Beyond P: Understanding NP-Completeness, Reductions, and Algorithmic Strategies

AAD COURSE PROJECT: TEAM AE AE DEE

Pragati Prasad (2024113028)
Mohit Nallagatla (2024101101)
AV Aditya (2024111031)
Nandini Chakaravarthy (2024111034)
Kimaya Kashyap (2024115001)

Acknowledgements

We would like to express our sincere gratitude to everyone who guided and supported us throughout this project.

First, we thank our instructors, **Professor Kannan Srinathan** and our TA, **Harpreet Singh**, for their invaluable guidance, insightful suggestions, and constant encouragement. Their expertise helped us navigate the complexities of this never-ending field.

We are also grateful to our peers and colleagues who provided feedback, shared ideas, and collaborated in discussions that enriched our understanding of the subject.

Finally, we would like to acknowledge the resources, textbooks, and research papers that provided the foundational knowledge and examples for this project.

This work would not have been possible without the combined support, inspiration, and knowledge from all these sources.

Github Link to the Project: https://github.com/moonIsHere11/AAD_courseproject-

Abstract

This project explores a range of **NP-Complete problems**, including **3SAT, Vertex Cover, Graph Coloring, Clique, and Set Cover**, with a focus on understanding their computational complexity,

designing algorithms, and analyzing practical solutions. For each problem, we implemented **brute-force approaches** to establish baseline solutions and examined **heuristic and approximation algorithms** to achieve efficient, feasible results in practice.

A key component of the project is the study of **polynomial-time reductions** between these problems, demonstrating their theoretical interconnections and illustrating why solving one efficiently would enable solutions for others. Through these reductions, we highlight the shared structural hardness of NP-Complete problems and provide insight into designing effective heuristics.

We analyze **time and space complexities**, compare **exact versus heuristic methods**, and discuss the limitations of approximation in problems such as MAX-Clique and Graph Coloring. The project demonstrates how **greedy and saturation-based heuristics**, as well as approximation strategies like **Set Cover's logarithmic greedy algorithm**, offer practical solutions where exact methods are computationally infeasible.

Overall, this work provides a **comprehensive exploration of NP-Complete problems**, bridging theoretical concepts with practical algorithm design, and illustrating the trade-offs between **optimality, feasibility, and computational efficiency**.

Contents

1 Problem Statement

"The P vs NP question is one of the deepest and most important open problems in all of mathematics and computer science. It asks whether every problem whose solution can be verified quickly (in polynomial time, the class NP) can also be solved quickly (the class P).

Discovery of NP-complete problems was one of the biggest revolutions in computability theory. Pick 3–4 NP-complete problems (e.g. SAT, 3SAT, Subset Sum, Hamiltonian Path, Vertex Cover) and show how each reduces to another. Implement both the reduction and a brute-force solver for small cases."

2 P vs NP

"The distinction between easy and hard problems is more than just a matter of degree; it defines the boundary of what can be computed efficiently in our universe."

2.1 Defining the Classes

At the heart of computer science lies the distinction between two fundamental classes of problems: **P** and **NP**.

- **P (Polynomial time)** represents problems that we can solve efficiently, problems for which a computer can find a solution quickly, even as the problem grows large.
- **NP (Nondeterministic Polynomial time)** represents problems for which, if someone hands us a candidate solution, we can **verify** it quickly, even if finding it from scratch might take an unimaginable amount of time.

This subtle difference, between **finding a solution** and **verifying a solution**, lies at the core of the most famous open question in theoretical computer science: **P vs. NP**.

2.2 The Significance of the P vs. NP Question

Solving this **would fundamentally reshape our understanding of computation, efficiency, and problem-solving itself**.

If **P = NP**, the implications are staggering:

- Every problem for which we can **verify a solution quickly** could also be **solved quickly**.

- This means that seemingly impossible tasks, cracking codes, optimizing massive networks, discovering new drugs by searching vast chemical spaces, could suddenly become computationally trivial.
- The world as we know it would change: encryption would collapse, AI could reach unprecedented efficiency, and countless problems we thought were "intractable" could be solved in polynomial time.

If, on the other hand, **P** $\neq$ **NP**, the universe retains its computational mysteries:

- There are problems that are fundamentally hard to solve, even if their solutions can be checked easily.
- This gap explains why cryptography works, why certain puzzles are always challenging, and why heuristics and approximations dominate practical problem-solving.
- It sets a clear boundary between what is efficiently computable and what lies beyond our current reach, a reminder that some problems are inherently complex, no matter how clever our algorithms become.

In essence, the **P vs. NP question is a window into the limits of knowledge and computation**. Solving it, one way or another, would either unlock a new era of possibility or confirm the tantalizingly hard boundaries of the computational universe.

3 NP Completeness, Reductions and Approximations

"the study of NP-Complete problems is not just about solving puzzles; it is about **mapping the landscape of computation itself**"

3.1 NP-Completeness and Its Significance

A problem being **NP-Complete** is like being at the pinnacle of computational difficulty. These are problems for which verifying a given solution is easy (in polynomial time), but finding that solution may be unimaginably hard. They sit at the intersection of **"verifiable" and "hard to solve"**, defining the frontier of what is efficiently computable. Unlike problems in **P**, which can be solved quickly, or **co-NP**, which focuses on disproving statements efficiently, NP-Complete problems are the ultimate test of computational limits.

3.2 Reductions: The Language of Complexity

A reduction is a way of saying, "If you can solve this problem, you can also solve that one." It is a formal bridge connecting seemingly different problems, allowing insights and algorithms to transfer across domains. When one NP-Complete problem reduces to another, it reveals a deep truth: their computational hardness is essentially the same.

When we find that problems are reducible to each other, it's like discovering a hidden network of difficulty. Solve one efficiently, and you unlock solutions for many others. Conversely, the inability to solve one efficiently implies a universal barrier, a proof that no known polynomial-time solution exists for the whole class. These chains of reductions help us understand both the **limits of algorithms** and the **possibilities of clever heuristics or approximations**.

3.3 Approximation Algorithms vs. Heuristic Algorithms

In the world of hard problems, not all algorithms are created equal. **Approximation algorithms** are like mathematically guaranteed shortcuts: they promise that the solution you find is within a known bound of the optimal. You might not get perfection, but you know how close you are. **Heuristic algorithms**, on the other hand, are more like intuition-driven explorers: they make smart choices to find good solutions quickly, but without any theoretical guarantee. They often work beautifully in practice, yet their performance can be unpredictable.

3.4 Our Aim Going Forward

In this project, we are charting these connections, exploring **brute-force, heuristic, and approximation algorithms** across NP-Complete problems. By analyzing **time and space complexities**, implementing reductions, and testing practical algorithms, we aim to **understand both the theory and the practice of solving hard problems**. We seek not just solutions, but insight: how far can we push computation, where do heuristics excel, and how do the limits of NP-Completeness shape what is feasible in the real world?

4 3SAT

3SAT or the 3-Satisfiability is a decision problem. It asks the question: "Given a Boolean formula in a specific form called 3-CNF, can you assign True or False values to its variables so that the whole formula becomes True?"

4.1 Definitions

- **Boolean formula** is a formula that has Variables and Operators. We will define the variables as x_1, x_2, x_3 and so on for the rest of this section. Similarly, the Operators are defined as AND ($\wedge$), OR ($\vee$), and NOT ($\neg$).
- **Literal** is either a variable or its negation.
- **Clause** is a group of variables joined by ORs.
- **CNF (Conjunctive Normal Form)** is when a bunch of clauses are joined by ANDs. For a Boolean Formula in CNF to be true, each of the clauses must be true.
- Example: $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$ is in CNF with two clauses and four literals.
- **3CNF** is when there are exactly 3 literals in each clause of the CNF of the given Boolean Formula.

Input: A 3CNF Boolean Formula
Question: Is there any assignment of True/False to the variables that makes the formula evaluate to True?
Output:

- Yes $\rightarrow$ if such an assignment exists (the formula is satisfiable)
- No $\rightarrow$ if such an assignment does not exist (the formula is unsatisfiable)

4.2 Circuit Diagram

For the rest of this section, we will assume that every 3CNF Boolean Formula can be represented as a circuit. For example, the equation $(a+b)(\overline{c}+A)(\overline{A}+B)$ can be drawn as:

We will use similar circuit diagrams to represent formulae whenever required henceforth.

4.3 Classifying 3SAT

3SAT can be verified in $O(m)$ time where m is the number of clauses provided. This is because for each clause given, we can simply check if any one of the literals is true. Since any given formula and its assignment can be verified in polynomial, i.e., $O(m)$ time, 3SAT belongs to NP.

To show that 3SAT (and all the other problems we will cover beyond this) are NP-Complete, we need to look at Cook-Levin Theorem which states that:

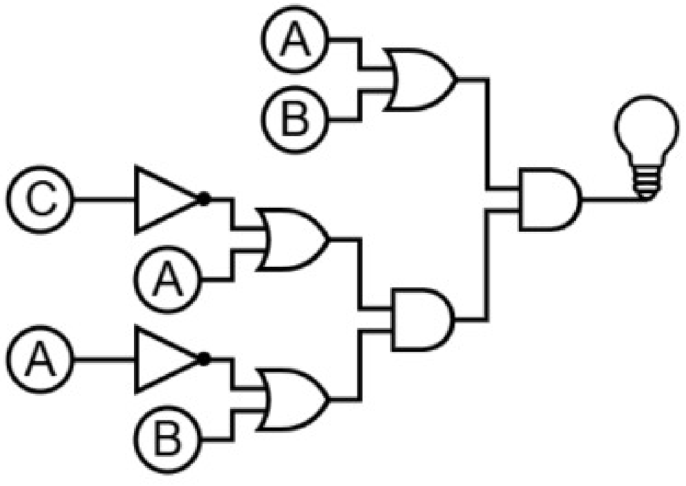


Figure 1: Circuit example

> "Every problem in NP can be reduced, in polynomial time, to an instance of SAT"

This theorem proved that the Boolean Satisfiability (SAT) problem is NP-complete, the first problem ever proven so.

