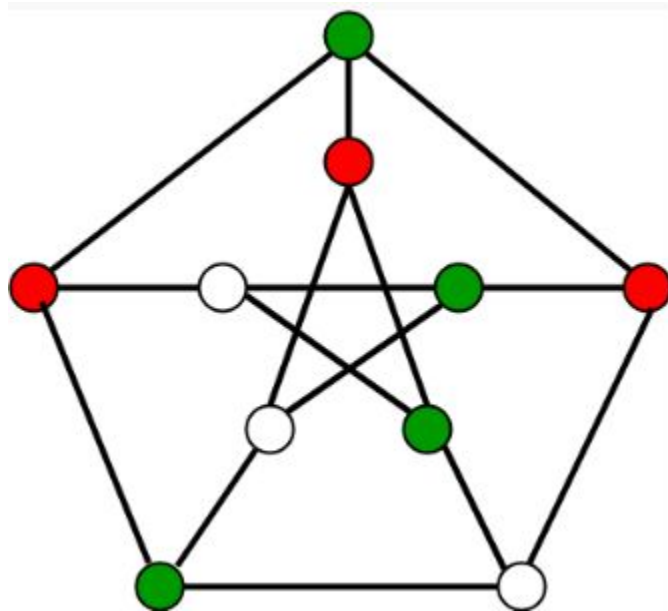


Min-Graph Coloring

David Nguyen and Christopher Simmons



Problem Presentation



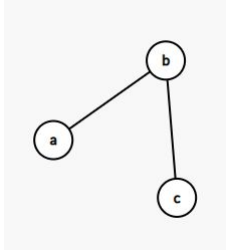
Exact Solution

Example Inputs and Outputs:

Incomplete Graphs:

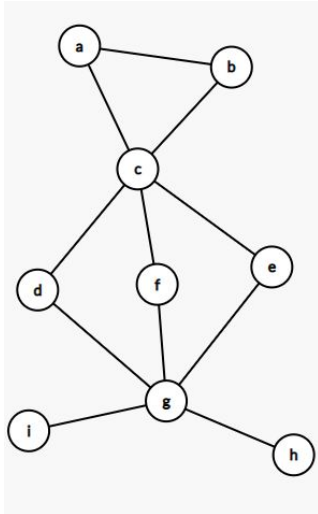
Input: 2

a b
b c



Input: 11

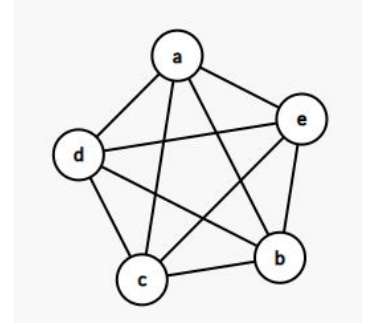
a b
a c
b c
c d
c e
c f
g d
g e
g f
g h
g i



Complete Graph:

Input: 10

a b
a c
a d
a e
b c
b d
b e
c d
c e
d e



Why is the problem important?

This problem has a wide range of applications in various fields such as:

1. Scheduling: This problem can be used to schedule tasks on a project network, where each vertex represents a task, and the edges represent the precedence constraints between the tasks.
2. Map coloring: This problem can be used to color regions on a map such that no two adjacent regions have the same color.
3. Sudoku: This game can be viewed as an instance of the problem, where the squares on the grid represent the vertices and the edges represent the constraints between the squares.

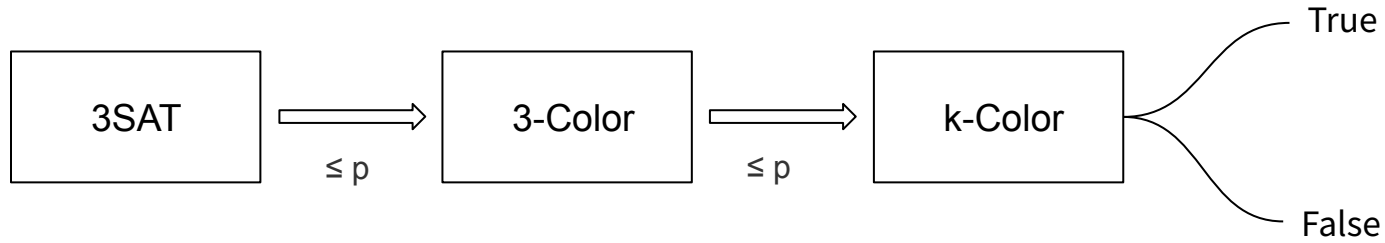
Explanation of the certifier process being polynomial

For each combination of colors, check if adjacent vertices have different colors. If any pair of adjacent vertices have the same color, the combination of colors is invalid.

The certifier process has a time complexity of $O(n^2)$ where n is the number of vertices. Therefore, the certifier process is polynomial.

Reduction problem

3SAT \rightarrow k-Coloring:



Explanation of the exact coded solution

Create a graph with the edges entered by the user.

Start at 1 color.

