

University of Isfahan

Carpet Project

Professor: Dr. Peyman Adibi

Authors:

Melika Shirian

Kianoosh Vadaei

Spring 2023

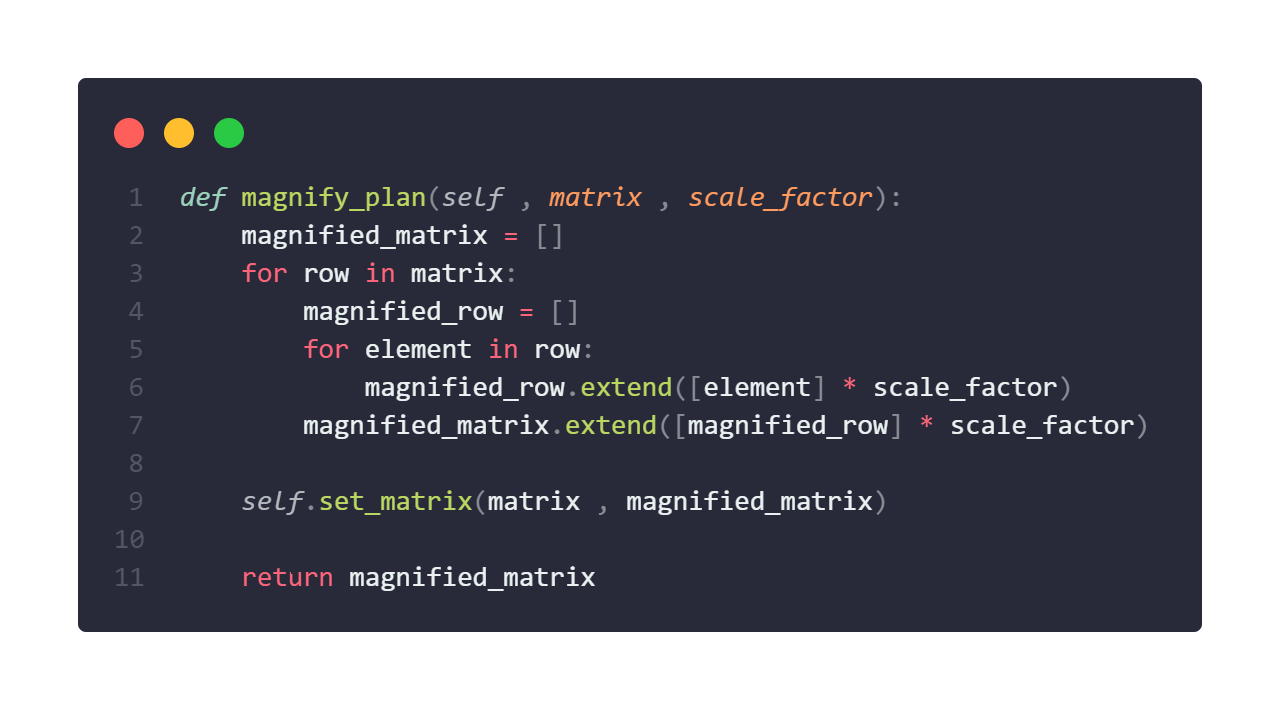
**Carpet Project**

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

**Set layout function:**

In this function, the 6x8 input matrix is ​​first converted to 300x400 with Magnify function, and for each non-zero value, the corresponding pixel in that coordinate is colored.



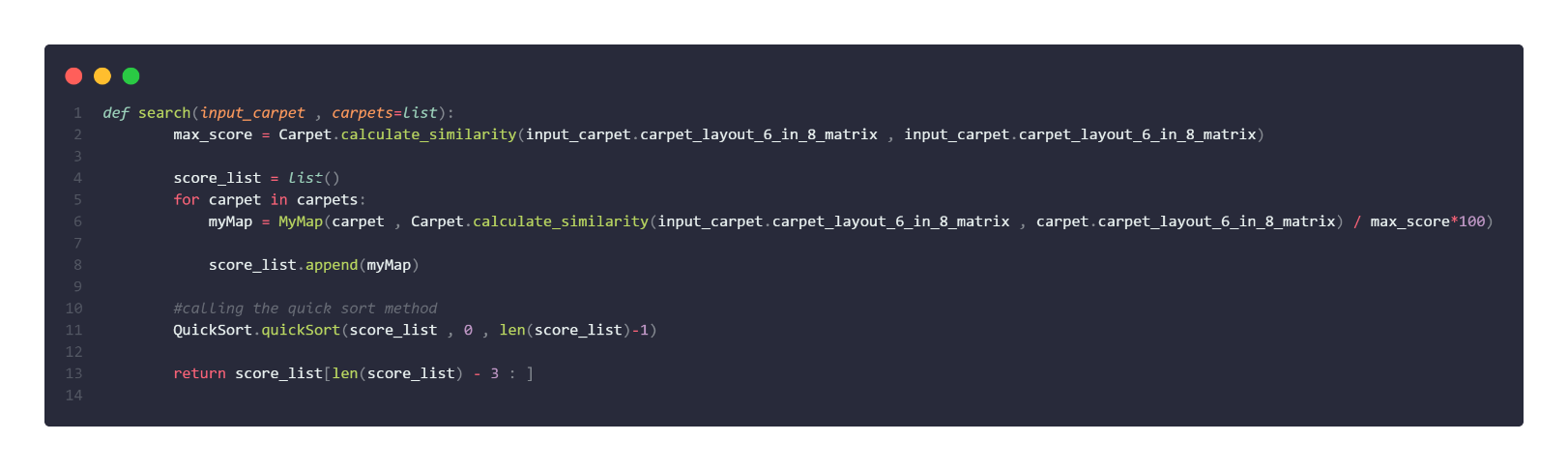
**Calculate similarity function:**

**** At first, input matrices are taken and converted into one-dimensional arrays. Then, with the Needleman–Wunsch algorithm, we form a matrix, equal to the length of the second array and the first array, as rows and columns, with zero elements, and fill the first row and the first column with the value of the gap penalty, which is -1. After that, we navigate through the matrix using two loops and find the three values of match, delete and insert respectively as follows:  
The similarity of two elements, the positional difference of an element, and the incremental difference of an element.  
After that, we put the maximum of these three values in the corresponding index.  
The similarity score of two input matrices is the last element of the formed matrix.

**Search function:**

In the search function, we first find the similarity score of the input matrix and itself and put it in a variable.

After that, we scroll through the list of matrices (carpets) and store the similarity percentage of that matrix and the input matrix in a map with the key of each matrix.

After that, using the quick sort algorithm, we sort the list based on the values ​​and return the last three that have the most points.

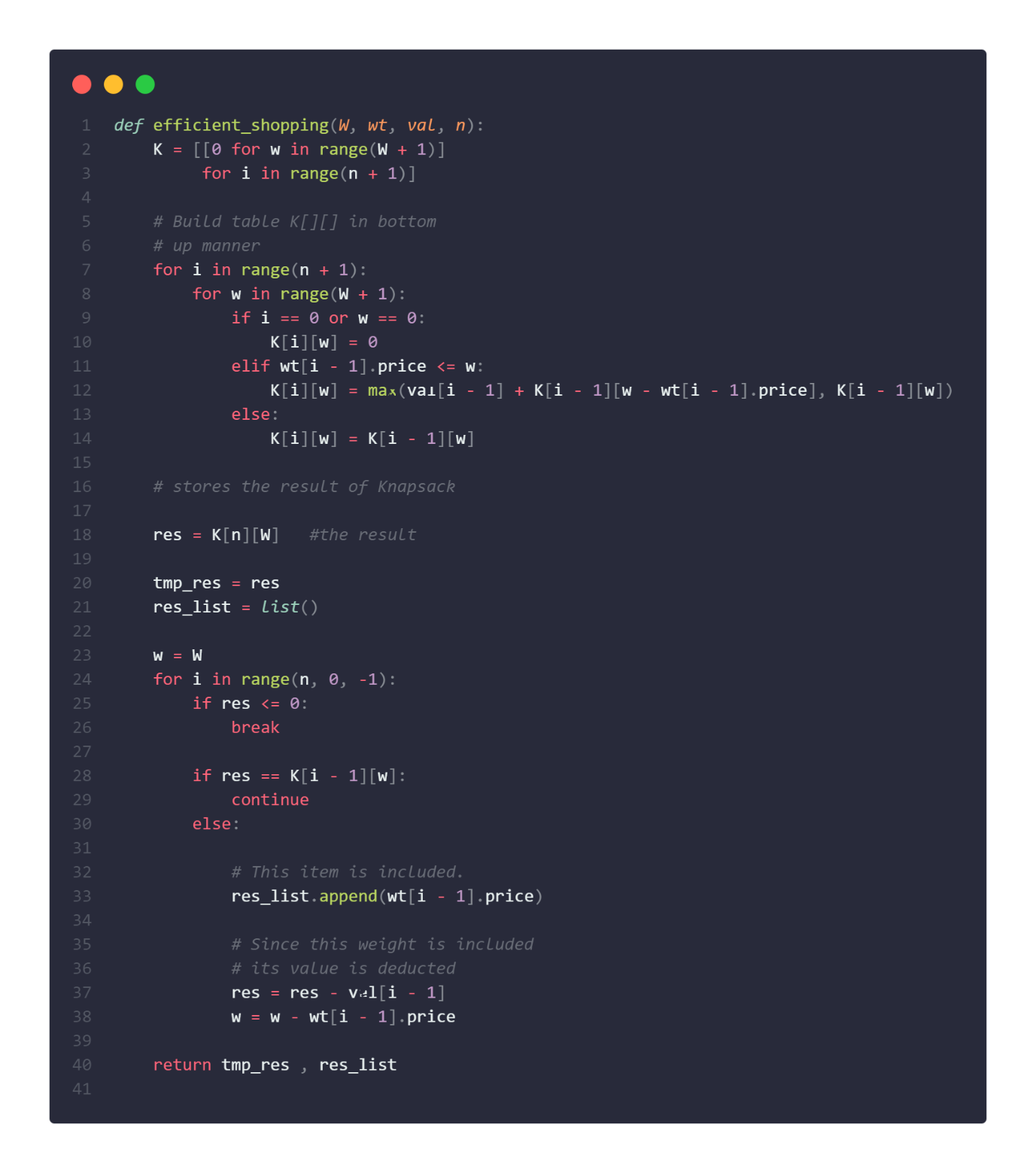
**Efficient Shopping function:**  
This function uses a dynamic programming algorithm to solve the knapsack problem optimally. The function takes inputs including the knapsack capacity (W), the weights of items (wt), the values of items (val), and the number of items (n).

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.



**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.



**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.



**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()".