It was also later shown that SAT can be reduced to 3SAT in polynomial time. As a result, if we can solve 3SAT efficiently, we can solve any NP problem efficiently. This matches the definition of NP-Complete. Thus, 3SAT can be classified as NP-Complete.

4.4 3SAT: Brute Force method

One very direct algorithm that we can use to solve this problem would be to directly assign all possible combinations of values to the variables, and to test if any of the assignments give us a True.

Let us assume that we are given a 3CNF formula $F(x_1, x_2, \ldots, x_n)$ with n variables and m clauses. Since each variable can be assigned a value of either 0 or 1, we have a total of 2^n possible combinations.

For each of the 2^n possible truth assignments:

- Evaluate F under that assignment.
- If F evaluates to True, return 1 (Satisfiable).
- If no assignment satisfies it, return 0 (Unsatisfiable).

Taking an example: For $F = (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_3)$ We try the following combinations, one by one:

- $x_1 = 0, x_2 = 0, x_3 = 0$
- $x_1 = 0, x_2 = 0, x_3 = 1$
- $\dots$
- $x_1 = 1, x_2 = 1, x_3 = 1$

Stop as soon as we find a satisfying assignment and return a 1. If none of these satisfies the equation, we can return a 0.

Time Complexity: Since we have n variables and m clauses, for each assignment we'd have to check m individual places. There are 2^n possible assignments. For each one, we check m clauses, and each clause has 3 literals (constant, so we don't have to take this into account). So total time = $O(2^n \times m)$. This approach gives us a time complexity which is exponential in n, which is why 3SAT is hard. There exists no solution that solves this problem within polynomial-time.

Space Complexity: We only need:

- Space to store the formula: $O(m)$
- Space for one assignment: $O(n)$ (it is only n here because we use only n bits to determine the values that the n variables take)

So total space taken is only $O(n + m)$ so it is of linear space complexity.

Why this is horrible: The time complexity being exponential is horrible for computation because as n increases, the time taken also increases very rapidly. As a result, using this (or any similar) brute force algorithm is detrimental and infeasible for larger n (even those > 100). Thus, approximation algorithms are used.

4.5 3SAT: Approximation Algorithm – 1: Randomization

Instead of dealing with 3SAT, we will be dealing with MAX-3SAT instead. MAX-3SAT is the optimization version of 3SAT that maximizes the number of satisfied clauses. The main difference is that while 3SAT asks whether there exists an assignment of variables that satisfies all clauses, MAX-3SAT simply finds an assignment that satisfies as many clauses as possible.

MAX-3SAT is not a decision problem like 3SAT is. It outputs OPT, which is the optimal number of clauses that can be satisfied in the best-case assignment. In 3SAT, if the formula is satisfiable, OPT $= m$ (all clauses satisfied). In Max-3SAT, the formula might not be fully satisfiable, so OPT $\leq m$. I.e., in this section, we will try to get as close to the maximum as possible, while keeping the computation and time comparatively feasible.

Input: A 3CNF Boolean Formula with n variables and m clauses
Question: What is the assignment of True/False to the variables that makes the formula maximize the number of True clauses?
Output: Number of satisfied clauses.

Algorithm: We assign random values (of 0 or 1) with a probability of 1/2 for each to all the n variables we have. I.e.: For each variable x_i (where $i = 1$ to n):

- Set $x_i = 1$ (True) with probability 1/2
- Set $x_i = 0$ (False) with probability 1/2

Substitute the values so assigned in F and count how many clauses are satisfied with this assignment. Return both the assignment and the number of satisfied clauses.

How good is it? We can prove that this algorithm is a 7/8 approximation algorithm. Let us take any clause. From the 3 literal bound that we set earlier, this clause will have only 3 literals. For this clause to fail, all three of them must be assigned 0. Since there are 2^3 combinations possible and only 1 one of them causes this particular clause to fail, we can say that this clause returns true with a probability of 7/8 after random allocation of variables. As a result, in expectation, this random assignment satisfies at least 7/8 of OPT. From the linearity of expectation, if this theory is true for one clause, it must be true for all clauses. So, we can guarantee that a minimum of 7/8 of m clauses will return true.

4.6 3SAT: Heuristic Algorithm – 1: Flipping Literals Method

Here too, we will continue dealing with MAX3SAT. The goal remains to find an assignment that satisfies as many clauses as possible, getting as close to the optimal number of satisfied clauses.

Input: A 3CNF Boolean formula F with n variables and m clauses
Question: What is an assignment of True/False to the variables that satisfies a large number of True clauses?
Output: The assignment and the number of satisfied clauses.

Algorithm: This algorithm starts with an initial assignment (either random or fixed) and iteratively tries to improve it by making small, localized changes.

1. **Initialization:** Choose a starting assignment A for all n variables. This can be done by randomly setting each variable x to True or False

with probability 1/2 (similar to the first approximation algorithm). Let $A = A_0$ be the current assignment.

2. **Repeat** the following procedure for a fixed, polynomial-bounded number of steps:
   (a) **Check for Improvement:** Examine the current assignment A.
   (b) **Identify a Candidate Flip:** Find a single variable x_i such that flipping its current truth value (i.e., changing it from True to False, or False to True) leads to a strict increase in the total number of satisfied clauses.
   (c) **Perform the Flip (Greedy Choice):** If such a variable exists, choose the flip that results in the largest increase (greedy approach) or simply choose any variable that improves the count, and perform the flip. Update the assignment A.
   (d) **If No Improving Flip Exists (Local Maximum):** The algorithm has reached a local optimum where no single variable flip can satisfy more clauses. Stop the search. Return the final assignment A and the count of satisfied clauses.

How Good Is It? The simple randomized local search (greedy flip) guarantees a performance similar to the purely random approach, but often performs much better in practice. This method is a heuristic algorithm because it uses a rule-of-thumb strategy i.e., iteratively flipping variables that improve the number of satisfied clauses. This doesn't guarantee an assignment that satisfies a specific fraction of clauses. Unlike a true approximation algorithm, which comes with a formal performance ratio or bound relative to the optimal solution, this method only guides the search toward better solutions in practice. It may get stuck in local maxima and does not provide a provable worst-case guarantee, making it heuristic rather than a formal approximation algorithm. However, this algorithm is quite practical and feasible due to its good time complexity.

When you pick an unsatisfied clause: Suppose the clause is $(l_1 \vee l_2 \vee l_3)$ and currently all three literals evaluate to false, that means at least one variable must change for this clause to become true. Flipping any one of the three variables in that clause guarantees this clause becomes satisfied (unless flipping breaks some other satisfied clause). So each flip has a 1/3 chance of picking a variable that helps fix the clause. We don't pick completely random variables, we only pick variables from unsatisfied clauses. That bias makes it much more likely to make progress, unlike flipping variables totally at random. Thereby, over many iterations, we may occasionally flip a wrong variable and temporarily break a satisfied clause, but repeated flips eventually correct this.

Time Complexity

- Initial random assignment $\rightarrow$ Assigning n variables takes $O(n)$ time.
- Iterations $\rightarrow$ suppose we allow $O(n^2)$ iterations for each random assignment.
- Each iteration involves:
  - Checking if the formula is satisfied $\rightarrow O(m)$ due to the m clauses.
  - Choosing a random unsatisfied clause $\rightarrow O(1)$ if we maintain a list of unsatisfied clauses.
  - Flipping a variable $\rightarrow O(1)$ and updating unsatisfied clauses might take $O(1)$ per affected clause, at most $O(3)$ per flip.
- So each iteration is roughly $O(m)$.
- With $O(n^2)$ iterations $\rightarrow O(m \cdot n^2)$ per random assignment.

Step 3: Restarts Suppose we try $O(\log n)$ random assignments to boost probability of success (every time we get stuck). Total expected time:

$$O(\text{restarts} \cdot \text{iterations} \cdot m) = O(\log n \cdot n^2 \cdot m) = O(mn^2 \log n)$$

Probability of success (Papadimitriou's analysis) While in the previous example we tried to find the fraction of clauses satisfied by random assignment, here, we find the probability that flipping the literals eventually leads to a full assignment (i.e., all clauses satisfied).

Since each variable is independently assigned True/False $\rightarrow$ some fraction of clauses satisfied. As calculated before, the probability of a flip improving a clause $= 1/3$. So, expected number of flips to satisfy all clauses: The number of steps to reach the satisfying assignment is $O(n^2)$ for 3-SAT. (Each step has constant probability of moving closer (1/3) or further (2/3). For a 1D unbiased random walk, expected hitting time = distance^2, so here, the order remains along the lines of $O(n^2)$). Overall probability after $O(n^2)$ flips results in at least $1/2$ chance of success for a single random assignment. Using multiple random restarts (say $\log n$ times as assumed in the time complexity) increases success probability to: $1 - (1 - 1/2)^{\log n} = 1 - 1/n$: which tends to 1 as the number of variables we take increases.

5 Vertex Cover

Vertex Cover is a decision problem. It asks the question: "Given a graph (a set of vertices connected by edges) and a number k, can you select at most k vertices such that every edge in the graph has at least one of its endpoints in your selected set?"

5.1 Definitions

$G(V, E)$ represents a graph with V vertices and E edges. Each edge can be written as $e(u, v)$ where the edge connects two vertices u, v belonging to V. A vertex cover is basically a set C such that for every $e(u, v)$ belonging to E, we have either u or v included in C.

