COMPUTER SCIENCE AND DATA ANALYTICS

Course: **AI – CS6511**

Project II: CSP Graph Coloring

Student: Shukurov Shamil

Student ID: G47855191

Instructor: Amrinder Arora

Baku 2023

Introduction

In computer science, the graph coloring problem is a well-known problem that involves coloring a graph so that no two nodes next to each other have the same color. This is an example of a Constraint Satisfaction Problem (CSP), where the goal is to find a solution that meets a set of rules. Let's define variable, domain and constraints for our problem:

1. Variables: Nodes of the graph
2. Domain of variable: Set of colors that node can be colored with
3. Constraints: No two adjacent node have the same color

The Backtracking algorithm is a common way to solve CSP problems. It tries to find a solution step by step by making decisions at each step and going back when a decision leads to a dead end. Several methods, such as Most Constrained Variable (MCV), Least Constrained Value (LCV), and Arc Consistency (AC-3), can be used to improve the performance of the Backtracking algorithm work.

In this assignment, I am going to implement the Backtracking algorithm along with MCV, LCV, and AC-3 techniques to solve the graph coloring problem.

Backtracking Algorithm

As I said earlier Backtracking algorithm is used to solve the graph coloring problem.

The Backtracking algorithm begins by picking a node for coloring. Then, the algorithm picks a color from the variable's domain and assign it to the node. Then, the algorithm checks to see if this assignment satisfies all of the constraints. If yes, the algorithm moves on to the next variable and gives that one a value. If the assignment doesn't satisfy the constraints, the algorithm backtracks and tries a different value for the current variable.

The Backtracking algorithm continues this process, selecting variables and values and checking the constraints, until a complete solution is found or all possible assignments have been tried. If a solution is found, the algorithm returns it. If no solution is found, the algorithm reports failure.

Pseudo-Code for Backtrack algorithm:

```
BackTrack(x,w,Domains):
    If x is complete assignment: update best and return

    Choose unassigned VARIABLE X_i --(MCV)

    Order VALUES Domain[i] of chosen X_i --(LCV)

    For each value v in that order:
        calculate partial weight

        If partial weight = 0 : continue

        Domains_new = LOOK_AHEAD(Domains) --(AC_3)

        If any element in Domains_new is empty: continue

        result = BackTrack(x U {X_i:v},w*partial_weight, Domains_new)

        if result is not FAILURE:
            return result
    return FAILURE
```

When we assign color to a node, `LOOK_AHEAD` function will reduce the domain of neighbors (maybe not only neighbors, all nodes, depending on algorithm) so that possible values for them is up to date.

Note that in our problem partial weights can be either 0 or 1, constraints are satisfied when partial weight is 1.

Now we have several questions to answer:

1. How to select a variable to color
2. Which color do we choose to assign a node
3. How we implement `LOOK_AHEAD`

In the next sections I will answer this questions and show how I implemented them in my code.

Which variable to pick: MCV

The MCV (Most Constrained Variable) or MRV (Minimum Remaining Values) is a heuristic to choose next variable to assign in Backtracking algorithm. MCV selects the variable that has the fewest possible values remaining in its domain. For our problem I implemented MRV, and in case of tie, the algorithm returns the variable that has most number of neighbors.

```
  """
      Select Most Constrained Variable: variable which has minimum length of
domain
      In case of tie, take the one that has most number of neighbours
  """
  def MCV(self, assignments: dict, domains: dict):
    # select all unassigned variables
    left_variables = list(
      filter(lambda x: x not in assignments, self.variables))

    # generate dictionary that holds neighbour count of each variable
    lv = {}
    for var in left_variables:
      lv[var] = len(self.graph[var])

    # sort lv by neighbour count
    sorted_lv = dict(sorted(lv.items(), key=lambda x: x[1], reverse=True))

    left_variables = list(sorted_lv.keys())

    #find most constrained variable
    min_var = left_variables[0]
    min_c = len(domains[min_var])

    for X_i in left_variables[1:]:
      if (len(domains[X_i]) < min_c):
        min_c = len(domains[X_i])
        min_var = X_i
    return min_var
```

In this implementation, first unassigned variables are sorted based on their neighbor count, then in that order most constrained variable is found. Therefore, it is most constrained variable and in case of tie it is the node that has most number of neighbors.

