A Report

On

K-Coloring Problem

Submitted in requirement for the course

Lab-Based Project(CSN-300)

of Bachelor of Technology in Computer Science and Engineering

by

Alankrit Anand(19114006)

Nivedit Rathore(19114059)

Rahul Katara (19114070)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, ROORKEE

ROORKEE- 247667 (INDIA)

4 May, 2022

[illegible]

R Niyogi 4.5.2022

K-Coloring Problem

Problem Statement:

We're going to work on `Graph Coloring` challenges. It's an `NP-hard problem` in which each node in the graph is supposed to be colored in such a way that any two nodes connected by an edge are colored differently. The goal is to utilize as few colors as possible.

We looked at several `heuristics methods` and found the following to be promising:

> Dsatur
> Recursive Largest First
> ImXRLF
> MCS

After implementing these algorithms, we intend to `compare their performance` on different kinds of graphs such as:

> Dense Graph
>
> Sparse Graph
>
> Bipartite Graph

The `score` can be assigned to algorithms based on the `number of colors` used by the algorithm, and the number of nodes and edges in the graph.

Finally, we can use these algorithms to `solve real-world problems` such as:

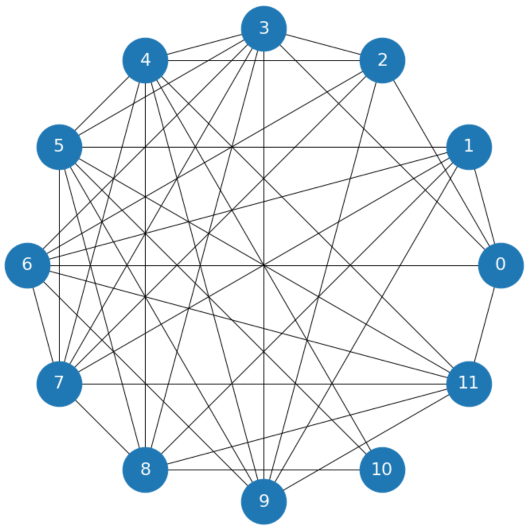
- Preparing exam schedule where every color represents a time slot. Exams are represented by nodes and two nodes are connected by an edge if they have a student in common.
- Solving sudoku.
- Checking if a graph is bipartite

Algorithms:

For graph coloring, a range of `heuristic-based approximation techniques` are available, which may frequently create quite pleasing results. In this report, we will look at three algorithms that work by assigning a color to each vertex one at a time using criteria designed to keep the **total number of colors as small as possible.**

We will **use random graphs** designated by $G_{n,p}$ for the experiment, where n represents the number of vertices and p signifies the percentage likelihood of an edge connecting any two sets of vertices.

To **demonstrate** how the algorithms function, we will use the following graph:



$G_{12,60}$

Greedy Approach:

Pick a vertex at random and assign it the smallest color that hasn't already been allocated to its neighbors. Its solutions may be `suboptimal`. However, given the suitable vertices sequence, **greedy may offer an optimum solution for any network**. In this greedy method, we select a random order and color the nodes in that order.

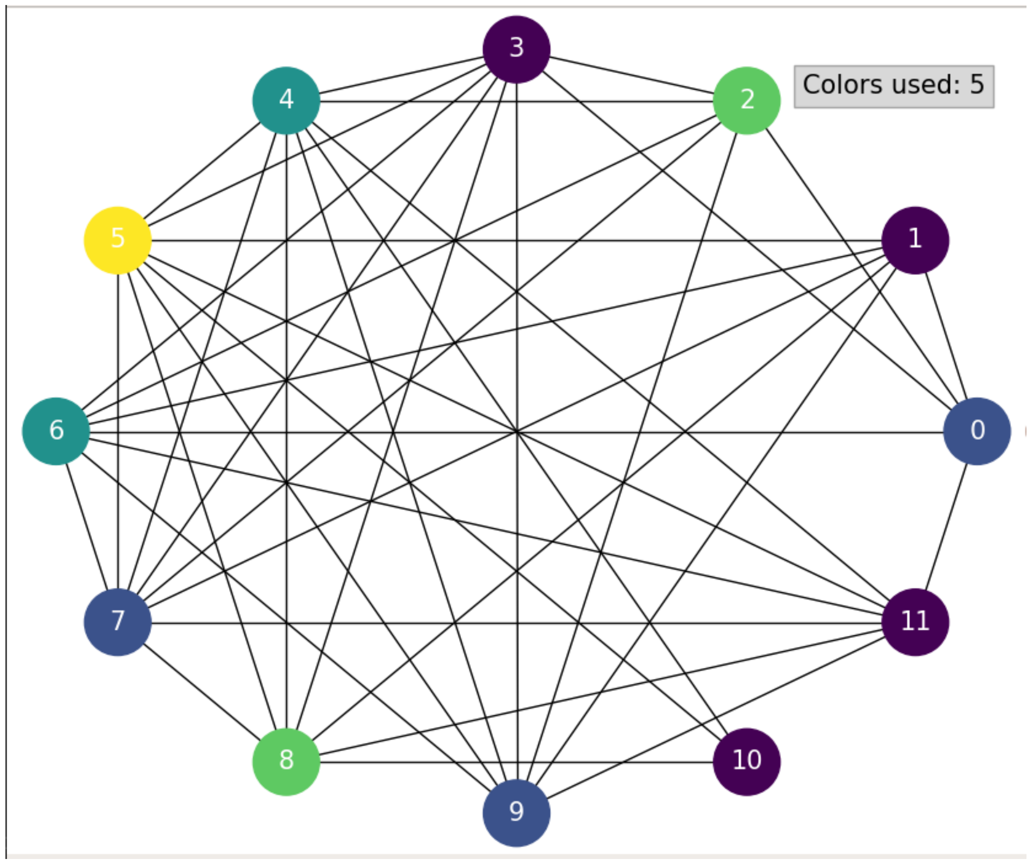
Pseudo Code:

```
GREEDY (S ← Φ, π)

     for  i ← 1 to |π| do

        for j ← 1 to |S|
            if (S ∪ {$\pi_i$}) is an independent set then
                $S_j$ ← $S_j$ ∪ {$\pi_i$}
                break
            else j ← j +1

        if j > |S| then
            $S_j$ ← {$\pi_i$}
```

