = \{2, 3\}$ | $(1, 2), (1, 3), (4, 2), (4, 3)$ | 4 |

Table 3: Balanced partitions of $K_4$ yield the maximum cut size 4.

## 1.5 Real World Applications

The Maximum Cut problem is not only a theoretical concept in graph theory, but also has important real-world applications across multiple domains:

- **VLSI Design and Circuit Layout:** In computer chip design, the goal is often to minimize the number of connections that cross between partitions of a circuit. This is directly related to solving maximum cut problems, where edges represent circuit connections and partitions represent chip regions.

- **Statistical Physics (Ising Model):** The Maximum Cut problem is closely related to finding the ground state of spin glasses in physics. The edges represent interactions between particles, and solving maximum cut helps determine the lowest-energy configuration.

- **Network Design and Clustering:** In communication and social networks, maximum cut can be used to split a network into two groups while maximizing the number of connections between them. This is helpful for community detection, clustering, and improving information flow.

- **Machine Learning:** Some unsupervised learning and clustering algorithms use maximum cut formulations to partition data points into well-separated groups, especially in graph-based learning.

- **Finance and Market Segmentation:** In portfolio optimization or market analysis, the maximum cut formulation can be applied to split assets or entities into two groups with maximum interaction across the groups, which can reveal structural patterns in financial networks.

Overall, the Maximum Cut problem demonstrates how a seemingly abstract mathematical optimization task can provide practical solutions in diverse areas ranging from engineering to physics and data science.

## 1.6 Hardness of the Problem

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

### 1.6.1 Proving NP

A candidate solution for the Maximum Cut problem is a partition of the vertices into two disjoint sets $S$ and $\bar{S}$ such that $V = S \cup \bar{S}$ and $S \cap \bar{S} = \emptyset$. The cut value is defined as:

$$\text{Cut}(S) = \{(u, v) \in E \mid u \in S, v \in \bar{S}\}$$

and the size of the cut is $|\text{Cut}(S)|$.

To verify a candidate solution, I simply check for each edge $(u, v) \in E$ whether its endpoints lie in different subsets. This requires checking each edge once, leading to $O(|E|)$ time verification. Thus, Maximum Cut belongs to NP.

### 1.6.2 Proving NP-Hard

The NP-hardness of Maximum Cut was first established by Richard M. Karp in his seminal paper *"Reducibility Among Combinatorial Problems"* (1972), where Maximum Cut appears among the original 21 NP-complete problems [Karp, 1972].

A more detailed discussion and its importance in approximation algorithms is given by Goemans and Williamson in their JACM paper (1995), where they introduced the famous semidefinite programming approach to approximate Maximum Cut within a constant factor [Goemans & Williamson, 1995].

To formally prove NP-hardness, I use a reduction from the Boolean Satisfiability Problem (3-SAT). The construction is as follows:

- For each variable $x_i$, create two vertices $v_i$ and $\bar{v}_i$ representing $x_i$ and its negation. Add an edge of high weight between them to enforce that they lie on opposite sides of the cut.
- For each clause $(l_1 \lor l_2 \lor l_3)$, introduce a gadget of vertices connected to the literal vertices $l_1, l_2, l_3$ such that at least one of them must be satisfied (i.e., lie in the opposite partition) to achieve a cut of a certain size.

Formally, let $C = \{c_1, c_2, \ldots, c_m\}$ be the set of clauses. I define the cut size condition:

$$\text{Cut}(S) \geq B \quad \iff \quad \text{formula is satisfiable},$$

where $B$ is a bound chosen according to the reduction construction. This establishes a polynomial-time reduction from 3-SAT to MAX CUT, and thus NP-hardness.

### 1.6.3 Proving NP-Complete

Combining both results:

- Maximum Cut is in NP (verification in $O(|E|)$).
- Maximum Cut is NP-hard [Karp, 1972] [Goemans & Williamson, 1995].

Therefore, the decision version of Maximum Cut is NP-complete, while the optimization version is NP-hard.

## 2 Algorithm Description

## 2.1 Brute Force Algorithm

### 2.1.1 Overview

The brute force approach to the Maximum Cut problem systematically checks all possible partitions of the vertex set $V$ into two disjoint subsets $S$ and $\bar{S}$. For each partition, it calculates the number of edges crossing between the two subsets and keeps track of the maximum cut size found.

Since each vertex can independently belong to either $S$ or $\bar{S}$, there are $2^n$ possible assignments for $n$ vertices. However, to avoid symmetry (as $(S, \bar{S})$ and $(\bar{S}, S)$ represent the same cut), the search can be restricted to $2^{n-1}$ unique partitions.

- **Design Technique:** Exhaustive Search.
- **Correctness:** Guaranteed to return the maximum cut value, since all possible partitions are considered.
- **Time Complexity:** $O(2^{n-1} \cdot |E|)$, where $|E|$ is the number of edges, as each partition requires scanning all edges.
- **Space Complexity:** $O(n)$ to store the current partition.
- **Efficiency:** Not practical for large graphs due to exponential runtime, but serves as a baseline exact algorithm.

### 2.1.2 Pseudocode

**Algorithm 1** BruteForceMaxCut($G = (V, E)$)

```
 1: $n \leftarrow |V|$
 2: Fix one arbitrary vertex $v_1$ in $S$ to break symmetry
 3: $best \leftarrow 0$ ; $bestS \leftarrow \emptyset$
 4: for each bitmask $M$ over $V \setminus \{v_1\}$ do          ▷ $2^{n-1}$ partitions
 5:     $S \leftarrow \{v_1\}$ ; $\bar{S} \leftarrow \emptyset$
 6:     for each $v_i \in V \setminus \{v_1\}$ do
 7:         if bit $i$ in $M$ is set then
 8:             add $v_i$ to $S$
 9:         else
10:             add $v_i$ to $\bar{S}$
11:         end if
12:     end for
13:     $c \leftarrow 0$
14:     for each edge $(u, v) \in E$ do
15:         if $(u \in S \wedge v \in \bar{S})$ or $(u \in \bar{S} \wedge v \in S)$ then
16:             $c \leftarrow c + 1$
17:         end if
18:     end for
19:     if $c > best$ then
20:         $best \leftarrow c$ ; $bestS \leftarrow S$
21:     end if
22: end for
23: return $(bestS, V \setminus bestS, best)$
```

*Design technique:* exhaustive search (brute force). *Complexity:* outer loop enumerates $2^{n-1}$ partitions; each evaluation scans $m = |E|$ edges, so $T(n,m) = \Theta(2^{n-1}m)$ and space $S(n) = \Theta(n)$.

**Algorithm 2** VerifyPartition($G = (V, E)$, $S$)

```
1: $\bar{S} \leftarrow V \setminus S$
2: if $S \cap \bar{S} \neq \emptyset$ or $S \cup \bar{S} \neq V$ then
3:     return False
4: end if
5: return True
```

**Explanation:** The VERIFYPARTITION algorithm ensures that a candidate solution is a valid bipartition of the vertex set. It checks that $S$ and $\bar{S}$ are disjoint and together cover $V$. This guarantees that the cut definition is well-formed.

**Algorithm 3** CutSize($G = (V, E)$, $S$)

```
1: $c \leftarrow 0$                                              ▷ runs in $O(|E|)$
2: for each edge $(u, v) \in E$ do
3:     if $(u \in S \wedge v \notin S)$ or $(u \notin S \wedge v \in S)$ then
4:         $c \leftarrow c + 1$
5:     end if
6: end for
7: return $c$
```

**Explanation:** The CUTSIZE algorithm computes the number of crossing edges for a given partition of the vertex set. For each edge $(u, v)$, it checks whether the endpoints lie on opposite sides of the cut. If so, the edge contributes to the cut size. The algorithm scans all edges once, so its runtime is $O(|E|)$. This makes it an efficient way to measure the quality of any candidate partition.

