به نام خدای رنگین کمان

مستند پروژه درس طراحی الگوریتم‌ها

استاد: دکتر ادیبی

نویسندگان:

ارشیا شفیعی (۴۰۰۳۶۲۳۰۱۹)، امیرعلی لطفی (۴۰۰۳۶۱۳۰۵۳)

طراحی فرش‌های جدید

در این مسئله قصد داریم با گرفتن نقشه فرش(ماتریس مجاورت) ناحیه‌های آن را رنگ‌آمیزی کنیم به طوری که کمترین تعداد رنگ استفاده شود.

ورودی: ماتریس مجاورت

خروجی: حداقل تعداد و رنگ‌های انتساب شده به هر راس

الگوریتم استفاده شده در این قسمت، الگوریتم m-coloring است. این الگوریتم با رهیافت backtracking سعی دارد با m رنگ یک گراف را رنگ کند به طوری که هیچ دو گره مجاوری با هم هم‌رنگ نباشند.

یک ArrayList به نام allColorings تعریف شده است که تمامی حالت‌های رنگ آمیزی گراف داده شده را ذخیره می‌کند.

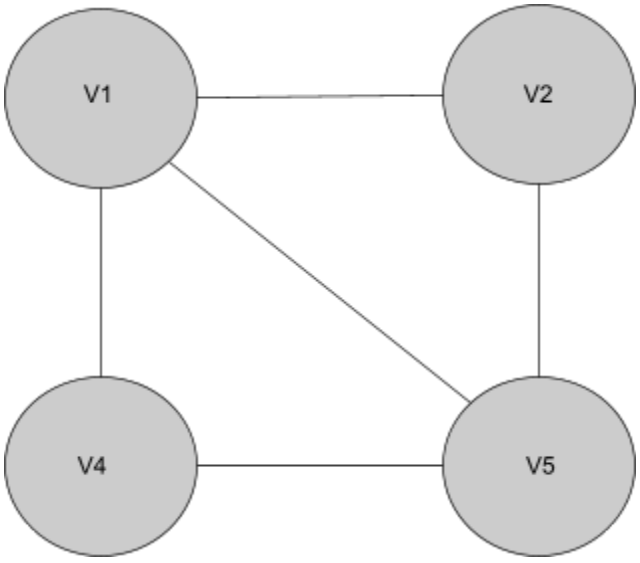
این تابع به صورت بازگشتی، هر بار برای یک گره صدا زده می‌شود. به ازای هر گره تمامی رنگ‌ها به ترتیب در آرایه coloring قرار می‌گیرند. در هرفراخوانی رنگ‌آمیزی مورد قبول است که هیچ دو گره مجاور هم یک رنگ نداشته باشند. وظیفه بررسی این شرط، بر عهده تابع promising است.

```
private void mColoring(int vertexIndex) {
    if (promising(vertexIndex, coloring)) {
        if (vertexIndex == n - 1) {
            int[] copy_coloring = coloring.clone();
            allMColorings.add(copy_coloring);
        } else {
            for (int color = 0; color < mColors; color++) {
                coloring[vertexIndex + 1] = color;
                mColoring( vertexIndex: vertexIndex + 1);
            }
        }
    }
}
```

تابع promising با در دست داشتن مقدار گره فعلی و رنگ‌آمیزی تا کنون و همچنین ماتریس مجاورت گراف به نام matrix، بررسی می‌کند که هر همسایه گره vertexIndex آیا با گره j هم رنگ است یا خیر.

```
private boolean promising(int vertexIndex, int[] coloring) {
    if (vertexIndex == -1) return true;

    boolean isPromising = true;
    int j = 0;

    int[][] matrix = graph.getAdjacencyMatrix();

    while (j < vertexIndex && isPromising) {
        if (matrix[vertexIndex][j] == 1 && coloring[vertexIndex] == coloring[j])
            isPromising = false;
        j++;
    }

    return isPromising;
}
```

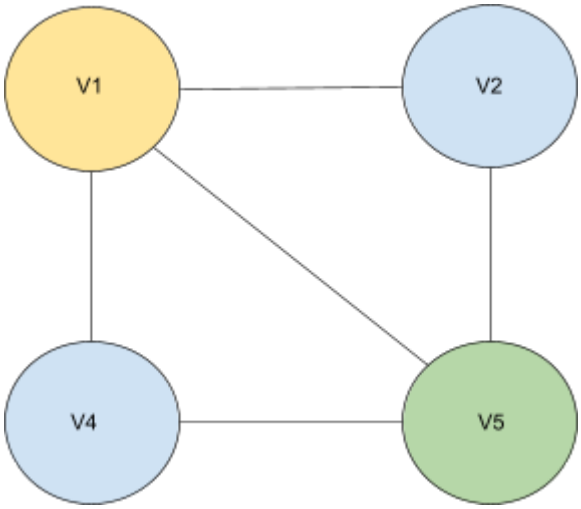
بررسی نمونه:



در این نمونه، هر گره یک سطح از فرش را مشخص می‌کند و یال‌ها نشان‌دهنده‌ی مجاورت هر بخش از با دیگری است. با اجرای الگوریتم m-coloring روی این گراف، خروجی زیر را برنامه به ما می‌دهد.

```
### Carpet Factory ###
-- Design a New Carpet --
Found 6 coloring with 3 colors.
Coloring #1:
Node #0: Color[0]
Node #1: Color[1]
Node #2: Color[2]
Node #3: Color[1]
Coloring #2:
Node #0: Color[0]
Node #1: Color[2]
Node #2: Color[1]
Node #3: Color[2]
Coloring #3:
Node #0: Color[1]
Node #1: Color[0]
Node #2: Color[2]
Node #3: Color[0]
Coloring #4:
Node #0: Color[1]
Node #1: Color[2]
Node #2: Color[0]
Node #3: Color[2]
Coloring #5:
Node #0: Color[2]
Node #1: Color[0]
Node #2: Color[1]
Node #3: Color[0]
Coloring #6:
Node #0: Color[2]
Node #1: Color[1]
Node #2: Color[0]
Node #3: Color[1]
```

همان‌طور که مشاهده می‌شود، الگوریتم ما ۶ نوع رنگ‌آمیزی مختلف با ۳ رنگ را پیدا کرده است. یکی از نمونه‌های رنگ‌آمیزی به صورت زیر است.



تحلیل زمان و حافظه:

تحلیل پیچیدگی زمانی برای کد مشخص شده نیازمند بررسی جزئیات بیشتر است. اما برای تحلیل اجمالی می‌توان به تعداد راس‌ها (n) و تعداد رنگ‌ها (m) در مسئله مقاله‌رنگ‌آمیزی (m-coloring) توجه کرد.

- تابع promising:

در این تابع، یک حلقه وجود دارد که به طول vertexIndex اجرا می‌شود. این حلقه یک مقایسه انجام می‌دهد و در صورت برقراری شرط، isPromising را به false تغییر می‌دهد. بنابراین پیچیدگی زمانی این حلقه O(vertexIndex) است.

- تابع mColoring:

در این تابع، ابتدا تابع promising صدا زده می‌شود که پیچیدگی زمانی آن قبلاً محاسبه شده است. سپس یک شرط if بررسی می‌شود که اگر برقرار باشد، یک شرط دیگر بررسی می‌شود که در صورت برقراری به انتهای راس‌ها رسیده است یا خیر. در صورتی که به انتهای راس‌ها رسیده باشیم، یک کپی از coloring ایجاد می‌شود و در لیست allMColorings قرار می‌گیرد. در غیر این صورت، یک حلقه وجود

دارد که به طول mColors اجرا می‌شود و داخل آن تابع mColoring صدا زده می‌شود. بنابراین پیچیدگی زمانی این حلقه O(mColors) است.

با ترکیب این دو تابع، می‌توان تحلیل زمانی کلی این کد را انجام داد. اگر از روش بازگشتی برای حل مسئله استفاده کنیم، تعداد تمام فرزندان درخت جستجوی حلقه for در تابع mColoring، برابر با mColors است. بنابراین تعداد تمام راس‌ها در این درخت بازگشتی برابر است با n × mColors. در نتیجه، تعداد تمام تماس‌های تابع promising نیز O(n × mColors) است.
بنابراین، تحلیل زمانی کلی این کد برابر است با O(n × mColors × vertexIndex)، که در آن vertexIndex نشان دهنده اندیس آخرین راس است.

برای تحلیل پیچیدگی فضایی کد مشخص شده، باید به استفاده از حافظه در طول اجرای برنامه توجه کنیم. در ادامه، تحلیلی از مصرف حافظه در این کد ارائه می‌دهم:
- متغیرهای مورد استفاده:
- coloring: یک آرایه صحیح که رنگ‌های مربوط به رئوس را نگهداری می‌کند. اندازه این آرایه برابر با تعداد رئوس است و بسته به اندازه ورودی، مصرف حافظه متغیر خواهد بود.
- allMColorings: یک لیست که تمام پاسخ‌های ممکن به مسئله m-coloring را نگهداری می‌کند. اندازه این لیست به تعداد پاسخ‌های ممکن بستگی دارد و مصرف حافظه متغیر است.
- matrix: یک ماتریس صحیح که ماتریس مجاورت گراف را نگهداری می‌کند. اندازه این ماتریس برابر با تعداد رئوس مسئله است و بسته به اندازه ورودی، مصرف حافظه متغیر است.

با توجه به موارد فوق، تحلیل فضایی کلی این کد برابر است با حافظه مورد استفاده در متغیرها و داده‌های ورودی. بنابراین، تحلیل فضایی آن به طور خلاصه به صورت زیر است:
- متغیرها و آرایه‌ها: O(n)
- ماتریس مجاورت: O(n^2)
- لیست allMColorings: بستگی به تعداد پاسخ‌های ممکن دارد.

به طور کلی، مصرف حافظه این کد به صورت خطی با تعداد رئوس و مصرف حافظه ماتریس مجاورت است. اگر تعداد رئوس گراف بسیار بزرگ باشد، مصرف حافظه نیز بسیار بالا خواهد بود.

جست‌وجو بر اساس طرح نقشه

در این بخش با دریافت یک ماتریس که طرح فرش را مشخص می‌کند باید فرش‌های مشابه با آن را که در سیستم مشخص شده است را نمایش دهیم.

ورودی: ماتریس که طرح فرش را مشخص می‌کند

خروجی: شبیه‌ترین فرش‌های داخل سیستم با طرح ورودی

در این بخش، ۷ ماتریس به صورت فایل به عنوان نمونه‌های فرش ذخیره شده است. هر فرش به صورت یک ماتریس از اعداد درنظر گرفته شده است که هر درایه می‌تواند بیانگر رنگ آن قسمت از فرش باشد. ما از یک الگوریتم برای تراز کردن ماتریس‌ها استفاده می‌کنیم. ابتدا هر ماتریس را به صورت یک آرایه یک بعدی درنظر گرفته، سپس یک آرایه‌ی دلخواه را به عنوان فرش معیار با دیگر آرایه‌ها به الگوریتم نیدلمن-وانچ می‌دهیم. این الگوریتم تا حد بهینه‌ای دو آرایه را با یکدیگر تراز می‌کند و یک عدد به عنوان درصد پنالتی ای عملیات گزارش می‌کند. هر چه این عدد کمتر باشد، بدین معنی است که دو آرایه شباهت بیشتری باهم دارند.

سپس بعد از به دست آوردن تمامی پنالتی‌ها، آن‌ها را با الگوریتم جستجوی سریع مرتب کرده و ۳ عدد کوچک‌تر را گزارش می‌کنیم.

بررسی نمونه:

برای ورودی یکی از فایل‌ها را انتخاب می‌کنیم. برای مثال فایل carpet3.txt. خروجی الگوریتم به صورت زیر خواهد بود.

```
### Carpet Factory ###
-- Searching for Similarity --
Found 7 carpets.
Choose the carpet index to search for similar ones [0- 6] >>> 3
All penalties sorted:
[1766, 1782, 1804, 0, 1770, 1758, 1790]
Similar ones and their scores:
[1] Carpet #5 with penalty 1758
[2] Carpet #0 with penalty 1766
[3] Carpet #4 with penalty 1770
```

مشاهده می‌شود که فرش‌های ۵، ۰ و ۴ بیشترین شباهت را با فرش ۳ دارند.

تحلیل زمان و حافظه:

تحلیل پیچیدگی زمانی برای کد مشخص شده، که برای الگوریتم Needleman-Wunsch استفاده می‌شود، به مراحل مختلف آن تقسیم می‌شود. زمانی که n طول رشته اول (x) و m طول رشته دوم (y) است، مراحل زمانی زیر را می‌توان تحلیل کرد:

1. مرحله مقداردهی اولیه جدول (Initializing the table):

دو حلقه تکرار (یکی برای رشته x و دیگری برای رشته y) وجود دارد که به تعداد (n + m) اجرا می‌شوند. بنابراین پیچیدگی زمانی این بخش O(n + m) است.

2. مرحله محاسبه مقادیر جدول (Calculating table values):

دو حلقه تکرار تودرتو (یکی درون دیگری) وجود دارد که به تعداد (n * m) اجرا می‌شوند. برای هر جفت (i, j) از اندیس‌های جدول، محاسبه مقدار جدول به صورت ثابت است و زمان ثابتی می‌برد. بنابراین پیچیدگی زمانی این بخش O(n * m) است.

3. مرحله بازسازی پاسخ (Reconstructing the solution):

یک حلقه تکرار وجود دارد که به تعداد (n + m) اجرا می‌شود. در هر مرحله، یک سری عملیات ثابت انجام می‌شود که زمان ثابتی می‌برد. بنابراین پیچیدگی زمانی این بخش نیز O(n + m) است.

بنابراین، با جمع کردن پیچیدگی زمانی این سه مرحله، می‌توان گفت که پیچیدگی زمانی کلی برای الگوریتم Needleman-Wunsch در این کد برابر است با O(n * m)، که n و m به ترتیب طول رشته اول و رشته دوم هستند.

تحلیل حافظه:

- متغیرها و آرایه‌ها:

- متغیرهای m و n: دو عدد صحیح که طول رشته‌های ورودی را نگهداری می‌کنند. این متغیرها به حافظه ثابتی نیاز دارند.

- ماتریس table: یک ماتریس صحیح که مقادیر جدول برای الگوریتم Needleman-Wunsch را نگهداری می‌کند. اندازه این ماتریس برابر با (n + m + 1) در (n + m + 1) است و مصرف حافظه آن به صورت متغیر است.

با توجه به موارد فوق، تحلیل فضایی کلی این کد برابر است با مصرف حافظه برای متغیرها و آرایه‌ها. بنابراین، تحلیل فضایی آن به طور خلاصه به صورت زیر است:

- متغیرها و آرایه‌ها: O(n + m)
- ماتریس O((n + m)^2)

به طور کلی، مصرف حافظه این کد به صورت خطی با جمع طول رشته‌های ورودی (n و m) و مصرف حافظه ماتریس table است. اگر طول رشته‌ها بسیار بزرگ باشد، مصرف حافظه نیز بسیار بالا خواهد بود.

خرید براساس میزان پول

در این مسئله با داشتن مقدار پولی باید بیشترین تعداد فرشی که می‌توان خرید را پیدا کرد.

ورودی: مقدار پول (گنجایش کوله‌پشتی)

خروجی: لیستی از فرش‌هایی که کاربر می‌تواند بخرد

برای حل این مسئله از الگوریتم کوله‌پشتی به روش برنامه‌نویسی پویا کمک می‌گیریم.

"K[i][w]" بیانگر بیشترین ارزش قابل دستیابی با در نظر گرفتن اولین "i" مورد و ظرفیت "w" است. این کد یک آرایه دوبعدی "K" به اندازه "(capacity + 1) × (n + 1)" را برای ذخیره‌سازی مقادیر بیشینه ایجاد می‌کند. همچنین یک آرایه بولین "inSack" به اندازه "n" را برای پیگیری آیتم‌های درون کوله‌پشتی تعریف می‌کند.

حلقه‌ی تودرتو روی هر آیتم ("i") و هر ظرفیت ممکن ("w") حرکت می‌کند. شرایط "if-else" اقلام پایه و رابطه تکرار اصلی مسئله کوله‌پشتی را اداره می‌کنند.

- اگر "i" برابر با 0 باشد (بدون آیتمی برای در نظر گرفتن) یا "w" برابر با 0 باشد (ظرفیت باقی‌مانده‌ای نداریم)، ارزش بیشینه برابر با 0 است.
- اگر وزن آیتم فعلی ("weights[i - 1]") کمتر یا مساوی با ظرفیت فعلی ("w") باشد، دو انتخاب برای ما وجود دارد: یا آیتم را در کوله‌پشتی قرار می‌دهیم ("values[i - 1] + K[i - 1][w - weights[i - 1]]") یا آن را نادیده می‌گیریم ("K[i - 1][w]")؛ ما بیشینه را از بین این دو مقدار انتخاب می‌کنیم و در "K[i][w]" ذخیره می‌کنیم.

- اگر وزن آیتم فعلی بیشتر از ظرفیت فعلی باشد، قادر به اضافه کردن آن نیستیم، بنابراین ارزش بیشینه همانند ارزش آیتم قبلی در همان ظرفیت باقی‌مانده خواهد بود ("K[i - 1][w]").

در نهایت، تابع آرایه "inSack" را برمی‌گرداند که هر خانه true به معنای آن است که کدام فرش‌ها انتخاب شده است.

```
26      public boolean[] knapsack(int capacity, int n, int[] weights, int[] values) {
27          int[][] K = new int[n + 1][capacity + 1];
28          boolean[] inSack = new boolean[n];
29          int includeValue, excludeValue;
30
31          for (int i = 0; i <= n; i++) {
32              for (int w = 0; w <= capacity; w++) {
33                  if (i == 0 || w == 0)
34                      K[i][w] = 0;
35                  else if (weights[i - 1] <= w) {
36                      includeValue = values[i - 1] + K[i - 1][w - weights[i - 1]];
37                      excludeValue = K[i - 1][w];
38                      if (includeValue > excludeValue) {
39                          K[i][w] = includeValue;
40                          inSack[i - 1] = true;
41                      } else {
42                          K[i][w] = excludeValue;
43                          inSack[i - 1] = false;
44                      }
45                  } else {
46                      K[i][w] = K[i - 1][w];
47                      inSack[i - 1] = false;
48                  }
49              }
50          }
51
52          return inSack;
53      }
```

```java
    public void printKnapsack(int capacity, int n, int[] weights, int[] values) {
        boolean[] sack = knapsack(capacity, n, weights, values);
        int sackWeight = 0;
        int sackValue = 0;
        for (int i = 0; i < sack.length; i++) {
            if (sack[i]) {
                System.out.println("The cost: " + weights[i] + " | The value: " + values[i]);
                sackValue += values[i];
                sackWeight += weights[i];
            }
        }
        System.out.println("Total sack value: " + sackValue + "|| Total sack weight: " + sackWeight);
    }
}

```

بررسی نمونه:

در این نمونه فرش‌هایی از قبل تعیین شده با قیمت‌هایی متفاوت وارد سیستم شده‌اند که با وارد کردن میزان پول خروجی آن مشخص می‌شود. دقت کنید که ارزش فرش‌ها همه یک در نظر گرفته شده است. برای تست این قسمت تابع runShopTest را اجرا می‌کنیم.

```java
public static void runShopTest() {
    ArrayList<Carpet> carpets = new ArrayList<>();
    carpets.add(new Carpet( cost: 200));
    carpets.add(new Carpet( cost: 100));
    carpets.add(new Carpet( cost: 400));
    carpets.add(new Carpet( cost: 900));
    carpets.add(new Carpet( cost: 500));
    Shop shop = new Shop(carpets);

    Scanner scanner = new Scanner(System.in);
    System.out.println("Please enter your money: ");
    int money = scanner.nextInt();
    shop.buyCarpet(money);
}

```

```
### models.Carpet Factory ###
-- Main Menu --
[0] Zero
[1] Design a new carpet
[2] Search by carpet pattern
[3] Purchase according to the budget
[4] Routing to the nearest carpet factory

Enter menu item >>> 3
Please enter your money:
700
The cost: 200 | The value: 1
The cost: 100 | The value: 1
The cost: 400 | The value: 1
Total sack value: 3|| Total sack weight: 700

Process finished with exit code 0
```

```
### models.Carpet Factory ###
-- Main Menu --
[0] Zero
[1] Design a new carpet
[2] Search by carpet pattern
[3] Purchase according to the budget
[4] Routing to the nearest carpet factory

Enter menu item >>> 3
Please enter your money:
350
The cost: 200 | The value: 1
The cost: 100 | The value: 1
Total sack value: 2|| Total sack weight: 300

Process finished with exit code 0
```

تحلیل زمان و حافظه:

با توجه به دو حلقه تودرتو در کد، زمان اجرای این الگوریتم به صورت مجموعه مربعی از ورودی‌هاست. اگر طول آرایه وزن‌ها را n و ظرفیت کوله‌پشتی را m در نظر بگیریم، زمان اجرای این کد O(n * m) است. از طرفی، هر یک از حلقه‌ها n+1 بار اجرا می‌شوند و داخل هر حلقه، مراحلی به تعداد m انجام می‌شود. بنابراین، تعداد کل عملیات‌ها برابر با (m+1) * (n+1) است.

در این کد، دو ساختار داده ذخیره می‌شود: آرایه دوبعدی K با اندازه (capacity+1) × (n+1) و آرایه inSack با اندازه n. در نتیجه، فضای مورد نیاز برای ذخیره سازی ماتریس K برابر با (capacity+1) × (n+1) و فضای مورد نیاز برای آرایه inSack برابر با n است. بنابراین، پیچیدگی فضایی کد برابر با O(n * capacity) است، که n طول آرایه‌های وزن‌ها و ارزش‌ها و capacity ظرفیت کوله‌پشتی است.

مسیریابی به نزدیک‌ترین فروشگاه

این مسئله از ما می‌خواهد که با توجه به ورودی کاربر، نزدیک‌ترین مسیر از راسی که کاربر وارد کرده تا بقیه فروشگاه‌ها (راس‌های دیگر) را بدست بیاوریم و خود مسیر و اندازه آن را به عنوان خروجی برگردانیم.

ورودی: یکی از فروشگاه‌ها (راس‌های داخل سیستم) که توسط اندیس آن مشخص می‌شود

خروجی: کوتاه‌ترین مسیر ممکن در گراف بین راس ورودی و بقیه راس‌ها و اندازه آن

برای حل این مسئله از الگوریتم دایجسترا به صورت حریصانه که کوتاه‌ترین مسیر از یک راس به بقیه راس‌های گراف را بدست می‌آورد، استفاده می‌کنیم.

این الگوریتم هر بار راسی که کمترین فاصله را دارد را بررسی می‌کند و فاصله‌های به روزرسانی شده را برای همسایه‌های آن راس محاسبه می‌کند.

در ابتدا، فاصله تمام راس‌ها را عدد بی‌نهایت تنظیم می‌کنیم، سپس فاصله راس مبدأ را صفر قرار می‌دهیم.

سپس در هر مرحله، راس با کمترین فاصله را از مجموعه راس‌ها انتخاب می‌کنیم. سپس برای همسایه‌های آن راس، مقدار فاصله را محاسبه کرده و اگر مقدار جدید کمتر از فاصله قبلی بود، فاصله را به روزرسانی می‌کنیم. همچنین، مشخص می‌کنیم که این گره چه گره پدری دارد. این فرایند ادامه می‌یابد تا وقتی که همه راس‌ها بررسی شوند و دیگر راسی برای بررسی باقی نماند.

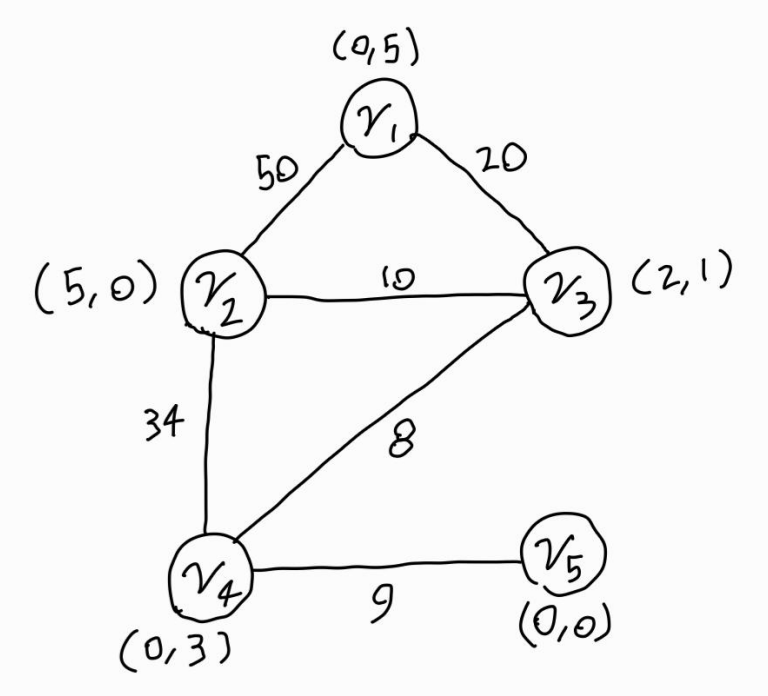
جواب مسئله در آرایه dist قرار دارد که در هر خانه آن مقدار کمترین مسیر از آن راس به راس مبدا قرار دارد. در آرایه prev هم نزدیک‌ترین راس قبلی برای هر راس ذخیره می‌شود تا در تابع findShortestPath دوباره مورد استفاده قرار بگیرد و بتواند کوتاه‌ترین مسیر را بسازد.

```java
public int[] dijkstra(int[][] graph, int srcIndex) {
    int size = graph.length;
    int[] dist = new int[size];
    PriorityQueue<Node> pq = new PriorityQueue<>();

    pq.add(new Node(srcIndex, distance: 0));
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[srcIndex] = 0;

    while (!pq.isEmpty()) {
        Node currentNode = pq.poll();
        int currentIndex = currentNode.index;
        int currentDistance = currentNode.distance;

        // We already found a better path before we got to
        // processing this node, so we can ignore it.
        if (dist[currentIndex] < currentDistance) continue;

        // Update distances of neighboring nodes
        for (int i = 0; i < size; i++) {
            int tempDist = currentDistance + graph[currentIndex][i];

            // If a shorter path is found, update the distance and parent node
            if (graph[currentIndex][i] != 0 && tempDist < dist[i]) {
                dist[i] = tempDist;
                prev[i] = currentIndex;
                pq.add(new Node(i, tempDist));
            }
        }
    }


    // find min
    int min = Integer.MAX_VALUE;
    int indexMin = -1;
    for (int i = 0; i < dist.length; i++) {
        if (dist[i] < min && dist[i] != 0) {
            min = dist[i];
            indexMin = i;
        }
    }
    return new int[]{min, indexMin};
}
```

```java
public ArrayList<Integer> getShortestPath(int destination) {
    ArrayList<Integer> path = new ArrayList<>();

    for (int at = destination; at != -1; at = prev[at]) {
        path.add(at + 1);
    }

    Collections.reverse(path);
    return path;
}
```

بررسی نمونه:

در این نمونه گراف فروشگاه‌ها از قبل تعیین شده با مختصاتی متفاوت وارد سیستم شده‌اند. برای تست این قسمت تابع runPathFinderTest را اجرا می‌کنیم. هر فروشگاه مختصاتی دارد که وزن بین یال‌ها توسط این مختصات ها بدست می‌آید. برای مثال ورودی زیر برابر با گراف زیر است.



```java
    public static void runPathFinderTest() {
        int[][] coordinates = {
                {0, 5},
                {5, 0},
                {2, 1},
                {0, 3},
                {0, 0}
        };

        int[][] edges = {
                {0, 1},
                {1, 2},
                {0, 2},
                {1, 3},
                {2, 3},
                {3, 4}
        };

        PathFinder pathFinder = new PathFinder(coordinates, edges);
        pathFinder.chooseVertexForShortestPath();
    }
```

```
### models.Carpet Factory ###
-- Main Menu --
[0] Zero
[1] Design a new carpet
[2] Search by carpet pattern
[3] Purchase according to the budget
[4] Routing to the nearest carpet factory

Enter menu item >>> 4
--------------------
Index: 1 || X: 0 | Y: 5
--------------------
Index: 2 || X: 5 | Y: 0
--------------------
Index: 3 || X: 2 | Y: 1
--------------------
Index: 4 || X: 0 | Y: 3
--------------------
Index: 5 || X: 0 | Y: 0
Please choose nearest vertex to yourself:
4
The length of minimum path is: 9
The Path: [4, 5]

Process finished with exit code 0
```

```
### models.Carpet Factory ###
-- Main Menu --
[0] Zero
[1] Design a new carpet
[2] Search by carpet pattern
[3] Purchase according to the budget
[4] Routing to the nearest carpet factory

Enter menu item >>> 4
--------------------
Index: 1 || X: 0 | Y: 5
--------------------
Index: 2 || X: 5 | Y: 0
--------------------
Index: 3 || X: 2 | Y: 1
--------------------
Index: 4 || X: 0 | Y: 3
--------------------
Index: 5 || X: 0 | Y: 0
Please choose nearest vertex to yourself:
1
The length of minimum path is: 20
The Path: [1, 3]

Process finished with exit code 0
```

تحلیل زمان و حافظه:

پیچیدگی زمانی الگوریتم دایکسترا به تعداد گره‌ها (V) و تعداد یال‌ها (E) در گراف وابسته است. در مرحله‌های اصلی الگوریتم، از یک حلقه به طول V استفاده می‌شود که به ترتیب هر یک از گره‌ها را یک بار بررسی می‌کند.

حلقه اصلی: پیچیدگی زمانی این مرحله$O((V + E) \log V)$ است. یک حلقه به طول V را برای بررسی هر یک از گره‌ها اجرا می‌کنیم. در هر مرحله، به تعداد یال‌های همسایه هر گره (مجموعه E) نگاه می‌کنیم و عملیات درج و حذف از صف اولویت را برای هر یال انجام می‌دهیم. هر عمل درج و حذف از صف اولویت با پیچیدگی $O(\log V)$ انجام می‌شود.

بازسازی مسیر کوتاه‌ترین مسیر (findShortestPath): پیچیدگی زمانی این مرحله $O(V)$ است. پس از پایان الگوریتم، با استفاده از اطلاعات prev، مسیر کوتاه‌ترین مسیر را بازسازی می‌کنیم. این بازسازی در حداکثر V مرحله انجام می‌شود.

بنابراین، پیچیدگی زمانی کل الگوریتم دایکسترا با استفاده از یک صف اولویت برابر $O((V + E) \log V)$ است.

پیچیدگی فضایی الگوریتم به تعداد گره‌ها (V) در گراف وابسته است و برابر $O(V)$ است. برای ذخیره فواصل کوتاه‌ترین مسیرها از گره مبدأ به سایر گره‌ها، نیازمند یک آرایه به طول V هستیم. در مورد نگهداری صف اولویت نیز، زمانی که همه گره‌ها در صف اولویت قرار دارند، فضای مصرفی این صف $O(V)$ است.