Input: An undirected Graph $G(V, E)$ and a value k
Question: Does there exist a Vertex Cover of size k or lesser?
Output:

- Yes $\rightarrow$ if such a cover exists
- No $\rightarrow$ if such a cover doesn't exist

5.2 Graph Visualization

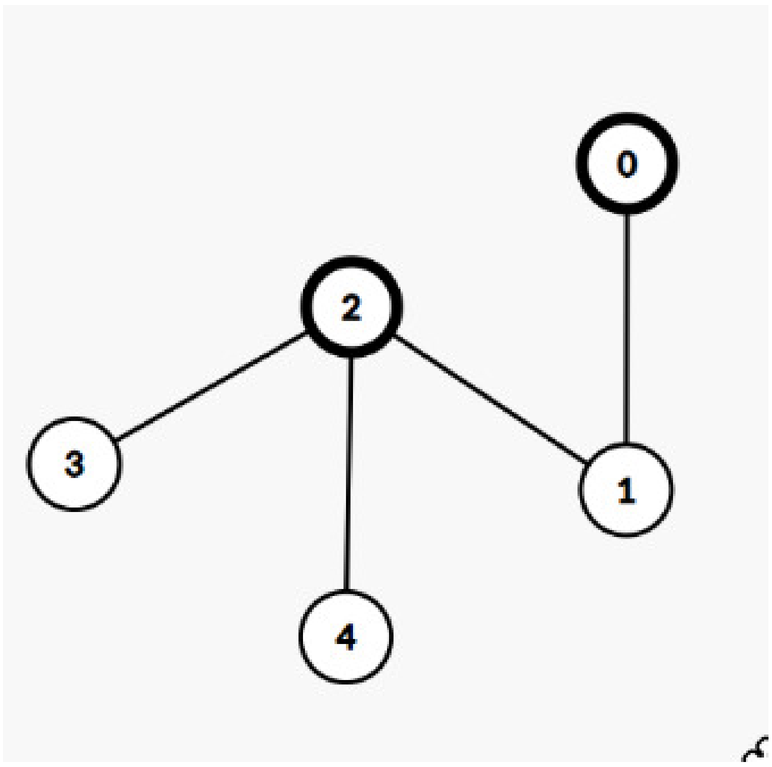


Figure 2: Vertex Cover Example

In this graph, we see that the set (0, 2) covers all the edges, and as a result, is a Vertex Cover.

5.3 Classifying Vertex Cover

Vertex Cover can be verified in $O(n)$ time where n is the number of nodes provided. This is because for each G and set C given, we simply have to check if all the vertices in C exist in G, and if all the edges of E are covered. This would take one pass of the nodes, giving us verification in poly time. To show that the Vertex Cover is NPC, we need to show that it is NP Hard. We will show that 3SAT can be reduced to Vertex Cover in polytime.

5.4 3SAT $\rightarrow$ Vertex Cover: Reduction

In order to show that Vertex Cover is NPC, we will show that 3SAT can be reduced to it in poly time. Since 3SAT is already proven to be NP complete, we know that all NP problems can be reduced to it. Thus, if every instance of 3SAT can be reduced to an instance of VC, then that would be sufficient to prove the hardness of the same.

Say we have the Boolean equation F with variables x_i (n variables, m clauses). We construct an undirected graph $G = (V, E)$ as follows:

1. **For each variable x_i in F, create two vertices:** One for x_i (the positive literal) and one for $\neg x_i$ (the negative literal). Connect these two vertices with an edge. This ensures that in a vertex cover solution, we must pick at least one of the literal or its negation — which corresponds to assigning True or False to that variable. We are basically forcing a choice: either the variable takes True or it takes False. With this our variable gadgets have been formed.
2. **Now, for each clause $C_j = (l_1 \vee l_2 \vee l_3)$:** Create 3 vertices, one for each literal in the clause. Connect all three vertices to form a triangle (3-cycle). The triangle ensures that to cover all edges in the triangle, you must pick at least 2 vertices. The one vertex left out corresponds to the literal that will be True in that clause. Our clause gadgets have been formed.
3. **Finally, connect clause vertices to variable vertices:** Each clause vertex is identified with the literal it represents. That is: The vertex representing literal l in a clause triangle is the same vertex as the literal in the variable gadget. This ensures consistency: if we pick a literal in the variable gadget (assign it True), it helps cover edges in the clause triangles.

VC size must be k:

- For each variable, we need 1 vertex from its pair $\rightarrow$ contributes n to the cover.

- For each clause triangle, we need 2 vertices to cover all edges $\rightarrow$ contributes $2m$ to the cover.
- $k = n + 2m$

Thus, if the graph has a vertex cover of size k, it corresponds to a satisfying assignment for the 3SAT formula.

Proof of Correctness:

- If 3SAT is satisfiable: Pick one literal per variable according to the satisfying assignment (True literal $\rightarrow$ pick in VC). For each clause triangle, the True literal in the clause can be left out, pick the other two $\rightarrow$ all edges covered. Total vertices picked $= n + 2m = k$.
- If G has a vertex cover of size k: Exactly one vertex must be picked from each variable pair (otherwise, k would be larger). In each clause triangle, exactly two vertices are picked $\rightarrow$ the one left out is True. This gives a satisfying assignment for the 3SAT formula.

Thereby, every instance of the Graph having a VC of k can be mapped to some formula F such that it results in a satisfying assignment and for every formula F that has a satisfying assignment, a graph having VC of k can be drawn. Clearly, graph construction is polynomial in size of the 3SAT formula. The reduction proves that $3SAT \leq_p$ Vertex Cover. Thus, Vertex Cover is NP Hard, and also, NP complete.

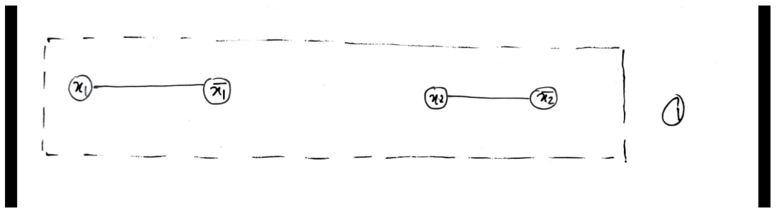


Figure 3: Step1: Connecting complement vertices

5.5 Vertex Cover: Brute Force method

One very direct algorithm that we can use to solve this problem would be to directly assign all possible combinations off the vertices possible to prospective vertex cover. If we have n vertices, then the power set of n: $P(n)$ consists of all the combinations of vertices $x_1, x_2, x_3 \ldots$ possible. We can now check each prospective VC, C', to find if it is a) a valid VC b) if it is of size k.

Algorithm:

1. Iterate Through All Subset Sizes, filter those of size k and below.

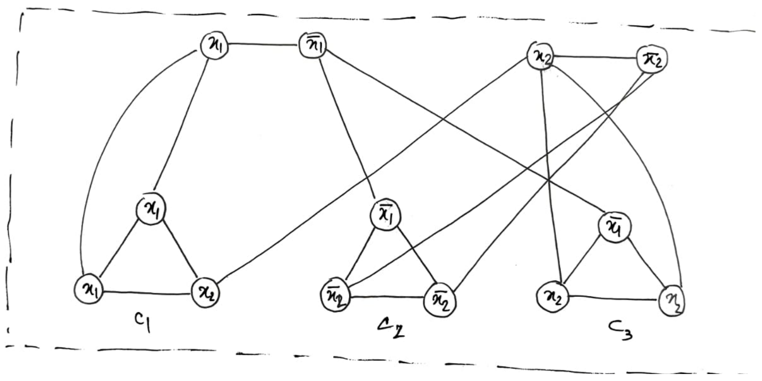


Figure 4: Step2: Connecting similar literals

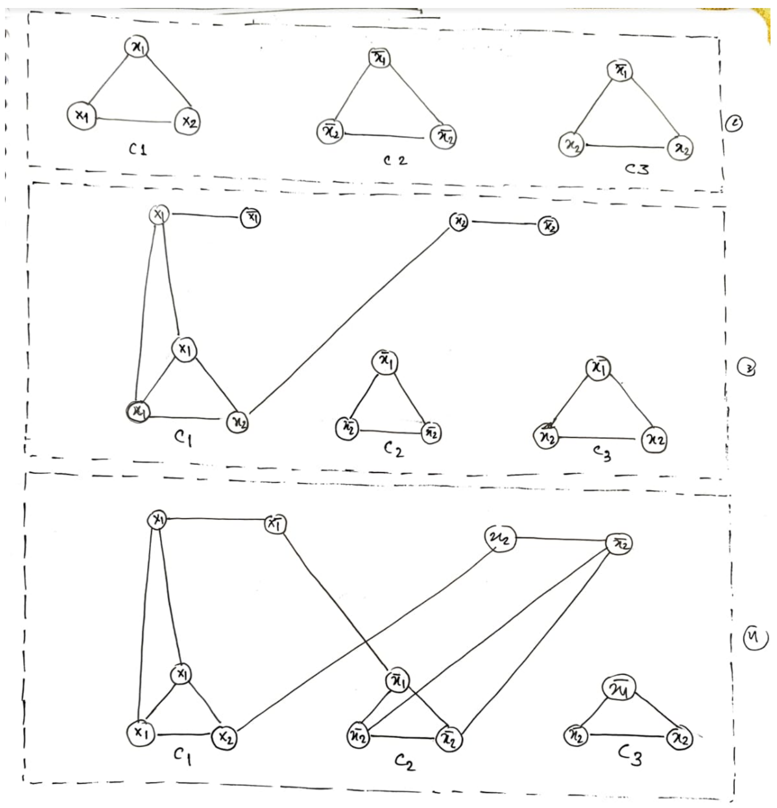


Figure 5: Step3: Final Comparison

2. Check if a given $\leq k$ sized subset is a valid vertex cover. For each generated subset V', check every edge $e \in E$ in the graph. If, for every edge $e = (u, v)$, either $u \in V'$ or $v \in V'$ (or both), then V' is a valid Vertex Cover.
3. Stop and Return: The moment a valid Vertex Cover V' is found for the smallest size k checked, stop and return V' and its size k. This k is the size of the minimum vertex cover (OPT).