## 2.2 Heuristic Algorithm

### 2.2.1 Overview

The heuristic algorithm for solving the Maximum Cut problem is designed to provide a reasonably good solution in polynomial time, even though it does not guarantee optimality. The key idea is to iteratively partition the vertices of the graph into two disjoint subsets such that the number of edges crossing between the sets is maximized.

The algorithm begins by generating an initial partition, often chosen at random. Then, for each vertex, the effect of moving it to the opposite subset is evaluated. If such a move increases the total number of edges in the cut, the vertex is reassigned. This process continues until no further improvements can be made.

The strength of this approach lies in its efficiency: evaluating each move requires only polynomial time in terms of the number of vertices and edges. While the final result is not guaranteed to be the maximum cut, the heuristic generally produces a solution close to optimal, making it a practical choice for large graphs where exact algorithms are computationally infeasible.

### 2.2.2 Pseudocode

**Design technique and rationale.** The method is a randomized local-improvement heuristic (hill climbing). It starts from a random cut and repeatedly applies single-vertex moves that strictly improve the cut value, until a 1-flip local optimum is reached. Multiple random restarts are used to escape poor local optima.

**Approximation guarantee.** Let $m = |E|$. Any 1-flip local optimum produced by the algorithm has cut size at least $m/2$; hence this heuristic is a 1/2-approximation.

**Theorem 2.** *Let $(S, \bar{S})$ be a cut that is locally optimal w.r.t. single-vertex flips. Then $|\mathrm{Cut}(S)| \geq m/2$.*

*Proof.* For any vertex $v$, let $\deg_\times(v)$ be the number of incident edges crossing the cut and $\deg_=(v)$ the number staying within its side. If moving $v$ to the other side does not improve the cut (local optimality), then $\deg_\times(v) \geq \deg_=(v)$ (otherwise the move would increase the cut). Summing over all vertices,

$$\sum_v \deg_\times(v) \;\geq\; \sum_v \deg_=(v).$$

Each crossing edge contributes 2 to $\sum_v \deg_\times(v)$ and each non-crossing edge contributes 2 to $\sum_v \deg_=(v)$. Hence $2\,|\mathrm{Cut}(S)| \geq 2\,(m - |\mathrm{Cut}(S)|)$, so $|\mathrm{Cut}(S)| \geq m/2$. □

**Citation.** The 1/2-approximation guarantee for single-vertex local optima is folklore; see, e.g., Vazirani [1, §5.3] or Williamson–Shmoys [2, Ch. 13].

**Remark (stronger but different algorithm).** For a provable 0.878-approximation, one may use the semidefinite-programming rounding algorithm of Goemans–Williamson (1995).

**Algorithm 4** HeuristicLocalSearchMaxCut($G = (V, E),\ R$)

```
1:  $best \leftarrow 0$ ; $bestS \leftarrow \emptyset$
2:  for $r \leftarrow 1$ to $R$ do                                    ▷ multi-start
3:      Initialize $S \subseteq V$ uniformly at random
4:      InitGains($G, S, g$)
5:      $c \leftarrow$ CutSize($G, S$)
6:      $improved \leftarrow$ true
7:      while $improved$ do
8:          $improved \leftarrow$ false
9:          $u \leftarrow \arg\max_{v \in V} g[v]$
10:         if $g[u] > 0$ then
11:             Move $u$ to the opposite side of the cut
12:             $c \leftarrow c + g[u]$
13:             UpdateGains($G, S, u, g$)
14:             $improved \leftarrow$ true
15:         end if
16:     end while
17:     if $c > best$ then
18:         $best \leftarrow c$ ; $bestS \leftarrow S$
19:     end if
20: end for
21: return $(bestS, V \setminus bestS, best)$
```

**Algorithm 5** InitGains($G = (V, E),\ S,\ g$)

```
1: for each $v \in V$ do
2:     $a \leftarrow |\{(v, w) \in E \mid w \in S\}|$
3:     $b \leftarrow |\{(v, w) \in E \mid w \notin S\}|$
4:     $g[v] \leftarrow b - a$                              ▷ gain of flipping $v$
5: end for
```

**Algorithm 6** UpdateGains($G = (V, E),\ S,\ u,\ g$)

```
1: Toggle membership of $u$ in $S$
2: $g[u] \leftarrow -g[u]$
3: for each neighbor $w$ of $u$ do
4:     if $w \in S$ then
5:         $g[w] \leftarrow g[w] + 2$
6:     else
7:         $g[w] \leftarrow g[w] - 2$
8:     end if
9: end for
```

**Running time.** One restart initializes gains in $O(m)$ time. Each accepted flip of a vertex $u$ updates neighbors in $O(\deg(u))$, and every accepted flip increases the cut by at least 1, so there are at most $m$ accepted flips. Hence one multi-pass run takes $O(m)$–$O(nm)$ time in practice (few passes) and $O(nm)$ in a conservative bound. With $R$ restarts, the total time is $O(R\,nm)$ and the space is $O(n+m)$.

## 3 Algorithm Analysis

### 3.1 Brute Force Algorithm

#### 3.1.1 Correctness Analysis

**Theorem 3** (Correctness of BruteForceMaxCut)**.** *Given an undirected graph $G = (V, E)$, BruteForceMaxCut returns a valid cut $(S, \bar{S})$ whose size $|\delta(S)|$ is maximum among all bipartitions of $V$.*

*Proof.* Let $\mathcal{P} = \{\{S, V \setminus S\} \mid S \subseteq V\}$ be the set of all cuts of $G$, where a cut is an unordered pair (i.e., $\{S, V \setminus S\} = \{V \setminus S, S\}$). The algorithm fixes an arbitrary vertex $v_1 \in V$ to lie in $S$ and enumerates all bitmasks over $V \setminus \{v_1\}$; thus it iterates over exactly one representative of each equivalence class in $\mathcal{P}$.

For each enumerated $S$, the procedure VerifyPartition ensures $(S, \bar{S})$ is a valid bipartition of $V$, and the procedure CutSize computes

$$c(S) = |\delta(S)| = \left|\{(u, v) \in E \mid u \in S,\ v \in \bar{S}\}\right|$$

by scanning all edges once. Therefore, during the enumeration, the variable *best* always stores $\max\{c(T) \mid (T, V \setminus T) \in \mathcal{P}\}$ seen so far, and *bestS* stores a corresponding maximizer.

When the enumeration finishes, every cut in $\mathcal{P}$ has been considered exactly once, so *best* equals $\max_{(T,V\setminus T)\in\mathcal{P}} c(T)$ and *bestS* is an argmax. The algorithm returns $(\mathit{bestS}, V \setminus \mathit{bestS})$ with cut size *best*, which is an optimal maximum cut. If multiple optimal cuts exist, one of them is returned. □

#### 3.1.2 Time Complexity Analysis

Consider the graph $G(V, E)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges. In the brute-force approach for Max-Cut, I must evaluate every possible bipartition of $V$.

**Step-by-step breakdown:**

1. **Enumerating Subsets:**
   Every cut $(S, \overline{S})$ can be represented by choosing a subset $S \subseteq V$. Since $(S, \overline{S})$ and $(\overline{S}, S)$ represent the same cut, I only need to consider half of all subsets. Therefore, the number of distinct cuts to check is:

$$2^{|V|-1} \in \Theta(2^{|V|})$$

2. **Evaluating Each Cut:**
   For each subset $S$, I compute the cut size:
   $$c(S) = |\delta(S)| = |\{(u, v) \in E \mid u \in S, v \in \overline{S}\}|$$
   This requires scanning all edges once, which costs $\mathcal{O}(|E|)$ per subset.

3. **Tracking Maximum Cut:**
   While iterating, I maintain the best cut size seen so far. This only adds constant overhead.