Explanation:

The algorithm takes an empty solution $S = \phi$ and an **arbitrary permutation of the vertices** π. In each outer loop, the algorithm takes the i^{th} vertex in the permutation π_i and attempts to find a color class $S_j \in S$ into which it can be inserted. If such a color class currently exists in S, then the vertex is added to it and the process moves on to consider the next vertex π_{i+1}. If not, a new color class is created for the vertex.

Output:



Coloring of $G_{12,60}$ by Greedy

DSatur:

`DStaur` (abbreviated from the degree of saturation) is similar to `Greedy` because it also **takes each vertex in some order and assigns it to the first acceptable color class**. The distinction is in how these vertex orderings are formed. The saturation degree of the vertices is used to determine the next node. $Saturation(v)$ is the number of **different colors allocated to the vertices of v** that are next to each other.

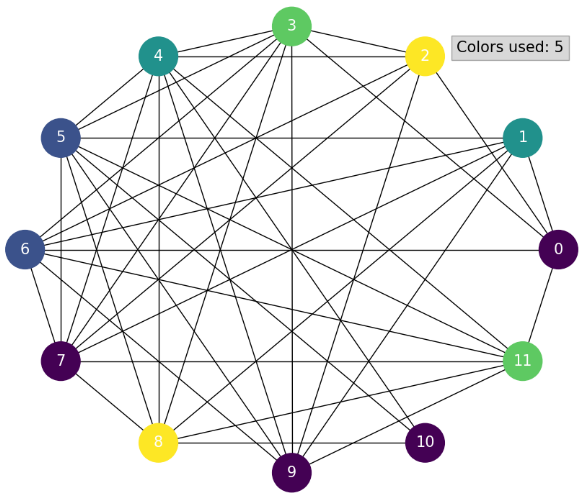
Pseudo Code:

```
DSATUR (S ←Φ, X ← V)

    while X = Φ do

        Choose v ∈ X
        for j ← 1 to |S|
              if (S_j ∪ {v}) is an independent set then
                    S_j ← S_j ∪ {v}
                    break
              else j ← j +1

        if j > |S| then
              S_j ← {v}
              S ← S ∪ S_j

        X ← X − {v}
```

Explanation:

The next vertex to be colored is the vertex in X with the **highest saturation degree**. If there is more than one vertex with the highest saturation degree, ties are broken by selecting the **vertex with the highest degree** from among them. The **maximum saturation degree heuristic** promotes vertices that are seen to be the most "limited" - that is, vertices with the fewest color alternatives accessible to them.

Output:



Coloring of $G_{12,60}$ by DSatur

Recursive Largest First (RLF):

It **colorizes a graph one color at a time, rather than one vertex at a time**. The technique use `heuristics` in each step to **select an independent collection of vertices** in the graph, which are subsequently colored with the same color. The graph is then cleared of this independent set. This process is continued **until the subgraph is empty**, at which time all of the vertices are colored.

Pseudo Code:

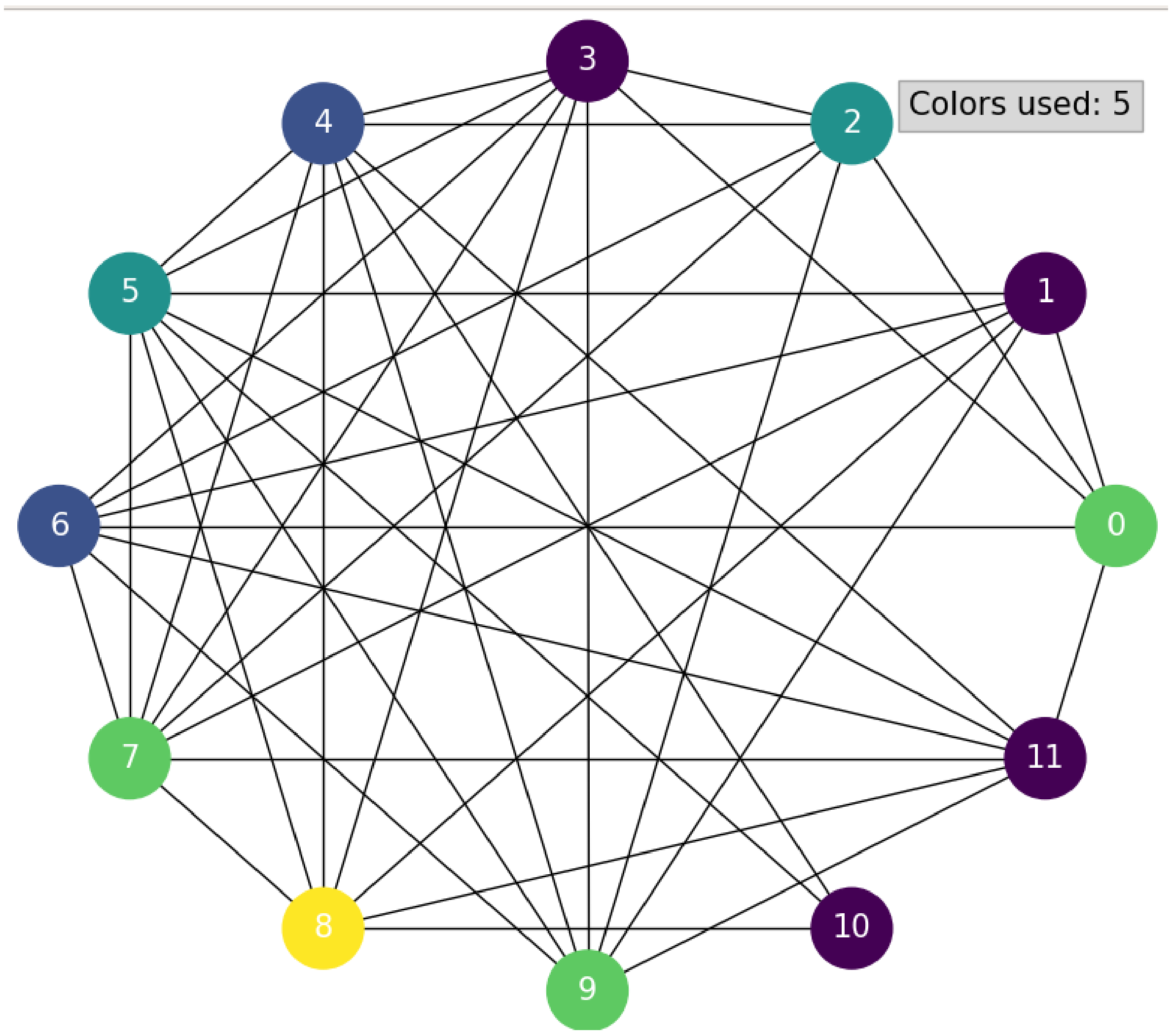
```
RLF (S ←Φ, X ← V, Y ← Φ, i ← 0)

        while X = Φ do
               i ← i + 1
               Sᵢ ← Φ

        while X  ≠ Φ do
              Choose v ∈ X
              Sᵢ ← Sᵢ ∪ {v}
              Y ← Y ∪ ΓX (v)
              X ← X − (Y ∪ {v})

        S ← S ∪ {Sᵢ}
        X ← Y
        Y ← 0
```