Time complexity of Brute Force Let n be the number of vertices and m be the number of edges. Checking Time: For any given subset V', verifying if it is a cover requires checking all m edges. The check for each edge takes constant time $O(1)$. So, checking one subset takes $O(m)$ time. Total Time Complexity: We must consider the worst-case scenario where the minimum vertex cover size is large (e.g., $k \approx n/2$). The total number of subsets checked is:

$$\sum_{k=1}^{n} \binom{n}{k}$$

Since the largest term in the summation is $\binom{n}{n/2}$, which is bounded by $O(2^n)$, the total number of subsets we must check is 2^n. Multiplying the number of subsets by the time to check each one:

$$\text{Total time} = O(2^n \times m)$$

Thus, the brute force algorithm results in an exponentially large time complexity.

Why this is horrible: The time complexity being exponential is horrible for computation because as n increases, the time taken also increases very rapidly. Thus for larger values of n (as the number of nodes increases) the calculation becomes incomputable. As a result, using this (or any similar) brute force algorithm is detrimental and infeasible for larger n (even those > 100). Thus, approximation algorithms are used.

5.6 Vertex Cover Approximation Algorithm – 1: Maximal Matching

Instead of dealing with the Vertex Cover decision problem (which asks if a cover of size k (or lesser than k) exists), we will be dealing with MinVC instead. Min-VC is the optimization version of Vertex Cover that minimizes the number of vertices in the cover, while not necessarily being at the absolute minimum. The main difference is that while the Vertex Cover decision problem asks whether there exists a subset of vertices V' of size $\leq k$ that covers all edges, Min-VC simply finds a subset V' that covers all edges while having the smallest possible size. We will now try to find a value for C that is as close to optimal OPT as possible. I.e., $C \geq \text{OPT}$, while keeping the computation feasible.

In a graph $G = (V, E)$, a matching M is a subset of the edges E such that no two edges in M share a common vertex. I.e., edges that are entirely separate/disjoint from each other. A matching M is maximal if it cannot be extended by adding any other edge from E. Maximal matching is locally optimal, i.e., it cannot be made any larger by adding any more edges. This is different from a maximum matching, which is globally optimal, the maximum possible number of edges in the matching.

Input: An undirected graph $G(V, E)$
Question: Find a subset of vertices $C \subseteq V$ that covers all edges E and has a size close to the minimum possible size, OPT.
Output: The approximate Vertex Cover C and its size $|C|$.

Algorithm

1. This is a simple greedy algorithm based on finding a maximal matching in the graph.
2. Initialize the resulting Vertex Cover C as an empty set: $C = \emptyset$.
3. Initialize the set of matched edges M as an empty set (the matching): $M = \emptyset$.
4. Let $E_R = E$ be the set of remaining edges after each transaction.
5. While the set of remaining edges E_R is not empty:
    (a) Arbitrarily choose any edge $e = (u, v)$ from E_R.
    (b) Add e to the matching: $M = M \cup \{e\}$.
    (c) Add both endpoints of e to the Vertex Cover: $V' = V' \cup \{u, v\}$.
    (d) Remove all edges from E_R that are incident to either u or v (since u and v are now in the cover, these edges are covered).
6. When all the edges are covered, we have achieved one such Vertex Cover.

How good is it? (2-Approximation) This algorithm guarantees that the size of the resulting cover $|C|$ is at most twice the size of the minimum possible vertex cover (OPT).

$$\frac{|C|}{\text{OPT}} \leq 2 \implies |C| \leq 2 \cdot \text{OPT}$$

There is a crucial relationship that exists between the set of maximal edges (M) and the optimal cover $\text{OPT} = C^*$. The algorithm adds exactly two vertices to C for every edge added to the matching M.

$$|C| = 2 \cdot |M| \quad \text{(Since } M \text{ is a set of disjoint edges)}$$

The optimal vertex cover, C^*, must cover every edge in the matching M. Since the edges in M are disjoint (meaning no two edges in M share an endpoint), for C^* to cover all $|M|$ edges, it must include at least one unique endpoint from each edge in M. This implies that the size of the optimal cover must be at least as large as the size of the matching.

$$\text{OPT} = |C^*| \geq |M|$$

By substituting the inequalities into the equations we have derived,

$$|C| = 2 \cdot |M| \leq 2 \cdot |C^*| = 2 \cdot \text{OPT}$$

Since $|C| \leq 2 \cdot \text{OPT}$, the algorithm is a 2-approximation for the Minimum Vertex Cover problem.

Time Complexity The time complexity of this algorithm is very efficient. In each step, we iterate over the remaining edges and pick one. This process continues until no edges remain. The total amount of work is proportional to the total number of edges examined and removed. We only need to examine each edge at most once. The complexity is dominated by iterating over and updating the set of edges and vertices, which is bounded by the number of vertices n and the number of edges m.

$$\text{Time Complexity} = O(n + m)$$

This is a polynomial time algorithm, which is highly feasible, even for large graphs. Thereby, we have a 2-approx algorithm for Vertex Cover, such that the size of the C we derive will always be less than 2 times the optimal size.

5.7 Vertex Cover Approximation Algorithm – 2: LP relaxation

This algorithm finds an approximate solution by first formulating the problem as an Integer Linear Program (ILP), relaxing the integer constraints to create a solvable Linear Program (LP), and then rounding the solution of the LP. Here too, instead of using VC, we will be using MINV to try to approximate the optimal vertex.

Defining MINVC in terms of ILP:

1. **Variables (The Choices):** For every vertex v in your graph, you create a variable x_v:
    - $x_v = 1$: Means you include vertex v in your cover C.

- $x_v = 0$: Means you exclude vertex v from your cover C.
- x_v must be strictly a 0 or a 1 (since I either include or don't include the node).

2. **Objective (The Goal):** You want to minimize the total size of your cover C. You add up all the chosen vertices:

$$\text{Minimize} \quad \sum_{v \in V} x_v$$

3. **Edge Constraints (The Rules):** Every edge (u, v) must be covered. For an edge to be covered, at least one of its endpoints (u or v) must be in the cover ($x_u = 1$ or $x_v = 1$). Constraint: For every edge (u, v), the sum of their variables must be at least 1:

$$x_u + x_v \geq 1$$

Now, ILP is hard simply because there is a hard 0 or 1 constraint on each x_i. If instead, we had a condition where we apply LP relaxation to make the problem less hard, we could relax the constraints in this form: Relaxed Constraint: $0 \leq x_v \leq 1$ This allows the probability of some vertice v getting into C being fractional instead of integral. We solve this new LP and get the optimal fractional solution, x^*. We get a value x_v^* for every vertex v. This value is the "weight" or "importance" of that vertex, where $0 \leq x_v^* \leq 1$. The sum of these values $L^* = \sum x_v^*$, will be less than or equal to the size of the true optimal integer cover, OPT. (Since we removed a rule, the answer can only get better/smaller). Relaxation to LP is complete. Now, we have to round the answer.

Rounding Rule Now we have a fast, fractional solution x^*. We must convert these fractions back into whole numbers (0 or 1) to get a valid Vertex Cover V'. We use a simple, consistent rounding rule: Rule: If a vertex's value x_v^* is **1/2** or greater, we include it in the cover. If it's less than 1/2, we exclude it.

$$\text{If } x_v^* \geq \frac{1}{2}, \text{ set } x_v^{\text{round}} = 1 \quad \text{(Add to } C\text{)}$$

$$\text{If } x_v^* < \frac{1}{2}, \text{ set } x_v^{\text{round}} = 0 \quad \text{(Exclude from } C\text{)}$$

Why this Rounding Guarantees a Cover This simple rounding guarantees that every edge (u, v) is covered in the final set C. We know from Step 2 that for every edge, $x_u^* + x_v^* \geq 1$. Since the sum of the two fractions is at least 1, it is mathematically impossible for both fractions to be less than 1/2. (If $x_u^* < 0.5$ and $x_v^* < 0.5$, then $x_u^* + x_v^* < 1$, which violates the constraint). Therefore, at least one endpoint (either u or v) must have $x_v^* \geq \frac{1}{2}$, guaranteeing it is included in V'.

Why this works? (2 approx guarantee) The size of the resulting cover $|C|$ is at most twice the size of the optimal vertex cover (OPT).

$$\frac{|C|}{\text{OPT}} \leq 2 \implies |C| \leq 2 \cdot \text{OPT}$$

The proof relies on two key facts relating the ILP solution (OPT) and the relaxed LP solution (L^*). To prove this we check the following:

- **Lower Bound of the LP Solution:** Since the LP relaxation has fewer constraints than the original ILP (allowing fractional values), the LP optimal value must be less than or equal to the ILP optimal value, OPT.

  $$L^* \leq \text{OPT}$$

  (we are essentially allowing more suitable changes to occur to our prospective set, so the minimum value will only decrease or remain the same, not increase).

- **Size of the Rounded Cover $|C|$:** The size of the final vertex cover C is the number of vertices v where $x_v^* \geq \frac{1}{2}$.

  $$|V'| = \sum_{v \in V'} 1 \quad \text{(Count of vertices in } V'\text{)}$$

  $$= \sum_{v \in V : x_v^* \geq 1/2} 1 \quad \text{(By the rounding rule)}$$

  Since every vertex v in C has $x_v^* \geq \frac{1}{2}$, it must be true that $1 \leq 2 \cdot x_v^*$ for all $v \in V'$.

  $$|C| = \sum_{v \in C} 1 \leq \sum_{v \in C} 2 \cdot x_v^* \leq \sum_{v \in V} 2 \cdot x_v^* = 2 \cdot \sum_{v \in V} x_v^* = 2 \cdot L^*$$

  Thus, we have the intermediate inequality: $|V'| \leq 2 \cdot L^*$.

The Approximation Bound: Combining the two results:

$$|V'| \leq 2 \cdot L^* \leq 2 \cdot \text{OPT}$$

Since $|V'| \leq 2 \cdot \text{OPT}$, the algorithm is a 2-approximation.

Time Complexity The complexity is dominated by the main LP solving step. Linear programs can be solved in polynomial time.

$$\text{Time Complexity} = \text{Polynomial Time}$$

Specifically, solving the LP takes $O(f(n, m))$ time, where f is a polynomial function of the number of variables (n) and the number of constraints (m). Steps 1 and 3 are essentially trivial in comparison. Thus, the time complexity is a polynomial function of n and m where n is the number of nodes and m is the number of edges.