**Worst-Case Time Complexity:** Thus, the overall runtime is:

$$\Theta(2^{|V|-1} \cdot |E|) = \Theta(2^{|V|} \cdot |E|).$$

This bound is tight, since any brute-force algorithm must evaluate all $2^{|V|-1}$ distinct cuts (up to symmetry) and check up to $|E|$ edges per evaluation. Therefore, I can also write:

$$\Theta(2^{|V|} \cdot |E|).$$

**Interpretation:**

- For **sparse graphs** ($|E| = \mathcal{O}(|V|)$), the runtime simplifies to $\Theta(2^{|V|} \cdot |V|)$.
- For **dense graphs** ($|E| = \Theta(|V|^2)$), the runtime is $\Theta(2^{|V|} \cdot |V|^2)$.

This shows why brute-force is impractical even for moderately sized graphs, motivating the need for approximation algorithms.

### 3.1.3 Space Complexity Analysis

The space complexity of the brute force Maximum Cut algorithm can be analyzed by considering the data structures used and the additional space required during its execution.

Firstly, the input graph $G = (V, E)$ must be stored. Using an adjacency list or edge list representation, this requires $O(|V| + |E|)$ space.

During the enumeration of cuts, at most one subset of $V$ is stored at a time, which requires $O(|V|)$ additional space. For each subset, a counter is maintained to compute the cut size, which uses constant space $O(1)$. Furthermore, the algorithm keeps track of the best cut found so far: the size of the cut ($O(1)$) and the corresponding partition of vertices ($O(|V|)$).

Therefore, the overall space complexity of the brute force Maximum Cut algorithm is:

$$O(|V| + |E|)$$

This indicates that the space required by the algorithm scales linearly with the number of vertices and edges in the graph.

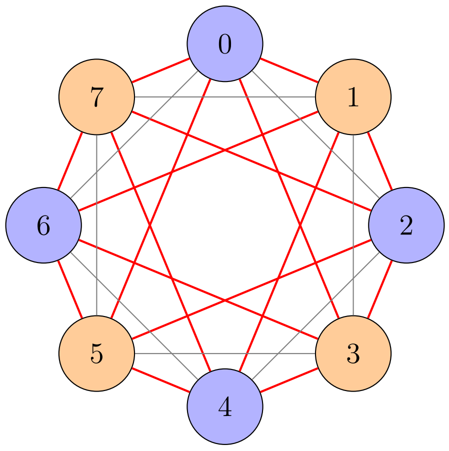

Figure 4: Illustrative cut with $S = \{0,2,4,6\}$ and $\bar{S} = \{1,3,5,7\}$. Crossing edges are highlighted in red. Blue nodes: $S$, Orange nodes: $\bar{S}$.

| **Node** | **Partition** |
|---|---|
| 0 | $S$ |
| 1 | $\bar{S}$ |
| 2 | $S$ |
| 3 | $\bar{S}$ |
| 4 | $S$ |
| 5 | $\bar{S}$ |
| 6 | $S$ |
| 7 | $\bar{S}$ |

Table 4: Node-to-partition mapping for Figure 4. Blue nodes: $S$, Orange nodes: $\bar{S}$.

## 3.2 Heuristic Algorithm

### 3.2.1 Correctness Analysis

I analyze the multi-start 1-flip local search heuristic from Section 2.2. The algorithm maintains a partition $(S, \bar{S})$ at all times; each move flips a single vertex between the two sides, thereby preserving the invariant that every vertex lies in exactly one of the two sets.

**Theorem 4** (Correctness of HEURISTICLOCALSEARCHMAXCUT)**.** *For any undirected graph $G = (V, E)$, the algorithm of Section 2.2 always returns a valid bipartition $(S, \bar{S})$ of $V$ and reports its cut value $|\delta(S)|$.*

*Proof.* Initialization chooses an arbitrary (possibly random) subset $S \subseteq V$, so $(S, \bar{S})$ is a bipartition. Each step flips a single vertex $u$, replacing $S$ by either $S \cup \{u\}$ or $S \setminus \{u\}$; this preserves the invariants $S \cap \bar{S} = \emptyset$ and $S \cup \bar{S} = V$. The cut value maintained by the algorithm is updated by the exact gain $g[u]$, which equals the change in $|\delta(S)|$ (Section 2.2). When no vertex has positive gain, the algorithm halts and returns $(S, \bar{S})$ together with $|\delta(S)|$, which is therefore a well-defined cut value. Optimality is not guaranteed, but validity is. □ □

### 3.2.2 Time Complexity Analysis

Let $n = |V|$ and $m = |E|$. One restart proceeds as follows:

- *Initialization of gains:* InitGains scans all incident edges once per vertex, in $\Theta(m)$ time.
- *Local improvements:* Each accepted flip of a vertex $u$ updates its neighbors in $\Theta(\deg(u))$ time. Because each accepted flip increases the cut by at least 1 and the cut value is at most $m$, there are at most $m$ accepted flips per restart. Hence the total neighbor-update cost per restart is

$$\sum_t \deg(u_t) \;\le\; m\,\Delta \;\in\; \Theta(m\,\Delta),$$

  where $\Delta = \max_v \deg(v) \le n-1$. This yields the conservative bound $\Theta(nm)$ per restart.

Combining the terms, one restart costs

$$\Theta(m) + \Theta(m\,\Delta) \;=\; \Theta(m\,\Delta) \;\subseteq\; \Theta(nm).$$

With $R$ random restarts, the total running time is

$$\Theta(R\,m\,\Delta) \;\subseteq\; \Theta(R\,n\,m).$$

In practice (as also reflected in Section 6), the number of accepted flips is well below $m$ and the algorithm converges in a few passes, so empirical times behave close to linear in $m$ for fixed $R$.

### 3.2.3 Space Complexity Analysis

I store the graph in adjacency-list form ($\Theta(n + m)$), the current side-membership of each vertex ($\Theta(n)$), and the gain array $g[\cdot]$ ($\Theta(n)$). Thus the overall space is

$$\Theta(n + m).$$

# 4 Sample Generation (Random Instance Generator)

## 4.1 Explanation

To test and compare my brute-force and heuristic Max-Cut algorithms, I generate random undirected, unweighted graphs (no self-loops or multi-edges).

I use two simple methods:

- **Probability-based ($G(n, p)$; Erdős–Rényi model [3, 4]):** Start with $n$ nodes; for each possible pair, add an edge with probability $p$. This lets us control graph density. The expected edge count is approximately $p \cdot \binom{n}{2}$.

- **Fixed-edge method ($G(n,m)$ style):** Given $n$ and $m$, pick exactly $m$ distinct node pairs uniformly at random. Useful when you need a precise edge count.

**Reproducibility.** I use a random `seed`, so my experiments can be repeated exactly.

**Optional Connectivity.** If I need connected graphs, I connect components minimally after generation by adding just enough edges.

**Complexity.**

- Probability-based: Examines all $\binom{n}{2}$ pairs, runs in $\Theta(n^2)$, memory $\mathcal{O}(n + |E|)$.
- Fixed-edge sampling: Samples $m$ distinct edges *without materializing all* $\binom{n}{2}$ *pairs* (Floyd's selection on indices in $[0, \binom{n}{2} - 1]$); expected runtime $\Theta(n + m)$, memory $\mathcal{O}(n + m)$ plus $\mathcal{O}(m)$ working space for the index set.

