



دانشگاه اصفهان
دانشکده مهندسی کامپیوتر

گزارش فنی پروژه الگوریتم

(کارخانه فرش بافی)

اعضای گروه:

زهرا معصومی (۴۰۰۳۶۲۳۰۳۴)

محدثه آخوندی (۴۰۰۳۶۱۳۰۰۲)

استاد: دکتر پیمان ادیبی

بهار ۱۴۰۲

فهرست مطالب

۱	شرح کلی	۳
۲	گزارش کار الگوریتم	۴
۱.۲	بخش اول: طراحی	۴
۲.۲	بخش دوم: بررسی فرش‌های مشابه	۹
۳.۲	بخش سوم: خرید بر اساس میزان پول	۱۳
۴.۲	بخش چهارم: مسیریابی به نزدیک‌ترین شعبه	۱۵

فهرست تصاویر

۱	کلاس FactoryManager	۳
۲	تابع ColorGraph	۴
۳	تابع isValidColor	۵
۴	گراف موردنظر برای رنگ‌آمیزی	۶
۵	نتیجه اجرای الگوریتم رنگ‌آمیزی	۷
۶	گراف رنگ‌آمیزی شده	۸
۷	شروع برنامه و لود کردن تصویر فرش‌ها در برنامه	۹
۸	تابع تعیین price	۱۰
۹	تابع get_minimum_penalty	۱۰
۱۰	فراخوانی sequenceAlignment برای هر سطر از ماتریس فرش‌ها	۱۱
۱۱	اجرای دو فرش مشابه : 4.070.531	۱۱
۱۲	اجرای دو فرش با شباهت کمتر : 5.067.044	۱۲
۱۳	تابع FillCarpetValue	۱۳
۱۴	تابع Knapstack	۱۴
۱۵	تابع FloydWarshal	۱۶
۱۶	تابع ConstructPath	۱۷
۱۷	تابع GetClosestFactoryVertex	۱۸
۱۸	نقشه شهر و خیابان‌ها	۱۸
۱۹	اجرای برنامه برای پیدا کردن کوتاه‌ترین مسیر	۱۹

۱ شرح کلی

این پروژه، عملیات یک کارخانه فرش‌بافی را انجام می‌دهد که نیاز به سامانه‌ای برای مدیریت و مکانیزه کردن کارهای کارخانه دارد. این سامانه شامل بخش مختلفی همچون طراحی، فروش، توزیع و ... می‌شود. عملیات طراحی برای طراحی فرش‌های جدید، فروش برای محاسبه حداکثر تعداد فرش‌ی که یک کاربر می‌تواند بخرد، بخش بررسی شباهت برای پیدا کردن فرش‌های مشابه، و بخش مسیریابی برای پیدا کردن نزدیک‌ترین شعبه به موقعیت کاربر پیاده‌سازی شده است. در این پروژه، عملیات اصلی کارخانه در کلاس FactoryManager پیاده‌سازی شده‌اند. این کلاس شامل property های زیر است:

```
namespace CarpetFactory;

public class FactoryManager
{
    public static List<Carpet> allCarpets = new List<Carpet>();
    private static int[,] Next;
    private static int[,] distance;
    private static CityGraph cityGraph = new CityGraph();
    private static int verticesCount;
}
```

شکل ۱: کلاس FactoryManager

- allCarpets که لیستی شامل همه فرش‌های کارخانه می‌باشد. در ابتدای برنامه، این لیست با ۱۰ فرش رندوم پر می‌شود.
 - Next آرایه‌ای دوبعدی شامل راس‌های کوتاه‌ترین مسیرها بین هر دو راس از گراف شهر.
 - distance آرایه دوبعدی شامل طول کوتاه‌ترین فاصله بین هر دو راس از راس‌های گراف شهر.
 - CityGraph گراف نقشه شهر.
 - verticesCount تعداد راس‌های گراف.
- در ادامه هر بخش از پروژه این سامانه به طور مجزا توضیح داده شده است.

۲ گزارش کار الگوریتم

۱.۲ بخش اول: طراحی

در این بخش، می‌خواهیم فرش‌های جدید را به سفارش مشتری طراحی کنیم. برای این کار، کاربر تعداد اشکال هندسی موردنظر و ارتباط آن‌ها را وارد می‌کند و در خروجی، حداقل تعداد رنگ برای رنگ‌آمیزی آن‌ها و رنگ هر شکل را تحویل می‌گیرد.

برای این کار، از الگوریتم "رنگ‌آمیزی گراف (m-coloring)" و رویکرد Backtracking کمک می‌گیریم. در تابع **ColorGraph** گراف موردنظر برای رنگ‌آمیزی (که شامل اشکال هندسی فرش و ارتباط آن‌ها است)، تعداد راس‌های گراف، آرایه‌ای از رنگ‌ها که هر خانه آن رنگ راس متناظر در گراف را نشان می‌دهد. در نهایت عددی به عنوان حداکثر تعداد رنگ‌های مجاز از ورودی می‌گیرد.




شکل ۲: تابع ColorGraph

در این الگوریتم، به ازای هر راس یک فراخوانی بازگشتی برای رنگ‌آمیزی آن داریم. به این صورت که مانند همه الگوریتم‌های Backtracking، ابتدا شرط امکان‌پذیر بودن بررسی می‌شود. در صورتی که رنگ‌آمیزی راسی با رنگ مشخص شده امکان‌پذیر بود، رنگ آن را در آرایه رنگ‌ها قرار داده و به سراغ راس بعدی می‌رویم. در صورتی که رنگ‌ها تمام شوند و رنگی برای آن راس پیدا نشود، مقدار آن خانه از آرایه را برابر -۱ قرار می‌دهیم و در پایان نیز مقدار false به معنای "عدم امکان‌پذیر بودن رنگ‌آمیزی گراف با رنگ‌های موجود" برمی‌گردانیم. (در این مرحله هرس کردن اتفاق می‌افتد.)

تابع **isValidColor** که برای بررسی شرط امکان‌پذیر بودن استفاده شد، راس و رنگ موردنظر را دریافت کرده و بررسی می‌کند که در همسایه‌های آن راس، راسی با رنگ یکسان وجود نداشته باشد. در این صورت تابع مقدار true برمی‌گرداند.

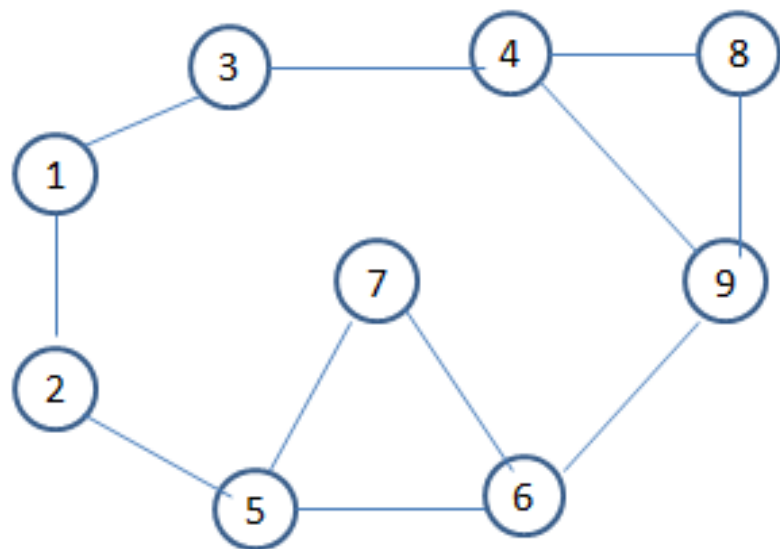
پیچیدگی زمانی این الگوریتم در بدترین حالت برابر $O(m^V)$ است. (m تعداد رنگ‌ها و V تعداد راس‌های گراف است.)

A screenshot of a code editor window titled 'FactoryManager.cs'. The code is in C# and defines a private static method 'IsValidColor'. The method takes four parameters: 'Graph graph', 'int vertex', 'int[] colors', and 'int color'. It returns a boolean. The logic is as follows: it iterates over the adjacent vertices of the given vertex using 'graph.adjList[vertex]'. For each adjacent vertex, it checks if the color of that vertex is equal to the color being tested. If it is, it returns false. If it completes the loop without finding a conflict, it returns true. The code is:

```
private static bool IsValidColor(Graph graph, int vertex, int[] colors, int color)
{
    // Check if any adjacent vertices have the same color
    foreach (int adjVertex in graph.adjList[vertex])
    {
        if (colors[adjVertex] == color)
        {
            return false;
        }
    }
    return true;
}
```

شکل ۳: تابع isValidColor

نمونه اجرای برنامه:

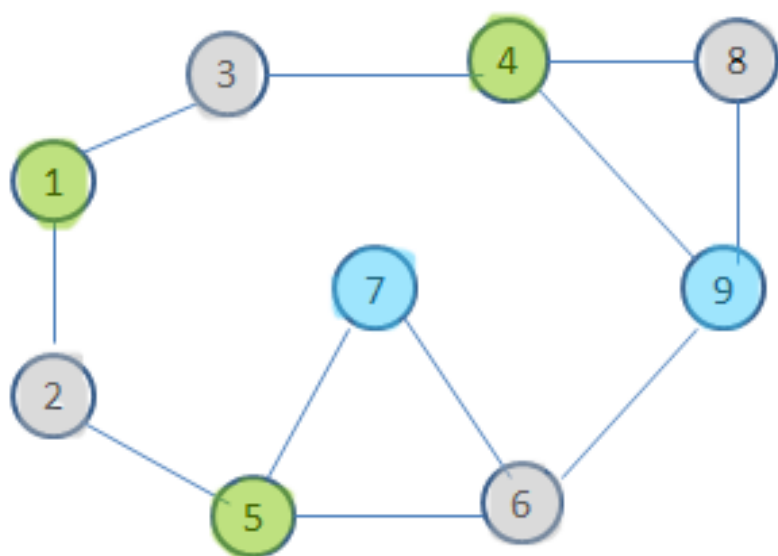


شکل ۴: گراف موردنظر برای رنگ‌آمیزی

```
---- Please Select Service Number: ----
[1] Design New Carpet
[2] Find Similar Carpets
[3] Buy Carpets
[4] Find Closest Branch
[5] Show All Carpets
[0] Exit

-> 1
Enter Number of Shapes:
9
Enter Connected Shapes in One Line: (Indexes Can Be Between 0 and 8)
[Enter -1 for End]
0 1
1 4
4 6
6 5
5 8
4 5
8 7
8 3
3 2
2 0
7 3
-1
Minimum Number of Required Colors is:3
Colored Shapes:
0 -> ■■
1 -> ■■
2 -> ■■
3 -> ■■
4 -> ■■
5 -> ■■
6 -> ■■
7 -> ■■
8 -> ■■
```

شکل ۵: نتیجه اجرای الگوریتم رنگ آمیزی



شکل ۶: گراف رنگ‌آمیزی شده

۲.۲ بخش دوم: بررسی فرش‌های مشابه

در این بخش برای پیدا کردن فرش‌های مشابه با یک فرش، از الگوریتم "هم‌ترازی دنباله‌ها" و کتابخانه‌های زبان پایتون استفاده می‌کنیم. با استفاده از کتابخانه Pillow تصویر فرش را در برنامه لود کرده و عرض آن‌ها را یکسان می‌کنیم. سپس برای هر پیکسل تصویر، مقدار RGB آن را به دست آورده و آن‌ها را در یک ماتریس ذخیره می‌کنیم.



```
import numpy as np
from PIL import Image
from numpy import ndarray

def photoToRGB(photo):
    return ndarray(photo)

def convertRGBToPhoto(numpydata, path):
    Image.fromarray(numpydata).save(path)

def modifySize(pho1, pho2, re):
    m = min(pho1.size[1], pho2.size[1]) // re
    pho1 = pho1.resize((pho1.size[0] // re, m), Image.Resampling.LANCZOS)
    pho2 = pho2.resize((pho2.size[0] // re, m), Image.Resampling.LANCZOS)
    return (pho1, pho2)

foo = Image.open('p1.jpg')
foo1 = Image.open('p2.jpg')

foo, foo1 = modifySize(foo, foo1, 5)

photoData1 = photoToRGB(foo)
photoData2 = photoToRGB(foo1)

photo1Sequence = list()
photo2Sequence = list()

for array2D in photoData1:
    photo1Sequence.append(array2D)

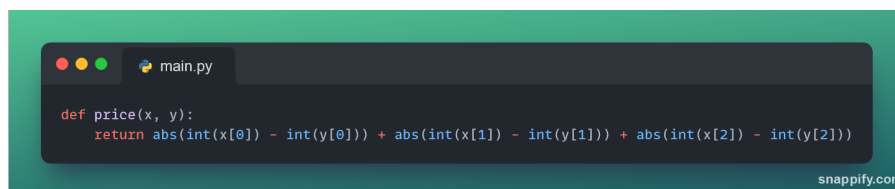
for array2D in photoData2:
    photo2Sequence.append(array2D)

print(len(photo1Sequence))
print(len(photo2Sequence))
```

snappify.com

شکل ۷: شروع برنامه و لود کردن تصویر فرش‌ها در برنامه

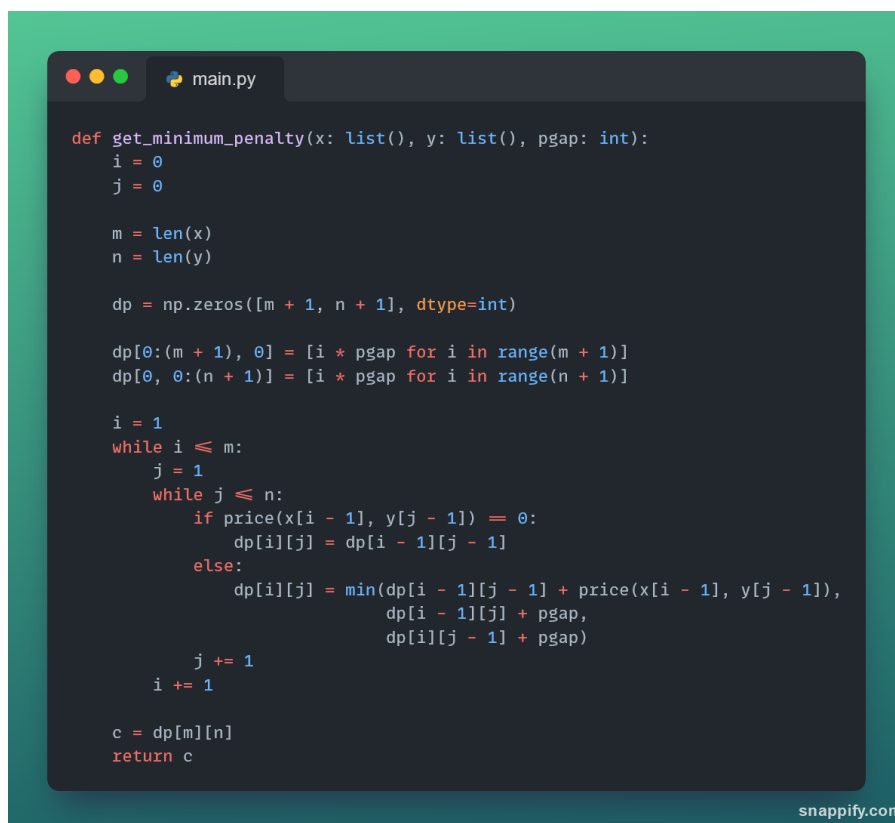
یا استفاده از الگوریتم هم‌ترازی دنباله‌ها، هر سطر از ماتریس طرح فرش را به تابع `get_minimum_penalty` می‌دهیم تا تصویر را به صورت افقی تراز کنیم و میزان شباهت را پیدا کنیم. در این الگوریتم نیاز به تعیین مقدار پناستی برای حالت برابر نبودن دو عنصر از دو آرایه داریم. ما این مقدار را برابر تفاوت مولفه‌های RGB هر دو آرایه در نظر می‌گیریم:

A screenshot of a code editor window titled 'main.py'. The code defines a function 'price(x, y)' that calculates the sum of absolute differences between corresponding elements of two lists x and y. The function signature is 'def price(x, y):' and the return statement is 'return abs(int(x[0]) - int(y[0])) + abs(int(x[1]) - int(y[1])) + abs(int(x[2]) - int(y[2]))'. The editor has a dark background with light-colored text. The URL 'snappify.com' is visible in the bottom right corner.

```
def price(x, y):  
    return abs(int(x[0]) - int(y[0])) + abs(int(x[1]) - int(y[1])) + abs(int(x[2]) - int(y[2]))
```

شکل ۸: تابع تعیین price

شیوه کار الگوریتم به این صورت است که با استفاده از رویکرد برنامه‌ریزی پویا، آرایه دوبعدی dp را برای ذخیره امتیاز هر دو عدد از دو آرایه ایجاد می‌کنیم. امتیاز هر دو عدد، با توجه به برابر بودن یا وجود فاصله‌ها تعیین می‌شود. اگر دو عدد برابر بودند چیزی به پناستی افزوده نمی‌شود، ولی در غیر این صورت، سه حالت با توجه به برابر نبودن دو عدد یا برابر نبودن طول آرایه‌ها و ایجاد فاصله (gap) ایجاد می‌شود که برای هر حالت یک مقدار مشخص پناستی به خانه مربوط به آن اضافه می‌کنیم. از بین آن‌ها کمترین مقدار را به عنوان مقدار خانه جدید dp در نظر می‌گیریم. نهایتاً خانه آخر این آرایه، کمترین مقدار پناستی‌های به دست آمده را به ما می‌دهد.

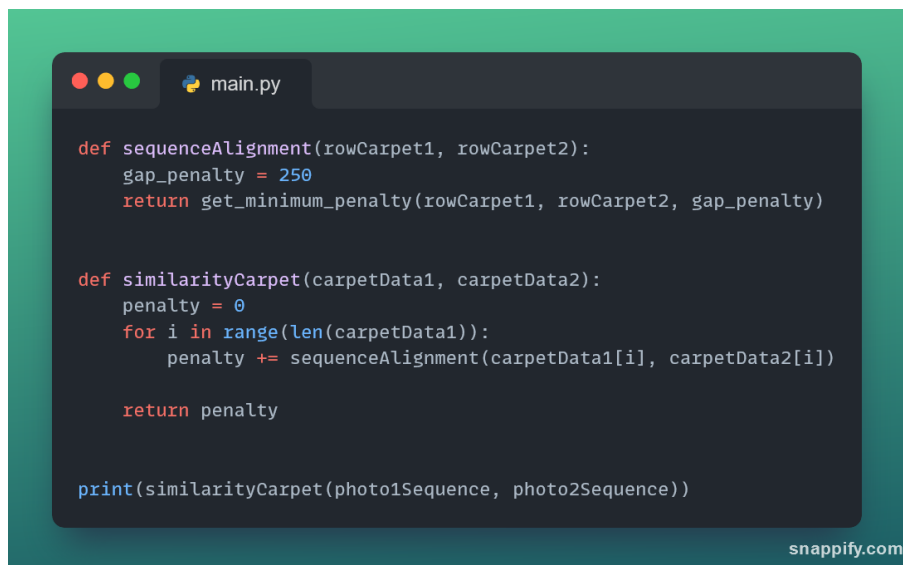
A screenshot of a code editor window titled 'main.py'. The code defines a function 'get_minimum_penalty(x: list(), y: list(), pgap: int):' which uses dynamic programming to find the minimum penalty. It initializes a 2D array 'dp' of size (m+1) x (n+1) with 'dtype=int'. The first row and first column are filled with values from 'pgap'. A nested loop iterates over indices i from 1 to m and j from 1 to n. For each (i, j), it checks if x[i-1] equals y[j-1]. If yes, dp[i][j] = dp[i-1][j-1]. If no, dp[i][j] = min(dp[i-1][j-1] + price(x[i-1], y[j-1]), dp[i-1][j] + pgap, dp[i][j-1] + pgap). Finally, it returns dp[m][n]. The editor has a dark background with light-colored text. The URL 'snappify.com' is visible in the bottom right corner.

```
def get_minimum_penalty(x: list(), y: list(), pgap: int):  
    i = 0  
    j = 0  
  
    m = len(x)  
    n = len(y)  
  
    dp = np.zeros([m + 1, n + 1], dtype=int)  
  
    dp[0:(m + 1), 0] = [i * pgap for i in range(m + 1)]  
    dp[0, 0:(n + 1)] = [i * pgap for i in range(n + 1)]  
  
    i = 1  
    while i <= m:  
        j = 1  
        while j <= n:  
            if price(x[i - 1], y[j - 1]) == 0:  
                dp[i][j] = dp[i - 1][j - 1]  
            else:  
                dp[i][j] = min(dp[i - 1][j - 1] + price(x[i - 1], y[j - 1]),  
                               dp[i - 1][j] + pgap,  
                               dp[i][j - 1] + pgap)  
            j += 1  
        i += 1  
  
    c = dp[m][n]  
    return c
```

شکل ۹: تابع get_minimum_penalty

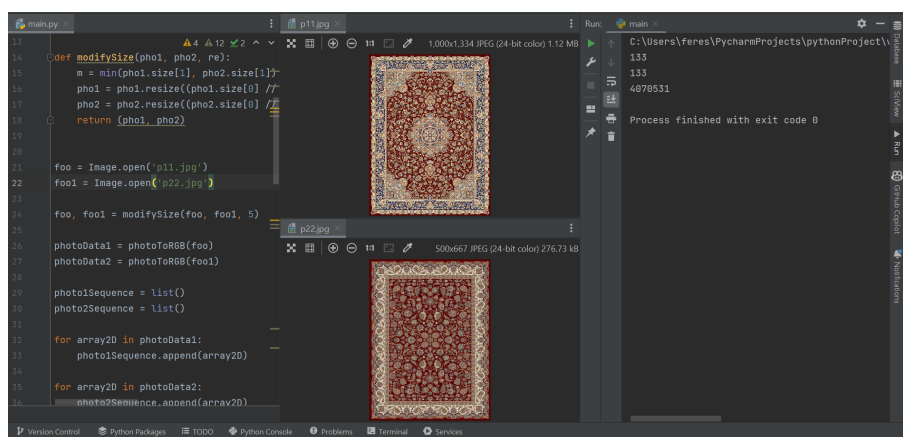
چون فرش‌های ما آرایه‌های دوبعدی هستند، برای همه سطرهای ماتریس طرح فرش، این متد را (با مقدار دلخواه ۲۵۰ برای پناستی (gap صدا زده و نتیجه آن‌ها را با هم جمع می‌کنیم. عدد حاصل، کمترین

پنالتی برای مقایسه فرش موردنظر و فرش دیگر است. با به دست آوردن این مقدار برای همه فرش‌ها، عددی که کمترین مقدار و در نتیجه بیشترین تشابه را با فرش موردنظر دارد، شبیه‌ترین فرش به آن است. پیچیدگی زمانی این الگوریتم در بدترین حالت برابر $O(m * n)$ است.



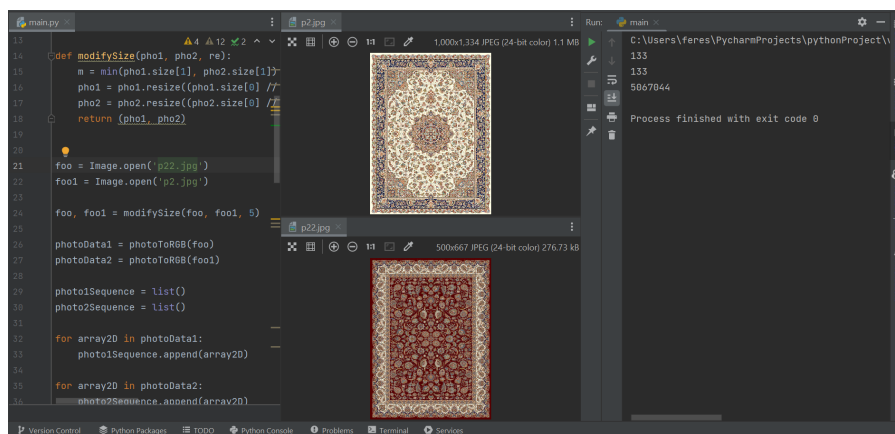
شکل ۱۰: فراخوانی sequenceAlignment برای هر سطر از ماتریس فرش‌ها

نمونه اجرای برنامه برای دو فرش مشابه:



شکل ۱۱: اجرای دو فرش مشابه : 4.070.531

نمونه اجرای برنامه برای دو فرش با شباهت کمتر:




شکل ۱۲: اجرای دو فرش با شباهت کمتر : 5.067.044

همان‌طور که مشاهده می‌کنیم، مقدار به دست آمده در اجرای اول که دو فرش مشابه‌تر هستند، کمتر از اجرای دوم (با دو فرش با شباهت کمتر) می‌باشد.
 (با توجه به طولانی بودن اجرای الگوریتم، حلقه برای مقایسه همه فرش‌ها برداشته شد و تنها دو تصویر باهم مقایسه می‌شوند.)

۳.۲ بخش سوم: خرید بر اساس میزان پول

در این بخش می‌خواهیم با داشتن میزان هزینه‌ای که کاربر می‌تواند پرداخت کند، حداکثر تعداد فرش‌هایی که می‌تواند بخرد را به او معرفی کنیم.

برای این کار از الگوریتم کوله‌پشتی و رویکرد برنامه‌ریزی پویا بهره می‌بریم. این تابع بر اساس ظرفیت کوله‌پشتی که همان میزان پول قابل پرداخت توسط کاربر است نوشته شده است. وظیفه این تابع این است که فرش‌هایی را به کاربر برای خرید پیشنهاد دهد که جمع قیمت آن‌ها برابر با پول قابل پرداخت توسط کاربر باشد؛ با این شرط که فرش‌ها را با بیشترین ارزش در اولویت قرار دهد. ابتدا توسط تابع `FillCarpetValue` فرش‌ها را بر اساس قیمت آن‌ها مرتب‌سازی می‌کنیم. در نتیجه این تابع، فرش‌ها با قیمت کمتر بیشترین ارزش را دارا می‌شود.



```
private static void FillCarpetValue()
{
    int val = (allCarpets.Count + 1);
    allCarpets.OrderBy(x => x.Price).ToList().ForEach(x => x.Value = --val);
}
```

شکل ۱۳: تابع `FillCarpetValue`

این تابع با دریافت میزان کل پول طرف به عنوان گنجایش کوله‌پشتی، قیمت هر فرش، ارزش هر فرش و تعداد کل فرش‌های موجود به عنوان پارامتر ورودی، به محاسبه تعداد فرش‌های قابل پرداخت برای فرد می‌پردازد. نهایتاً تعداد فرش‌ها و هم فرش‌های انتخاب شده را به کاربر نشان می‌دهد. پیچیدگی زمانی این الگوریتم در بدترین حالت برابر $O(capacity * itemCount)$ است.



شکل ۱۴: تابع Knapstack

این تابع تمامی فرش‌ها را تک‌به‌تک چک می‌کند و سپس آرایه دوبعدی K را با محاسبه ماکسیمم مقدار بین فرش‌ها در مقایسه با بقیه، پر می‌کند.

۴.۲ بخش چهارم: مسیریابی به نزدیک‌ترین شعبه

از آن جایی که کارخانه شعبات زیادی دارد، این سامانه دارای بخشی است که کاربر می‌تواند با وارد کردن مختصات خود، نزدیک‌ترین شعبه به خود را بیابد و مسیر رفتن به آن نقطه را پیدا کند. شهر فرضی ما شامل چهارراه‌هایی است که به یکدیگر متصل هستند. این نقاط به همراه خیابان‌های بین آن‌ها از قبل به سیستم معرفی می‌شوند.

برای پیاده‌سازی این بخش، از الگوریتم فلوید-وارشال و رویکرد برنامه‌ریزی پویا استفاده می‌کنیم. در این الگوریتم، دو آرایه دوبعدی distance و Next داریم که در اولی، کوتاه‌ترین فاصله بین دو راس از گراف و در دومی، کوتاه‌ترین مسیر برای رسیدن از یک راس به راس دیگر را ذخیره می‌کنیم. سپس این دو ماتریس را مقداردهی اولیه می‌کنیم.

در شروع الگوریتم، هر دو راس i و j از گراف را در نظر می‌گیریم. اگر بین آن‌ها راسی مانند k وجود داشته باشد، به صورتی که مجموع فاصله i تا k و فاصله k تا j کمتر از مسیر مستقیم از i به j باشد، آن را در خانه i, j از آرایه ذخیره می‌کنیم و در ماتریس Next هم آن را به مسیر اضافه می‌کنیم. در غیر این صورت تغییر خاصی نمی‌کند و همان مقدار قبلی (فاصله مستقیم از i به j) می‌ماند. به همین صورت پیش رفته و ماتریس distance و Next را پر می‌کنیم.



```
//-----Floyd-Warshal
private const int INF = 10000;

private static void FloydWarshall(int[,] graph, int verticesCount)
{
    distance = new int[verticesCount, verticesCount];
    Next = new int[verticesCount, verticesCount];
    FactoryManager.verticesCount = verticesCount;

    for (int i = 0; i < verticesCount; i++)
    {
        for (int j = 0; j < verticesCount; j++)
        {
            // No edge between node i and j
            if (graph[i, j] == INF)
                Next[i, j] = -1;
            else
                Next[i, j] = j;

            distance[i, j] = graph[i, j];
        }
    }

    for (int k = 0; k < verticesCount; k++)
    {
        for (int i = 0; i < verticesCount; i++)
        {
            for (int j = 0; j < verticesCount; j++)
            {
                if (distance[i, k] + distance[k, j] < distance[i, j])
                {
                    distance[i, j] = distance[i, k] + distance[k, j];
                    Next[i, j] = Next[i, k];
                }
            }
        }
    }
}
```

snappify.com

شکل ۱۵: تابع FloydWarshal

سپس در تابع ConstructPath راس‌های موجود در مسیر بین دو راس u و v را به لیستی اضافه کرده و آن را ریترن می‌کنیم.



شکل ۱۶: تابع ConstructPath

در نهایت، با استفاده از تابع `GetClosestFactoryVertex` مختصات کاربر را گرفته و راسی که در آن قرار دارد را پیدا می‌کنیم. سپس در ماتریس `distance` و در سطر مربوط به آن راس، جستجو کرده و نزدیک‌ترین راس به آن را پیدا می‌کنیم. سپس با تابع `ConstructPath`، مسیر بین آن دو را پیدا کرده و به صورت یک لیست برمی‌گرداند.

پیچیدگی زمانی این الگوریتم در بدترین حالت برابر $O(V^3)$ است. (V تعداد راس‌های گراف است).

```

FactoryManager.cs

public static List<int> GetClosestFactoryVertex(int x, int y)
{
    FloydWarshall(cityGraph.adjMatrix, cityGraph.Size);

    int[] personLocCloseVertices = new int [verticesCount];
    var personLoc = cityGraph.Vertices.Where(c => c.x == x && c.y == y).FirstOrDefault();
    var index = cityGraph.Vertices.IndexOf(personLoc); //u
    for (int i = 0; i < verticesCount; i++)
    {
        personLocCloseVertices[i] = distance[index, i];
    }

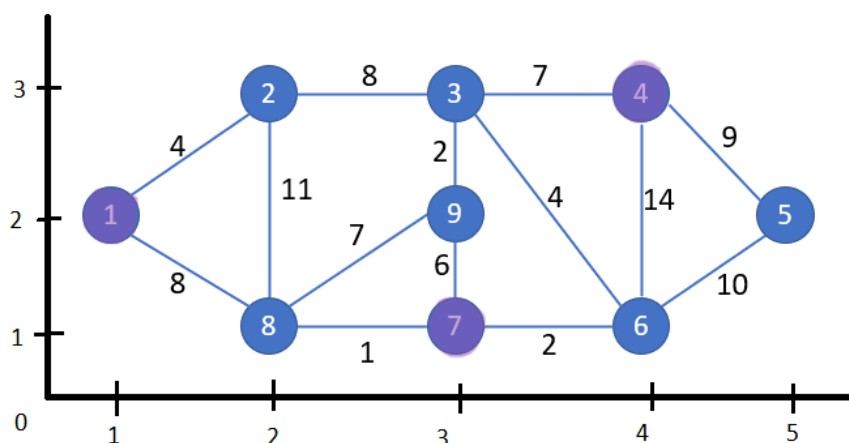
    var carpetBranchIndex = -1;
    do
    {
        var min = personLocCloseVertices.ToList().Min();
        var index2 = personLocCloseVertices.ToList().IndexOf(min); //index of closest vertex
        var v = cityGraph.Vertices[index2];
        carpetBranchIndex = v.isCarpetBranch ? index2 : -1;
        // if (carpetBranchIndex == -1) personLocCloseVertices.ToList().Remove(min);
        if (carpetBranchIndex == -1) personLocCloseVertices[index2] = Int32.MaxValue;
    } while (carpetBranchIndex == -1);

    return ConstructPath(index, carpetBranchIndex);
}

```

شکل ۱۷: تابع GetClosestFactoryVertex

نمونه اجرای برنامه:



شکل ۱۸: نقشه شهر و خیابان‌ها

```
---- Please Select Service Number: ----  
[1] Design New Carpet  
[2] Find Similar Carpets  
[3] Buy Carpets  
[4] Find Closest Branch  
[5] Show All Carpets  
[0] Exit  
-> 4  
Enter Your Location:  
3  
3  
You Are in Intersection 2  
The Closest Branch to Your Location is: 6  
You Can Follow This Path to Get to The Closest Branch:  
3 -> 6 -> 7
```

شکل ۱۹: اجرای برنامه برای پیدا کردن کوتاه‌ترین مسیر

همین‌طور که می‌بینیم، ما در موقعیت (۳،۳) قرار داریم که راس ۳ در نقشه شهر است. نزدیک‌ترین شعبه به موقعیت کاربر، راس ۷ است که از مسیر راس ۶ می‌گذرد.