Nandita Venkatesh
G37588483

Project 2 Report

Project Description

A graph is given as a text file that is supposed to be colored. The proper vertex coloring is such that each vertex is assigned a color, and no two adjacent vertices are assigned the same color.

Write a CSP algorithm to solve this coloring problem. The CSP algorithm should have the following components:

• Search algorithm to solve the CSP

• Heuristics (min remaining values, least constraining value)

• Constraint propagation using AC3.

Solution:

Search Algorithm to Solve CSP

The Search Algorithm for CSP was implemented using backtracking. This algorithm runs recursively and applies the constraints based on Minimum Remaining Values (MRV), Least Constraining Values (LCV), and Arc Consistency.

The first step would be to define the nodes and the edges of the graph. From the given text file, I compute the graph, and this is done with the help of the functions **readColor** and **readGraph**.

```
# This function reads the file to get the corresponding colors
def readColor(file):
```

```
# This function reads the file to get a list of nodes and corresponding edges
def readGraph(file):
```

The graph is stored as a list of lists containing nodes, and each node is associated with its neighbors. The length of this list is equal to the number of nodes present in the graph. Then we ensure that no duplicates remain by converting the list into a set and then the set back into a list. The heuristics are then implemented on this list of lists.

Heuristics

To choose the next node that we start with, we use **Minimum Remaining Values (MRV)** constraint. This is implemented in the function **mrv_heuristic.** Here the first node chosen is the node with the most neighbors. Then the heuristic selects the node with the least domain values left. In doing so, it selects the node that, if not chosen, will have a high chance of failure in the future.

```python
# Implementing MRV heuristic to choose a node for coloring
def mrv_heuristic(adjList, domainList, assigned):
    minColorsAvailable = 1000
    nextNode = -1
    if 1 in assigned:
        for i in range(Nodes):
            if len(domainList[i]) < minColorsAvailable and assigned[i] != 1:
                minColorsAvailable = len(domainList[i])
                nextNode = i
        return nextNode
    else:
        nextNode = -1
        length = -1
        for x in range(len(adjList)):
            if len(adjList[x]) > length:
                nextNode = x
                length = len(adjList[x])
        return nextNode
```

```python
# Implementing the LCV to choose the priority of coloring depending on the restriction a node imposes on its neighbours
def lcv_heuristic(nextNodeChosen, adjList, domainList):
    orderOfColoring = []
    for colors in domainList[nextNodeChosen]:

        minValue = 100
        for i in adjList[nextNodeChosen]:

            noOfColorsRemaining = len(domainList[i])
            if colors in domainList[i]:
                noOfColorsRemaining = noOfColorsRemaining - 1
            if noOfColorsRemaining < minValue:
                minValue = noOfColorsRemaining
        orderOfColoring.append([colors, minValue])
        sortedOrderOfColoring = sorted(orderOfColoring, key=lambda x: x[1], reverse=True)
        colorOrderPriority = [item[0] for item in sortedOrderOfColoring]
    return colorOrderPriority
```

Once the node is selected, it is assigned as the variable **nextNode** then, the **Least Constraining Value (LCV)** heuristic is applied. This is implemented in **lcv_heuristic** function. Here we check the number of colors remaining in the domain for each node and then order the list with the node having the least constraints first. This ensures that the chosen node has the least effect on its neighbor and is less likely to cause failure.

Constraint propagation using AC3

This algorithm forward-checks the neighbors and removes the color assigned to the current node. This way, we can remove the inconsistent values and make the CSP backtracking algorithm more efficient. If the domain of any node becomes empty, then we tag it as a false condition, the node is assigned a new value, and AC-3 is applied again.

```python
# Implementing the AC-3 on the neighbours of the selected node
def ac3Check(nextNode, l, adjList, domainListUpdated):
    while len(l) != 0:
        a = l.popleft()
        [remove, domainListUpdated] = remove_inconsistent_values(a, domainListUpdated)

        if remove == 1:
            if len(domainListUpdated[a[0]]) == 0:
                return 0, domainListUpdated
            adjlistTemp = [item for item in adjList[a[0]] if item != a[1]]

            for j in adjlistTemp:
                b = [j, a[0]]
                if isAssigned[j] != 1:
                    l.append(b)

    return 1, domainListUpdated
```

```python
# This function removes the domain of connected nodes when one node is assigned with a color
def remove_inconsistent_values(a, domainListUpdated):
    removed = 0
    if len(domainListUpdated[a[1]]) == 1:
        # print("a[1]:",a[1])
        c = domainListUpdated[a[1]][0]
        # print("c:",c)
        if c in domainListUpdated[a[0]]:
            domainListUpdated[a[0]].remove(c)
            removed = 1
    return removed, domainListUpdated
```

Once the assignment of the colors is completed, the values are verified as a final check by going over the graph and ensuring that no neighbors are assigned the same color.

GitHub Link

The code was written in Python 3.7; The following steps are to be followed to run the Program on the local machine:

1. Open the GitHub link provided below.
2. Download the code and Extract the File on your Local System.
3. Open the Project in your IDE.
4. Run the **unitTest_GraphColoring.py** file to see the result of the unit tests.

GitHub Link - https://github.com/nanditavenkatesh/CSCI6511_AI_Project2.git