6 MaxClique

Clique is a decision problem. It asks the question: "Given a graph G and a number k, does G contain a subset of k vertices such that every pair of vertices in the subset is connected by an edge?" In other words, can you find a complete subgraph of size k inside the given graph?

6.1 Definitions

- **Clique** C is a subgraph of given graph G, of size k such that every pair of vertices in C is connected by an edge.
- **Complete graph** is a graph in which all vertices are connected to all other vertices.

Input: An undirected graph $G(V, E)$ and number k
Question: Is there a subgraph of size k or more such that every vertex is connected to every other vertex?
Output:

- Yes $\rightarrow$ if such a C exists
- No $\rightarrow$ if such a C doesn't exist

6.2 Circuit Diagram

This is a graph with a clique of size 4 (nodes 1, 3, 4, 5 are all interconnected).

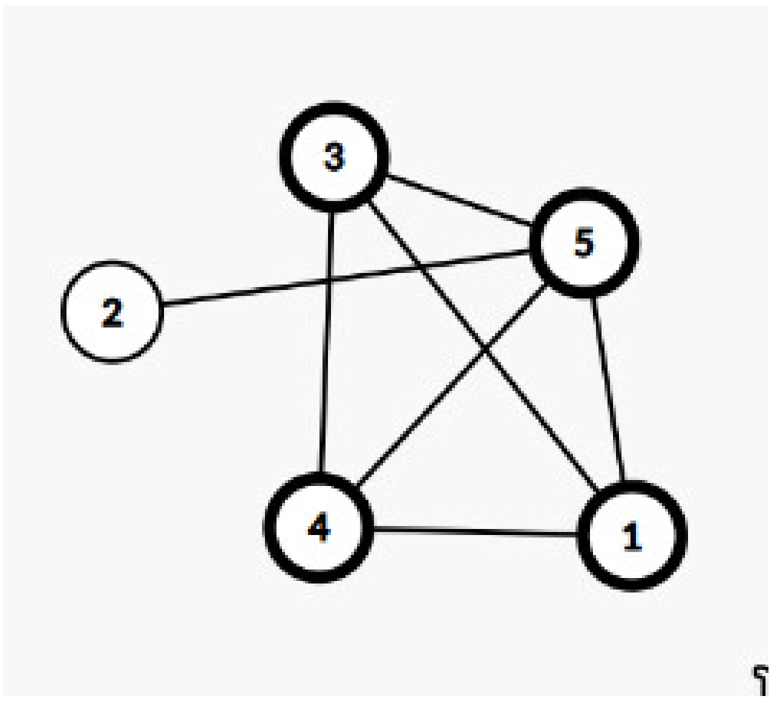


Figure 6: CLique Example

6.3 Classifying Clique

CLIQUE can be verified in $O(k^2)$ time, where k is the size of the proposed clique. This is because once we are given a candidate set of k vertices, we simply check whether every pair of vertices in the set has an edge between them. Since there are $\binom{k}{2}$ such pairs, verification takes polynomial time. Thus, CLIQUE belongs to NP. To show that CLIQUE (and many other problems we will discuss) is NP-Complete, we rely on the fact that Vertex Cover can be reduced to CLIQUE in polynomial time, since VC is already NPC (from the previous proofs).

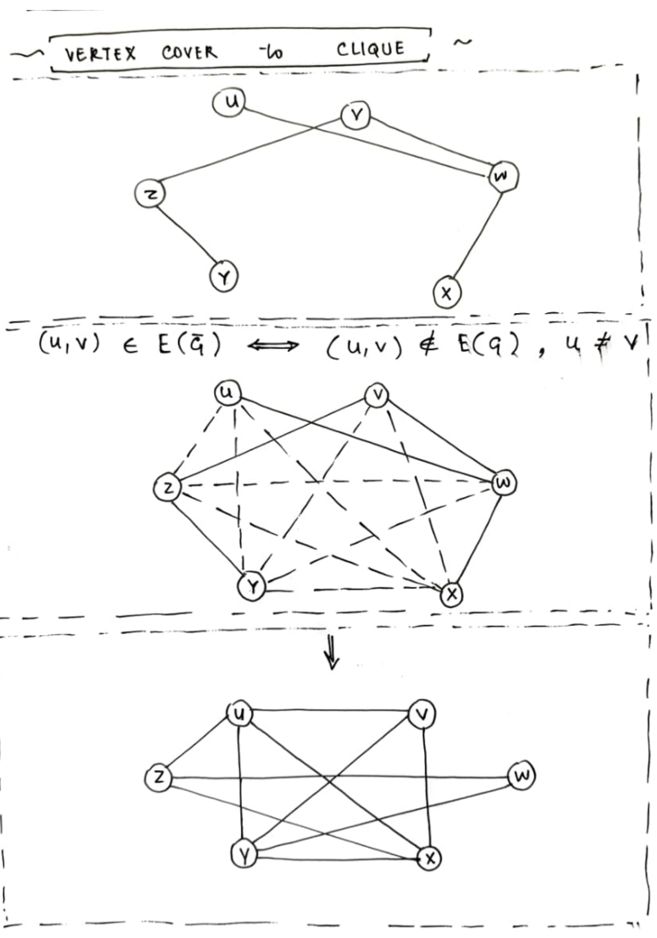


Figure 7: Vertex Cover to Clique Reduction

6.4 Vertex Cover → CLIQUE: Reduction

Since VC is an NPC problem and CLIQUE has been proven to belong to NP, simply showing that VC can be reduced to CLIQUE is enough to show that all NP problems can be reduced to CLIQUE, i.e., CLIQUE is NP Hard and thereby, NP complete.

To map every instance of VC into an instance of CLIQUE, we want a polynomial-time reduction:

$$(G, k) \longrightarrow (H, k')$$

such that: G has a vertex cover of size $k \iff H$ has a clique of size $|V| - k$. Vertex Cover in graph G corresponds to Clique in the complement graph $\bar{G}$.

This means that for every graph G that has a vertex cover of size k, in its complement graph ($\bar{G}$, where if an edge exists in G, it's removed in $\bar{G}$ and vice versa). Just by this fact alone, we can construct an instance of graph $\bar{G}$ for every G.

Say we have an instance of VC, in graph $G = (V, E)$, with a vertex cover of size k. To construct the complement graph:

- Form a new graph $H = \bar{G}$ with: Same set of vertices V.
- An edge between u and v iff (u, v) is not an edge in G.
- $k' = |V| - k$.

Why this works (Proof of correctness)

1. **If G has a vertex cover of size k, then $\bar{G}$ has a clique of size $|V| - k$:** Let C be a vertex cover in G with $|C| = k$. Consider the remaining vertices: $I = V \setminus C$. These vertices I form an independent set in G, because any edge between two vertices in I would not be covered by C. So no pair of vertices in I is connected in G. Therefore, in the complement graph $\bar{G}$, all vertices in I are connected to each other. Thus, I forms a clique of size $|V| - k$ in $\bar{G}$.
2. **If $\bar{G}$ has a clique of size $|V| - k$, then G has a vertex cover of size k:** Let K be a clique of size $|V| - k$ in $\bar{G}$. Consider the remaining vertices: $C = V \setminus K$. Since K is a clique in the complement graph, for any two vertices in K, there is no edge between them in the original graph G. Thus, K is an independent set in G. Now, for every edge in G, at least one endpoint must lie in C. Otherwise, if both endpoints were in K, that would contradict the fact that K contains no edges in G. Thus, C is a vertex cover of size k in G.

Since:

- The transformation is polynomial-time, and
- Vertex Cover solution exists $\iff$ Clique solution exists,

we conclude: Vertex Cover $\leq_p$ Clique. Thus, CLIQUE can also be classified as NPC.

6.5 CLIQUE: Brute Force method

One direct way would be to try every possible set of k vertices and test whether that set forms a clique.

Algorithm For each subset $S \subseteq V$ of exactly k vertices:

- Check every pair of vertices $u, v \in S$.

- If every pair u, v is in E, return 1 (Clique of size k exists).
- If no such subset works, return 0 (No clique of size k).

Time Complexity: Number of subsets of size k: $\binom{n}{k}$. For each subset we must check all $\binom{k}{2}$ pairs to verify edges (or check adjacency in $O(1)$ if adjacency matrix used). So running time is $O(\binom{n}{k} \cdot k^2)$ (checking costs $\Theta(k^2)$ per subset). Two useful special-case bounds:

- If k is constant, time is polynomial in n: $O(n^k \cdot k^2)$.
- In the general worst case (if we try all subset sizes or $k \approx n/2$), $\binom{n}{k}$ is $O(2^n/\sqrt{n})$ at peak, so brute force is essentially exponential, e.g. $O(2^n \cdot n^2)$.

Hence for the decision version with arbitrary k the time is exponential in n: $O(\binom{n}{k}k^2)$ (worst-case $\approx O(2^n n^2)$).

Space Complexity: Space to store the graph:

- Adjacency matrix: $O(n^2)$.
- Adjacency lists: $O(n + m)$.
- Space for one candidate subset / bookkeeping: $O(k)$ (or $O(n)$ if you store a bitmask of chosen vertices).

So total space: $O(n^2)$ (matrix) or $O(n + m)$ (lists) plus $O(n)$ for temporary storage — overall linear-to-quadratic depending on representation. The time complexity is clearly non polynomial, and so is highly incomputable for larger values of n. Thus we need to adopt approximation algorithms.

6.6 CLIQUE Heuristic Algorithm – 1: Greedy Algorithm

As covered earlier, heuristic algorithms are practical methods that try to find a good solution quickly. There is no guarantee on how close the solution is to the optimal or the performance in the worst case. I.e., theoretically, there is no solid equation that could describe the outcome of the algorithm.

