طراحی الگوریتم

گزارش فنی پروژه‌ی کارخانه‌ی فرشبافی

دکتر پیمان ادیبی

فرناز موحدی
۴۰۰۳۶۱۳۰۵۷
رضا چراخ
۴۰۰۳۶۱۳۰۱۸

خرداد ۱۴۰۲

توضیح کلی پروژه و اهداف آن:

این پروژه شامل دو بخش بوده است که بخش اول آن طراحی و بخش دوم آن فروش فرش است. این دو بخش کاملا مجزا از هم هستند و پیاده‌سازی های متفاوتی نیز دارند. در ادامه همه‌ی قسمت های پیاده‌سازی شده توضیح داده خواهند شد.
هدف از این پروژه به کار بردن الگوریتم های بررسی شده در طول ترم با استفاده از روش های مختلف طراحی الگوریتم است.

۱) طراحی

۱-۱) طراحی فرش‌های جدید

خواسته‌ی مسئله: فرش‌ها به‌گونه‌ای رنگ شوند که نواحی همسایه همرنگ نباشند و در عین حال کمترین تعداد رنگ لازم برای رنگ کردن آنها استفاده شود. در انتها رنگ انتساب داده شده به هر یک از نواحی هم باید نوشته شود.

روش کلی حل: هر ناحیه از فرش به یک راس از یک گراف نسبت داده شده است و همسایه بودن این نواحی هم با یال های گراف نشان داده می شود. با این مدل‌سازی می توان از الگوریتم رنگ آمیزی گراف استفاده کرد.

شرح گام‌به‌گام حل مسئله به همراه کد: در تابع graphColoring() آرایه ای وجود دارد که به هر عنصر از آن، رنگ راس با اندیس i ام نسبت داده می شود. رنگ ها عدد هستند و از ۰ شروع میشوند.

```
void graphColoring(int [][]graph, int numOfVertices)
{
    color = new int[numOfVertices];
    for (int i = 0; i < numOfVertices; i++)
        color[i] = 0;
    graphColoringUtil(graph, color, currentVertex: 0, numOfVertices);
    printSolution(color, numOfVertices);
}
```

بخش اصلی الگوریتم در تابع graphColoringUtil() انجام می شود و بر پایه ی روش عقب گرد است. شرط ابتدایی این تابع هنگامی اجرا می شود که به همه ی راس ها یک رنگ نسبت داده شده باشد. در ادامه از تابع isSafe() برای بررسی همسایه بودن یا نبودن دو راس استفاده می شود و در صورت همسایه نبودن رنگ قبلی به این راس هم نسبت داده میشود. اگر دو راس همسایه باشند، درخت روش عقب گرد از آن برگ ادامه نمی یابد و درخت مربوط به آن هرس می شود. همان طور که واضح است این کار تا وقتی که شرط اولیه برقرار باشد، برای همه ی راس ها انجام میشود.

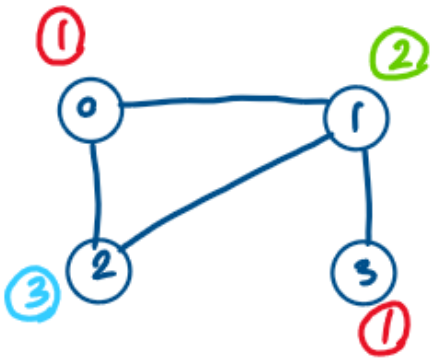
```
boolean isSafe(int currentVertex, int [][]graph, int []color, int currentColor, int numOfVertices)
{
    for (int i = 0; i < numOfVertices; i++)
        if (graph[currentVertex][i] == 1 && currentColor == color[i])
            return false;
    return true;
}
2 usages
int graphColoringUtil(int [][]graph, int []color, int currentVertex, int numOfVertices)
{
    if (currentVertex == numOfVertices)
        return 0;
    for (int c = 1; c <= numOfVertices; c++) {
        if (isSafe(currentVertex, graph, color, c, numOfVertices)) {
            color[currentVertex] = c;
            if (graphColoringUtil(graph, color, currentVertex: currentVertex + 1, numOfVertices) == 0)
                return 0;
            color[currentVertex] = 0;
        }
    }
    return -1;
}
```

تحلیل زمانی کد: با توجه به ترکیبات مختلفی که در طول اجرای تابع از رنگ ها ساخته می شود، از مرتبه ی O(c*v) خواهد بود که c تعداد رنگ ها و v تعداد راس های گراف است.
تحلیل حافظه ای کد: در تابع graphColoring() به علت بازگشتی بودن و استفاده از stack از مرتبه ی O(v) خواهد بود که v تعداد راس های گراف است.

نمونه ورودی و خروجی آن:

```
          ***          WELCOME TO OUR CARPET COMPANY!          ***
                       YOU CAN DESIGN NEW CARPETS OR BUY SOME!
                       1 --> DESIGN A CARPET
                       2 --> BUY A CARPET
                       3 --> EXIT
1
          ***          DESIGNING A CARPET          ***
                       Enter the number of regions on your carpet:
                       This is the format --> 5 --> number of regions = 5
4
                       Enter the number of intersections between regions on your carpet:
                       This is the format --> 5 --> number of intersection = 5
4
                       Enter the regions that are in the neighborhood together:
                       This is the format --> 0 3 --> 0 is in the neighborhood of 3
0 1
0 2
1 2
1 3
The minimum colors that you can color your carpet with them is:
 1  2  3  1

Process finished with exit code 0
```

گراف ورودی به صورت زیر است و مشاهده می شود که تعداد رنگ ها و انتساب آنها درست انجام شده است:



۲) فروش

۲-۱) جست‌وجو بر اساس طرح نقشه

خواسته‌ی مسئله: در این قسمت باید با دریافت طرح یک فرش توسط کاربر، فرش‌های موجود با بیشترین تشابه را به کاربر معرفی کنیم.

روش کلی حل: فرش‌ها که در یک آرایه دوبعدی قرار دارند را وارد آرایه یک‌بعدی می‌کنیم. سپس با الگوریتم هم‌ترازی دنباله‌ها هر دو آرایه را هم‌تراز می‌کنیم و سپس با یکدیگر مقایسه می‌کنیم.(فرش وارد شده توسط کاربر با تمام فرش‌های موجود مقایسه می‌شود)

شرح گام‌به‌گام حل مسئله به همراه کد: ابتدا آرایه دوبعدی فرش که کاربر وارد کرده را به آرایه یک بعدی تبدیل می‌کنیم. سپس در یک حلقه فور آن را با تمام فرش‌های موجود چک می‌کنیم. در این حلقه ابتدا فرش ـاام به یک آرایه یک‌بعدی تبدیل می‌شود. سپس یک ماتریس (آرایه دوبعدی) با ابعاد [n+1][m+1] که m در آن طول آرایه یک‌بعدی بدست آمده از فرش وارد شده کاربر است و n طول آرایه یک‌بعدی در لیست فرش‌ها.

```
static ArrayList<Integer> similarityPercentage = new ArrayList();
1 usage
static void carpetSuggestion(int[][] basePattern, int rows, int columns) {

    int[] oneDimentionalPattern = new int[rows * columns];
    int i, j, k = 0;
    for (i = 0; i < rows; i++) {
        for (j = 0; j < columns; j++) {
            k = i * columns + j;
            oneDimentionalPattern[k] = basePattern[i][j];
            k++;
        }
    }
    for (int l = 0; l < Carpet.carpets.size(); l++) {
        int z = 0;
        int [][] patternToCompare = Carpet.carpets.get(l).pattern;
        int[] oneDtoCompare = new int[patternToCompare.length * patternToCompare[0].length];
        for (int m = 0; m < patternToCompare.length; m++) {
            for (int n = 0; n < patternToCompare[m].length; n++) {
                z = m * patternToCompare[0].length + n;
                oneDtoCompare[z] = patternToCompare[m][n];
                z++;
            }
        }
        int[][] opt = new int[oneDtoCompare.length + 1][oneDimentionalPattern.length + 1];
```

سپس در حلقه فور اول ستون آخر و در فور دوم سطر آخر را بر اساس الگوریتم هم‌ترازی دنباله‌ها پر می کنیم.

```
for (int m = 0; m <= oneDtoCompare.length; m++) {
    opt[m][oneDimentionalPattern.length] = 2 * (oneDtoCompare.length - m);
}
for (int m = 0; m <= oneDimentionalPattern.length; m++) {
    opt[oneDtoCompare.length][m] = 2 * (oneDimentionalPattern.length - m);
}
```

در ادامه برای پر کردن بقیه ماتریس از این سطر و ستون استفاده می‌کنیم و در دو حلقه فور یکی یکی مینیمم گزینه های زیر را برای درج عنصر i , j انتخاب می‌کنیم.
و مقادیر بدست آماده را در یک arrayList اضافه می‌کنیم.

$$opt(i,j) = \min(opt(i+1,j+1) + penalty, opt(i+1,j) + 2, opt(i,j+1) + 2).$$

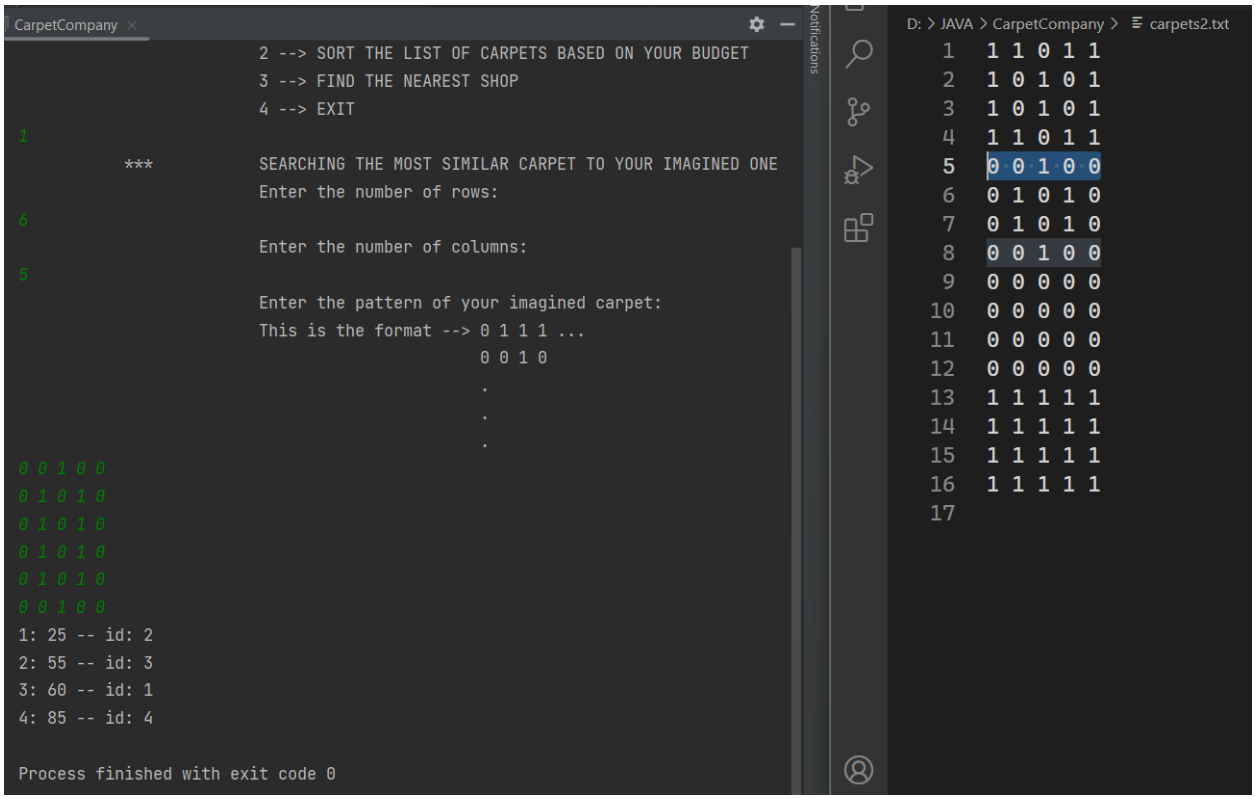
```
for (int m = oneDtoCompare.length - 1; m >= 0 ; m--) {
    for (int n = oneDimentionalPattern.length - 1; n >= 0 ; n--) {
        int penalty = 0;
        if (oneDtoCompare[m] != oneDimentionalPattern[n])
            penalty = 5;
        opt[m][n] = Math.min(Math.min( opt[m + 1][n + 1] + penalty , opt[m + 1][n] + 7 ), opt[m][n + 1] + 7);
    }
}
similarityPercentage.add(opt[0][0]);
//System.out.println(similarityPercentage.get(l));
}
quickSort(similarityPercentage, low: 0, high: similarityPercentage.size() - 1);
```

سپس الگوریتم Quick Sort را برای آن ArrayList فراخوانی میکنیم.(این الگوریتم در بخش ۲-۲ شرح داده شده است.)

تحلیل زمانی کد: با توجه به ترکیبات مختلفی که در طول اجرای تابع ساخته می شود و بدست آوردن کمینه ی آنها در هر حالت، از مرتبه ی O(n+1*m+1) خواهد بود که m در آن طول آرایه یک‌بعدی بدست آمده از فرش وارد شده کاربر است و n طول آرایه یک‌بعدی در لیست فرش‌ها.
تحلیل حافظه ای کد: با همان تحلیل زمانی میتوان ادعا کرد که از مرتبه ی O(n+1*m+1) خواهد بود.

نمونه ورودی و خروجی آن:

برای چک کردن درست بودن کد، فایل با نمونه های ۵ * ۴ نوشته شده است و ورودی ای با سطر و ستون ۶ * ۳ به آن داده ایم:



در قسمت سمت راست فایل مشاهده می‌شود که شامل چهار فرش است.
به ترتیب با ID های ۱ و ۲ و ۳ و ۴.
بعد از مقایسه ی فرش ها با ورودی کاربر خروجی میزان penalty است که مرتب شده است.
هر چقدر penalty کمتر باشد، فرش مورد نظر به فرش وارد شده ی کاربر شبیه تر است.
بنابراین شبیه ترین فرش ها به ترتیب: فرش ۲ و ۳ و ۱ و ۴ هستند.

۲-۲) خرید بر اساس میزان پول

خواسته‌ی مسئله: فرش هایی که در قسمت ۱-۲ به سیستم معرفی شده‌اند، در این قسمت هم به کار می آیند. به این صورت که این فرش ها دارای قیمت مشخصی هستند و کاربر با وارد کردن حداکثر میزان پول خود، بیشترین تعداد فرشی که می تواند بخرد را مشاهده می‌کند.

روش کلی حل: فرش ها بر اساس قیمت، به صورت صعودی در لیستی مرتب می شوند. این مرتب سازی برای اطمینان از سریع بودن آن توسط Quick sort با روش تقسیم و حل انجام میشود. سپس از ابتدای این لیست تا جایی که بودجه ی کاربر اجازه می دهد فرش‌ها در لیست فرش‌های خریداری شده قرار می گیرند.

نکته: در راهنمایی این بخش آمده است که می توان از الگوریتم کوله پشتی استفاده کرد. اما واقعیت امر این است که در الگوریتم کوله پشتی تعداد اشیا مهم نیست و همچنین ویژگی ای وجود دارد علاوه بر وزن که بتوان بر اساس آن شی را انتخاب کرد.(مثلا سود نهایی) اما در این قسمت ما فرش هایی داریم که ویژگی دیگری غیر از قیمت(که در مدل سازی با مسئله ی کوله پشتی همان وزن اشیا می شود) وجود ندارد و البته تعداد اشیا هم مهم است. با توجه به این توضیحات، استفاده از الگوریتم Quicksort و انتخاب اشیا با استفاده از آن راه بهتری است.

شرح گام‌به‌گام حل مسئله به همراه کد: تابع ()partition آخرین عنصر را به عنوان pivot در نظر می گیرد. سپس pivot را در محل مناسب آن قرار می دهد. به نحوی که همه ی عناصر سمت چپ از آن کوچکتر و همه ی عناصر سمت راست از آن بزرگتر باشند. به این صورت که اندیس های i و j به ترتیب برای اعداد بزرگتر و کوچکتر از pivot هستند و در هر مرحله جای عنصر این اندیسها با هم عوض می شود. نیاز است که هم قیمت ها در لیست arr و هم لیست مربوط به فرش ها هم زمان مرتب شوند.

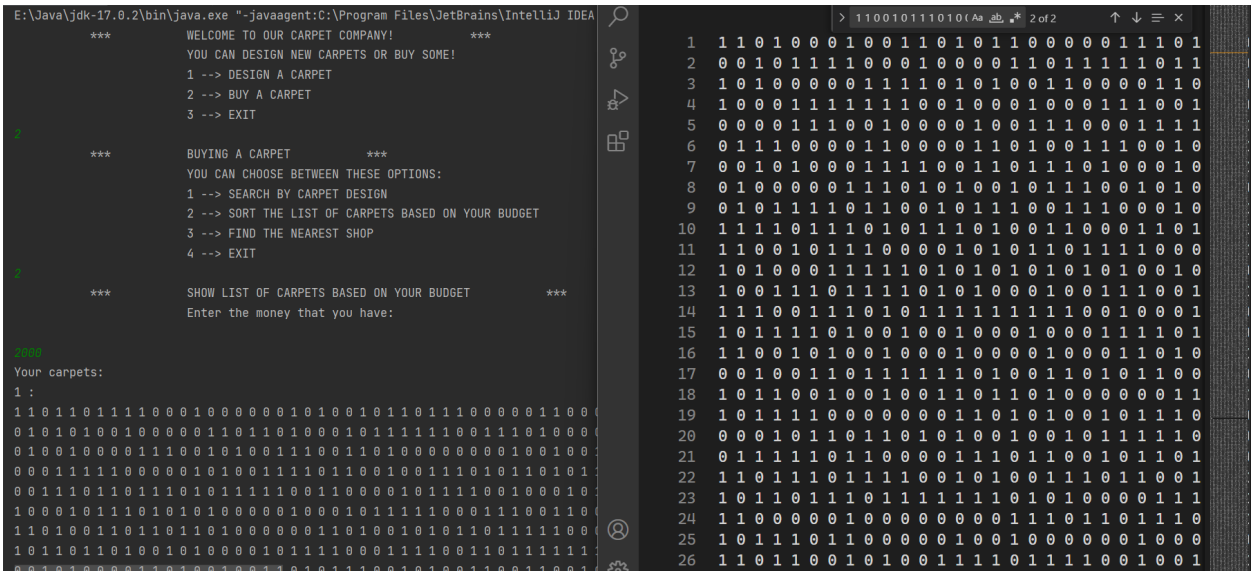
```
static int partition(ArrayList<Integer> arr, int low, int high , ArrayList<ArrayList<Integer>> carpets) {
    int pivot = arr.get(high);

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr.get(j) < pivot) {
            i++;
            swap(arr, i, j);
            ArrayList<Integer> temp = carpets.get(i);
            carpets.set(i, carpets.get(j));
            carpets.set(j, temp);
        }
    }
    swap(arr,  i: i + 1, high);
    ArrayList<Integer> temp = carpets.get(i+1);
    carpets.set(i+1, carpets.get(high));
    carpets.set(high, temp);
    return (i + 1);
}
```

در تابع quickSort() هم زیر مجموعه های چپ و راست هم به صورت بازگشتی مرتب می شوند.

```
static void quickSort(ArrayList<Integer> arr, int low, int high, ArrayList<ArrayList<Integer>> carpets) {
    if (low < high) {
        int pi = partition(arr, low, high, carpets);
        quickSort(arr, low,  high: pi - 1, carpets);
        quickSort(arr,  low: pi + 1, high, carpets);
    }
}
```

تحلیل زمانی کد: از آنجایی که مقادیر آرایه وابسته است تحلیل حالت متوسط را انجام می دهیم که از مرتبه ی O(n*logn) خواهد بود. البته باز هم در بدترین حالت ممکن است که به O(n^2) برسد و از الگوریتم هایی مثل merge sort بدتر بنظر برسد اما در عمل می بینیم که در نمونه های زیاد بهتر عمل میکند.
تحلیل حافظه ای کد:

نمونه ورودی و خروجی آن:




```
price: 507

2 :
```

۳-۲) مسیریابی به نزدیک‌ترین فروشگاه کارخانه

خواسته‌ی مسئله: شهر فرضی ای وجود دارد که شامل چهارراه‌هایی است. کاربر با وارد کردن تقاطع یا چهارراهی که در آن حضور دارد، نزدیک ترین تقاطع هایی را می یابد که در آنها شعبه های فروشگاه وجود دارند.

روش کلی حل: شهر فرضی ما شامل ۱۰ تقاطع است که این تقاطع ها راس های گراف و چهارراه های آن یال های گراف را تشکیل می دهند. همچنین فاصله ی بین این تقاطع ها با وزن یال های گراف نشان داده می شود. در سه تقاطع از این شهر به شماره‌ی ۳ و ۵ و ۹ شعبه های فروشگاه وجود دارند. این شهر توسط یک ماتریس ۱۰ در ۱۰ در فایل نوشته شده و از قبل به سیستم معرفی می شود.

شرح گام‌به‌گام حل مسئله به همراه کد: در ابتدای تابع dijkstra() آرایه ای تعریف می‌شود که قرار است در آن کوتاه ترین مسیر بین گره ی مبدا تا دیگر گره ها ذخیره شود. در ادامه آرایه ی boolean ای تعریف می شود که در صورت اتمام پیدا کردن کوتاه ترین مسیر بین مبدا و یک راس مقدار آن عنصر true می شود. در ابتدا همه ی مقادیر بزرگترین مقدار ممکن هستند. فاصله ی یک گره تا خودش هم همیشه ۰ است. آرایه ی parents[ ] نگه دارنده ی درخت کوتاه ترین مسیر است.

```
static void dijkstra(int[][] adjacencyMatrix,
                     int startVertex)
{
    int nVertices = adjacencyMatrix[0].length;
    int[] shortestDistances = new int[nVertices];

    boolean[] added = new boolean[nVertices];

    for (int vertexIndex = 0; vertexIndex < nVertices;
         vertexIndex++)
    {
        shortestDistances[vertexIndex] = Integer.MAX_VALUE;
        added[vertexIndex] = false;
    }

    shortestDistances[startVertex] = 0;

    int[] parents = new int[nVertices];

    parents[startVertex] = NO_PARENT;
```

هر گره در ابتدا بیشترین فاصله را دارد و اگر فاصله ی کمتری پیدا شد، مقدار آن در آرایه ویرایش می شود. در ادامه برای همسایه های گره ی مورد نظر هم مقدار فاصله را ویرایش میکند.

```
for (int i = 1; i < nVertices; i++)
{
    int nearestVertex = -1;
    int shortestDistance = Integer.MAX_VALUE;
    for (int vertexIndex = 0;
         vertexIndex < nVertices;
         vertexIndex++)
    {
        if (!added[vertexIndex] &&
                shortestDistances[vertexIndex] <
                        shortestDistance)
        {
            nearestVertex = vertexIndex;
            shortestDistance = shortestDistances[vertexIndex];
        }
    }
    added[nearestVertex] = true;
    for (int vertexIndex = 0;
         vertexIndex < nVertices;
         vertexIndex++)
    {
        int edgeDistance = adjacencyMatrix[nearestVertex][vertexIndex];

        if (edgeDistance > 0
                && ((shortestDistance + edgeDistance) <
                shortestDistances[vertexIndex]))
        {
            parents[vertexIndex] = nearestVertex;
            shortestDistances[vertexIndex] = shortestDistance +
                    edgeDistance;
```

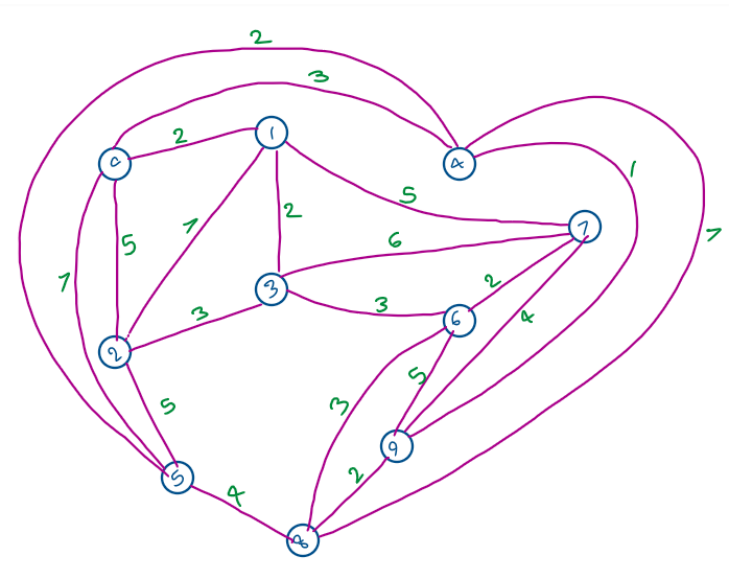
تحلیل زمانی کد:از مرتبه ی O(v^2) است که v تعداد راس های گراف است.

تحلیل حافظه ای کد: O(v^2) است که v تعداد راس های گراف است.

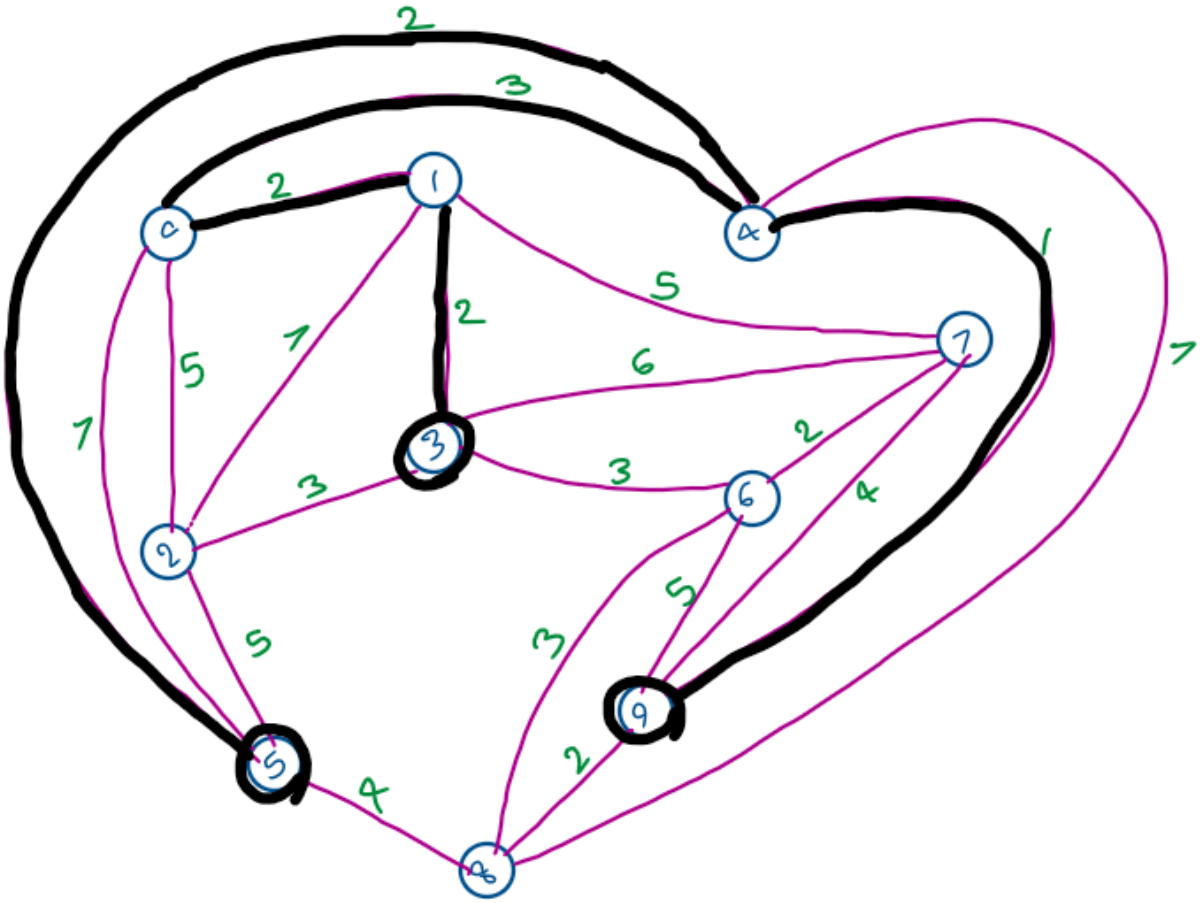
نمونه ورودی و خروجی آن:

```
                    3 --> FIND THE NEAREST SHOP
                    4 --> EXIT
3
          ***       FINDING THE NEAREST SHOP TO YOU          ***
                    As you know there are 10 intersections in our town.
                    Enter the intersection that you are in it in rage (0 - 9):
                    Our shops are in intersections : 3 , 5 , 9!
4
0 2 5 0 3 7 0 0 0 0
2 0 7 2 0 0 0 5 0 0
5 7 0 3 0 5 0 0 0 0
0 2 3 0 0 0 3 6 0 0
3 0 0 0 0 2 0 0 7 1
7 0 5 0 2 0 0 0 4 0
0 0 0 3 0 0 0 2 3 5
0 5 0 6 0 0 2 0 0 4
0 0 0 0 7 4 3 0 0 2
0 0 0 0 1 0 5 4 2 0

Vertex   Distance   Path
4 -> 3       7      4 0 1 3
4 -> 5       2      4 5
4 -> 9       1      4 9                    You can see the nearest one with the distance and the intersections that you have to go!

                    NOW YOU CAN CHOOSE BETWEEN THESE OPTIONS:
                    1 --> BACK TO MAIN MENU
                    2 --> EXIT
```

شهر فرضی ما:



مسیرها از راس ۴ به شعبه ها:


a
1
4
7
3
6
2
9
5
8