Explanation:

The i^{th} color class S_i is constructed in each outer loop. The algorithm employs two sets: X, which includes uncolored vertices that may presently be added to S_i without producing a collision, and Y, which contains uncolored vertices that cannot be added to S_i without generating a clash. At the start of the execution, $X = V$ and $Y = \phi$.

To begin, a vertex v from X is picked and added to S. Then, in the subgraph created by X, any vertices surrounding v are switched to Y to indicate that they can no longer be allocated to S_i. Finally, v and its neighbors are removed from X since they are no longer considered candidates for inclusion in color class S_i . When $X = \phi$, no more vertices can be added to the current color class S_i, and the algorithm proceeds to construct color class S_{i+1}.

Output:



Coloring of $G_{12,60}$ by RLF

Genetic Algorithm:

`Genetic Algorithms` are adaptive heuristic search algorithms that simulate the process of natural selection. **Each generation consists of a population of individuals and each individual represents a possible solution**. Here, `chromosome string` $= c_0c_1...c_{n-1}$, where $c_i = color\ of\ i^{th}\ node.$

$$Fitness = No.\ of\ edges\ with\ different\ colors\ at\ the\ ends + No.\ of\ nodes - No.of\ colors\ used$$

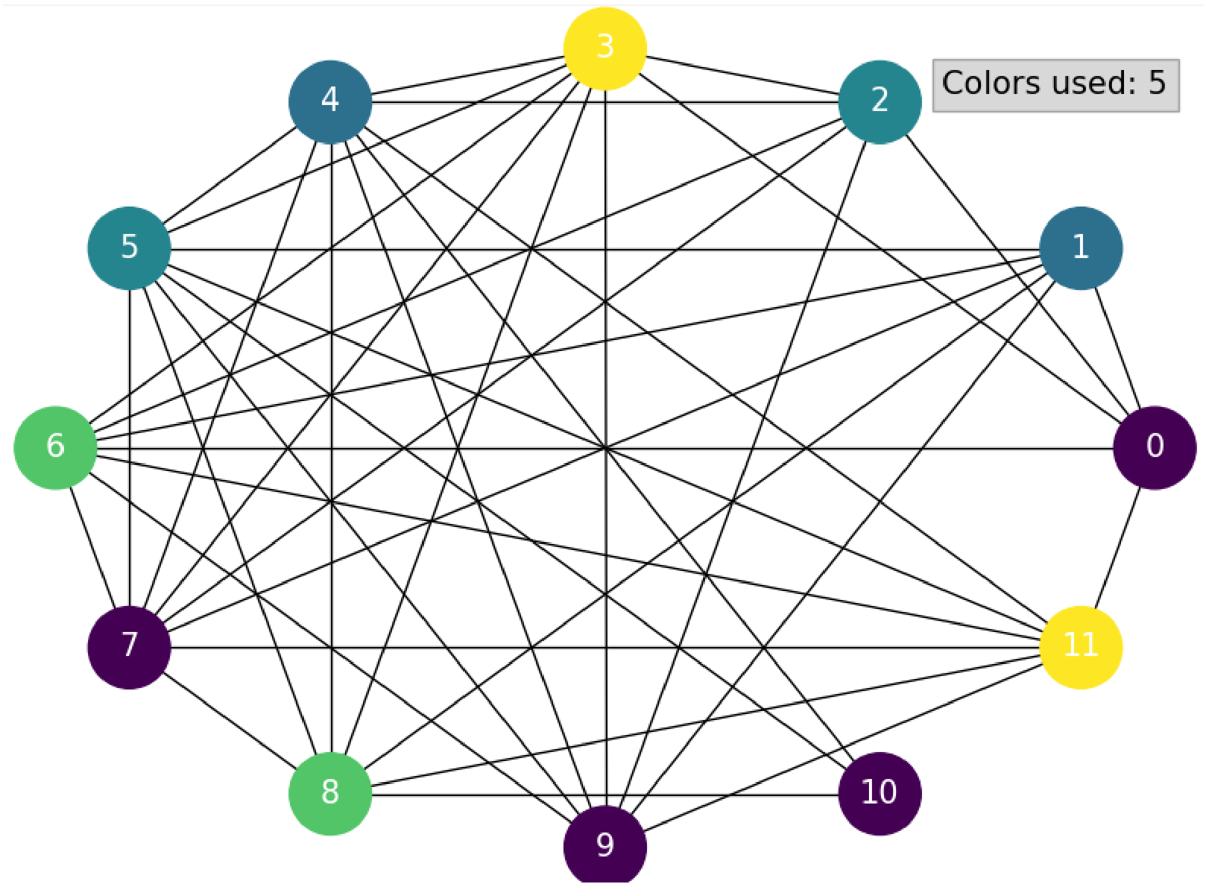
Pseudo Code:

```
GA (S)

  parameter(s): S - set of blocks
  output: superstring of set S

  Initialization:

    t ← 0
    Initialize $P_t$ to random individuals from S*

    EVALUATE-FITNESS-GA(S, $P_t$)

    while termination condition not met
    do
        Select individuals from $P_t$ (fitness proportionate)

        Recombine individuals

        Mutate individuals

        EVALUATE-FITNESS-GA(S, modified individuals)

        $P_{t+1}$ ← newly created individuals

        t ← t + 1

    return (superstring derived from best individual in $P_t$)

    procedure EVALUATE-FITNESS-GA(S, P)

      S - set of blocks
      P - population of individuals

      for each individual i ∈ P

          generate derived string s(i)

          m ← all blocks from S that are not covered by s(i)

          s'(i) ← concatenation of s(i) and m

          fitness(i) ← $1/\|s'(i)\|^2$
```

Explanation:

1. Create the first generation by generating the individuals randomly.
2. Choose pair of parents from the current population and crossover them to generate a new generation.
3. Choose some individuals from the new generation and mutate them.
4. Discard the individuals with lesser fitness value.
5. Repeat steps 2 - 4 multiple times.
6. Choose the fittest individual for the solution.

Output:



Coloring of $G_{12,60}$ by Genetic Algorithm

Simulated Annealing:

`Simulated Annealing` is a **stochastic global search optimization algorithm**. The algorithm is inspired by annealing in metallurgy where metal is heated to a high temperature quickly, then cooled slowly, which increases its strength and makes it easier to work with.

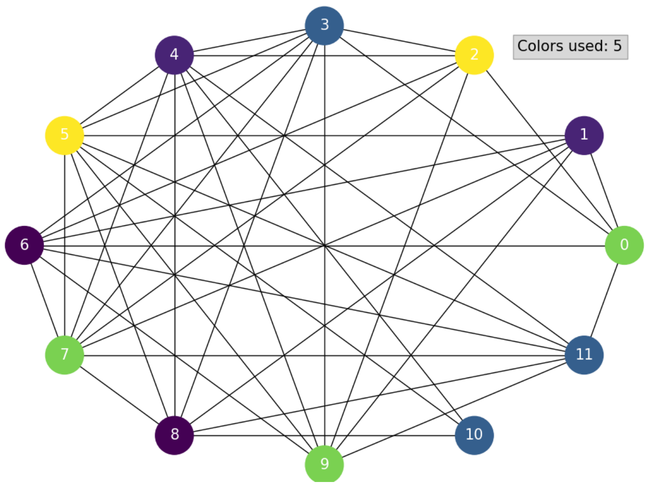
Pseudo Code:

```
SA (S)

  Let s = $s_0$

  For k = 0 through $k_{max}$ (exclusive):
      T ← temperature( 1 - (k+1)/$k_{max}$ )
      Pick a random neighbour, s' ← neighbour(s)
      If P(E(s), E(s'), T) ≥ random(0, 1):
          s ← s'

  Output: the final state s
```

Explanation:

1. Maintain a single candidate solution and takes steps of a random but constrained size from the candidate in the search space.
2. If the new solution is better than the current solution, then the current solution is replaced with the new solution.
3. This process continues for a fixed number of iterations.
4. New points that are not as good as the current point are accepted sometimes.

Output:



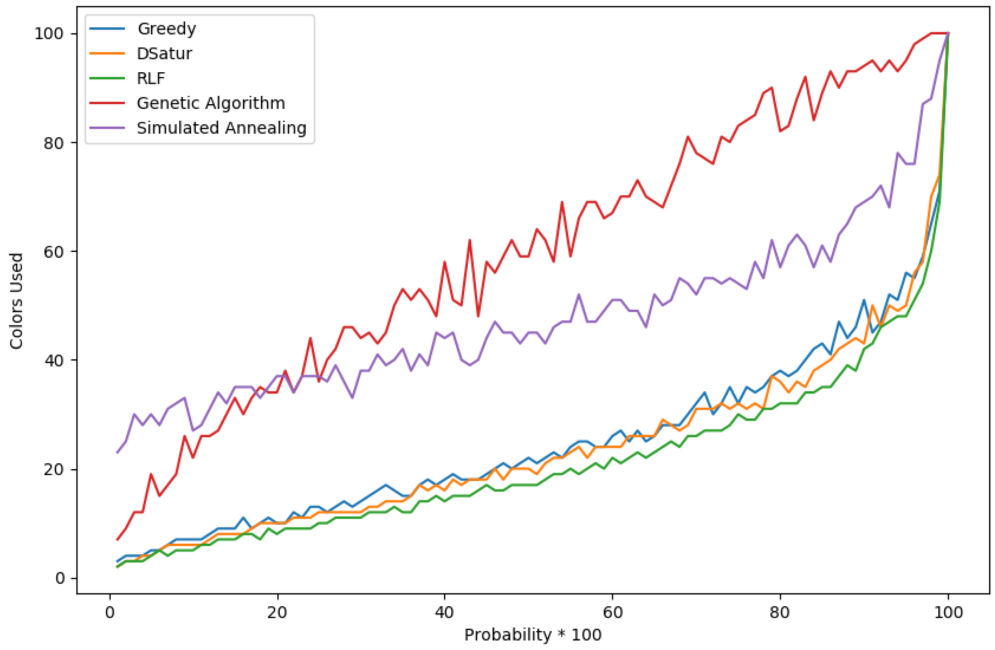
Coloring of $G_{12,60}$ by Simulated Annealing

Empirical Comparison:

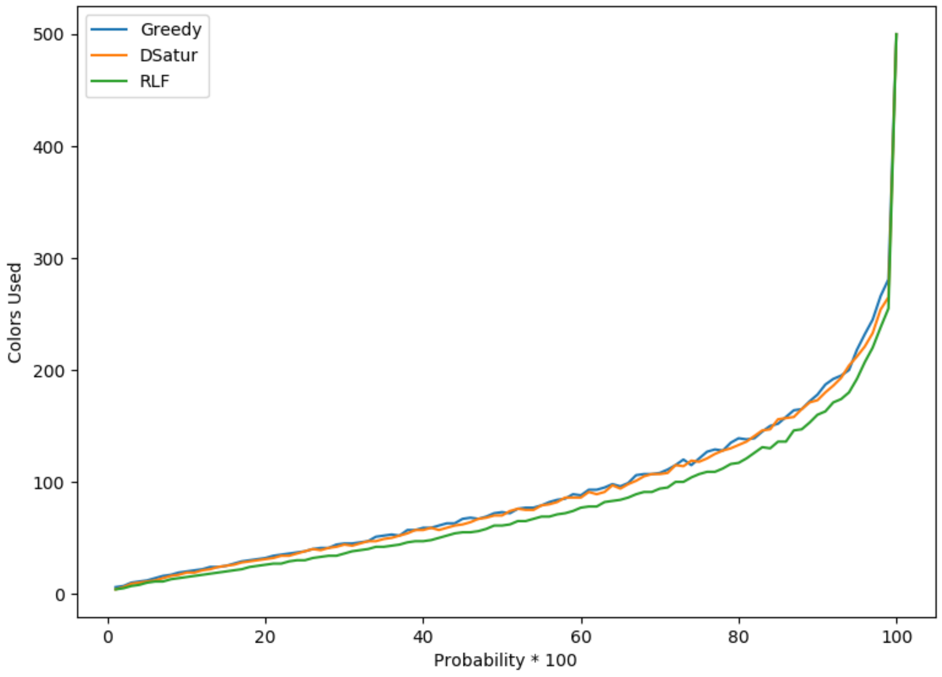
We now give a comparison of the three algorithms, focusing on the quality of solutions produced and the run time needs.

Colors Used:

We first fixed the number of vertices and then vary the values of p from $1\ to\ 100$. The $X-axis$ represents the **value of** p and $Y-axis$ represents the **number of colors used**. The number of iterations for `AI algorithms` is fixed to be 10000. The **size of the population** for the `genetic algorithm` is fixed to be 20.



Colors used for $G_{n,p}$ with n = 100



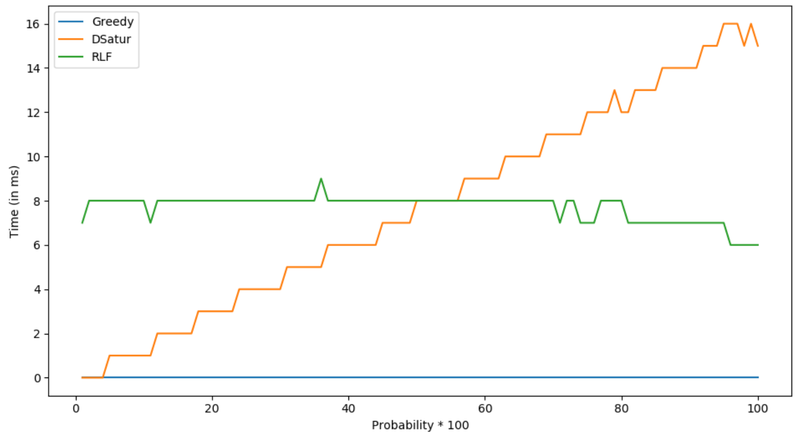
Colors used for $G_{n,p}$ with n = 500

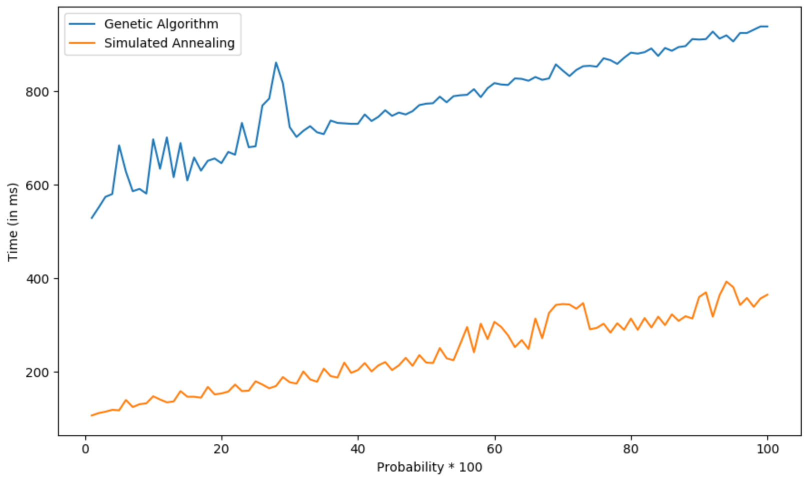
Result and Analysis:

- RLF constantly employs the least colors, proving its superiority over DSatur and Greedy across the board (from sparse to dense).
- For sparse graphs, the variation in color is minor. However, as the graph grows denser, the gap deepens.
- On random graphs, the efficiency of Greedy algorithms' random ordering looks roughly identical to DSatur's ordering based on saturation degree.
- The number of colors used by search-based heuristics algorithms is significantly larger than the other 3 algorithms.
- The genetic algorithm works better than simulated annealing on sparse graphs whereas the reverse is true for the denser graphs.

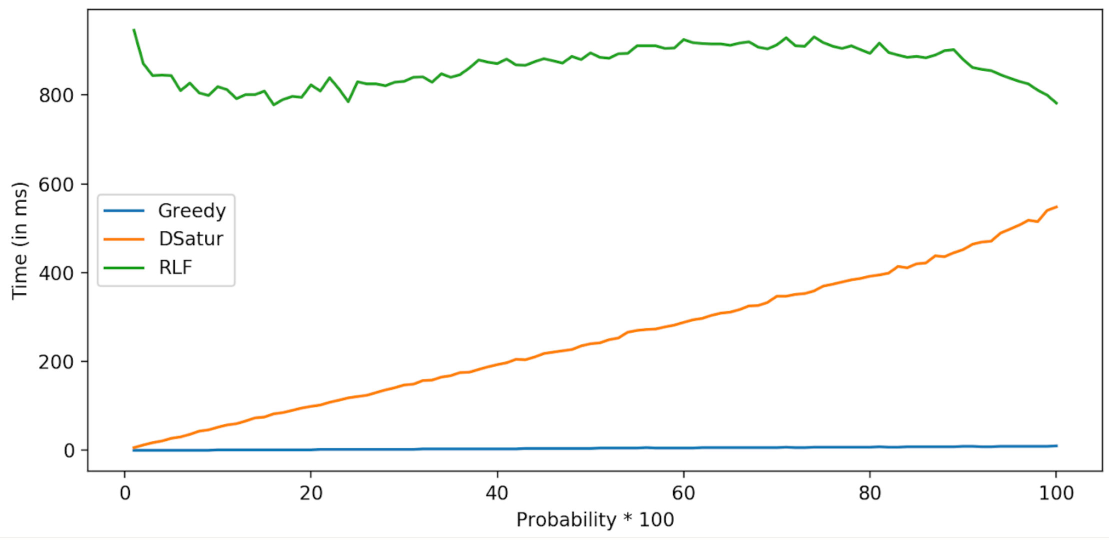
Time Consumption:

We started with a fixed number of vertices and then altered p from $1\ to\ 100$. The $X-axis$ shows the value of p, while the $Y-axis$ shows the **time consumed**. The **number of iterations** is fixed to be 100 for the `Genetic Algorithm` and 1000 for `Simulated annealing`.

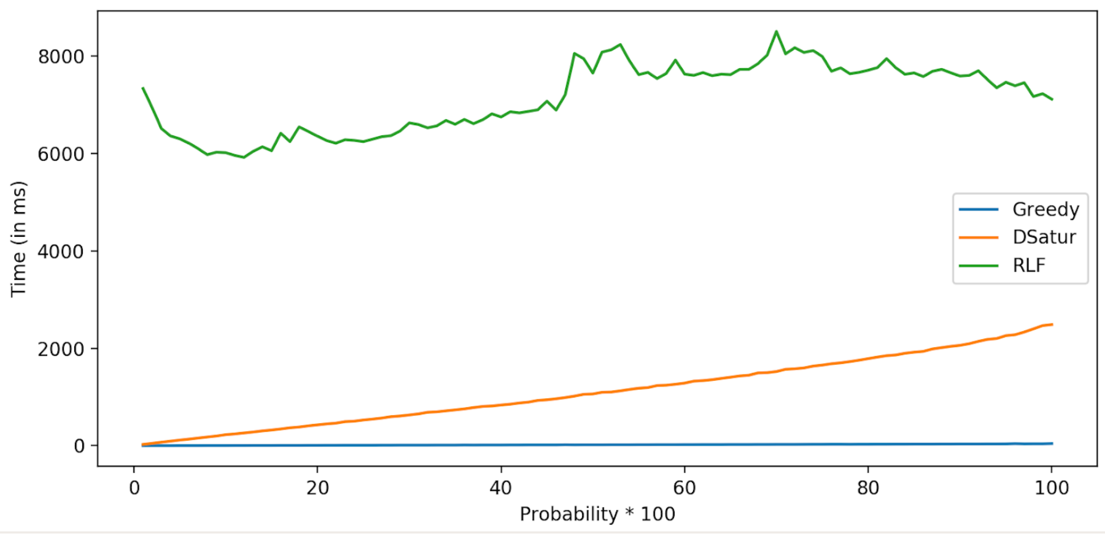




Time consumed for $G_{n,p}$ with n = 100



Time consumed for $G_{n,p}$ with n = 500



Time consumed for $G_{n,p}$ with n = 1000

Result and Analysis:

- The runtime of greedy is negligible as compared to DSatur and RLF
- The runtime of DSatur increases linearly with p (or the number of edges).
- The runtime of RLF remains unaffected by the value of p.
- Comparing the 2nd and 3rd graphs, the runtime increases by 4 times and 8 times respectively for Dsatur and RLF respectively when the number of nodes is doubled, indicating that the time complexity of:

$$
\begin{aligned}
DSatur &= O(n^2) \\
RLF &= O(n^3)
\end{aligned}
$$

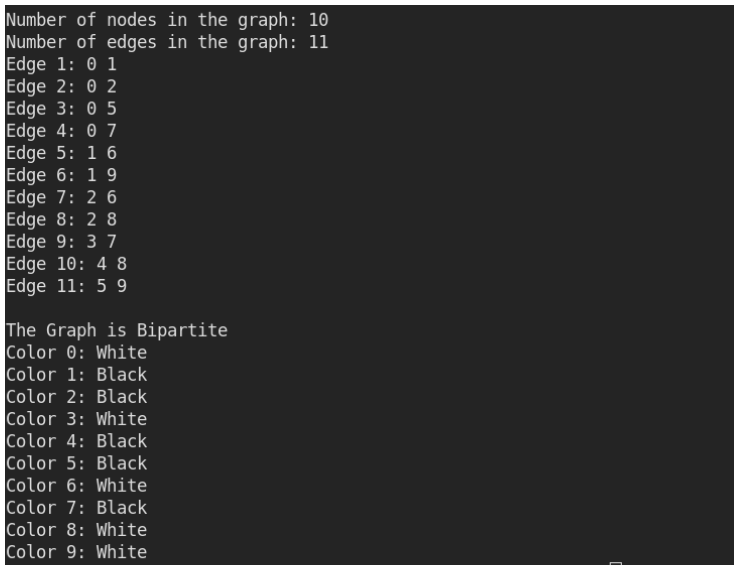
- The runtime of the search-based algorithm is significantly greater than the other 3 algorithms.
- The runtime of the genetic algorithm and simulated annealing increases linearly with the density of the graph.
- The genetic algorithm is more unpredictable in runtime with sparse graphs whereas simulated annealing is more unpredictable with denser graphs.

Applications

Bipartite Checker

It uses `DSatur algorithm` to determine if the graph is bipartite or not. If it is a bipartite graph, it additionally prints the color of each node. `Dsatur algorithm` has a special property that it can **always color a bipartite graph in two colors**.

Working:

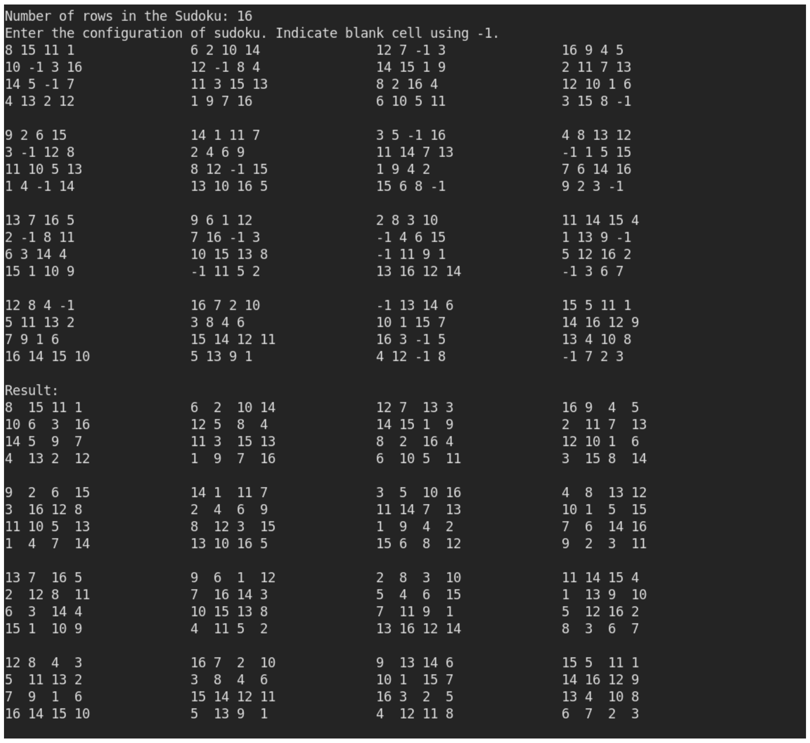


```
Number of nodes in the graph: 10
Number of edges in the graph: 11
Edge 1: 0 1
Edge 2: 0 2
Edge 3: 0 5
Edge 4: 0 7
Edge 5: 1 6
Edge 6: 1 9
Edge 7: 2 6
Edge 8: 2 8
Edge 9: 3 7
Edge 10: 4 8
Edge 11: 5 9

The Graph is Bipartite
Color 0: White
Color 1: Black
Color 2: Black
Color 3: White
Color 4: Black
Color 5: Black
Color 6: White
Color 7: Black
Color 8: White
Color 9: White
```

Sudoku Solver

It uses `DSatur algorithm` to solve any $n * n$ sudoku grid, where n is a **perfect square**. First, It constructs a graph where every node represents one of the cell. Edge is added between two nodes if they can't have same value. The **final coloring represents one of the possible solutions of the sudoku.**

Working:



```
Number of rows in the Sudoku: 16
Enter the configuration of sudoku. Indicate blank cell using -1.
8 15 11 1               6 2 10 14               12 7 -1 3               16 9 4 5
10 -1 3 16              12 -1 8 4               14 15 1 9               2 11 7 13
14 5 -1 7               11 3 15 13              8 2 16 4                12 10 1 6
4 13 2 12               1 9 7 16                6 10 5 11               3 15 8 -1

9 2 6 15                14 1 11 7               3 5 -1 16               4 8 13 12
3 -1 12 8               2 4 6 9                 11 14 7 13              -1 1 5 15
11 10 5 13              8 12 -1 15              1 9 4 2                 7 6 14 16
1 4 -1 14               13 10 16 5              15 6 8 -1               9 2 3 -1

13 7 16 5               9 6 1 12                2 8 3 10                11 14 15 4
2 -1 8 11               7 16 -1 3               -1 4 6 15               1 13 9 -1
6 3 14 4                10 15 13 8              -1 11 9 1               5 12 16 2
15 1 10 9               -1 11 5 2               13 16 12 14             -1 3 6 7

12 8 4 -1               16 7 2 10               -1 13 14 6              15 5 11 1
5 11 13 2               3 8 4 6                 10 1 15 7               14 16 12 9
7 9 1 6                 15 14 12 11             16 3 -1 5               13 4 10 8
16 14 15 10             5 13 9 1                4 12 -1 8               -1 7 2 3

Result:
8  15 11 1              6  2  10 14             12 7  13 3              16 9  4  5
10 6  3  16             12 5  8  4              14 15 1  9              2  11 7  13
14 5  9  7              11 3  15 13             8  2  16 4              12 10 1  6
4  13 2  12             1  9  7  16             6  10 5  11             3  15 8  14

9  2  6  15             14 1  11 7              3  5  10 16             4  8  13 12
3  16 12 8              2  4  6  9              11 14 7  13             10 1  5  15
11 10 5  13             8  12 3  15             1  9  4  2              7  6  14 16
1  4  7  14             13 10 16 5              15 6  8  12             9  2  3  11

13 7  16 5              9  6  1  12             2  8  3  10             11 14 15 4
2  12 8  11             7  16 14 3              5  4  6  15             1  13 9  10
6  3  14 4              10 15 13 8              7  11 9  1              5  12 16 2
15 1  10 9              4  11 5  2              13 16 12 14             8  3  6  7

12 8  4  3              16 7  2  10             9  13 14 6              15 5  11 1
5  11 13 2              3  8  4  6              10 1  15 7              14 16 12 9
7  9  1  6              15 14 12 11             16 3  2  5              13 4  10 8
16 14 15 10             5  13 9  1              4  12 11 8              6  7  2  3
```

Source Code:

GitHub - alankrit-anand/Graph-Coloring

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your session.

https://github.com/alankrit-anand/Graph-Coloring

alankrit-anand/**Graph-Coloring**

1 Contributor 0 Issues 0 Stars 0 Forks

It consists of three folders:

- ▼ **Header**

  Contains header files of the coloring algorithms.

- ▼ **Src**

  Contains source code of the coloring algorithms

- ▼ **Simulation**

  Contains different files for simulations.

- ▼ **Application**

  Contains files to run applications

- ▼ **Input**

  Contains input files for applications

Dependencies required:

> G++
> Python
> Matplotlib Library
> Networkx Library

How to Simulate:

- Go to simulation directory:

  👉 cd simulation

- Compile and run the simulator file

  👉 g++ simulator.cpp && ./a.out

- Choose the type of simulation you want to run
- Give the required inputs such as number of nodes n or probability percentage p
- The result is displayed.

How to run applications:

- Go to the application directory:

```
👉 cd application
```

- Compile and run the application file

```
👉 g++ application.cpp && ./a.out
```

- Choose the type of application you want to run
- Give the required inputs.
- The result is displayed.

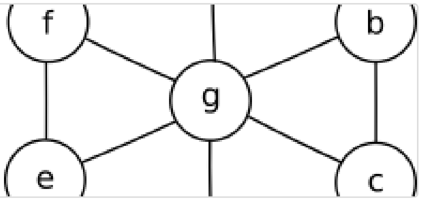
References

https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/A%20Guide%20to%20Graph%20Colouring_%20Algorithms%20and%20Applications%20%5BLewis%202015-10-27%5D.pdf

DSatur - Wikipedia

DSatur is a graph colouring algorithm put forward by Daniel Brélaz in 1979. Similarly to the greedy colouring algorithm, DSatur colours the vertices of a graph one after another, adding a previously unused colour when needed.

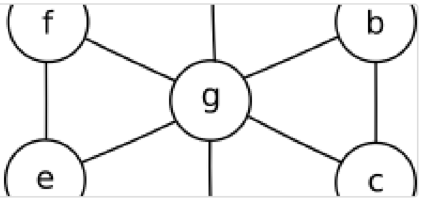
https://en.wikipedia.org/wiki/DSatur



Recursive largest first algorithm - Wikipedia

The Recursive Largest First (RLF) algorithm is a heuristic for the NP-hard graph coloring problem. It was originally proposed by Frank Leighton in 1979. The RLF algorithm assigns colors to a graph's vertices by constructing each color class one at a time.

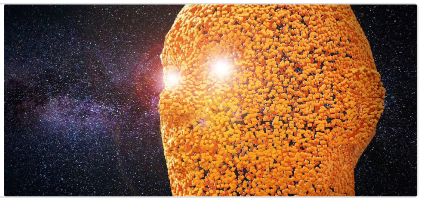
https://en.wikipedia.org/wiki/Recursive_largest_first_algorithm



Experience the power of the Genetic Algorithm

Genetic Algorithm is an evolutionary computing technique based on the concepts of Genetics and Natural Selection. It is mostly used to find a near-optimal solution for many optimizations and tougher problems where a deterministic polynomial solution is infeasible. The Genetic Algorithm

https://towardsdatascience.com/experience-the-power-of-the-genetic-algorithm-4030adf0383f



Optimization Techniques-Simulated Annealing

The Simulated Annealing algorithm is based upon Physical Annealing in real life. Physical Annealing is the process of heating up a material until it reaches an annealing temperature and then it will be cooled down slowly in order to change the material to a desired structure.

https://towardsdatascience.com/optimization-techniques-simulated-annealing-d6a4785a1de7