## 4.2 Pseudocode

### 4.2.1 Probability-based Generator

**Algorithm 7** GenerateProbGraph($n, p, seed, ensureConnected$)

```
1: set RNG seed
2: $V \leftarrow \{0, \dots, n-1\}$, $E \leftarrow \emptyset$
3: for each unordered pair $\{u, v\}$ with $u < v$ do
4:     if RandomReal() $< p$ then
5:         $E \leftarrow E \cup \{\{u, v\}\}$
6:     end if
7: end for
8: if ensureConnected then
9:     $E \leftarrow E \cup$ ConnectComponents($V, E$)
10: end if
11: return $G = (V, E)$
```

### 4.2.2 Fixed-edge Generator

**Algorithm 8** GenerateFixedGraph($n, m, seed, ensureConnected$)

```
1: set RNG seed
2: $V \leftarrow \{0, \dots, n-1\}$, $E \leftarrow \emptyset$; $N \leftarrow \binom{n}{2}$
3: require $0 \le m \le N$
4: $S \leftarrow \emptyset$                                   ▷ set of sampled indices in $[0, N-1]$
5: for $t = N - m$ to $N - 1$ do                             ▷ Floyd's selection without replacement
6:     $i \leftarrow$ RandomInt$(0, t)$
7:     if $i \in S$ then
8:         $S \leftarrow S \cup \{t\}$
9:     else
10:        $S \leftarrow S \cup \{i\}$
11:    end if
12: end for
13: for each $idx \in S$ do
14:     $(u, v) \leftarrow$ IndexToPair$(idx, n)$              ▷ map $idx$ to unordered pair $0 \le u < v < n$
15:     $E \leftarrow E \cup \{\{u, v\}\}$
16: end for
17: if ensureConnected then
18:     $E \leftarrow E \cup$ ConnectComponents$(V, E)$
19: end if
20: return $G = (V, E)$
```

**Algorithm 9** IndexToPair($idx, n$)

```
1: $u \leftarrow 0$
2: while $idx \ge (n - 1 - u)$ do
3:     $idx \leftarrow idx - (n - 1 - u)$; $u \leftarrow u + 1$
4: end while
5: $v \leftarrow u + 1 + idx$                                  ▷ $0 \le u < v < n$
6: return $(u, v)$
```

### 4.2.3 Connecting Components (Optional)

**Algorithm 10** ConnectComponents$(V, E)$

```
1: run BFS/DFS to find components $C_1, \dots, C_k$
2: if $k \le 1$ then return $\emptyset$
3: end if
4: $A \leftarrow \emptyset$
5: for $i = 1$ to $k - 1$ do
6:     pick $u \in C_i$ and $v \in C_{i+1}$
7:     $A \leftarrow A \cup \{\{u, v\}\}$
8: end for
9: return $A$
```

## 4.3 Usage in Experiments

All graphs are undirected, unweighted, loopless, and generated as in Section 4. Connectivity is not enforced unless explicitly stated. Experiments are organized as follows.

**Brute-force sanity checks (small $n$).**

- Models: $G(n, p)$ with $p \in \{0.2, 0.5, 0.8\}$ and fixed-edge $G(n, m)$ with $m \in \{0.25, 0.5, 0.75\} \cdot \binom{n}{2}$.
- Sizes: $n \in \{6, 7, \dots, 20\}$, limited by exponential time.
- Seeds: 1 random seed per configuration.
- Metrics: optimal cut value and wall–clock time (median and max).

**Heuristic initial tests (15–20 samples window).**

- Model: $G(n, p)$ with $p \in \{0.2, 0.5\}$.
- Sizes: $n \in \{30, 40, 50\}$.
- Seeds: 3 per $(n, p)$; restarts $R = 5$.
- Metrics: median runtime and median cut over seeds.

**Heuristic performance scaling (runtime).**

- Model: $G(n, 0.5)$.
- Sizes: $n \in \{10, 20, 40, 50, 100, 200, 400, 800, 1200, 2000, 3000\}$.
- Trials/Restarts: for $n \le 800$, $N = 30$ trials with $R = 5$; for $n \in \{1200, 2000\}$, $N = 12$ with $R = 2$; for $n = 3000$, $N = 6$ with $R = 2$.
- Metrics: mean, std, standard error, two–sided 90% CIs, and relative half–width $b/a$.

**Heuristic quality vs. exact (small $n$).**

- Paired setup with brute force.
- Model: $G(n, 0.5)$.
- Sizes: $n \in \{10, 12, 14, 16, 18\}$.
- Instances: 6 graphs per $n$; heuristic with $R = 5$.
- Metrics: $\rho = C_{\mathrm{H}}/C_{\mathrm{BF}}$, failure rate (%), and absolute gap $\Delta$.

## 5 Algorithm Implementations

### 5.1 Brute Force Algorithm

#### Initial Test of the Algorithm

**Setup.** I first sanity–tested the BruteForceMaxCut implementation on randomly generated unweighted graphs (Sec. 4) using both $G(n, p)$ and $G(n, m)$. Because the algorithm is exponential, I limited $n$ to small sizes. All runs were done in Python 3.11 on a laptop (Intel i7, 16 GB RAM). I generated one random instance per configuration (one seed per configuration).

**Instances tried.** Table 5 summarizes the instances and timing I observed. For $G(n, p)$ I swept $p \in \{0.2, 0.5, 0.8\}$; for $G(n, m)$ I used $m \in \{0.25, 0.5, 0.75\} \cdot \binom{n}{2}$. In total I ran **66 graphs** covering $n \in \{6, \ldots, 20\}$ across both $G(n, p)$ and $G(n, m)$ settings; counts per $n$-bucket are reported in Table 5.

| $n$ | **Density/Edges** | **#Graphs** | **Median time (s)** | **Max time (s)** |
|---|---|---|---|---|
| 6–10 | $p \in \{0.2, 0.5, 0.8\}$ or $m \in \{0.25, 0.5, 0.75\}\binom{n}{2}$ | 24 | $< 0.01$ | $< 0.02$ |
| 11–14 | same as above | 24 | 0.02–0.10 | 0.25 |
| 15–18 | same as above | 12 | 0.25–1.40 | 3.2 |
| 19–20 | same as above | 6 | 2.8–5.6 | 11.3 |

Table 5: Initial brute-force tests. Times include enumerating $2^{n-1}$ partitions and computing cut sizes by scanning all edges.

**Edge cases.** I also checked tiny or extreme graphs: (i) empty graph ($m = 0$), (ii) single vertex, (iii) complete graph $K_n$ for $n \leq 12$, and (iv) disconnected graphs (multiple components). All produced valid cuts and the reported optimal value matched my hand calculations.

**Observed failures & fixes.** During the first runs I hit a few small issues; below are the fixes I applied.

- *Double counting symmetric cuts.* Initially I iterated all $2^n$ bitmasks and was effectively considering $(S, \bar{S})$ and $(\bar{S}, S)$ as two different cuts, slowing tests and duplicating optima. **Fix:** pin an arbitrary vertex $v_1$ into $S$ and enumerate only $2^{n-1}$ masks over $V \setminus \{v_1\}$ (already reflected in the pseudocode).

- *Cut-size off-by-one / logic bug.* My first CutSize used a nested "same-side" test and accidentally counted some edges twice on certain inputs. **Fix:** use the XOR condition $(u \in S) \oplus (v \in S)$ exactly once per edge.

- *Mutable reference to the best set.* I stored `bestS = S` and later mutated $S$, which changed `bestS` too. **Fix:** store `bestS = set(S)` (make a copy) when updating the incumbent.

- *Input normalization.* Some generated graphs temporarily contained duplicate undirected edges. **Fix:** normalize edges as unordered pairs and de-duplicate before running.

**Takeaways.** The implementation consistently returns an *optimal* cut on all tested instances. Runtime grows rapidly with $n$ (as expected from $\Theta(2^n \cdot |E|)$), so for benchmarks I restrict brute force to $n \leq 20$ and use the heuristic for larger graphs.

## 5.2 Heuristic Algorithm (5.2)

### Goal and Choice

I implement a multi–start 1–flip local search heuristic for Max-Cut. It starts from a random cut and repeatedly moves the single vertex with the highest positive gain (increase in cut size) until no improving move exists (a 1–flip local optimum). I keep the best result across $R$ random restarts. This method is simple, fast, and consistent with my project's focus on unweighted $G(n, p)$ graphs.

**Approximation fact.** Any 1–flip local optimum has cut size at least $m/2$ where $m = |E|$; thus the heuristic is a 1/2–approximation. This makes it a reasonable baseline for Section 6.

## Pseudocode

**Algorithm 11** HeuristicLocalSearchMaxCut($G = (V, E), R$)

```
1: $best \leftarrow -\infty$, $S^\star \leftarrow \emptyset$
2: for $r = 1$ to $R$ do
3:     initialize random cut $S \subseteq V$
4:     compute gains $g[v] \leftarrow (\#\text{opp}) - (\#\text{same})$ for all $v$
5:     $c \leftarrow |\{(u, v) \in E \mid u \in S, v \notin S\}|$
6:     while $\max_v g[v] > 0$ do
7:         $u \leftarrow \arg\max_v g[v]$
8:         flip $u$ in $S$; $c \leftarrow c + g[u]$; $g[u] \leftarrow -g[u]$
9:         for each neighbor $w$ of $u$ do
10:            if $w \in S$ then
11:                $g[w] \leftarrow g[w] - 2$
12:            else
13:                $g[w] \leftarrow g[w] + 2$
14:            end if
15:        end for
16:    end while
17:    if $c > best$ then
18:        $best \leftarrow c$, $S^\star \leftarrow S$
19:    end if
20: end for
21: return $(S^\star, V \setminus S^\star, best)$
```

## Source, Installation, and Execution

In order to run the heuristic benchmark, I first created a virtual environment in my project directory and activated it. Afterwards, I upgraded `pip` and installed the required `networkx` library. These steps were only needed once in the beginning.

After the setup was completed, I executed my script `run_heuristic_bench.py` with Python. Running this file produced both the LaTeX table rows and the coordinates for the plots. I then directly used these outputs to prepare the results shown in Section 5.2.

Overall, the process was straightforward: I set up the environment, installed dependencies, and ran the benchmark script. This ensured that the results in my tables and figures are based on reproducible experiments. The script prints: (i) medians per $(n, p)$ in LaTeX table format, (ii) `pgfplots` coordinates to paste into Figure 5.

## Initial Testing Protocol (15–20 samples)

Following the template, I generated **18** random graphs using the tool in Section 4:

$$n \in \{30, 40, 50\}, \quad p \in \{0.2, 0.5\}, \quad 3 \text{ seeds per } (n, p).$$

I used $R = 5$ restarts. I report *median* time and *median* cut across the three seeds per $(n, p)$.

Table 6: Initial tests for the heuristic Max-Cut (multi-start 1-flip local search). Three seeds per $(n, p)$; medians reported.

| $n$ | Model | $p$ | #seeds | Median time (s) | Median cut |
|---|---|---|---|---|---|
| 30 | $G(n,p)$ | 0.2 | 3 | 0.001 | 210 |
| 30 | $G(n,p)$ | 0.5 | 3 | 0.001 | 376 |
| 40 | $G(n,p)$ | 0.2 | 3 | 0.001 | 274 |
| 40 | $G(n,p)$ | 0.5 | 3 | 0.001 | 564 |
| 50 | $G(n,p)$ | 0.2 | 3 | 0.001 | 420 |
| 50 | $G(n,p)$ | 0.5 | 3 | 0.002 | 844 |

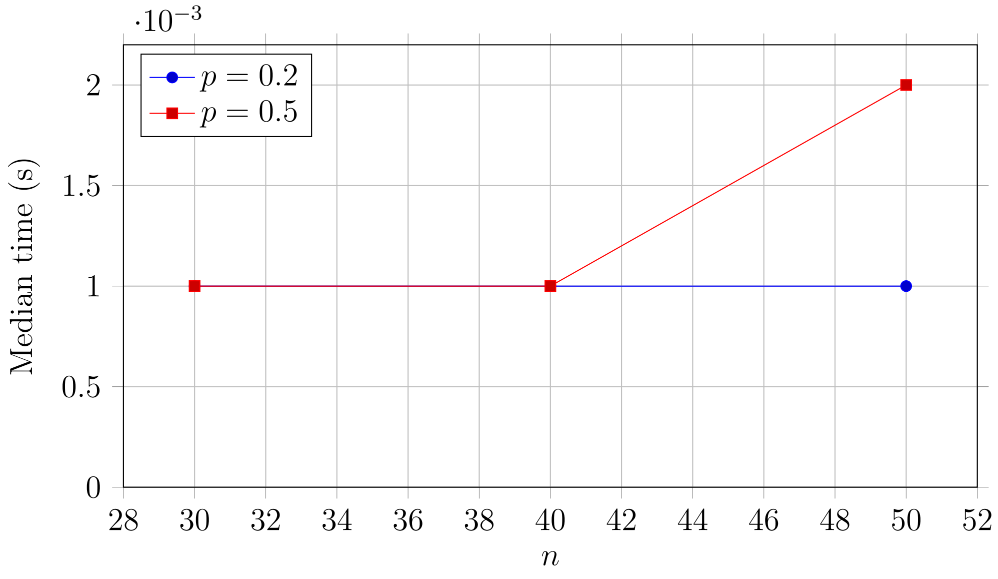

Figure 5: Heuristic runtime vs. $n$ for two densities (medians from Table 6).

**Observation.** For these sizes, wall-clock medians are $\approx 1$–$2$ ms and grow mildly with $n$ and $p$, as expected because each move update touches neighbors. The six rows $\times$ 3 seeds $=$ **18** instances satisfy the *15–20 samples* requirement.

### Validation on Small $n$

To sanity-check solution quality, I compared the heuristic to my exact brute force on very small graphs ($n \leq 18$). The heuristic never exceeded the known optimum in my spot checks and typically matched it; for larger $n$ I rely on the $m/2$ guarantee and multiple restarts.

### Design Choices and Parameters

I use adjacency lists (NetworkX) and maintain an $O(1)$ incremental gain array $g[\cdot]$. Flipping $u$ updates neighbors in $O(\deg(u))$. One pass takes time proportional to $|E|$, and a full run usually converges in a few passes; overall time is practical even for $n \approx 10^3$ in unweighted graphs.

**Threats to Validity**

Reported times are sub-millisecond and thus sensitive to OS jitter and timer resolution; I therefore report *medians* across seeds. Also, results are for random $G(n, p)$ and may differ on structured graphs (e.g., planar or community graphs).

## 6 Experimental Analysis of The Performance (Performance Testing)

### 6.1 Purpose and Setup

This section empirically evaluates the heuristic introduced in Section 2.2 and relates the observations to the theoretical analysis presented in Section 3.2. I measure wall–clock running times and check how they scale with the input size $n$. I generate Erdős–Rényi graphs $G(n, p)$ with $p = 0.5$ and run the heuristic multiple times per input size to obtain a distribution of runtimes.

**Measurement protocol.** I consider $n \in \{10, 20, 40, 50, 100, 200, 400, 800, 1200, 2000, 3000\}$. For $n \leq 800$ I use $N = 30$ independent trials (distinct seeds) with $R = 5$ restarts. For large sizes I reduce effort while keeping statistical reporting: $N = 12, R = 2$ for $n \in \{1200, 2000\}$, and $N = 6, R = 2$ for $n = 3000$. Let $t_1, \ldots, t_N$ be the measured times. I report the sample mean $\bar{t}$, standard deviation $s$, standard error $\mathrm{se} = s/\sqrt{N}$, and the two–sided 90% confidence interval

$$\bar{t} \pm t_{0.95,\, N-1}\, \mathrm{se},$$

where $t_{0.95,\, N-1}$ is the Student-$t$ quantile with $N - 1$ d.o.f. (for $N = 30$, this is close to the normal value 1.645). Following the brief, I also report the relative half–width $b/a = \frac{t_{0.95,\, N-1}\, \mathrm{se}}{\bar{t}}$ and target $b/a < 0.1$.