Why approximation algorithms CANNOT exist for CLIQUE An approximation algorithm must be Polynomial-time and give a provable guarantee, i.e., a proof that claims something like:

$$\mathrm{ALG}(G) \geq \frac{1}{\alpha} \cdot \mathrm{OPT}(G)$$

where α is the approximation ratio. However, For MAX-CLIQUE, good approximation algorithms do not exist (unless P=NP). No constant-factor approximation known and we have no mathematical guarantee. Thus, no approximation algorithm exists. We instead look at its heuristic algorithm, one that doesn't provide a worst case guarantee but is fast and used for practical purposes.

Instead of looking at the CLIQUE decision problem, we will instead look at the MAXCLIQUE optimization problem instead. The new problem statement becomes: "Given graph $G = (V, E)$, find a subset $C \subseteq V$ that is a clique and has maximum possible size OPT, i.e., the largest possible clique". Ideally $|C| = \text{OPT}$. However, in this approximation, we seek $|C|$ as close to OPT while keeping computation feasible.

Input: graph $G = (V, E)$
Question: What is the closest C we can reach to C^*, where C^* is the optimal solution to CLIQUE(G)
Output: Clique C

Algorithm:

1. Initialize the best clique found so far: $R_{\text{best}} = \emptyset$.
2. (Optional but recommended): Order all vertices in V by some heuristic (e.g., decreasing degree).
3. For each vertex v_{start} in the ordered set V: Now, we can start iterating over the general vertices:
   (a) **Start Candidate Clique:** Initialize the current candidate clique R with the starting vertex: $R = \{v_{\text{start}}\}$.
   (b) **Candidate Set:** Define the set of available vertices that are neighbors of the starting vertex: $\text{Cand} = \text{Neighbors}(v_{\text{start}})$.
   (c) **Greedy Extension:** While Cand is not empty:
      - **Pick Vertex:** Pick a vertex $v_{\text{next}} \in \text{Cand}$ (choose using a heuristic, e.g., highest degree within the subgraph induced by Cand, or choose randomly).
      - **Check Adjacency (Clique Property):** Check if v_{next} is adjacent to every vertex already in the current clique R.
      - **If Yes (Valid Addition):** Add v_{next} to R, and update Cand to only include vertices that are neighbors of v_{next} and still in the original candidate set:

$$R = R \cup \{v_{\text{next}}\}$$

$$\text{Cand} = \text{Cand} \cap \text{Neighbors}(v_{\text{next}})$$

      - **If No (Invalid Addition):** Remove v_{next} from Cand and try the next available candidate.

(d) **Update Best Result:** If $|R| > |R_{\text{best}}|$, set $R_{\text{best}} = R$.

4. Return R_{best}, the largest maximal clique found.

Why this is only a heuristic (no good worst-case guarantee) and not an approx No constant-factor guarantee: there is no polynomial-time algorithm that always returns a clique of size within a constant factor of OPT for general graphs (unless $P = NP$). Stronger hardness: Some PCP results imply that MAX-CLIQUE is extremely hard to approximate: for every $\epsilon > 0$, it is NP-hard to approximate MAX-CLIQUE within factor $n^{1-\epsilon}$. This simply means that it is very hard to get a polynomial expression for the approximation ratio. So, the greedy algorithm has no provable approximation ratio in the worst case. However, the greedy approach often performs well on real/structured instances and datasets.

How good is this? Unlike an approx. algorithm, we don't measure a heuristic by any guaranteed quality, since it doesn't exist. The heuristic is measured by its fast, polynomial-time performance. Speed: The total time complexity is typically bounded by $O(n^3)$ or $O(n \cdot (n+m))$, depending on the graph representation and the efficiency of the inner loops. This makes the algorithm highly feasible for large graphs where finding the true maximum clique is impossible. Trade-off: The algorithm chooses speed and feasibility over optimality and guaranteed quality.

Time complexity:

- Sorting vertices by degree: $O(n \log n)$.
- Worst case: if you run the inner loop many times, the cost can be $O(n^2)$ per seed, and doing it for all seeds gives $O(n^3)$ worst-case in naive implementations.
- Practical realistic bound with adjacency matrix: about $O(n^2/w)$ per seed $\rightarrow$ often $O(n^2)$ overall with pruning/degeneracy ordering.
- So typical complexity (practical) $\approx O(n^2)$ to $O(n^3)$ depending on implementation and graph density.

Space complexity:

- Adjacency matrix: $O(n^2)$.
- Adjacency lists: $O(n + m)$.
- Candidate/clique bookkeeping: $O(n)$.

As a result, we have a polytime heuristic for the MAXCLIQUE problem.

7 Graph Colouring

Graph Colouring is a decision problem that asks the following question: "Given a graph and a number k, can you assign colours to each vertex so that no two adjacent vertices share the same colour?"

7.1 Definitions

- **Colouring:** A colouring of a graph assigns a colour (usually represented as numbers like 1, 2, 3) to each vertex.
- **Proper colouring:** A colouring is proper if no two adjacent vertices have the same colour.

Input: A graph $G = (V, E)$ and an integer k.
Question: Does a proper colouring exist using at most k colours?
Output:

- Yes $\rightarrow$ if such an assignment exists
- No $\rightarrow$ if such an assignment does not exist

7.2 Graph Visuals

7.3 Classifying Graph Colouring

Graph Colouring can be verified in $O(n^2)$ time, where n is the number of vertices in the graph. This is because once we are given a candidate colouring, we simply check whether every edge connects vertices of different colors. Since there are at most $\binom{n}{2}$ such pairs of vertices, verification takes polynomial time. Thus, Graph Colouring belongs to NP. To show that Graph Colouring is NP-Complete, we rely on the fact that CLIQUE can be reduced to Graph Colouring in polynomial time, since CLIQUE is already NP-Complete (from the previous proofs).

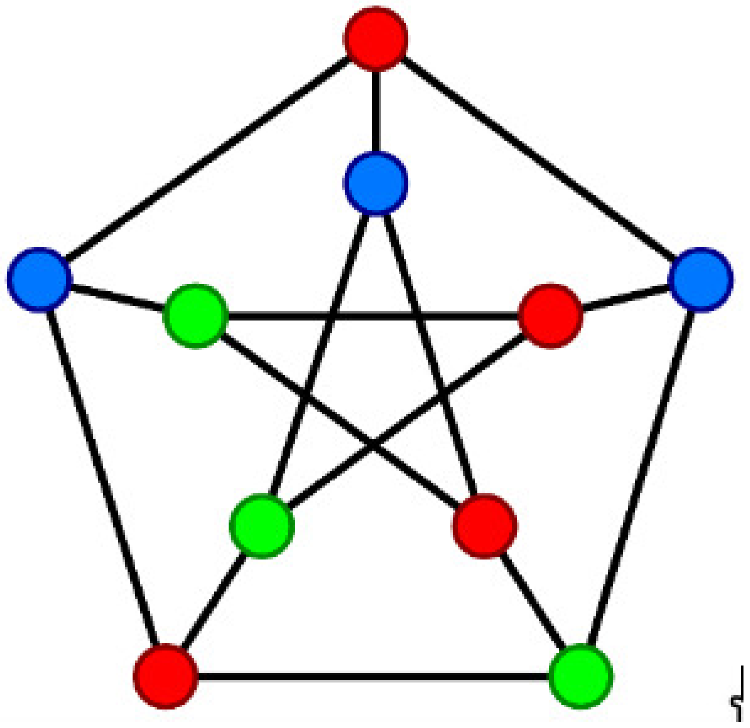


Figure 8: Graph Coloring example

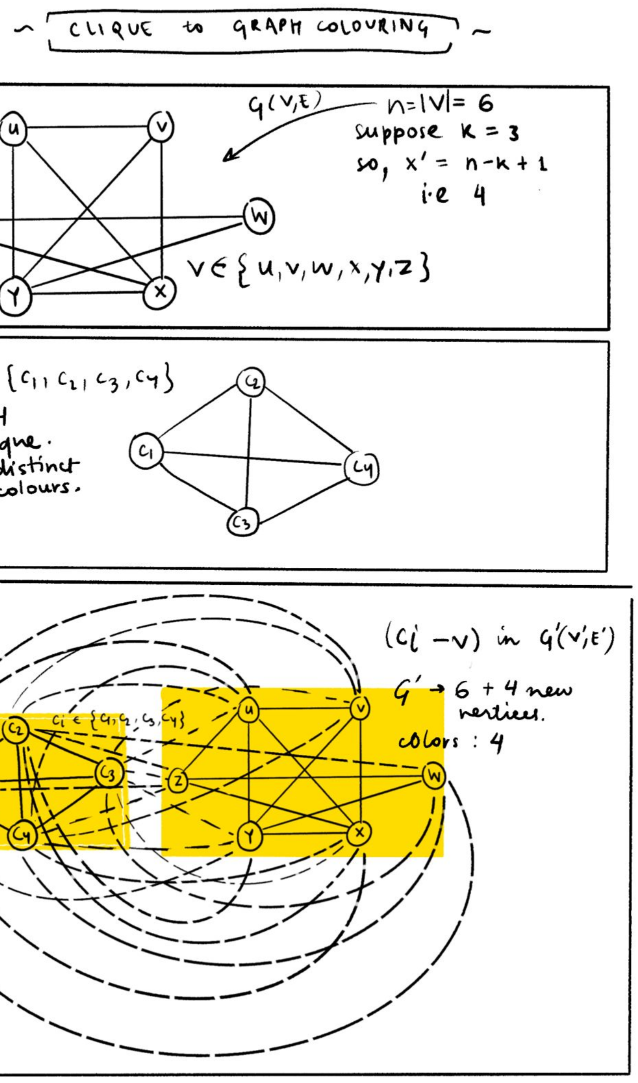


Figure 9: Clique to Graph Coloring Reduction

7.4 Graph Colouring: Brute Force Method

One direct way to solve Graph Colouring is to try every possible assignment of k colors to the n vertices and check whether it is valid.

Algorithm For each possible colouring of all n vertices using at most k colors:

- For every edge $\{u, v\} \in E$, check if the endpoints u and v have different colors.
- If every edge satisfies this condition, return 1 (a valid k-colouring ex-

ists).

- If no colouring works, return 0 (no valid k-colouring exists).

Time Complexity: Number of possible colourings: k^n (each of the n vertices can take any of k colors). For each colouring, we must check all m edges to ensure adjacent vertices have different colors. So, running time is $O(k^n \cdot m)$.

Space Complexity: Space to store the graph:

- Adjacency matrix: $O(n^2)$
- Adjacency lists: $O(n + m)$
- Space to store one colouring: $O(n)$

So total space: $O(n^2)$ (matrix) or $O(n+m)$ (lists) plus $O(n)$ temporary storage. The time complexity is clearly non-polynomial, making this approach infeasible for larger graphs. Hence, for practical purposes, we adopt heuristic or approximation algorithms.

7.5 Graph Colouring: Greedy Heuristic Algorithm

As covered earlier, heuristic algorithms are practical methods that try to find a good solution quickly. There is no guarantee on how close the solution is to the optimal colouring or the performance in the worst case. That is, theoretically, there is no solid formula that describes the outcome of the algorithm.

Why Approximation Algorithms Cannot Exist (for general k-colouring)
An approximation algorithm must be polynomial-time and give a provable guarantee, e.g.:

$$\text{ALG}(G) \leq \alpha \cdot \text{OPT}(G)$$

where α is the approximation ratio. However, for general Graph Colouring (chromatic number minimization), no polynomial-time algorithm is known that guarantees a colouring within a constant factor of the optimal number of colors, unless P = NP. Hence, we cannot have a formal approximation algorithm for general graphs. Instead, we use heuristic algorithms, which are fast and often perform well in practice. Here, similar to the previous cases, we will be using the optimization version of the Graph Colouring problem, namely, MINColour (to minimize the number of colours we use).

Input: Graph $G = (V, E)$
Question: Assign colors to vertices so that adjacent vertices get different colors, using as few colors as possible.
Output: A valid colouring (number of colors may not be optimal).

Greedy Algorithm

1. Order vertices using a heuristic (commonly by decreasing degree).
2. Initialize color assignment: uncolored for all vertices.
3. Iterate over vertices: For each vertex $v \in V$ in the chosen order:
   - **Determine Forbidden Colors:** Collect all colors already assigned to neighbors of v.
   - **Assign Smallest Possible Color:** Assign v the smallest numbered color not used by its neighbors.

We can return the colours of the graphs and the number of colours used. This number would be $\geq$ to the optimal number.

Why This is Only a Heuristic (Not an Approximation Algorithm)

- **No constant-factor guarantee:** There is no polynomial-time algorithm that always returns a colouring using a bounded multiple of the optimal number of colors (unless P=NP).
- **Local decisions:** The greedy algorithm assigns colors vertex by vertex based only on current neighbors; it does not globally optimize the total number of colors.
- **Practical performance:** Often works well on structured or sparse graphs, but for adversarial or dense graphs, it can use many more colors than optimal.

Thus, greedy colouring is heuristic, not an approximation algorithm.

How Good Is This? Heuristic algorithms are measured by how fast they are. **Speed:** The greedy algorithm runs fast in polynomial time: Checking neighbors for each vertex: $O(n + m)$ total for all vertices with adjacency lists. Sorting vertices (optional, e.g., by degree): $O(n \log n)$. So total time: $O(n+m)$ to $O(n \log n + m)$ depending on implementation. **Trade-off:** The algorithm prioritizes speed and feasibility over optimality and guaranteed quality.

Time Complexity

- Sorting vertices: $O(n \log n)$.
- Assigning colors: Each vertex checks its neighbors to find forbidden colors: $O(n + m)$ with adjacency lists, $O(n^2)$ with adjacency matrix.
- Overall: $O(n + m)$ to $O(n^2)$, depending on representation.

Space Complexity

- Adjacency matrix: $O(n^2)$.
- Adjacency lists: $O(n + m)$.

- Color bookkeeping: $O(n)$.
- Total: Linear-to-quadratic depending on representation.

The greedy colouring algorithm provides a fast polynomial-time heuristic for the graph colouring problem. It does not guarantee the minimum number of colors, but works well in practice on many graph instances, making it a useful practical tool despite lacking a formal approximation ratio.

7.6 GRAPH COLORING: Heuristic Algorithm – DSATUR

DSATUR (Degree of Saturation) is another heuristic for the k-colouring question that performs better than the simple greedy approach. It dynamically chooses the next vertex to color based on how **constrained** it is, i.e., how many distinct colors its neighbors already has. Like the previous cases, DSATUR does not guarantee an optimal number of colors, but in practice it often produces near-optimal colorings efficiently. Thereby, it is not an approximation algorithm, but simply an approximation. Here too, we will be using the optimization problem k-colouring to apply DSATUR to.

Input: Graph $G = (V, E)$
Question: Assign colors to vertices so that adjacent vertices get different colors, using as few colors as possible.
Output: A valid coloring (may not be optimal, but it must be feasible).

Algorithm (DSATUR Heuristic)

1. Initially, all vertices are uncolored.
2. For each vertex v, we track its saturation degree as the number of different colors assigned to its neighbors. Initially, all saturation degrees are 0.
3. **Select First Vertex:** Pick the vertex with the highest degree (number of neighbors) as the first to color. Assign it the first color (say, color 1).
4. **Iterate:** Repeat until all vertices are colored:
   (a) **Choose Vertex:** Pick an uncolored vertex with the highest saturation degree (largest number of distinct colors among neighbors). Break ties by choosing the vertex with the largest degree.
   (b) **Assign Color:** Assign the smallest possible color that does not conflict with neighbors.
   (c) **Update Saturation Degrees:** After coloring, update the saturation degree of all uncolored neighbors of the chosen vertex.
5. This is the final colouring of the vertices.

Why This is a Heuristic (Not an Approximation Algorithm)

- **No provable worst-case guarantee:** DSATUR does not guarantee that the number of colors used is within a factor of the optimal chromatic number.
- **Local decisions:** It colors the vertex with highest saturation first but does not backtrack; future decisions may require extra colors.
- **Practical effectiveness:** Often yields near-optimal colorings, especially for sparse or structured graphs, but can use more colors on dense/adversarial graphs.

Thus, DSATUR is heuristic, not approximation.

How Good Is This? Speed: DSATUR runs in polynomial time: Selecting vertex with max saturation: $O(n)$ per step $\rightarrow O(n^2)$ overall. Updating saturation degrees: $O(m)$ total. Overall typical complexity: $O(n^2 + m)$. Trade-off: Achieves better colorings than naive greedy at a small extra computational cost.

Time Complexity

- Vertex selection: $O(n)$ per step $\rightarrow O(n^2)$ total.
- Saturation updates: $O(m)$ total.
- Overall: $O(n^2 + m)$ for adjacency lists; $O(n^2)$ for adjacency matrix.

Space Complexity

- Graph representation: Adjacency matrix $O(n^2)$ or Adjacency lists $O(n+m)$.
- Saturation tracking: $O(n)$.
- Color assignment: $O(n)$.

DSATUR is a polynomial-time heuristic for the graph coloring problem that often performs better than simple greedy coloring. It prioritizes vertices that are most constrained by previously assigned colors, but does not guarantee the minimum number of colors. It is fast, practical, and widely used for real-world graphs.

8 Set Cover

Set Cover is a classic decision and combinatorial problem that asks this: "Given a universe of elements and a collection of sets whose union equals the universe, can you select at most k sets whose union still covers the entire universe?"

8.1 Definitions

- **Universe U:** A set of elements $\{e_1, e_2, \dots, e_n\}$.
- **Collection of subsets $S = \{S_1, S_2, \dots, S_m\}$:** Each $S_i \subseteq U$.
- **Set cover:** A selection of subsets from S such that their union equals U.

Input: universe U of n elements, collection of subsets $S = \{S_1, S_2, \dots, S_m\}$ and integer k
Question: Can we select at most k subsets from S whose union equals U?
Output:

- Yes $\rightarrow$ if such a selection exists (decision version satisfiable)
- No $\rightarrow$ if no such selection exists

8.2 Classifying Set Cover (NP-Completeness)

Set Cover can be verified in polynomial time. Given a candidate selection of at most k sets, we can check if the union of these sets equals the universe U in $O(n \cdot k)$ time. Thus, Set Cover belongs to NP. To show that Set Cover is NP-Complete, we can rely on a polynomial-time reduction from Graph Coloring to SC, since we have already proved that Graph Colouring is NPC previously.

Start with a Graph Coloring instance: **Input:** Graph $G = (V, E)$ and an integer k.

Construct a Set Cover instance

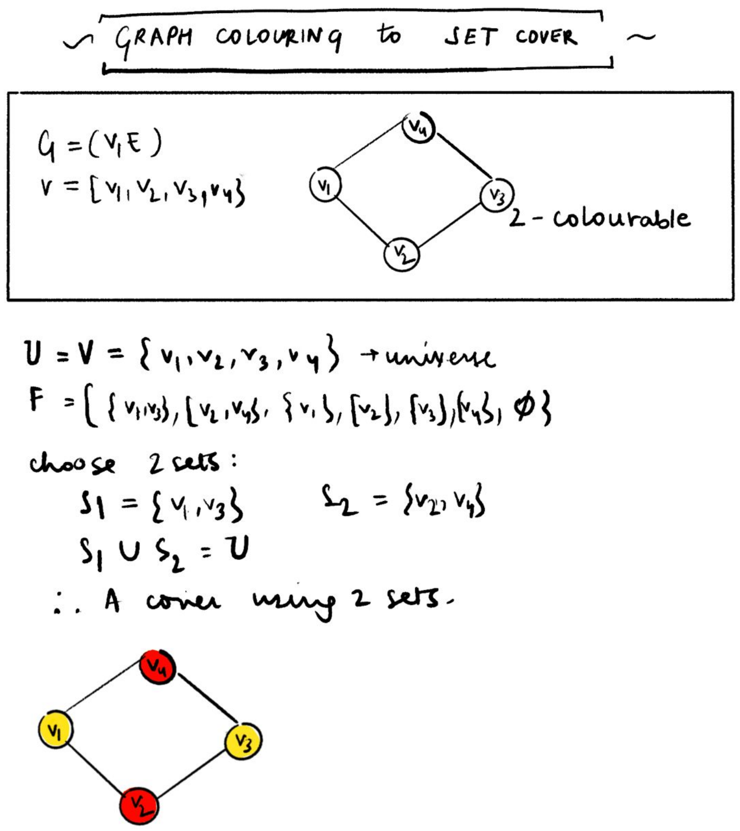
- **Universe U:** Let the universe U consist of all vertices in the graph: $U = V = \{v_1, v_2, \dots, v_n\}$. Each element of U corresponds to a vertex that needs to be "covered" by some color.
- **Collection of subsets S:** Think of each **color class** as a subset of vertices. A color class can contain any set of vertices that are **independent** (no edges between them) because vertices in the same color class must not be adjacent. So S consists of all possible independent sets in the graph. Each independent set is a valid subset that can be assigned a single color.

Set cover question: Can we select at most k subsets from S whose union covers all vertices in V? If yes, each chosen subset can be assigned a unique color. Since each subset is an independent set, no adjacent vertices get the same color.

Why this is correct?

- If the graph is k-colorable then: There exists a coloring of G with k colors. Each color class is an independent set (no edges inside), so selecting these k subsets from S covers all vertices. Thus a Set cover exists.
- If the Set Cover instance has a solution using at most k subsets: Each selected subset is an independent set (by construction), and together they cover all vertices. Assigning a unique color to each subset gives a valid k-coloring of the original graph. Thus Kcolour is satisfied.

Constructing the universe U and the collection S can be done in polynomial time (for small enough k or using implicit representation of independent sets). Therefore, the reduction is valid. Since **Graph Coloring is NP-Complete**, and we have reduced it to Set Cover in polynomial time, **Set Cover is also NP-Complete**.



~ GRAPH COLOURING to SET COVER ~

$G = (V, E)$

$V = \{v_1, v_2, v_3, v_4\}$

$U = V = \{v_1, v_2, v_3, v_4\} \rightarrow$ universe

$F = \{\{v_1, v_3\}, \{v_2, v_4\}, \{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \phi\}$

choose 2 sets :

$S_1 = \{v_1, v_3\}$ $S_2 = \{v_2, v_4\}$

$S_1 \cup S_2 = U$

$\therefore$ A cover using 2 sets.

Figure 10: Graph Coloring to Set Cover reduction

8.3 Set Cover: Brute Force Method

One direct way to solve Set Cover is to try every combination of k subsets and check whether their union equals the universe.

Algorithm: For each subset $C \subseteq S$ of size at most k:

- Compute the union of subsets in C.
- If $\bigcup_{S_i \in C} S_i = U$, return Yes.
- If no combination works, return No.

Time Complexity: Number of combinations: $\binom{m}{k}$ (choose k sets from m). For each combination, computing the union takes $O(n \cdot k)$. Total running time: $O(\binom{m}{k} \cdot n \cdot k)$. Worst-case: exponential in m (infeasible for large instances).

Space Complexity:

- Store universe: $O(n)$.
- Store subsets: $O(\sum |S_i|)$.
- Temporary storage for candidate union: $O(n)$.
- Total: $O(n + \sum |S_i|)$.

The time complexity is clearly non-polynomial, making brute force infeasible for practical purposes. Hence, heuristic or approximation algorithms are used.

8.4 Set Cover: Greedy Heuristic Algorithm

For practical purposes, we use a greedy algorithm that iteratively selects the set covering the largest number of uncovered elements. This algorithm is fast and provides a good solution, though not necessarily optimal.

Input: Universe U, Collection of subsets $S = \{S_1, \ldots, S_m\}$
Output: A selection of subsets $C \subseteq S$ that covers all elements of U.

Algorithm:

1. **Initialize:** $C = \emptyset$ (current cover), $U_{\text{remaining}} = U$ (elements yet to cover).
2. **Repeat until $U_{\text{remaining}}$ is empty:**
   (a) **Pick Set:** Choose subset $S_i \in S$ that covers the largest number of elements in $U_{\text{remaining}}$.
   (b) **Add to Cover:** Add S_i to C.
   (c) **Update Remaining:** Remove elements of S_i from $U_{\text{remaining}}$.
3. Return C.

Why This is a Heuristic / Approximation

- **Polynomial-time heuristic:** The greedy algorithm runs fast ($O(m \cdot n)$) but does not always give the minimum number of sets.
- **Provable bound:** Unlike Graph Coloring, Set Cover's greedy algorithm has a logarithmic approximation guarantee: it uses at most $H(n)$ times the optimal number of sets ($H(n) = 1 + 1/2 + \cdots + 1/n \approx \ln n$).

- **Practical effectiveness:** Often performs very well in real-world instances.

How Good Is This? Speed: Each iteration selects the set covering the most uncovered elements, costing $O(m \cdot n)$ in naive implementations. Trade-off: Chooses speed and feasibility over absolute optimality. In practice, it's very effective.

Time Complexity

- Picking best set per iteration: $O(m \cdot n)$.
- At most n iterations (each iteration covers at least one new element).
- Total: $O(m \cdot n^2)$ in naive implementation.
- Using clever data structures (heap, bitsets): can reduce to $O(m \cdot n \log n)$.

Space Complexity

- Store universe: $O(n)$.
- Store subsets: $O(\sum |S_i|)$.
- Store current cover and bookkeeping: $O(n)$.
- Total: $O(n + \sum |S_i|)$.

The greedy Set Cover algorithm is a fast, polynomial-time heuristic/approximation algorithm for covering problems. It does not guarantee an optimal cover, but the logarithmic approximation factor ensures it is never far from the optimal in the worst case. However, it is much faster than the brute/other algorithms and hence, a valid heuristic.

9 Reductional Analysis

"All NPC problems are the same problem, just wearing different costumes"

Throughout this project, we have performed the following reductions:

$$3SAT \rightarrow \text{Vertex Cover} \rightarrow \text{CLIQUE} \rightarrow \text{Graph Colouring} \rightarrow \text{Set Cover}$$

Clearly, through examination of all of these algorithms, one key observation emerges: **all the NP-Complete problems we studied are interconnected through polynomial-time reductions**. By reducing one problem to another, we can:

1. **Demonstrate NP-Completeness:** For example, **Graph Coloring** can be reduced to **Set Cover**, and **Vertex Cover** can be reduced to **Clique**. These reductions show that if we could solve one problem

efficiently (in polynomial time), we could also solve all others efficiently, since ultimately, all NPC problems are basically the same problem.

2. **Understand relative difficulty:** Reductions reveal that the "hardness" of these problems is essentially the same. Problems like Clique, Graph Coloring, and Set Cover, while superficially different, are all computationally equivalent in the sense that solving one allows solving the others.

3. **Guide algorithm design:** By knowing the reductions, we can adapt algorithms from one problem to another. For example, greedy heuristics for Set Cover can inspire heuristics for coloring or other covering problems, since ultimately, all the problems have the same hardness.

4. **Highlight the role of exact vs. heuristic approaches:** Brute-force algorithms are universally infeasible due to exponential growth. Heuristics and approximation algorithms provide practical solutions, but **their design often leverages insights gained from reductions**. This will be elaborated on in the benchmarking results and graphs attached.

5. **Illustrate theoretical boundaries:** Reduction chains emphasize **why approximation guarantees are limited** in some problems (like MAX-Clique or Graph Coloring), versus problems like Set Cover, which allow logarithmic approximation.

10 Summary

In this project, we explored a wide range of **NP-Complete problems**, analyzing both their **exact, heuristic and approximative approaches**. We carefully examined their **time and space complexities**, practical feasibility, and theoretical limits.

- **3SAT:** Developed **randomized and literal-flipping heuristics** for MAX-3SAT, demonstrating how local, greedy improvements can effectively increase the number of satisfied clauses.

- **Vertex Cover:** Explored the **brute-force approach** and two polynomial-time **approximation algorithms**: one based on **maximal matching** and another using **LP relaxation** with a formal proof of approximation ratio.

- **Graph Coloring:** Investigated **brute-force coloring**, the **greedy heuristic**, and **DSATUR**, a saturation-based heuristic. Highlighted why **approximation algorithms are theoretically hard**, emphasizing that no polynomial-time algorithm guarantees a constant-factor coloring for general graphs.

- **Clique:** Studied **brute-force enumeration** and **greedy** methods, illustrating the dramatic difference in running times between naive search and heuristic methods. Discussed why **approximation is infeasible** for MAX-Clique due to known hardness results from PCP theory.
- **Set Cover:** Applied **brute-force search** and the **greedy approximation algorithm**, noting that the greedy heuristic provides a **logarithmic approximation guarantee** while remaining efficient in practice.

Overall Insights:

- **Brute-force algorithms** are universally infeasible for larger instances due to exponential growth.
- **Heuristic and approximation strategies** allow practical solutions, but their guarantees vary: some (like Set Cover) have provable bounds, while others (like MAX-Clique and Graph Coloring) are purely heuristic.

11 Conclusion

The systematic reduction of NP-Complete problems in this report not only **demonstrates their theoretical equivalence** but also provides **practical insights** for designing efficient heuristics. Understanding these connections helps explain why certain algorithmic strategies work across multiple problems and why exact solutions are generally infeasible.

Bibliography

- https://www.youtube.com/watch?v=YX40hbAHx3s
- https://www.claymath.org/wp-content/uploads/2022/06/pvsnp.pdf
- https://www.scottaaronson.com/papers/pnp.pdf
- https://en.wikipedia.org/wiki/Karp%27s_21_NP-complete_problems
- https://en.wikipedia.org/wiki/List_of_NP-complete_problems