Which color to use: LCV

The LCV heuristic works by considering each value in the domain of the current variable and computing the number of values that would be eliminated from the domains of the other variables in the CSP if that value were selected. The LCV heuristic chooses the value that eliminates the fewest number of values from the domains of the other variables. Or we can say that LCV heuristic chooses the value such that neighbors of the node has most consistent values.

```
""" Select Least Constrained Value :
    Order values of selected variable by decreasing number of consistent values
of neighboring variables.
  """

  def LCV(self, assignments: dict, domains: dict, X_i: int):
    colors_domain = []
    color_index = 0
    for col in domains[X_i]:
      cv = 0  #consistent values
      for X_j in self.neighbours(X_i):
        cv = cv + len(domains[X_j])
        # if color is in the domain of the neighbour, we decrease consistent
values
        if (col in domains[X_j]):
          cv = cv - 1
      colors_domain.append((cv, color_index, col))
      color_index = color_index+1

    # Order domain by decreasing number of consistent values
    colors_domain.sort(reverse=True)

    ordered_domain_values = list(map(lambda x: x[2], colors_domain))

    return ordered_domain_values
```

In above implementation, for each color we calculate how many consistent values that neighbors will have if we choose that color. Then we sort them by decreasing number of consistent values of neighboring variables. In case of tie, color that comes first in the domain of variable will be selected.

AC_3

Two variables are arc consistent if for every value in the domain of one variable, there exists at least one value in the domain of the other variable that satisfies the binary constraint between them. In other words, if we have two variables X and Y, X and Y are arc consistent if for every value x in the domain of X, there exists at least one value y in the domain of Y such that (X=x, Y=y) satisfies the constraint.

```
  def enforce_arc_consistency(self, domains: dict, X_i: int, X_j: int):
    d_copy = copy_domains(domains)

    if (len(d_copy[X_j]) == 0):
      return None
    if (len(d_copy[X_j]) == 1):
      val = d_copy[X_j][0]
      d_copy = self.reduce_domain(d_copy, val, X_i)
    return d_copy
```

`enforce_arc_consistency` removes values from domain of X_i to make X_i arc consistent with respect to X_j. In our specific graph coloring problem, enforce only happens if there is only one value in the domain of X_j.

Now that we know what arc consistent means, we can talk about the main idea behind AC_3. The AC-3 algorithm simply enforces arc consistency until no domains change. In forward checking, when we assign a variable X_i to a value, we are actually enforcing arc consistency on the neighbors of X_i with respect to X_i. Why stop there? AC-3 doesn't. In AC-3, we start by enforcing arc consistency on the neighbors of X_i (forward checking). But then, if the domains of any neighbor X_j changes, then we enforce arc consistency on the neighbors of , etc.

Implementation of AC_3:

```
  def AC_3(self, domains: dict, X_j: int):
    domains_old = copy_domains(domains)
    domains_new = copy_domains(domains)

    S = [X_j]
    while (len(S) > 0):
      X_j = S.pop(0)
      for X_i in self.neighbours(X_j):
        domains_new = self.enforce_arc_consistency(domains_old, X_i, X_j)
        if (domains_new == None):
          return None
        if (len(domains_old[X_i]) != len(domains_new[X_i])
            and len(domains_new[X_i]) == 1):
          S.append(X_i)
        domains_old = domains_new.copy()
    return domains_new
```

The algorithm starts by initializing a queue, S, with the variable X_j. The while loop continues until the queue is empty. In each iteration of the loop, the algorithm dequeues a variable from the queue, X_j, and enforces arc consistency between X_j and its neighboring variables X_i using the enforce_arc_consistency method.

If enforce_arc_consistency returns None, it means the domain of X_i is empty, and the algorithm has failed. Otherwise, if the domain of X_i has been modified, the algorithm checks if the size of the domain has been reduced to one. If it has, it adds X_i to the queue. Finally, the domains_old dictionary is updated with the domains_new dictionary.

Back to Backtracking

Now that we've talked about MCV, LCV and AC_3 we can implement Backtracking algorithm:

```
  def BackTracking(self, assignments, domains,verbose=1):
    # Check if assignment is complete, if complete then return assignment
    if self.check_complete(assignments) == 1:
      return assignments
    assignments_new = copy_assignments(assignments)

    # Select Most Constrained Variable
    X_i = self.MCV(assignments_new, domains)
    # Order values according to LCV
    ordered_vals = self.LCV(assignments_new, domains, X_i)

    for v in ordered_vals:
      assignments_new[X_i] = v

      # if partial weight is zero, continue
      pw = self._partial_weight(assignments_new, X_i)
      if (pw == 0):
        if verbose > 1:
          print("Constraints do not meet for variable {} and value
{}".format(X_i,v))
        assignments_new = copy_assignments(assignments)
        continue

      domains_new = copy_domains(domains)
      domains_new[X_i] = [v]
      domains_new = self.AC_3(domains_new, X_i)
```

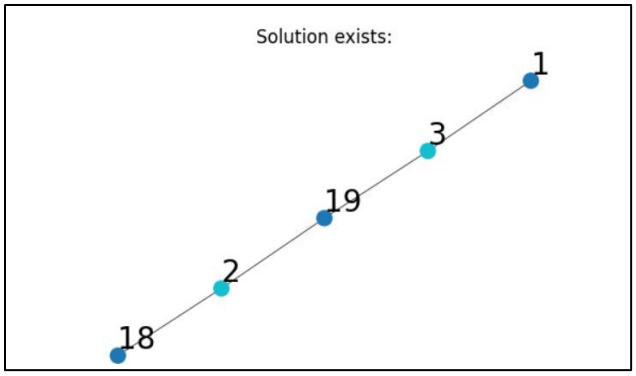
```
      #If any domains_i is empty, continue
      if domains_new is None or any(
          len(domains_new[d_i]) == 0 for d_i in domains_new):
        assignments_new = copy_assignments(assignments)
        domains_new = copy_domains(domains)
        continue
      result = self.BackTracking(assignments_new, domains_new,verbose)
      if result != {}:
        return result
    return {}
```

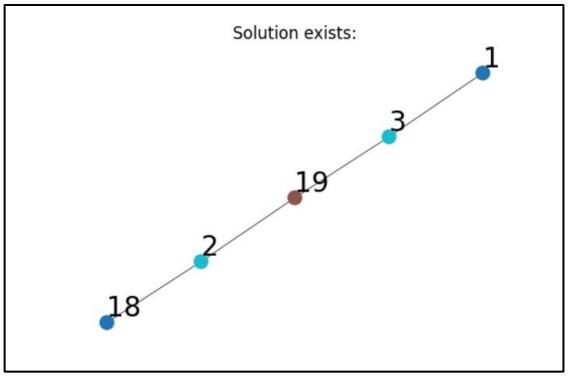
And I have another function to call **BackTracking** function:

```
def BackTracking_Search(self, verbose=1, shuffle=False):
    if(verbose>0):
        print("Backtracking algorithm started...")
        #For measuring algorithm time
        start_time = timer()

    assignments = {}
    domains = copy_domains(self.init_domains)
    if(shuffle):
        domains= shuffle_domains(domains)

    res = self.BackTracking(assignments, domains)
    self.solution = res

    if verbose>0:
        end_time = timer()
        print("Algorithm Ended in {} seconds".format(round(end_time-
start_time,4)))
    return res
```

This function creates assignments, domains and result dictionaries then calls Backtracking. Note that both **BackTracking** and **BackTracking_Search** has argument called verbose, this argument is for displaying information on the screen, when it is set to 1 it measures the time algorithm worked, when it is set to 2 it also shows when algorithm backtracks to previous values and nodes.

We also have option to shuffle the domain of each node before calling backtrack. The reason that we have it is that when we use LCV, in case of tie, it will always return the first color that comes in the domain, therefore our algorithm uses last colors in very constrained situations. To overcome this, shuffling could be option. Suppose that we want to color below graph with 3 colors:





Solution without shuffling domains Solution with shuffling domains

Of course both version are correct, that is why I made shuffling or not shuffling optional, if user wants to see more diverse colors they can shuffle.