Use the itertools to create all possible combinations of colors for n vertices.

For each combination, check if it is valid.

If false, increment the number of colors, and do the same process.

If true, return that number of colors, and the color assigned to each vertex.

```
while not check:
    nums = []

    for i in range(num_colors):
        nums.append(i)

    lst = itertools.product(nums, repeat=len(graph))

    for val in lst:
        check, colors, nums = check_independent_set(val, graph)
        if check:
            break

    num_colors += 1
```

```
def check_independent_set(nums, graph):
    pair = zip(graph, nums)
    colors = {}

    for x in pair:
        colors[x[0]] = x[1]

    for u in graph:
        for v in graph[u]:
            if colors[u] == colors[v]:
                return False, colors, nums

    return True, colors, nums
```


Approximation

Degree of Saturation (DSatur) Approximation

The DSatur algorithm is a greedy approximation approach for the min-coloring problem

1. Iteratively colors each vertex in the graph by selecting the vertices with the highest degree of saturation (DS)
2. Selected vertices are colored with the first available color.
3. The the DS values of the adjacent uncolored vertices after each color assignment
4. Repeat until all vertices are colored.

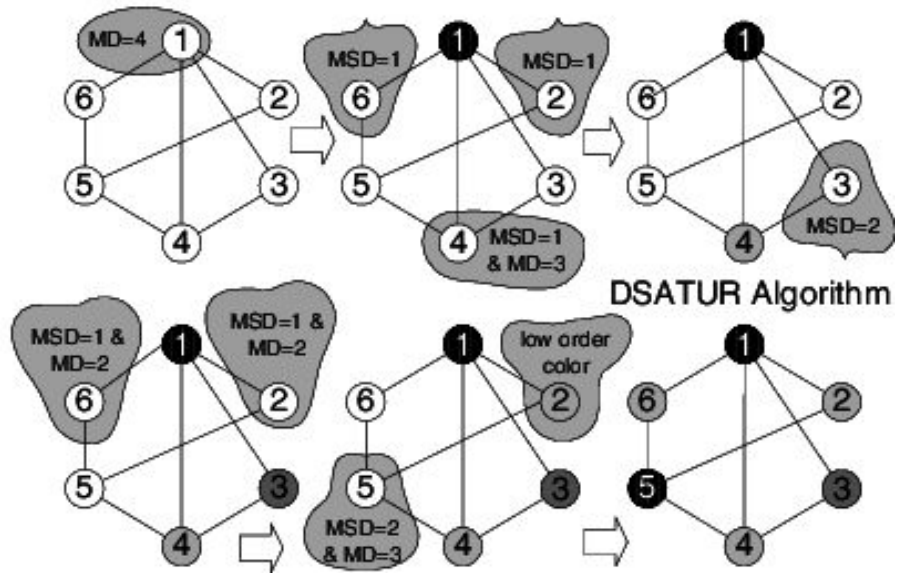


Figure by Darko Kirovski

Pseudocode

1. Choose a vertex with maximum degree and color it with the first available color.
2. While there are uncolored vertices:
 - a. Choose the uncolored vertex with the highest DS value and color it with the first available color.
 - b. Update the DS values of all uncolored vertices adjacent to the newly colored vertex.
 - c. If there are no vertices with $DS > 0$, choose a vertex with maximum degree that has not been colored and color it with the first available color.
3. Return the final coloring.

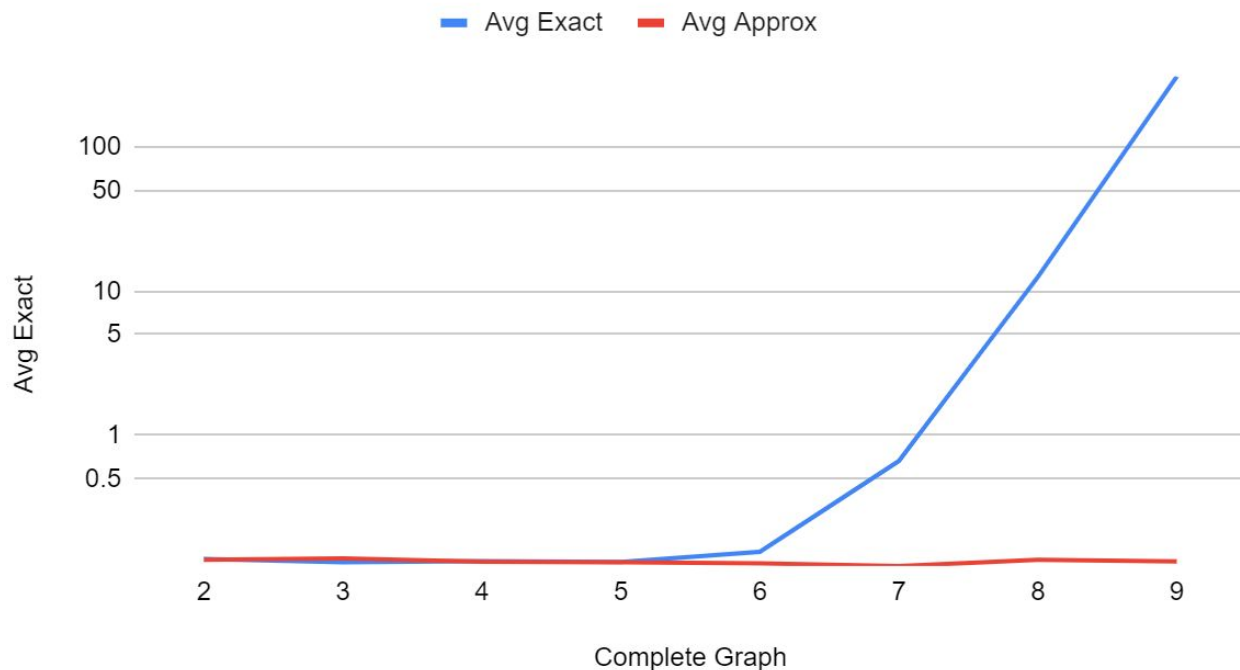
This algorithm has a runtime of $O(n^2)$, what is interesting is DSatur is an exact algorithm if the input graph is a Bipartite, Cycle, or a Wheel graph.

