GROVER SEARCH ALGORITHM FOR THE LIST COLORING PROBLEM

PRESENTED BY - SETHURATHIENAM IYER FOR THE IIT-DELHI'S COURSE ON QUANTUM COMPUTING AND MACHINE LEARNING-02

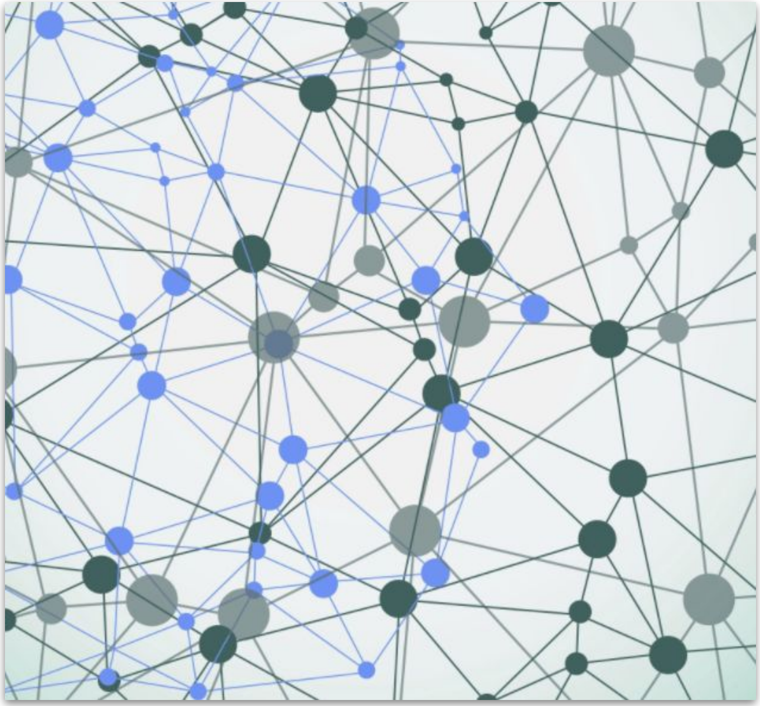
Table of Contents

Graphs - Introduction

Graphs are **mathematical structures** that **consist** of **nodes** (vertices) and **connections** between them (edges). They are used to model relationships and connections between different elements.

Nodes in a graph can **represent** various **entities**, such as cities, people, or objects, while **edges represent** the **relationships** or connections between these entities.

Graphs can be **directed** (where edges have a specific direction) or **undirected** (where edges have no direction).



Graphs are like a mathematical way of showing connections between things. It's a handy way to model relationships and understand connections between different elements.

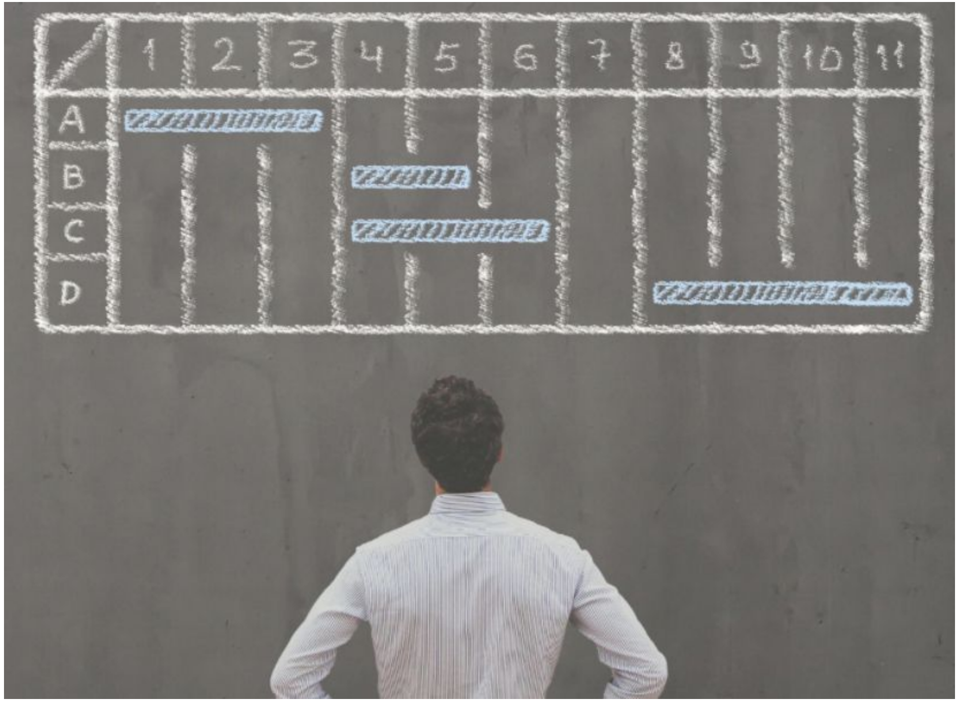
Understanding the Graph Coloring and List Coloring Problem