Unit Tests

The test cases are written for checking the Backtracking algorithm as well as parts of it. First off all we define a graph on setUp function, we will test **`MCV,LCV, AC_3`** and also other utility functions like **`check_complete, reduce_domain,`**

```
def setUp(self):
  # Add setup code here
  edge_list = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]
  colors = [1,2,3]
  graph = {0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 3], 3: [1, 2]}
  init_domains = {0: [1,2,3], 1: [1,2,3],
                  2: [1,2,3], 3: [1,2,3]}
  self.gc = GraphColoring(edge_list, colors, graph, init_domains)
```

Tests for **`AC_3`**, **`check_complete, reduce_domain, MCV,LCV`**

```
def test_AC_3(self):
  # We coloured 0 with 1, now we will call AC_3 on 0
  domains = {0: [1], 1: [1, 2, 3], 2: [1, 2, 3], 3: [1, 2, 3]}
  domains_new = self.gc.AC_3(domains, 0)
  # Assert the output is as expected
  expected_domains = {0: [1], 1: [2, 3], 2: [2, 3], 3: [1,2,3]}
  self.assertEqual(domains_new, expected_domains)

def test_check_complete(self):
```

```
    #Test if check_complete returns 0 if the assignments are incomplete or not
correct,
    #and 1 otherwise
    assignments = {0: 1, 1: 2, 2: 3, 3: 1}
    self.assertEqual(self.gc.check_complete(assignments), 1)
    assignments = {0: 1, 1: 2, 2: 3, 3: 2}
    self.assertEqual(self.gc.check_complete(assignments), 0)

  def test_reduce_domain(self):
    #Test if reduce_domain remove the specified value from the domain of the
specified variable
    domains = {0: [1,2,3], 1: [1,2,3], 2: [1,2,3], 3: [1,2,3]}
    new_domains = self.gc.reduce_domain(domains, 1, 1)
    self.assertEqual(new_domains[1], [2,3])

  def test_LCV(self):
    # Test if LCV returns the domain of the variable ordered
    # by decreasing number of consistent values of neighboring variables
    # if tie LCV returns reverse order of values
    assignments = {3:1}
    domains = {0: [1,2,3], 1: [2, 3], 2: [2, 3], 3: [1]}
    self.assertEqual(self.gc.LCV(assignments, domains, 0), [1,3,2])

  def test_MCV(self):
    assignments = {0:1}
    domains = {0:1, 1:[2,3], 2:[2,3], 3:[1,2,3]}
    self.assertEqual(self.gc.MCV(assignments, domains), 1)
```

I also tested backtrack algorithm on different example graphs. I am writing one of the test

```
def test_simple_graph(self):
  # Enter code here
    # initialize graph
    edge_list = [(1,2),(2,3),(3,4),(4,1)]
    colors = [1,2,3]
    graph = {1: [2, 4], 2: [1, 3], 3: [2, 4], 4: [1, 3]}
    init_domains = {1: [1,2,3], 2: [1,2,3], 3: [1,2,3], 4: [1,2,3]}

    # initialize graph coloring object
    g = GraphColoring(edge_list, colors, graph, init_domains)

    # perform graph coloring with backtracking and MCV, LCV, AC_3
    solution = g.BackTracking_Search()
    # assert that solution is valid
    self.assertTrue(g.check_complete(solution))
```

Overall, these unit tests provide a comprehensive evaluation of the implementation of the graph coloring problem and help ensure its correctness.

Conclusion

In conclusion, the backtracking algorithm is an effective approach to solving the map coloring problem. By incorporating various techniques such as Minimum Remaining Values (MCV), Least Constraining Value (LCV) and AC-3, the algorithm was able to efficiently search for a valid solution.

MCV and LCV heuristics were used to select the next variable to be assigned a value, and to determine the order in which the values should be assigned. This approach helped in reducing the search space and finding a solution quickly.

In addition, the AC-3 algorithm was used to prune inconsistent values from the domain of variables, which further reduced the search space and improved the efficiency of the algorithm.

Source Code

The source code can be found on my GitHub repo: https://github.com/ShamilShukurov/Graph-Coloring-CSP-

For running the program use the following command:

```
python3 main.py <filepath>
```

where filepath is a filepath to input file.

For drawing the graph matplotlib and networkx are used, for running the program these modules should be installed

Some Examples

Some example generated: