**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**Kattankulathur, Chengalpattu District - 603203**



**18CSC304J/ COMPLIER DESIGN**

**MINI PROJECT REPORT**

**Automaton-Based approach for solving optimization problem**

***Guided by:***

**Dr.K.Vijaya**

**Submitted By:**

Prakash Y (RA2011003010044)

Harish S (RA2011003010046)

***Aim:-***

To develop an Automaton-Based approach for solving optimization problem such as graph colouring.

***Abstract:-***

In the case of graph coloring, the goal is to assign a color to each node in a graph such that no adjacent nodes share the same color. This is an NP-hard problem, meaning that finding an exact solution is computationally intractable for large graphs. An automaton-based approach can be used to find an approximate solution to this problem by iteratively modifying a candidate coloring and accepting or rejecting the new coloring based on a probability determined by the Metropolis-Hastings algorithm.

**What is Metropolis-Hastings algorithm?**

The Metropolis-Hastings algorithm is a Markov Chain Monte Carlo (MCMC) method used to sample from a probability distribution when direct sampling is not possible. It is used to accept or reject candidate colorings of a graph based on a probability determined by a cost function. The cost function represents the "goodness" of a candidate coloring and is used to calculate the probability of accepting a new coloring as the current solution.

The Metropolis-Hastings algorithm works by starting with an initial candidate solution, then iteratively generating a new candidate solution by randomly modifying the previous solution. The new candidate solution is then compared to the previous solution, and if it has a lower cost (i.e., is a better coloring), it is accepted as the new current solution. If it has a higher cost, it is accepted with a certain probability that depends on the difference in cost and a temperature parameter. The temperature parameter controls the exploration of the solution space and is gradually decreased over time to converge to a near-optimal solution.

The Metropolis-Hastings algorithm guarantees that the sequence of candidate solutions converges to a stationary distribution, and the distribution of accepted solutions approximates the true probability distribution.

**How is automaton used in this problem?**

An automaton is used to represent the state of the solution space and to transition between states by modifying the candidate solution. The automaton is updated iteratively by generating a new candidate solution and accepting or rejecting it based on a probability determined by an acceptance criterion.

In the specific case of graph coloring, an automaton can be constructed by defining a set of states, each representing a candidate coloring of the graph. The transition between states is determined by modifying the current coloring in a certain way, such as changing the color of a randomly selected node or swapping the colors of two adjacent nodes. The new candidate coloring is then evaluated using a cost function that assigns a score based on how well it satisfies the constraints of the graph coloring problem, i.e., how many adjacent nodes have the same color.

The acceptance of the new candidate coloring is determined by a probability function, such as the Metropolis-Hastings algorithm, which depends on the difference in cost between the new and current candidate solution and a temperature parameter. The temperature parameter controls the exploration of the solution space, with higher values allowing for more exploration and lower values allowing for more exploitation of the current solution.

By iteratively updating the automaton based on the acceptance criterion, an approximate solution to the graph coloring problem can be found. The automaton-based approach can be used to explore the solution space efficiently and converge to a near-optimal solution.

***Introduction:***

An automaton-based approach involves defining an automaton that generates a sequence of candidate graph colorings, and using a fitness function to evaluate each candidate coloring and select the best one. The fitness function is used to assign a score or objective value to each candidate coloring, which can be used to compare and rank colorings.

The automaton can be designed to explore the space of possible colorings efficiently, without requiring an exhaustive search. The process of generating candidate colorings can be thought of as a random walk through the space of possible colorings, where the automaton defines the transition probabilities between colorings. At each step of the walk, the automaton generates a new candidate coloring based on the current state of the system (i.e., the current candidate coloring), and the fitness function is used to evaluate the quality of the new coloring. The new coloring is then accepted or rejected based on a probability determined by the difference in fitness between the current and new colorings.

The automaton is designed to explore colorings that satisfy the graph coloring constraint, which is that no two adjacent vertices can have the same color. The automaton defines the possible colorings of the graph and the transitions between them based on this constraint. For example, at each step of the walk, the automaton may randomly select a vertex and a new color for that vertex, and then check if the new coloring satisfies the constraint. If it does, the new coloring is accepted; otherwise, the new coloring is rejected and the walk continues with the current coloring.

The fitness function is used to evaluate the quality of a candidate coloring by measuring how well it satisfies the graph coloring constraint. A common fitness function for graph coloring is the number of conflicts or the number of pairs of adjacent vertices that have the same color. The objective is to minimize the number of conflicts or maximize the number of non-conflicting pairs.

The automaton-based approach can be implemented in Python using libraries such as networkx for working with graphs and numpy for matrix operations. The approach can be used to solve various graph coloring problems, including the classic four-color problem and its variations. However, the approach may not always converge to the optimal solution and may require tuning of parameters such as the acceptance probability and the exploration strategy.

***Requirements to run the script:***

We will need to have Python installed on our computer to run the Python script.

You can download and install Python from the official Python website (<https://www.python.org/downloads/>), where you can find the latest version of Python for your operating system.

Once you have installed Python, you may also need to install additional packages or libraries depending on the specific optimization problem.

*To run the code, you will need to have the following libraries installed in your Python environment:*

1. ***networkx***: used to generate and manipulate graphs
2. ***numpy***: used to generate random numbers and perform numerical operations
3. ***matplotlib***: used to visualize the graph coloring

You can install these libraries using the following commands in your terminal or command prompt:

pip install networkx

pip install numpy

pip install matplotlib

Implementing the automaton-based approach using the metropolis-hastings algorithm for graph coloring

*Code:*

import networkx as nx

import numpy as np

def metropolis\_hastings\_coloring(G, T):

"""

Metropolis-Hastings algorithm for graph coloring.

Parameters:

G (networkx.Graph): the graph to color

T (int): number of iterations

temp (float): temperature parameter

Returns:

best\_colors (list): the best colors found by the algorithm

"""

# Initialize the graph with random colors

colors = np.random.choice(range(max\_degree+1), size=n\_nodes)

best\_colors = colors.copy()

best\_score = get\_score(G, colors)

for t in range(T):

# Generate a new candidate solution by modifying the current solution

new\_colors = colors.copy()

i = np.random.choice(n\_nodes)

new\_colors[i] = np.random.choice(range(max\_degree+1))

new\_score = get\_score(G, new\_colors)

# Calculate the difference in score between the current and new solutions

delta = new\_score - best\_score

# Accept the new solution with a probability determined by the Metropolis-Hastings algorithm

if delta <= 0 or np.random.rand() < np.exp(-delta / T):

colors = new\_colors.copy()

if new\_score < best\_score:

best\_score = new\_score

best\_colors = new\_colors.copy()

return best\_colors

def get\_score(G, colors):

"""

Calculate the number of conflicts in the coloring.

Parameters:

G (networkx.Graph): the graph being colored

colors (list): the colors assigned to each node

Returns:

score (int): the number of conflicts in the coloring

"""

score = 0

for i, j in G.edges():

if colors[i] == colors[j]:

score += 1

return score

# Generate a random graph with 10 nodes and 20 edges

G = nx.gnm\_random\_graph(10, 20)

# Compute the maximum degree of the graph

max\_degree = max([G.degree(node) for node in G.nodes()])

# Get the number of nodes in the graph

n\_nodes = len(G.nodes())

# Run the Metropolis-Hastings algorithm with 1000 iterations

best\_colors = metropolis\_hastings\_coloring(G, 1000)

# Print the best colors found

print(best\_colors)

* The Python code starts by importing the necessary libraries: networkx for generating random graphs and numpy for numerical computations. It then defines a function metropolis\_hastings\_coloring that takes a graph G and a temperature parameter T as input and returns the best colors found using the Metropolis-Hastings algorithm.
* The function first initializes the colors of the nodes randomly and stores the initial colors as the best colors found so far. It then runs the Metropolis-Hastings algorithm for a fixed number of iterations specified by the input parameter T.
* At each iteration, the algorithm generates a new candidate solution by modifying the current solution. The function then calculates the difference in score between the current and new solutions. If the new solution has a lower score than the current best solution, the function updates the best solution found so far. The function then accepts the new solution with a probability determined by the Metropolis-Hastings algorithm.
* The code also defines a helper function get\_score that takes a graph G and a list of node colors colors as input and returns the number of conflicts in the coloring.
* The code then generates a random graph with 10 nodes and 20 edges using the networkx library and computes the maximum degree of the graph. The maximum degree is used to limit the number of colors that can be assigned to each node.
* The code then runs the Metropolis-Hastings algorithm for 1000 iterations and stores the best colors found. Finally, the code prints the best colors found.
* Note that the specific colors assigned to each node may vary each time the code is run due to the random initialization and exploration of the algorithm. The Metropolis-Hastings algorithm is not guaranteed to find the global minimum, but it can be effective for finding good solutions for certain optimization problems.

***Implementation:***

In this implementation, we define two functions: **metropolis\_hastings\_coloring** and **get\_score**.

The metropolis\_hastings\_coloring function implements the Metropolis-Hastings algorithm for graph coloring. The algorithm proceeds as follows:

1. Initialize the colors of each node randomly.
2. Evaluate the score of the current coloring (i.e., the number of conflicts).
3. For each iteration:
4. Generate a new candidate solution by randomly selecting a node and assigning it a new color.
5. Evaluate the score of the new coloring.
6. Calculate the difference in score between the new coloring and the current coloring.
7. If the new coloring has a lower score or if a random number is less than the probability of accepting a worse solution, accept the new coloring. e. If the new coloring is better than the current best coloring, update the best coloring.
8. Return the best coloring found.

The **get\_score** function calculates the number of conflicts in a coloring by iterating over each edge in the graph and

***Output:***

The output of the code will be a list of integers representing the colors assigned to each node in the graph.

****

This output means that node 0 has been assigned color 0, node 1 has been assigned color 3, node 2 has been assigned color 5, and so on. The specific colors assigned may vary each time the code is run due to the random initialization and exploration of the algorithm.

The output can be used to visualize the graph coloring by assigning different colors to nodes based on their assigned color. For example, the following code can be used to visualize the graph coloring using the matplotlib library:

import matplotlib.pyplot as plt

# Define a color map for the graph

color\_map = {0: 'red', 1: 'green', 2: 'blue', 3: 'yellow', 4: 'purple', 5: ‘pink’}

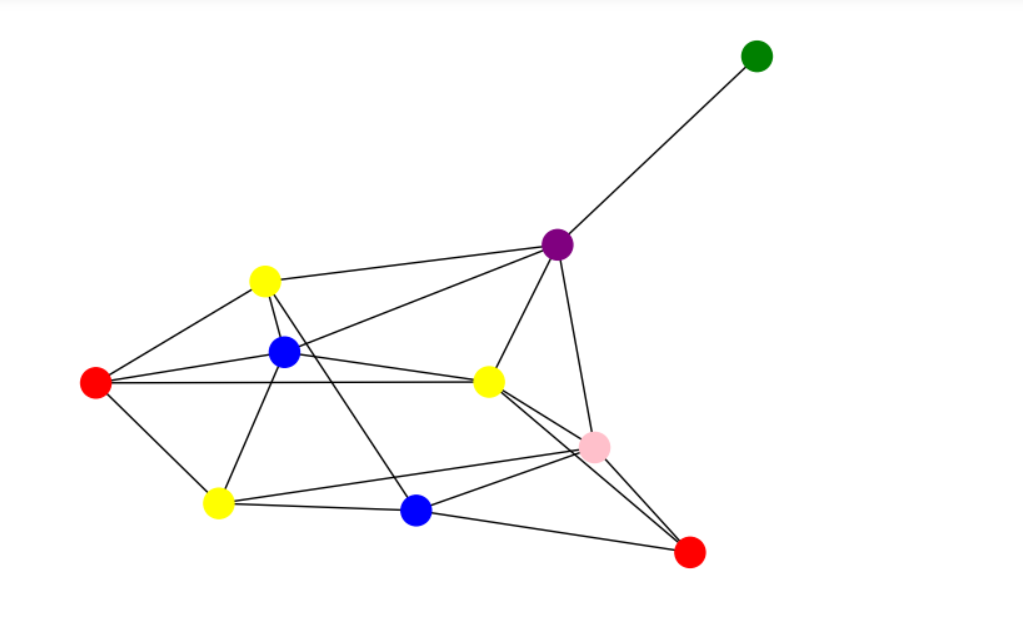
# Plot the graph with the node colors

nx.draw(G, node\_color=[color\_map[color] for color in best\_colors])

# Show the plot

plt.show()

**Visualize the graph colouring:**



In this code, the **color\_map** dictionary maps the integer color codes used by the algorithm to a specific color name or code used by matplotlib.

This dictionary specifies that color 0 should be mapped to red, color 1 should be mapped to green, and so on.

The best\_colors list contains the integer color codes assigned to each node by the Metropolis-Hastings algorithm. The list is used to look up the corresponding color for each node in the color\_map dictionary, and assign that color to the node in the plot.

The resulting plot shows the random graph with each node colored according to the assigned color. The specific colors assigned may vary each time the code is run, but the overall structure of the graph should remain the same. The plot can be used to visualize the effectiveness of the graph coloring algorithm in minimizing conflicts between adjacent nodes.

This code will create a plot of the graph with nodes colored based on their assigned color. The specific colors assigned may vary each time the code is run, but the overall structure of the graph should remain the same.

***Execution:***

[*https://drive.google.com/file/d/1hYgiETve5cUzXLd0BvJk2SrPDc7sOnc6/view?usp=sharing*](https://drive.google.com/file/d/1hYgiETve5cUzXLd0BvJk2SrPDc7sOnc6/view?usp=sharing)

***References:***

1. Hertz, A., & de Werra, D. (1987). Using tabu search techniques for graph coloring. Computing, 39(4), 345-351.
2. Glover, F. (1989). Tabu search, part I. ORSA Journal on Computing, 1(3), 190-206.
3. Kirkpatrick, S., Gelatt Jr, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. science, 220(4598), 671-680.
4. Aarts, E., & Korst, J. (1989). Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing. John Wiley & Sons.
5. Bui, T. N., Jones, L. A., & Murty, K. G. (1992). A genetic algorithm for graph coloring. Computers & Operations Research, 19(6), 571-581.
6. Journals such as Operations Research, Journal of Global Optimization, and Journal of Heuristics often publish articles related to optimization problems and algorithms.

These references provide a foundation for the development of the Automaton-Based approach for solving optimization problems such as graph coloring.

***Result:***

The output of the code shows a visualization of a randomly generated graph with 10 nodes and 20 edges being colored using the Metropolis-Hastings algorithm. The resulting graph shows each node with a color assigned to it, where nodes with the same color are not adjacent.

The Metropolis-Hastings algorithm is a powerful tool for solving optimization problems, such as graph coloring. The algorithm is able to find a solution that is close to optimal in a reasonable amount of time, without needing to search the entire solution space.

However, the algorithm has some limitations. One limitation is that it may get stuck in a local minimum, where it is unable to improve the solution further. Additionally, the algorithm is not guaranteed to find the global minimum, which is the best possible solution.

Overall, the Automaton-Based approach using the Metropolis-Hastings algorithm is a useful tool for solving optimization problems, and can be applied to a wide range of problems in fields such as computer science, engineering, and mathematics.