The Graph Coloring Problem involves assigning colors to the nodes of a graph in such a way that **no two adjacent nodes share the same color.**

The main objective is to minimize the number of colors used while satisfying the coloring constraints.

It has important real-world applications, such as scheduling tasks with time constraints, assigning frequencies in wireless networks, and solving register allocation in computer programming.

The List Coloring Problem is a variant of the Graph Coloring Problem where **each node is associated with a list of permissible colors**, and the goal is to find a proper coloring while adhering to these color lists.



The goal is to color these nodes in such a way that no two connected nodes have the same color while using as few colors as possible. The List Coloring Problem is a slight twist, where each node has a list of allowed colors.

Boolean Satisfiability and NP-Completeness

Boolean Satisfiability, or SAT, is a fundamental problem in computer science and logic which **determines whether a given Boolean formula can be satisfied** in a way that the entire formula evaluates to **true**.

It is widely used in various areas, including automated reasoning, hardware and software verification, and artificial intelligence.

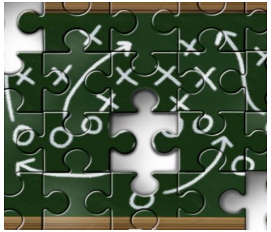
P is a complexity class that represents the set of problems that can be **solved in polynomial time**.

NP (Nondeterministic Polynomial time) is a complexity class that represents the set of problems for which a solution can be **verified in polynomial time.**

All problems in P are also in NP, but not all problems in NP are known to be in P.

SAT was the first problem proven to be **NP-complete**, as per the **Cook-Levin theorem**. This means all problems in the NP complexity class, which includes many decision and **optimization problems**, are at most as **hard to solve as SAT**.

The proof that **SAT** is **NP-Complete** involves two parts: showing that **SAT is in NP**, and that **any NP-Complete problem is reducible to SAT**



How List Coloring can be written as a SAT Problem

Consider the graph on the right. There are **5 vertices** (V1,V2..V5) and **3 colours** (RED, GREEN & BLUE).

Define Variables: Assign a Boolean variable for each node and colour pair, e.g., V1RED, V1GREEN, V1BLUE for node V1 and colours RED, GREEN, and BLUE.

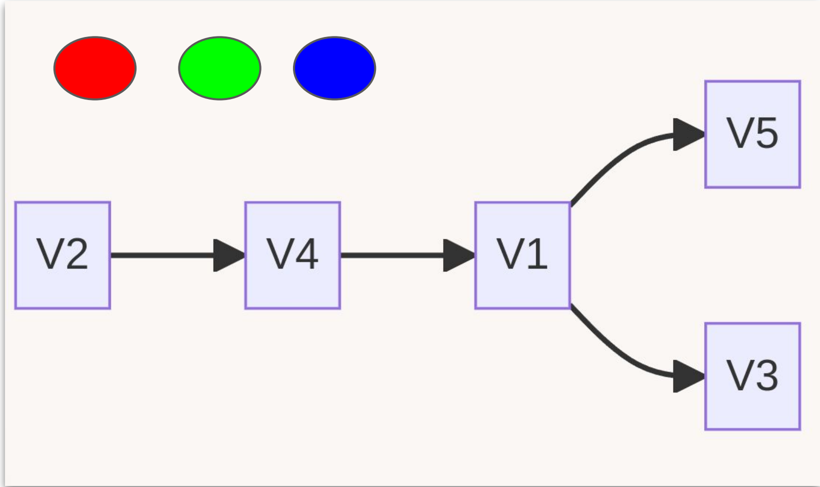
Node Colour Assignment: Ensure each node gets exactly one colour by creating a logical OR expression for each node, e.g., '(V1RED | V1GREEN | V1BLUE)'.

Adjacent Node Constraints: Prevent adjacent nodes from sharing a colour by using logical expressions for adjacent node pairs, e.g., '~(V1RED & V4RED)' ensures V1 and V4 don't both get RED.