Exact vs Approximation, side by side

The difference between $O(e^n)$ and $O(n^2)$ can be seen in this chart comparing the exact and DSatur approaches on complete graphs.

- Exact solution ran in >5 minutes at 9 vertices

Avg Exact vs. Complete Graph

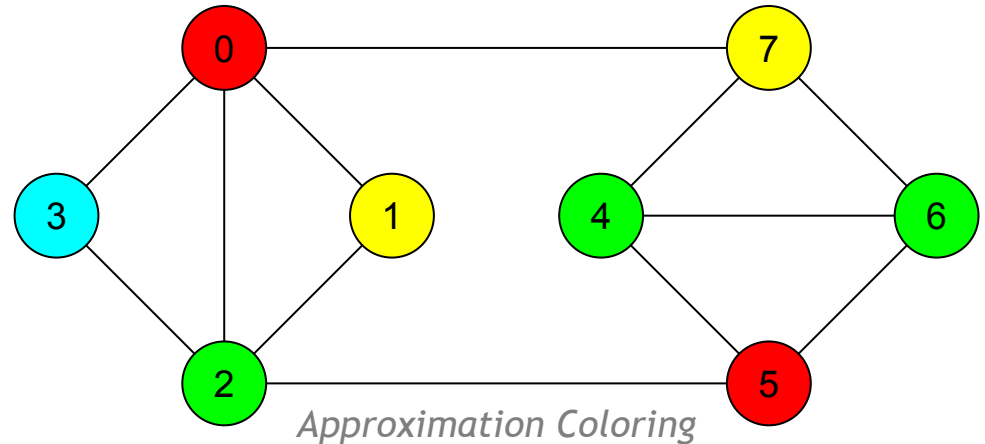
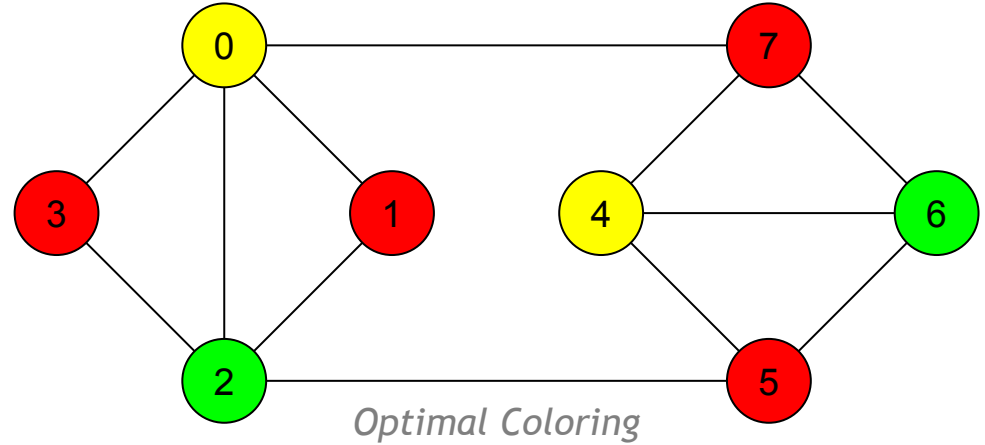


Fail Cases

Since the approximation is Heuristic, the algorithm can only analyze locally optimal choices for the coloring assignment.

This means that there are cases where the algorithm will not produce the chromatic number of the graph.

Figures are Test-9



Upper bounding

Brooke's Theorem:

For any connected undirected graph G with maximum degree Δ , the chromatic number of G is at most Δ , unless G is a complete graph or an odd cycle, in which case the chromatic number is $\Delta + 1$.

Upper bounds for graph coloring can be found via Brooks' Theorem, a tool helps us state chromatic numbers:

