University of Isfahan

Carpet Project

Professor: Dr. Peyman Adibi

Authors:

Melika Shirian

Kianoosh Vadaei

Spring 2023

Table of Contents

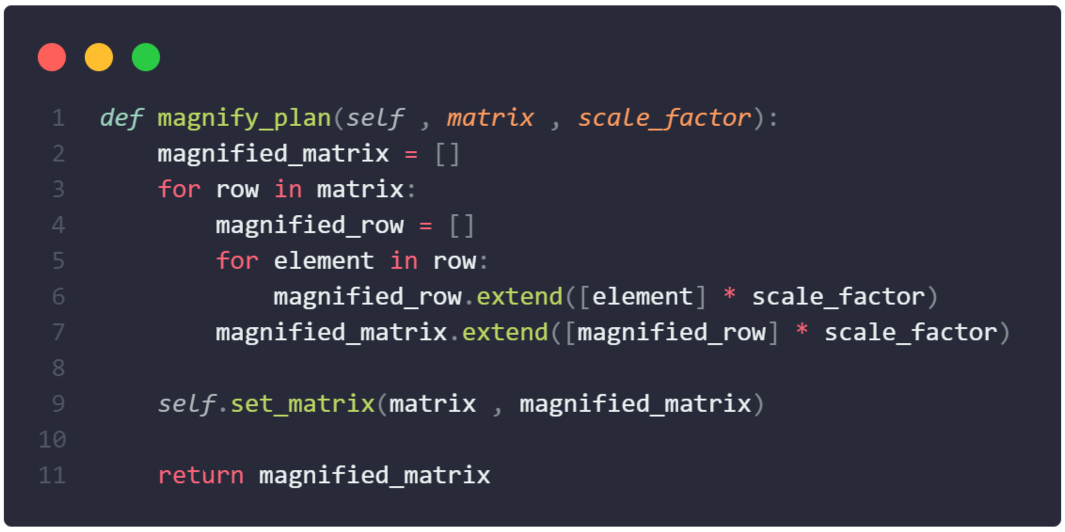
In the following section, we explain the important functions of the program.

Magnify plan function:

In this function, the 8*6 input matrix is converted into a 400*300 matrix by the following algorithm.

In this way, each element of the input matrix expands to 50 elements in both column and row directions.

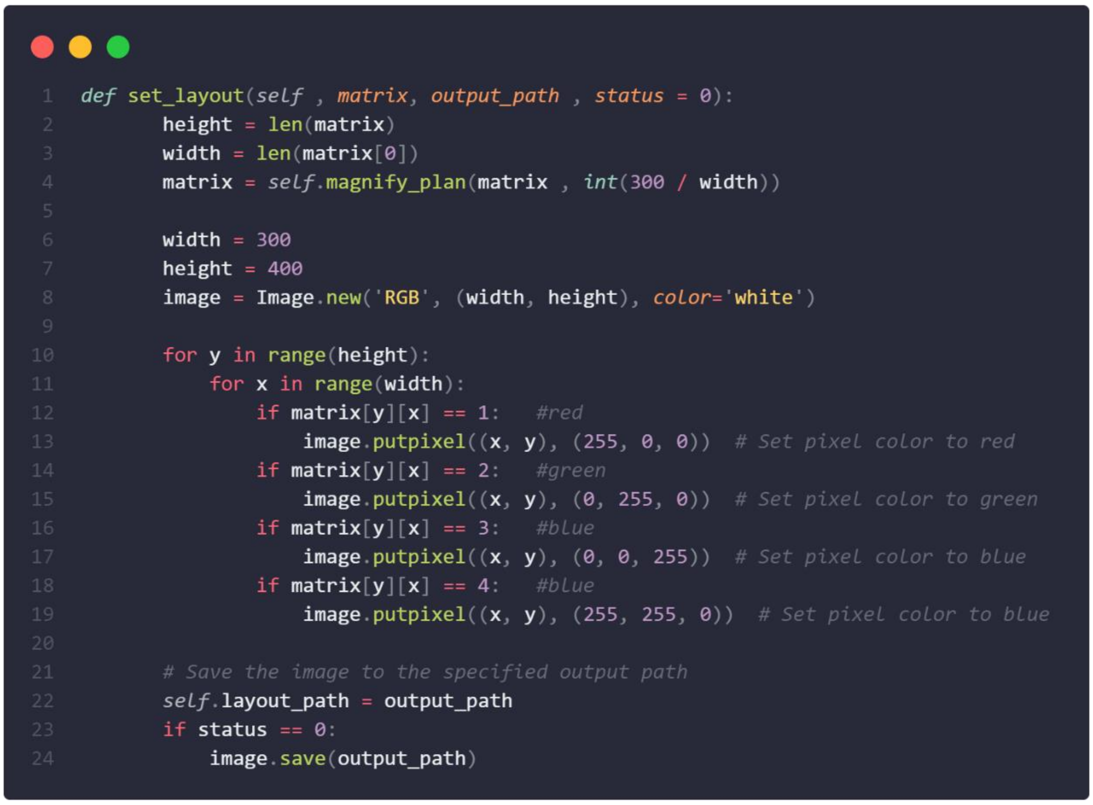
Also, in the reverse magnify function, the exact opposite of this happens



```python
def magnify_plan(self , matrix , scale_factor):
    magnified_matrix = []
    for row in matrix:
        magnified_row = []
        for element in row:
            magnified_row.extend([element] * scale_factor)
        magnified_matrix.extend([magnified_row] * scale_factor)

    self.set_matrix(matrix , magnified_matrix)

    return magnified_matrix
```

Set layout function:

The "magnify_plan" method takes a matrix and a scale factor as input and returns a magnified version of the matrix. It achieves this by creating an empty magnified matrix and then iteratively enlarging each element in the original matrix by repeating it horizontally "scale_factor" (which is 50 here) times and vertically "scale_factor" times. The resulting magnified matrix is then returned.



```python
def set_layout(self , matrix, output_path , status = 0):
        height = len(matrix)
        width = len(matrix[0])
        matrix = self.magnify_plan(matrix , int(300 / width))

        width = 300
        height = 400
        image = Image.new('RGB', (width, height), color='white')

        for y in range(height):
            for x in range(width):
                if matrix[y][x] == 1:   #red
                    image.putpixel((x, y), (255, 0, 0))  # Set pixel color to red
                if matrix[y][x] == 2:   #green
                    image.putpixel((x, y), (0, 255, 0))  # Set pixel color to green
                if matrix[y][x] == 3:   #blue
                    image.putpixel((x, y), (0, 0, 255))  # Set pixel color to blue
                if matrix[y][x] == 4:   #blue
                    image.putpixel((x, y), (255, 255, 0))  # Set pixel color to blue

        # Save the image to the specified output path
        self.layout_path = output_path
        if status == 0:
            image.save(output_path)
```

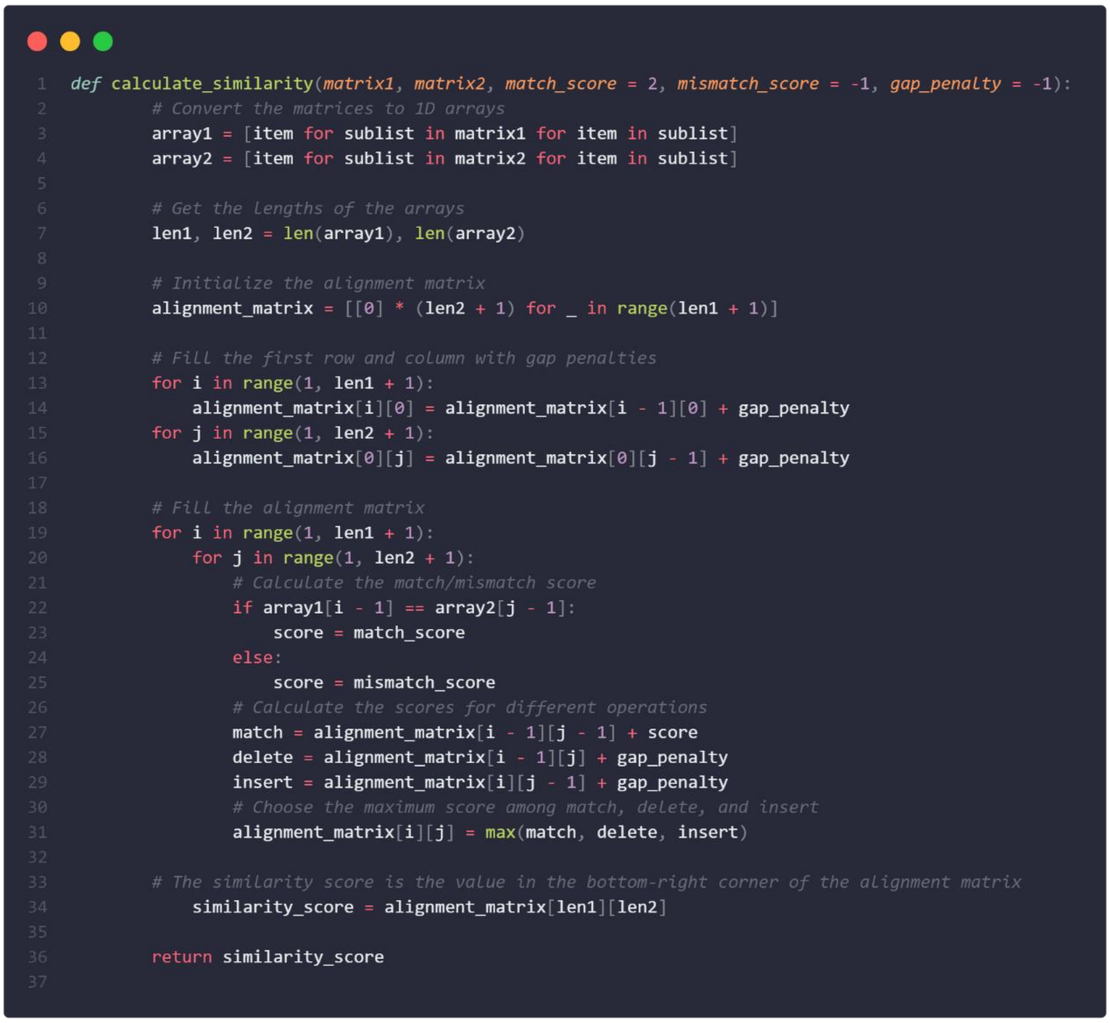
Calculate similarity function:

The "calculate_similarity" function takes two matrices (matrix1 and matrix2) along with optional parameters for match score, mismatch score, and gap penalty. It calculates the similarity score between the two matrices using the Needleman-Wunsch algorithm.

First, the function converts the matrices into 1D arrays (array1 and array2). Then, it initializes an alignment matrix with dimensions based on the lengths of the arrays.

The function fills the first row and column of the alignment matrix with gap penalties. Then, it iterates through the remaining cells of the alignment matrix and calculates the match/mismatch score based on the corresponding elements of array1 and array2. It calculates the scores for three possible operations: match (diagonal movement), delete (vertical movement), and insert (horizontal movement). The maximum score among these operations is stored in the current cell of the alignment matrix.

Finally, the similarity score is obtained by retrieving the value in the bottom-right corner of the alignment matrix, and it is returned as the result of the function.



```python
def calculate_similarity(matrix1, matrix2, match_score = 2, mismatch_score = -1, gap_penalty = -1):
        # Convert the matrices to 1D arrays
        array1 = [item for sublist in matrix1 for item in sublist]
        array2 = [item for sublist in matrix2 for item in sublist]

        # Get the lengths of the arrays
        len1, len2 = len(array1), len(array2)

        # Initialize the alignment matrix
        alignment_matrix = [[0] * (len2 + 1) for _ in range(len1 + 1)]

        # Fill the first row and column with gap penalties
        for i in range(1, len1 + 1):
            alignment_matrix[i][0] = alignment_matrix[i - 1][0] + gap_penalty
        for j in range(1, len2 + 1):
            alignment_matrix[0][j] = alignment_matrix[0][j - 1] + gap_penalty

        # Fill the alignment matrix
        for i in range(1, len1 + 1):
            for j in range(1, len2 + 1):
                # Calculate the match/mismatch score
                if array1[i - 1] == array2[j - 1]:
                    score = match_score
                else:
                    score = mismatch_score
                # Calculate the scores for different operations
                match = alignment_matrix[i - 1][j - 1] + score
                delete = alignment_matrix[i - 1][j] + gap_penalty
                insert = alignment_matrix[i][j - 1] + gap_penalty
                # Choose the maximum score among match, delete, and insert
                alignment_matrix[i][j] = max(match, delete, insert)

        # The similarity score is the value in the bottom-right corner of the alignment matrix
            similarity_score = alignment_matrix[len1][len2]

        return similarity_score
```

Search function:

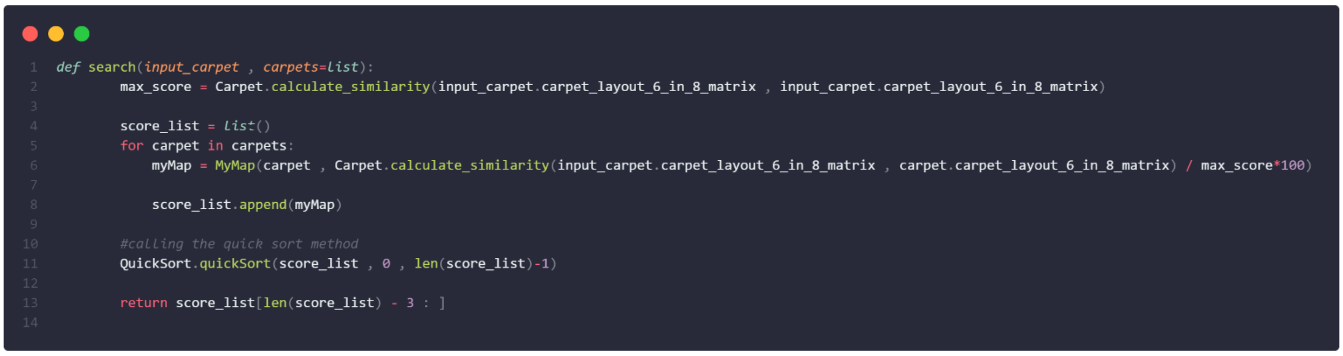
The "search" function searches for an input carpet (input_carpet) within a list of carpets (carpets). It first calculates the similarity score between the input carpet and itself as the maximum score.

Then, it calculates the similarity scores between the input carpet and each carpet in the list. These scores are stored in a list called **score_list**, where each score is associated with the corresponding carpet.

The "quicksort" method is called to sort the "score_list" in descending order based on the similarity scores.

Finally, the function returns the top three carpets with the highest similarity scores, which can be accessed from "score_list[len(score_list) - 3:]".

In summary, the "search" function finds the most similar carpets to the input carpet from a given list of carpets based on their similarity scores.



```
def search(input_carpet , carpets=list):
        max_score = Carpet.calculate_similarity(input_carpet.carpet_layout_6_in_8_matrix , input_carpet.carpet_layout_6_in_8_matrix)

        score_list = list()
        for carpet in carpets:
            myMap = MyMap(carpet , Carpet.calculate_similarity(input_carpet.carpet_layout_6_in_8_matrix , carpet.carpet_layout_6_in_8_matrix) / max_score*100)

            score_list.append(myMap)

        #calling the quick sort method
        QuickSort.quickSort(score_list , 0 , len(score_list)-1)

        return score_list[len(score_list) - 3 : ]
```

Efficient Shopping function:

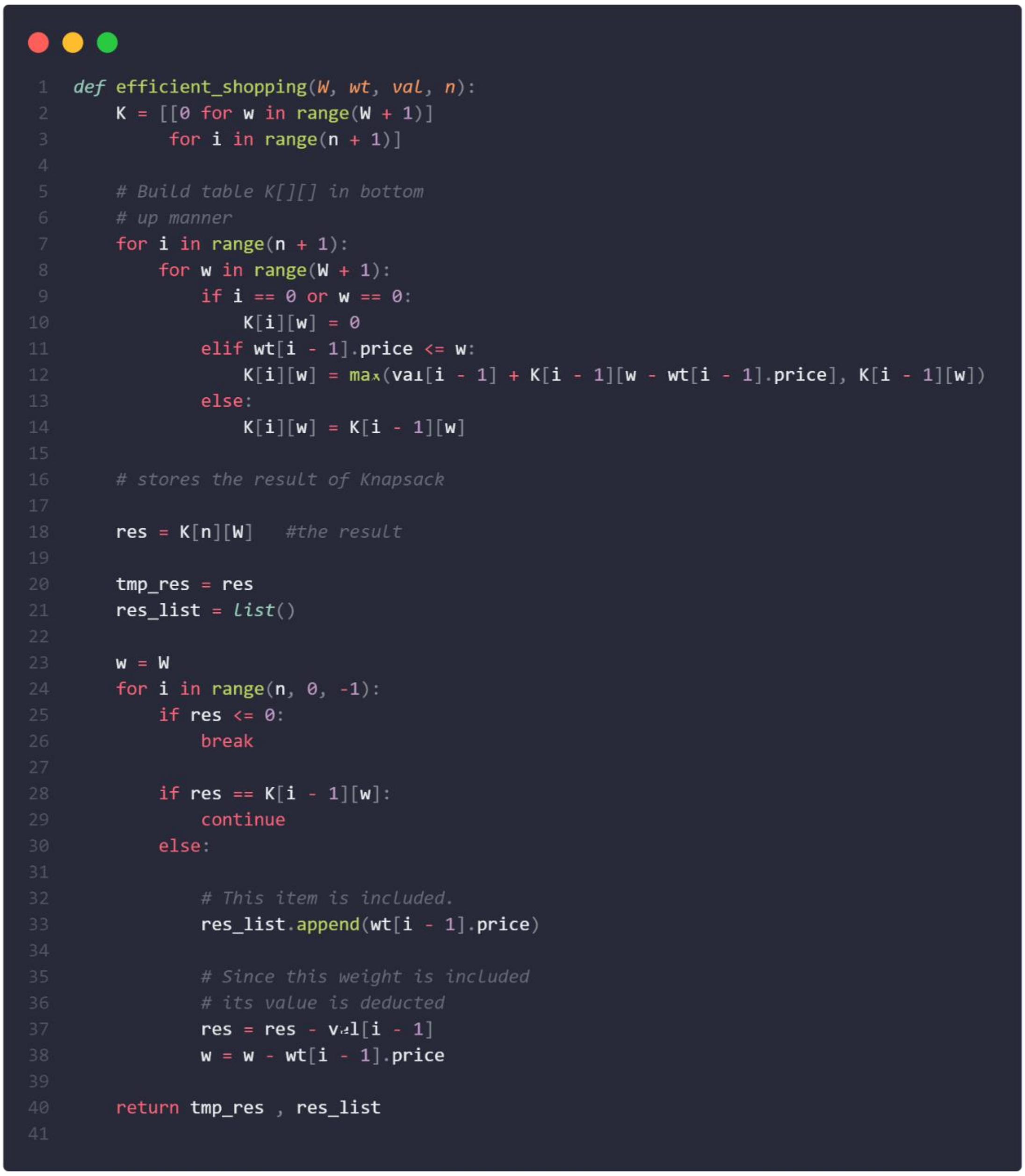
This function uses a dynamic programming algorithm to solve the knapsack problem optimally. The function takes inputs including the knapsack capacity (W), the weights of items (wt), the values of items (val), and the number of items (n).

The algorithm calculates the optimal values in a bottom-up manner using a table called K. In this table, K[i][w] represents the maximum value achievable by considering the first i items and a knapsack capacity of w.

The result of the knapsack problem (the optimal value) is stored in the variable res. Then, by traversing backwards, the code tracks the items that contributed to this result and stores their weights in the res_list list.

Finally, the function returns the optimal value (tmp_res) and the list of weights (res_list).

This algorithm efficiently calculates the maximum achievable value in the knapsack by utilizing dynamic programming and reusing previous computations.



```python
def efficient_shopping(W, wt, val, n):
    K = [[0 for w in range(W + 1)]
         for i in range(n + 1)]

    # Build table K[][] in bottom
    # up manner
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i - 1].price <= w:
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1].price], K[i - 1][w])
            else:
                K[i][w] = K[i - 1][w]

    # stores the result of Knapsack

    res = K[n][W]    #the result

    tmp_res = res
    res_list = list()

    w = W
    for i in range(n, 0, -1):
        if res <= 0:
            break

        if res == K[i - 1][w]:
            continue
        else:

            # This item is included.
            res_list.append(wt[i - 1].price)

            # Since this weight is included
            # its value is deducted
            res = res - val[i - 1]
            w = w - wt[i - 1].price

    return tmp_res , res_list

```

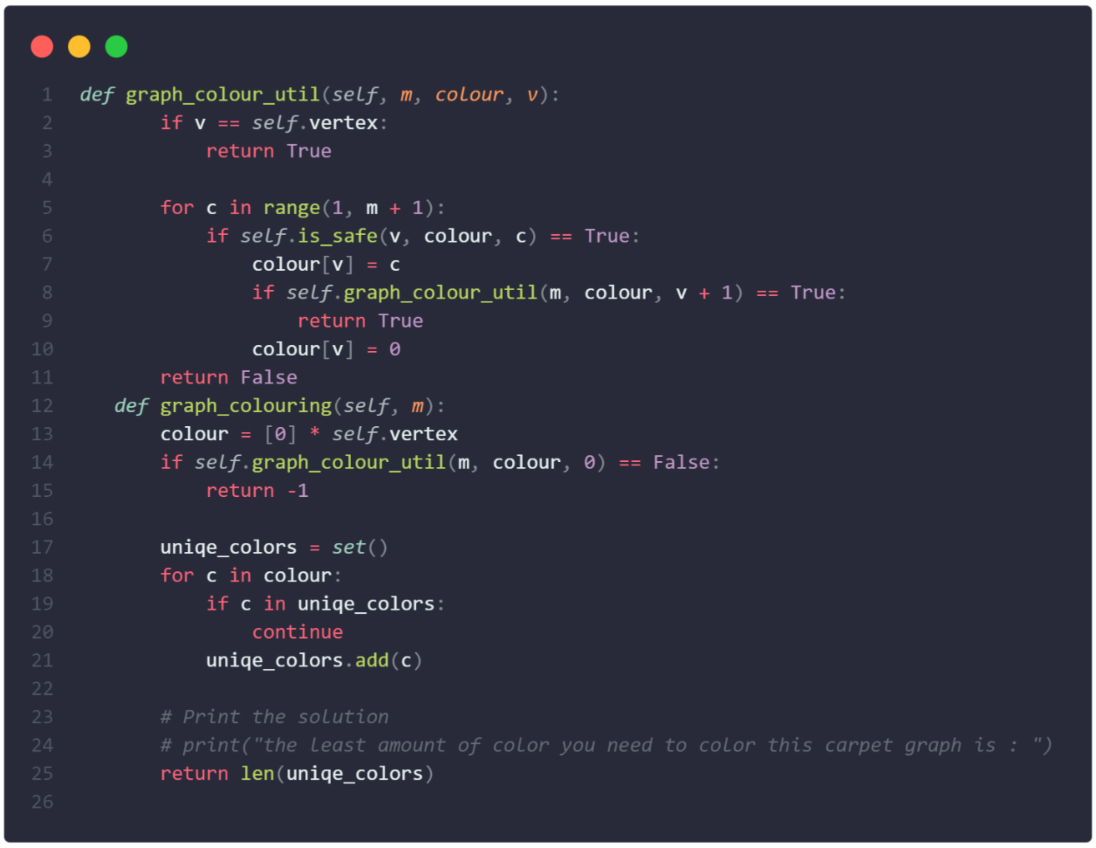
Graph Color Util and Graph Coloring functions:

In this part we implemented a graph coloring algorithm for coloring a graph using a minimum number of colors. The algorithm aims to assign colors to vertices of the graph in such a way that no adjacent vertices have the same color.

The "graph_colour_util" function is a recursive helper function that explores different color combinations for each vertex. It starts with the first vertex and checks all possible colors (from 1 to m) for that vertex. If a color is safe to assign (determined by the "is_safe" function), it assigns the color to the vertex and recursively moves on to the next vertex. If a valid coloring is achieved for all vertices, it returns True. Otherwise, it backtracks by resetting the color assigned to the current vertex and tries the next available color.

The "graph_colouring" function initializes a color array and calls the "graph_colour_util" function to perform the graph coloring. If a valid coloring is not possible (indicated by "graph_colour_util" returning False), it returns -1. Otherwise, it calculates the number of unique colors used by iterating over the color array and adding each color to a set. The function returns the count of unique colors used.

The algorithm efficiently finds the minimum number of colors required to color the graph by backtracking and exploring different color assignments for each vertex. It ensures that adjacent vertices have different colors, resulting in a valid coloring solution for the graph.



```
def graph_colour_util(self, m, colour, v):
        if v == self.vertex:
            return True

        for c in range(1, m + 1):
            if self.is_safe(v, colour, c) == True:
                colour[v] = c
                if self.graph_colour_util(m, colour, v + 1) == True:
                    return True
                colour[v] = 0
        return False
    def graph_colouring(self, m):
        colour = [0] * self.vertex
        if self.graph_colour_util(m, colour, 0) == False:
            return -1

        uniqe_colors = set()
        for c in colour:
            if c in uniqe_colors:
                continue
            uniqe_colors.add(c)

        # Print the solution
        # print("the least amount of color you need to color this carpet graph is : ")
        return len(uniqe_colors)
```

Dijkstra functions:

This code implements the Dijkstra's algorithm for finding the shortest path in a weighted graph from a given source vertex (src) to all other vertices. The algorithm works on a graph represented by an adjacency matrix.

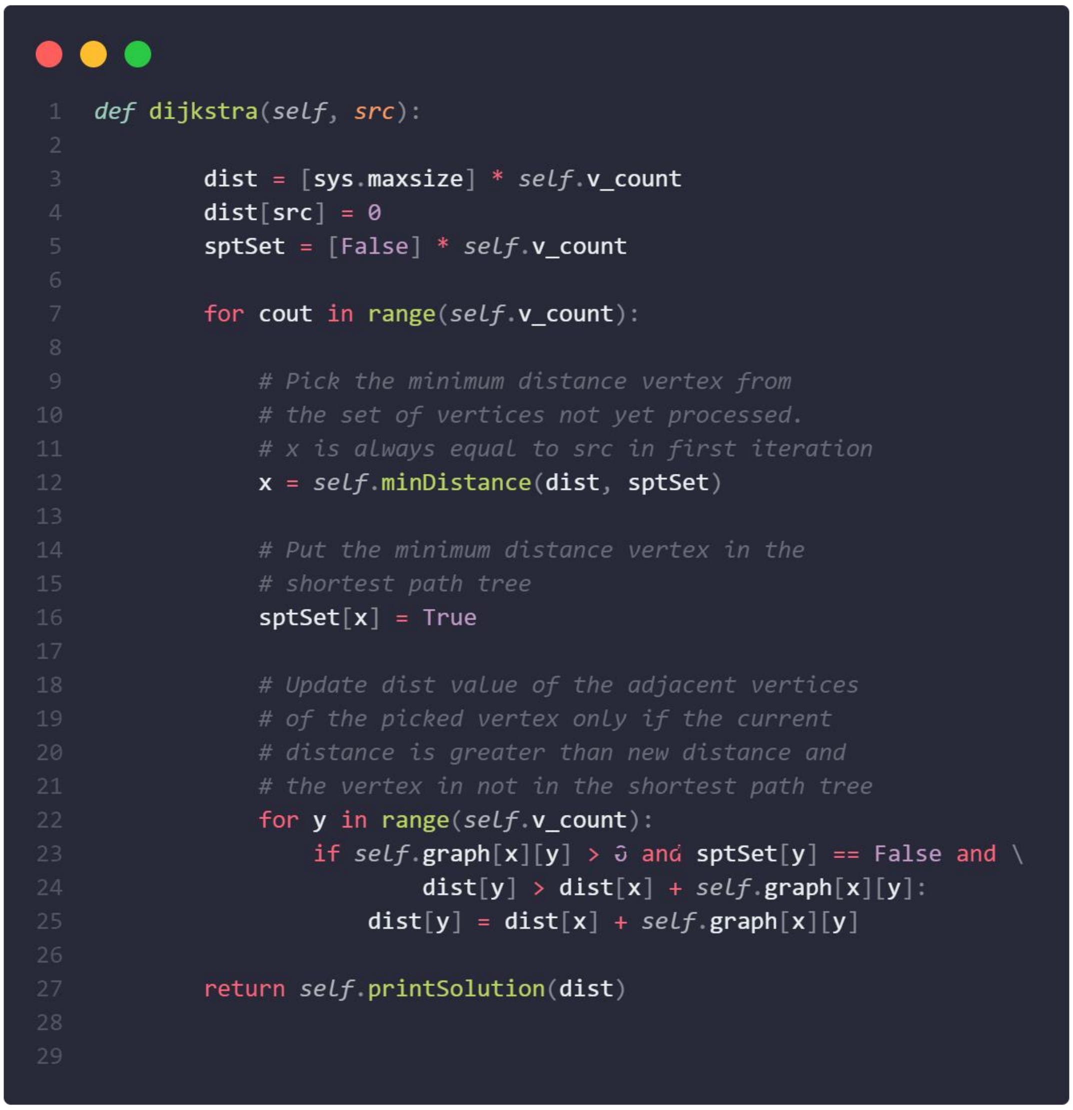
The "Dijkstra" function takes the source vertex as input and initializes the distance array (dist) with maximum values except for the source vertex which is set to 0. It also initializes a boolean array (sptSet) to keep track of the vertices included in the shortest path tree.

The algorithm iterates for all vertices in the graph. In each iteration, it selects the vertex with the minimum distance value (using the "minDistance" function) among the set of vertices not yet processed. Initially, the source vertex is chosen. The selected vertex is marked as processed by setting its corresponding value in the sptSet array to True.

Then, the algorithm updates the distance values of the adjacent vertices of the selected vertex. It checks if there is an edge between the selected vertex (x) and the adjacent vertex (y), and if the adjacent vertex is not already included in the shortest path tree (sptSet[y] == False). If the current distance to vertex y is greater than the sum of the distance to vertex x and the weight of the edge between x and y, it updates the distance value of y accordingly.

After processing all vertices, the algorithm returns the result by calling the "printSolution" function, which can be used to display the calculated shortest distances from the source vertex to all other vertices.

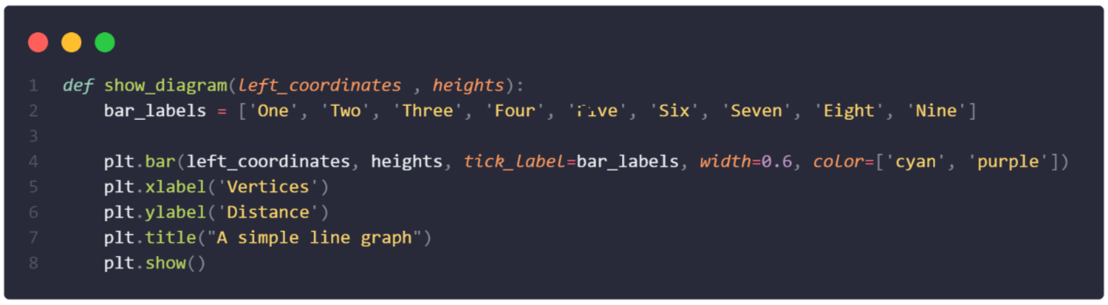
Overall, this code efficiently computes the shortest paths from a source vertex to all other vertices in a weighted graph using Dijkstra's algorithm.



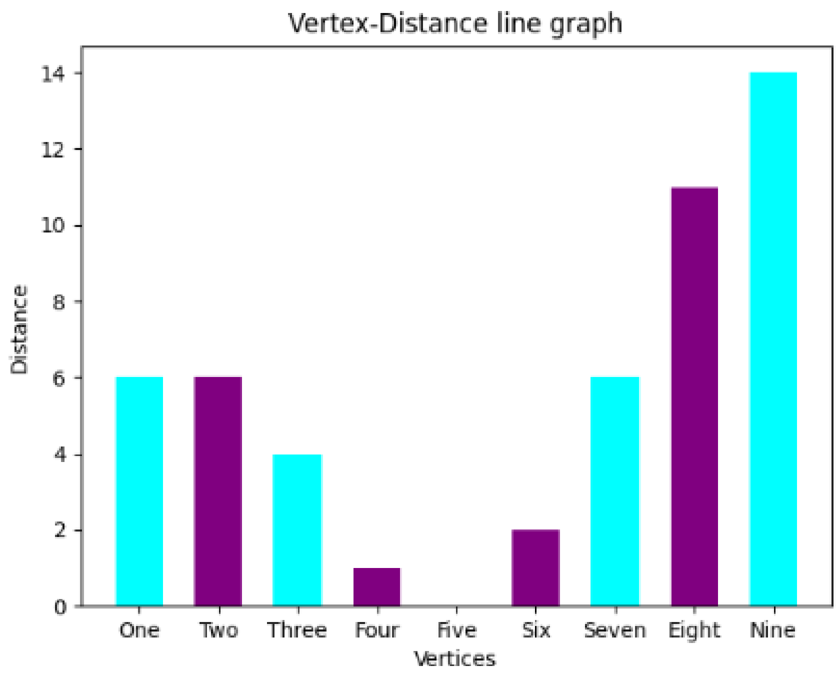
```python
def dijkstra(self, src):

        dist = [sys.maxsize] * self.v_count
        dist[src] = 0
        sptSet = [False] * self.v_count

        for cout in range(self.v_count):

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # x is always equal to src in first iteration
            x = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the
            # shortest path tree
            sptSet[x] = True

            # Update dist value of the adjacent vertices
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shortest path tree
            for y in range(self.v_count):
                if self.graph[x][y] > 0 and sptSet[y] == False and \
                        dist[y] > dist[x] + self.graph[x][y]:
                    dist[y] = dist[x] + self.graph[x][y]

        return self.printSolution(dist)
```

Show Diagram function :

The "show_diagram" static method is responsible for displaying a bar diagram using the "matplotlib.pyplot" library. It takes two parameters: "left_coordinates", which represents the x-coordinates of the bars, and "heights", which represents the heights (distances) of the bars. The method creates a bar plot using the "plt.bar" function, setting the tick labels, width, and color of the bars. It also sets the x-axis label to "Vertices", the y-axis label to "Distance", and the title of the plot to "A simple line graph". Finally, it displays the plot using "plt.show()".



```python
def show_diagram(left_coordinates , heights):
    bar_labels = ['One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine']

    plt.bar(left_coordinates, heights, tick_label=bar_labels, width=0.6, color=['cyan', 'purple'])
    plt.xlabel('Vertices')
    plt.ylabel('Distance')
    plt.title("A simple line graph")
    plt.show()
```



Inputs and outputs

Design:

Input sample:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Output:

- The least amount of color you need to color this carpet graph is: 3

Navigation to the nearest factory store:

Input sample:

five

Output:

- The nearest department to you is: 4 (And the diagram)

Note that **Search by map layout** part has been implemented in such a way that it takes the photo as input. Input examples are placed in the project files.

Also, in **Purchase based on amount of money** part, the values in the program are generated randomly, so they have no fixed input and output

Sources:

Geeks for Geeks (https://www.geeksforgeeks.org/): We drew inspiration and obtained valuable insights from the renowned website Geeks for Geeks during the development of this project. The site provided valuable resources, tutorials, and examples that influenced our approach and contributed to the overall design and implementation of our solution.