List Color Constraints: By not including certain colours in the logical OR expression for a node, we can exclude those colours from being assigned to that node.

*For example, if we want to omit the colour **RED** from node V1, we would write the expression as '(V1GREEN | V1BLUE)'.*

Reducing the problem space Map the three colours to two variables to reduce the number of overall variables.




```
~((~v3 & ~v4) & (~v7 & ~v8)) & ~((~v3 & v4) & (~v7 & v8)) & ~((v3 & ~v4) & (v7 & ~v8)) & ~((~v7 & ~v8) & (~v3 & ~v4))
& ~((~v7 & v8) & (~v3 & v4)) & ~((v7 & ~v8) & (v3 & ~v4)) & ~((~v7 & ~v8) & (~v1 & ~v2)) & ~((~v7 & v8) & (~v1 & v2))
& ~((v7 & ~v8) & (v1 & ~v2)) & ~((~v1 & ~v2) & (~v7 & ~v8)) & ~((~v1 & v2) & (~v7 & v8)) & ~((v1 & ~v2) & (v7 & ~v8))
& ~((~v1 & ~v2) & (~v5 & ~v6)) & ~((~v1 & v2) & (~v5 & v6)) & ~((v1 & ~v2) & (v5 & ~v6)) & ~((~v5 & ~v6) & (~v1 &
~v2)) & ~((~v5 & v6) & (~v1 & v2)) & ~((v5 & ~v6) & (v1 & ~v2)) & ~((~v1 & ~v2) & (~v9 & ~v10)) & ~((~v1 & v2) & (~v9
& v10)) & ~((v1 & ~v2) & (v9 & ~v10)) & ~((~v9 & ~v10) & (~v1 & ~v2)) & ~((~v9 & v10) & (~v1 & v2)) & ~((v9 & ~v10) &
(v1 & ~v2)) & ((~v1 & ~v2) | (~v1 & v2) | (v1 & ~v2)) & ~(v1 & v2) & ((~v3 & ~v4) | (~v3 & v4) | (v3 & ~v4)) & ~(v3 &
v4) & ((~v5 & ~v6) | (~v5 & v6) | (v5 & ~v6)) & ~(v5 & v6) & ((~v7 & ~v8) | (~v7 & v8) | (v7 & ~v8)) & ~(v7 & v8) &
((~v9 & ~v10) | (~v9 & v10) | (v9 & ~v10)) & ~(v9 & v10)
```

5 vertices, 3 colors (represented in two bits) = 10 bits

Using Grover's Search Algorithm as Q-SAT Solver

Grover's Algorithm: Introduced by Lov Grover in 1996, this quantum algorithm **can perform unstructured searches in an unordered database faster than classical algorithms**. It is often used as a subroutine in other algorithms.

The algorithm starts with **all states in a superposition.** This is our **unordered database**.

Conditional Phase Oracle with Phase Shift - 1 is applied to the superposition state, changing the signs of the amplitudes for the target states followed by **H** on each qubit followed by **conditional phase shift of -1** on Computational Basis except **|0>** followed by **H** on each qubit in the register.

This process is **iterated** a number of times **proportional to the square root of the size of the database**, until a good solution is found, reducing the search time from linear (O(N)) to square root of N (O(√N)).

Grover's algorithm is a probabilistic algorithm, which means that even in the best case it has a non-zero failure probability.

Consider the SAT expression $\neg x \wedge y$ This evaluates to true if x=0 and y=1. We use **PhaseOracle** in Qiskit to compile this into phase oracle which marks the state.

The register starts in the state: |**register**⟩=|00⟩. After applying H to each qubit the register's state transforms to: |register⟩=1√4∑i∈{0,1}2|i⟩

=1/2 (|00⟩+|01⟩+|10⟩+|11⟩) (We will omit ½ from now)

Then, the phase oracle is applied to get: |register⟩= (|00⟩−|01⟩+|10⟩+|11⟩)

Then H acts on each qubit again to give: |register⟩=(|00⟩+|01⟩−|10⟩+|11⟩)

Now the conditional phase shift is applied on every state except |00⟩

|register⟩=(|00⟩−|01⟩+|10⟩−|11⟩)

Finally, the first Grover iteration ends by applying H

again to get: |register⟩=|01⟩

By following the above steps, the valid item is found in a single iteration. This is because for N=4 and a single valid item(M=1), K_*optimal*=1

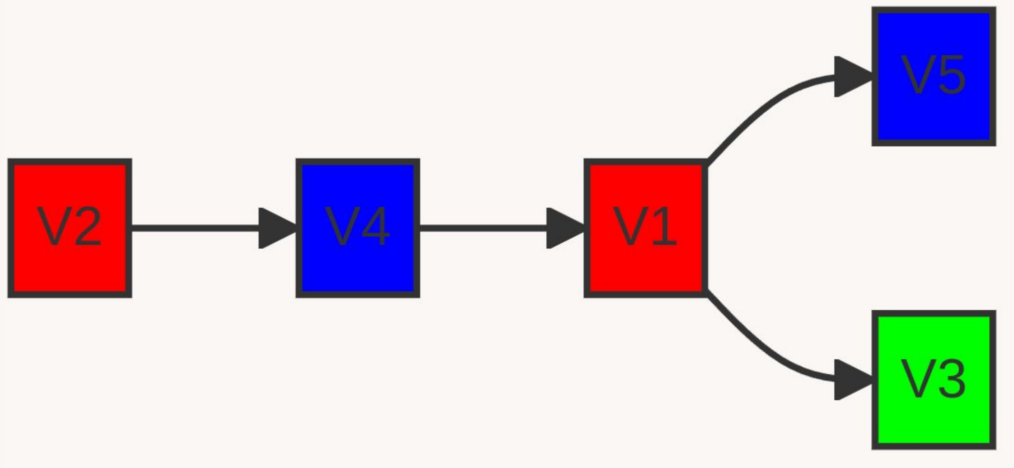
$$K_{\text{optimal}} = \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{M}} - \frac{1}{2} \right\rfloor$$

Graph + Color Lists => Constraint Satisfaction Problem => SAT Expression => Oracle of the SAT Problem => Grover's Algorithm => Measurement of Possible Marked State => Possible Valid Colouring

Code Overview - The Core Logic

```python
# Constraints for color assignments expressed in Boolean logic -
# Here we consider 4 colors - BLUE, RED, GREEN AND COLORLESS but 4rth is excluded
expression = [
    # Constraints for V2 and V4
    '~(V2RED & V4RED)', '~(V2GREEN & V4GREEN)', '~(V2BLUE & V4BLUE)',
    '~(V4RED & V2RED)', '~(V4GREEN & V2GREEN)', '~(V4BLUE & V2BLUE)',

    # Constraints for V4 and V1
    '~(V4RED & V1RED)', '~(V4GREEN & V1GREEN)', '~(V4BLUE & V1BLUE)',
    '~(V1RED & V4RED)', '~(V1GREEN & V4GREEN)', '~(V1BLUE & V4BLUE)',

    # Constraints for V1 and V3
    '~(V1RED & V3RED)', '~(V1GREEN & V3GREEN)', '~(V1BLUE & V3BLUE)',
    '~(V3RED & V1RED)', '~(V3GREEN & V1GREEN)', '~(V3BLUE & V1BLUE)',

    # Constraints for V1 and V5
    '~(V1RED & V5RED)', '~(V1GREEN & V5GREEN)', '~(V1BLUE & V5BLUE)',
    '~(V5RED & V1RED)', '~(V5GREEN & V1GREEN)', '~(V5BLUE & V1BLUE)',

    # Color assignment constraints
    '(V1RED | V1GREEN | V1BLUE) & ~V1COLORLESS',
    '(V2RED | V2GREEN | V2BLUE) & ~V2COLORLESS',
    '(V3RED | V3GREEN | V3BLUE) & ~V3COLORLESS',
    '(V4RED | V4GREEN | V4BLUE) & ~V4COLORLESS',
    '(V5RED | V5GREEN | V5BLUE) & ~V5COLORLESS'
]

# Combine all the constraints using logical AND
expression = ' & '.join(expression)

# Reduce the expression by replacing color assignments with Boolean logic expression
reduced_expr = get_reduced_expression(expression)

oracle = PhaseOracle(reduced_expr)
```

```python
# Run Grover's algorithm to find a valid color assignment
backend = Aer.get_backend('aer_simulator')
quantum_instance = QuantumInstance(backend, shots=32)
problem = AmplificationProblem(oracle, is_good_state=is_good_state)
result = Grover(quantum_instance=quantum_instance).amplify(problem)

# Interpret and print the results
interpret_results(result.assignment)
```



Code Overview - The Helper Functions

```python
def replace_color(vertex, color, pos):
    """
    Replaces the color in a vertex with corresponding Boolean logic expressions.

    Args:
        vertex (str): Vertex identifier.
        color (str): Color of the vertex.
        pos (int): Position of the vertex in the expression.

    Returns:
        str: Boolean logic expression representing the vertex color.
    """
    if color == 'RED':
        # RED: 00
        return '(~v{} & ~v{})'.format(pos, pos+1)
    elif color == 'GREEN':
        # GREEN: 01
        return '(~v{} & v{})'.format(pos, pos+1)
    elif color == 'BLUE':
        # BLUE: 10
        return '(v{} & ~v{})'.format(pos, pos+1)
    else:
        return '(v{} & v{})'.format(pos, pos+1)

def get_reduced_expression(expression):
    """
    Replaces the color assignments in the expression with Boolean logic expressions.

    Args:
        expression (str): Original expression.

    Returns:
        str: Expression with replaced color assignments.
    """
    vertices = ['V1', 'V2', 'V3', 'V4', 'V5']
    colors = ['RED', 'GREEN', 'BLUE', 'COLORLESS']

    for index, vertex in enumerate(vertices):
        pos = index * 2 + 1
        for color in colors:
            to_replace = vertex + color
            replace_with = replace_color(vertex, color, pos)
            expression = expression.replace(to_replace, replace_with)
    return expression
```

```python
def has_one_color(bitstring):
    """
    Checks if each vertex in a given bitstring has exactly one color.

    Args:
        bitstring (str): Bitstring representing a color assignment.

    Returns:
        bool: True if each vertex has one color, False otherwise.
    """
    for i in range(0, len(bitstring), 2):
        if int(bitstring[i:i+2], 2) > 2:
            return False
    return True

def has_different_colors(bitstring):
    """
    Checks if connected vertices in a given bitstring have different colors.

    Args:
        bitstring (str): Bitstring representing a color assignment.

    Returns:
        bool: True if connected vertices have different colors, False otherwise.
    """
    edges = [(2, 4), (4, 1), (1, 3), (1, 5)]  # Reflecting the graph
    for v1, v2 in edges:
        if bitstring[2*(v1-1):2*v1] == bitstring[2*(v2-1):2*v2]:
            return False
    return True

def is_good_state(bitstring):
    """
    Checks if a given bitstring represents a valid color assignment.

    Args:
        bitstring (str): Bitstring representing a color assignment.

    Returns:
        bool: True if the color assignment is valid, False otherwise.
    """
    return has_one_color(bitstring) and has_different_colors(bitstring)
```

```python
def get_color_map(color_code):
    """
    Maps a color code to the corresponding color name.

    Args:
        color_code (str): Two-bit color code.

    Returns:
        str: Color name corresponding to the color code.
    """
    color_map = {'00': "RED", '01': "GREEN", '10': "BLUE", "11": "COLORLESS"}
    return color_map[color_code]

def interpret_results(assignment):
    """
    Interprets and prints the assignment of colors to vertices.

    Args:
        assignment (str): Bitstring representing the color assignment.

    Returns:
        None
    """
    for x in range(0, len(assignment)-1, 2):
        vertex_number = x // 2 + 1
        color_code = assignment[x:x+2]
        color_name = get_color_map(color_code)
        message_str = f"Vertex Number {vertex_number} is assigned the colour {color_name}"
        print(message_str)
```

```
Vertex Number 1 is assigned the colour RED
Vertex Number 2 is assigned the colour RED
Vertex Number 3 is assigned the colour GREEN
Vertex Number 4 is assigned the colour BLUE
Vertex Number 5 is assigned the colour BLUE
```

Please Ask your Questions!

References

1. "Lectures on Satisfiability," A. Aho, Columbia University, [Online]. Available: http://www.cs.columbia.edu/~aho/cs3261/Lectures/L20-Satisfiability.html

2. "3-Coloring is NP-Complete," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/3-coloring-is-np-complete/

3. "Grovers Algorithm," IBM Quantum Computing, [Online]. Available: https://quantum-computing.ibm.com/composer/docs/iqx/guide/grovers-algorithm

4. "Grovers Algorithm," Grove Documentation, [Online]. Available: https://grove-docs.readthedocs.io/en/latest/grover.html

5. "Exploring Grovers Algorithm," Microsoft Quantum Katas, [Online]. Available: https://github.com/microsoft/QuantumKatas/tree/main/tutorials/ExploringGroversAlgorithm