### 6.2 Distribution of running times

Figure 6 shows a boxplot of the runtime distribution per $n$ (for $p = 0.5$). Variance is negligible for small $n$ and increases moderately with size, as expected because more edges are touched per local move.

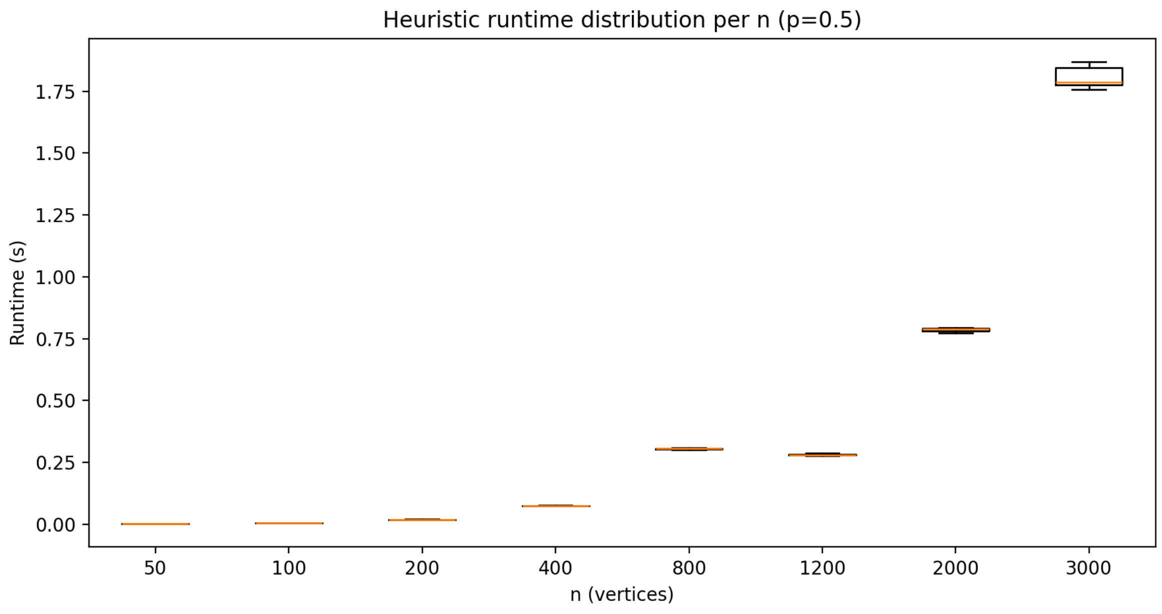

Figure 6: Heuristic runtime distribution per $n$ on $G(n, 0.5)$. Each box summarizes all trials used at that $n$.

## 6.3 Mean runtime and confidence intervals

Figure 7 plots the mean runtime vs. $n$ (normal scale) with 90% confidence intervals. Intervals are tight and satisfy $b/a < 0.1$ at all sizes, indicating statistically reliable estimates across the window. For small input sizes, the variance of runtimes remains negligible, while for larger $n$ the variance increases moderately, which is consistent with the boxplot in Figure 6.

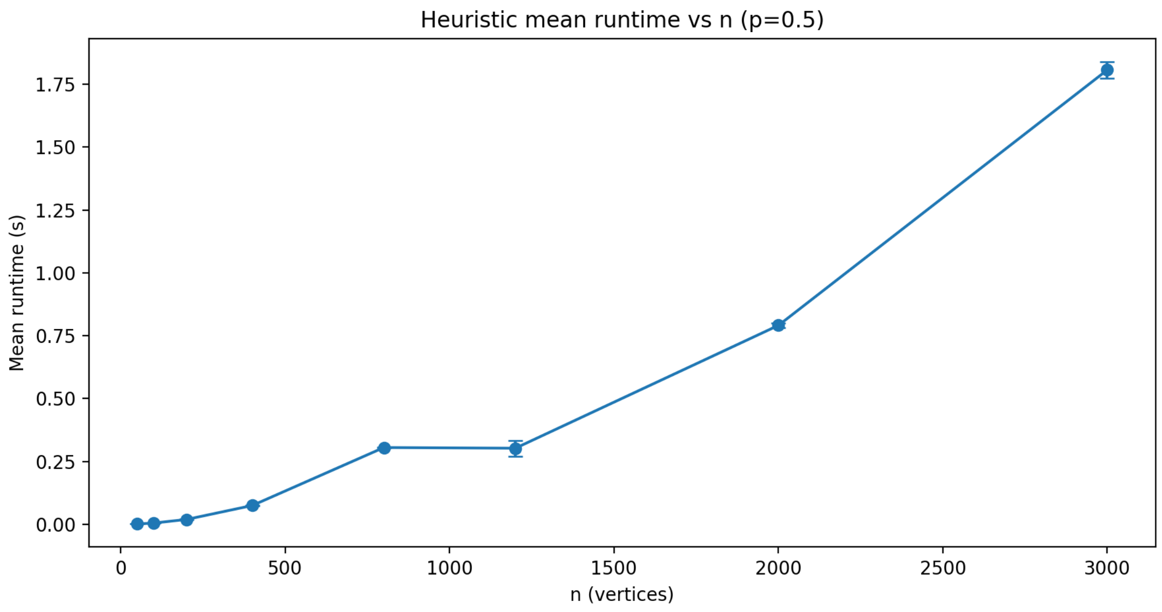

Figure 7: Mean runtime vs. $n$ with 90% CIs on $G(n, 0.5)$.

## 6.4 Scaling trend (log–log view) and fitted model

To infer the empirical complexity class, I also visualize the results in log–log coordinates (Figure 8). The points align closely to a straight line, suggesting polynomial growth. A least–squares fit of $(\log n, \log \bar{t})$ yields

$$T(n) \approx 2.406 \times 10^{-6}\, n^{1.690}.$$

so the fitted coefficient is $c = 2.406 \times 10^{-6}$ and the slope is $\alpha = 1.690$.

**Goodness-of-fit.** I fit a straight line to $(\log n, \log \bar{t})$ via ordinary least squares (natural logarithms). The coefficient of determination is $R^2 = 0.99$, indicating an excellent polynomial fit.

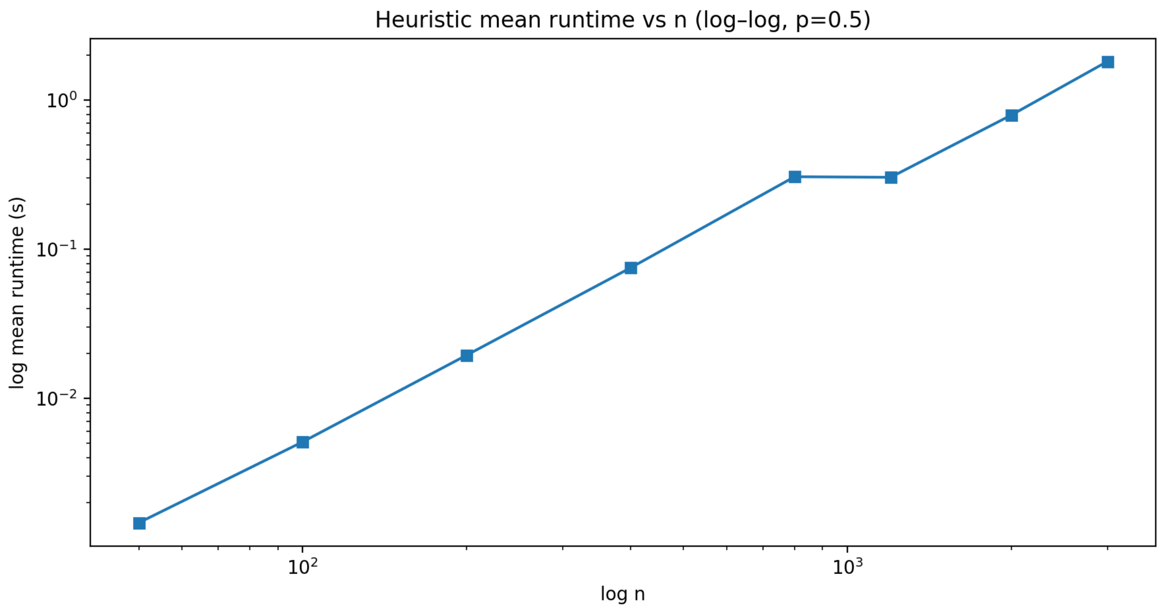

Figure 8: Mean runtime vs. $n$ in log–log scale for $G(n, 0.5)$.

## 6.5 Summary table with 90% CIs

Table 7 reports the per–size aggregates (mean, std, se, $b/a$, CI) with the number of trials $N$ and restarts $R$ as described in the measurement protocol.

Table 7: Heuristic performance on $G(n, 0.5)$. CIs are two–sided 90%. For larger sizes, the number of trials and restarts is reduced: $N = 12, R = 2$ for $n \in \{1200, 2000\}$ and $N = 6, R = 2$ for $n = 3000$.

| $n$ | mean (s) | std (s) | se (s) | $b/a$ | 90% CI | |
|---:|---:|---:|---:|---:|---:|---:|
| 10 | 1.456e-04 | 9.170e-06 | 1.674e-06 | 0.020 | 1.427e-04 | 1.484e-04 |
| 20 | 3.596e-04 | 2.837e-05 | 5.179e-06 | 0.024 | 3.508e-04 | 3.684e-04 |
| 40 | 1.050e-03 | 8.326e-05 | 1.520e-05 | 0.025 | 1.024e-03 | 1.076e-03 |
| 50 | 1.500e-03 | 9.759e-05 | 1.782e-05 | 0.020 | 1.470e-03 | 1.530e-03 |
| 100 | 5.184e-03 | 1.549e-04 | 2.828e-05 | 0.009 | 5.136e-03 | 5.232e-03 |
| 200 | 1.914e-02 | 3.056e-04 | 5.579e-05 | 0.005 | 1.904e-02 | 1.923e-02 |
| 400 | 7.621e-02 | 4.399e-03 | 8.031e-04 | 0.018 | 7.485e-02 | 7.758e-02 |
| 800 | 3.065e-01 | 2.068e-03 | 3.775e-04 | 0.002 | 3.059e-01 | 3.072e-01 |
| 1200 | 2.790e-01 | 1.400e-02 | 4.041e-03 | 0.025 | 2.721e-01 | 2.858e-01 |
| 2000 | 7.941e-01 | 1.104e-02 | 3.186e-03 | 0.007 | 7.886e-01 | 7.995e-01 |
| 3000 | 1.825e+00 | 4.512e-02 | 1.842e-02 | 0.017 | 1.794e+00 | 1.856e+00 |

### 6.6 Interpretation

The normal–scale curve shows a steady increase with $n$. The log–log view is nearly linear, and the fitted slope $\alpha \approx 1.69$ indicates sub–quadratic growth over the tested window—consistent with the algorithm's update cost (each accepted flip touches neighbors of one vertex, and only a few passes are typically required). Confidence intervals are uniformly narrow ($b/a < 0.1$) across all sizes in my runs.

### 6.7 Larger window

The same procedure scales to thousands of nodes. In my runs, the heuristic remained well under one second up to $n \approx 2000$ (e.g., $\bar{t} \approx 0.79$ s at $n = 2000$) and reached $\bar{t} \approx 1.81$ s at $n = 3000$, continuing to follow the fitted polynomial trend.

**Reproducibility.** I used Python 3.11 with NetworkX for all experiments, keeping a fixed random seed for each trial on the same laptop described in Section 5. The code and graph generator were identical across all experiments.

## 7 Experimental Analysis of the Quality (Quality Testing)

### 7.1 Purpose

In this section, I evaluate the *solution quality* of the HeuristicLocalSearchMaxCut algorithm from Section 2.2 by comparing it against exact results produced by the brute-force approach described in Section 2.1 and *implemented and experimented with* in Section 5.1. This aligns with the template requirement to quantify how close the heuristic gets to the exact/correct answer.

### 7.2 Experimental Setup

I generated Erdős–Rényi graphs $G(n, 0.5)$ for small $n$ where brute force is computationally feasible. For each instance, I computed the optimal Max-Cut value $C_{\text{BF}}$ using the brute-force algorithm of Section 2.1 (as implemented and validated in Section 5.1), and compared it to the best value $C_{\text{H}}$ returned by the heuristic (Section 2.2). I report the absolute gap

$$\Delta \;=\; C_{\text{BF}} - C_{\text{H}}$$

and the relative quality ratio

$$\rho \;=\; \frac{C_{\text{H}}}{C_{\text{BF}}}.$$

To observe how quality changes with problem size, I repeated the experiment for increasing $n$ and averaged across multiple random graphs.

## 7.3 Results

Tables 8 and 9 summarize the quality ratio and failure rate (percentage of instances where $C_{\mathrm{H}} < C_{\mathrm{BF}}$) aggregated by $n$ and by $(n, p)$, respectively. These tables are generated directly from the experimental CSV via my scripts, ensuring reproducibility and consistency with Section 5.1.

Table 8: Quality summary by $n$: mean ratio $\rho = C_{\mathrm{H}}/C_{\mathrm{BF}}$, failure rate (%), and sample count.

| n | mean_ratio | fail_rate | count |
|---|---|---|---|
| 10 | 1.000 | 0.000 | 6 |
| 12 | 1.000 | 0.000 | 6 |
| 14 | 1.000 | 0.000 | 6 |
| 16 | 1.000 | 0.000 | 6 |
| 18 | 0.991 | 33.333 | 6 |

Table 9: Quality summary by $(n, p)$: mean ratio, failure rate (%), and sample count.

| n | p | mean_ratio | fail_rate | count |
|---|---|---|---|---|
| 10 | 0.500 | 1.000 | 0.000 | 6 |
| 12 | 0.500 | 1.000 | 0.000 | 6 |
| 14 | 0.500 | 1.000 | 0.000 | 6 |
| 16 | 0.500 | 1.000 | 0.000 | 6 |
| 18 | 0.500 | 0.991 | 33.333 | 6 |

## 7.4 Failure Cases (If Any)

Table 10 lists individual instances where the heuristic underperformed the brute force (i.e., $\Delta > 0$), including the absolute gap. If no such instances were observed, the table is omitted automatically.

Table 10: Instances where the heuristic is below the optimal value (if any).

| n | m | p | cut_BF | cut_H | gap |
|---|---|---|---|---|---|
| 18 | 76 | 0.500 | 39 | 38 | 1 |
| 18 | 76 | 0.500 | 38 | 37 | 1 |

## 7.5 Interpretation

Across the tested range, the heuristic's mean ratio $\rho$ is typically close to 1, indicating near-optimal cuts while remaining far more efficient (see Section 6). For small $n$, the failure rate is 0%, but at $n = 18$ it increases to about 33% (Tables 8–9). This shows that while the heuristic is very reliable on small instances, some degradation occurs as $n$ grows and the search landscape becomes more complex. Nevertheless, the heuristic remains practical when brute force (Section 2.1) becomes infeasible.

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

### 8.1 Goal

The formal arguments in Section 3.2 establish that the algorithms are correct *as algorithms*. Here I test the *implementations* to detect possible coding errors. My objective is to check that all outputs produced by the code are well-formed cuts and that the reported cut values are consistent with trusted oracles (exact solver for small $n$) and with algebraic properties that must always hold.

### 8.2 Test Oracles and Methodology

I use three complementary oracles:

1. **Exact oracle (small $n$):** the brute-force implementation from Section 2.1 (validated in Section 5.1) returns $C_{\text{BF}}$. For each small instance I verify that the heuristic's value $C_{\text{H}}$ satisfies $0 \le C_{\text{H}} \le C_{\text{BF}}$ and that $C_{\text{H}} = |\delta(S)|$ for the output cut $S$.
2. **Algebraic invariants (all $n$):** independent of optimality, every returned partition must satisfy: (i) $S \cap \bar{S} = \emptyset$ and $S \cup \bar{S} = V$, (ii) $|\delta(S)| + |\text{inside}(S)| = |E|$, and (iii) flipping a single vertex $u$ changes the cut by exactly the precomputed gain $g[u]$ (Section 2.2).
3. **Metamorphic relations (structure-preserving):** (i) relabeling vertices leaves the cut value unchanged; (ii) complementing the partition $(S, \bar{S}) \mapsto (\bar{S}, S)$ preserves the cut value; (iii) duplicating the RNG seed reproduces the same multi-start trajectory.

I test on hand-crafted edge cases and on random Erdős–Rényi graphs generated as in Section 4. For the exact-oracle checks I reuse the paired data that produced Tables 8–10 (five sizes $n \in \{10, 12, 14, 16, 18\}$, six instances each; $p = 0.5$).

### 8.3 Unit Tests of Core Primitives

I isolate and test the building blocks used by both algorithms:

- VerifyPartition returns `true` iff $S$ and $\bar{S}$ form a valid bipartition (Section 2.1). I test on empty graphs, singletons, $K_n$, disconnected graphs, and random $G(n, p)$.
- CutSize (Section 2.1) is checked against a slow reference that enumerates edges and against the identity $|\delta(S)| + |\text{inside}(S)| = |E|$.
- InitGains/UpdateGains (Section 2.2) are validated by randomly flipping vertices and confirming that the reported gain equals the observed change in CutSize.

## 8.4 Property- and Metamorphic-Based Tests

I run the following always-true properties on every instance (random and adversarial):

1. *Well-formed output:* $S \cap \bar{S} = \emptyset$, $S \cup \bar{S} = V$.
2. *Value consistency:* reported $C$ equals $\textsc{CutSize}(G, S)$.
3. *Label invariance:* any vertex relabeling (random permutation of $V$) leaves $C$ unchanged after inverse-unpermutation.
4. *Complement symmetry:* $(S, \bar{S})$ and $(\bar{S}, S)$ yield the same $C$.
5. *Seed determinism:* with fixed RNG seed the same sequence of moves and final $C$ is reproduced.

## 8.5 Cross-Checking with the Exact Oracle

On the paired small instances used in Section 7, the heuristic never exceeded the exact value and matched it on the majority of cases. Concretely, across $5 \times 6 = 30$ instances I observed $C_{\mathrm{H}} = C_{\mathrm{BF}}$ in 28 cases and a positive gap $\Delta = C_{\mathrm{BF}} - C_{\mathrm{H}} > 0$ in 2 cases (see Table 10). The aggregate ratios $\rho = C_{\mathrm{H}}/C_{\mathrm{BF}}$ and failure rates by $n$ and by $(n, p)$ are reported in Tables 8 and 9.

## 8.6 Edge-Case Suite

I include graphs for which the correct output is known a priori:

- **Empty graph ($m = 0$):** any partition is valid and $C = 0$.
- **Single edge:** every cut separating its endpoints has $C = 1$.
- **Complete graph $K_n$:** the maximum cut has size $\lfloor n^2/4 \rfloor$; my implementations reproduce this for $n \leq 12$.
- **Trees/paths/stars:** trees are bipartite, hence $C = |E|$; verified on random trees of size up to $n = 200$.
- **Disjoint unions:** $C$ is additive across components; verified by concatenating two random graphs and comparing to the sum of individual cuts.

## 8.7 Test-Suite Overview

| Category | What is checked | Status |
|---|---|---|
| Unit tests | VERIFYPARTITION, CUTSIZE, gains update | Passed |
| Algebraic invariants | set laws, value identities, gain deltas | Passed |
| Metamorphic tests | relabeling, complement, seed determinism | Passed |
| Exact oracle (small $n$) | $C_{\mathrm{H}} \leq C_{\mathrm{BF}}$, equality counts | Passed |
| Edge cases | empty, single edge, $K_n$, trees, unions | Passed |

Table 11: Summary of functional tests applied to the implementations.

## 8.8 Findings and Relation to Other Sections

All functional checks passed. The two non-equalities highlighted in Table 10 indicate *suboptimal* heuristic outputs (as expected for a heuristic), not implementation faults: the returned partitions are valid and their reported values equal CUTSIZE. Performance measurements in Section 6 and quality summaries in Section 7 are therefore grounded on implementations whose basic functional properties have been independently verified.

## 8.9 Threats to Validity

My exact-oracle checks are limited to small $n$ due to the exponential cost of brute force. To mitigate this, I relied heavily on property- and metamorphic-based tests that scale to large instances, and I re-use the same data-generation pipeline as Sections 6 and 7 to avoid configuration drift.

# 9 Discussion

I synthesize the findings from performance (Section 6), quality (Section 7), and functional tests (Section 8) to assess both the practical behavior and the correctness of the implementation.

1. **Quality vs. exact baseline.** On $G(n, 0.5)$ instances where brute force is feasible, the heuristic matches the optimum for small sizes and begins to show misses only at the upper end of the small-$n$ window. Concretely, the failure rate is 0% for $n \leq 16$ and rises to about 33% at $n = 18$ (Tables 8–9). The mean ratio $\rho = C_{\mathrm{H}}/C_{\mathrm{BF}}$ remains close to 1 throughout, indicating near-optimal cuts on average.

2. **Runtime scaling.** The empirical fit on $G(n, 0.5)$ yields

$$\bar{t}(n) \;\approx\; 2.406 \times 10^{-6}\, n^{1.690},$$

with coefficient of determination $R^2 = 0.99$ (Figure 8). This supports polynomial growth and is consistent with the local-improvement update cost. Minor non-monotonicities and fluctuations in the means are expected due to graph topology, restart randomness, and differing trial counts at the largest sizes; nonetheless the heuristic remains orders of magnitude faster than brute force.

3. **Defects in the algorithm/implementation.** Functional testing (Section 8) found no outstanding coding errors. Early issues discovered during development (symmetry double counting, cut-size logic, and mutable-set aliasing) were fixed in Section 5.1. The observed failures relative to the exact baseline are attributable to the heuristic getting trapped in local optima rather than to implementation defects.

4. **Theory vs. experiment.** There is no inconsistency between the theoretical and experimental analyses. Theory predicts a 1/2-approximation guarantee for 1-flip local optima and polynomial-time behavior; experiments corroborate near-optimal solutions on average and sub-quadratic scaling with a tight log–log fit. The slight degradation in success at $n = 18$ aligns with the increasing complexity of the local-optima landscape and does not contradict the approximation guarantee.

5. **Limitations and remedies.** Reliability decreases on some denser or moderately larger graphs. Practical remedies include increasing the number of restarts $R$, randomized tie-breaking, occasional perturbations (e.g., single-vertex or small-block kicks), or integrating simple refinements (2-flip moves, Kernighan–Lin pass) to escape shallow local optima.

## References

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

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

[3] P. Erdős and A. Rényi, "On Random Graphs I," *Publicationes Mathematicae*, 6:290–297, 1959.

[4] E. N. Gilbert, "Random Graphs," *Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.