Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Інститут комп’ютерних систем
Кафедра інформаційних систем

Розрахунково-графічна робота
з дисципліни «Теорія алгоритмів»
Тема: «Алгоритм визначення найкоротшого шляху для пасажира метрополітену»

Виконав:
студент групи АІ-226
Якубовський М.Р.
Прийняли:
Арсірій О.О.
Смик С.Ю.

Одеса 2023

Завдання: Розробіть та реалізуйте алгоритм визначення найкоротшого шляху для пасажира метрополітену, який дозволяв би йому переміщатися від однієї заданої станції до іншої заданої станції, використовуючи граф підземних комунікацій.

Мета: розробити та реалізувати алгоритм для знаходження найкоротшого маршруту в метрополітені між заданими станціями та оцінити його ефективність. Визначити переваги та недоліки всіх алгоритмів.

1. Опис вхідних та вихідних даних

Вхідні дані: граф підземних комунікацій(вершинами графа виступають станції, а ребрами шляхи між станціями), початкова станція, кінцева станція. За основу взяті реальні карти метро.

Вихідні дані: найкоротший шлях (назви станцій), довжина або час найкоротшого шляху, час виконання алгоритму.

2. Розробка та опис загальної схеми алгоритму

Для розробки було вирішено вибрати два алгоритми : Дейкстри , Флойда-Уоршелла . Розглянемо кожен з цих алгоритмів.

Алгоритм Дейкстри

Алгоритм Дейкстри - це алгоритм для пошуку найкоротшого шляху в графі з невід'ємними вагами ребер. Ось загальні відомості про цей алгоритм:

Огляд:

- Тип алгоритму: Алгоритм Дейкстри відноситься до класу алгоритмів пошуку найкоротшого шляху.

- Застосування: Алгоритм Дейкстри використовується для знаходження найкоротшого шляху в графі з неорієнтованими або орієнтованими ребрами, де ваги ребер є невід'ємними (не можуть бути від'ємними).

- Вимоги до графа: Граф має бути зваженим і ациклічним (або не містити циклів з від'ємними вагами). Це означає, що немає циклів зі зменшенням суми ваг на шляху.

Опис алгоритму:

1. Ініціалізація: Всі вершини графа помічаються як недосяжні, крім початкової вершини, яка має відстань 0. До всіх інших вершин присвоюється початкова відстань, яка рівна нескінченності.

2. Оновлення відстаней: Для кожної вершини, до якої можна дістатися з поточної вершини, розглядається можливий коротший шлях через поточну вершину. Якщо знайдено коротший шлях, відстань до цієї вершини оновлюється

3. Вибір наступної вершини: Вибирається вершина з найменшою відстанню серед непомічених вершин і позначається як поточна вершина.

4. Повторення: Шаги 2 і 3 повторюються, поки не буде відзначено всі вершини або не буде досягнута кінцева вершина.

5. Кінцевий результат: По завершенню алгоритму ми маємо відстані від початкової вершини до всіх інших вершин у графі, а також інформацію про оптимальний шлях до кожної вершини.

Наприклад розглянемо виконання алгоритму на прикладі(рис.2.1 та рис 2.2.).

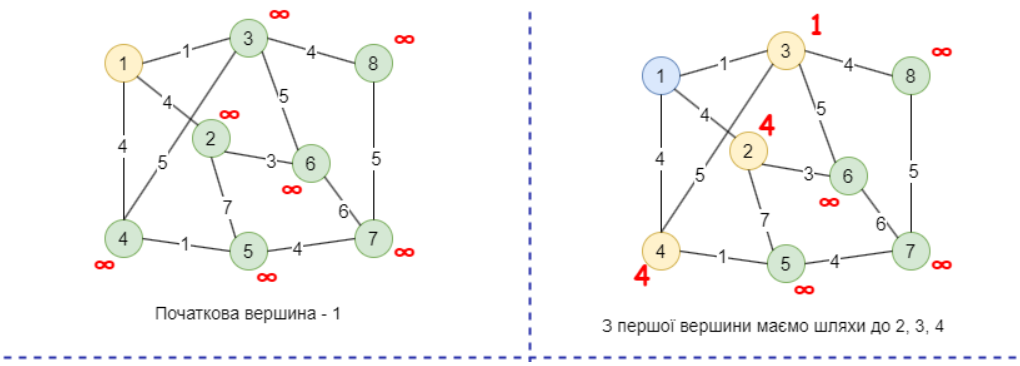

Рисунок 2.1. – Приклад алгоритму Дейкстри

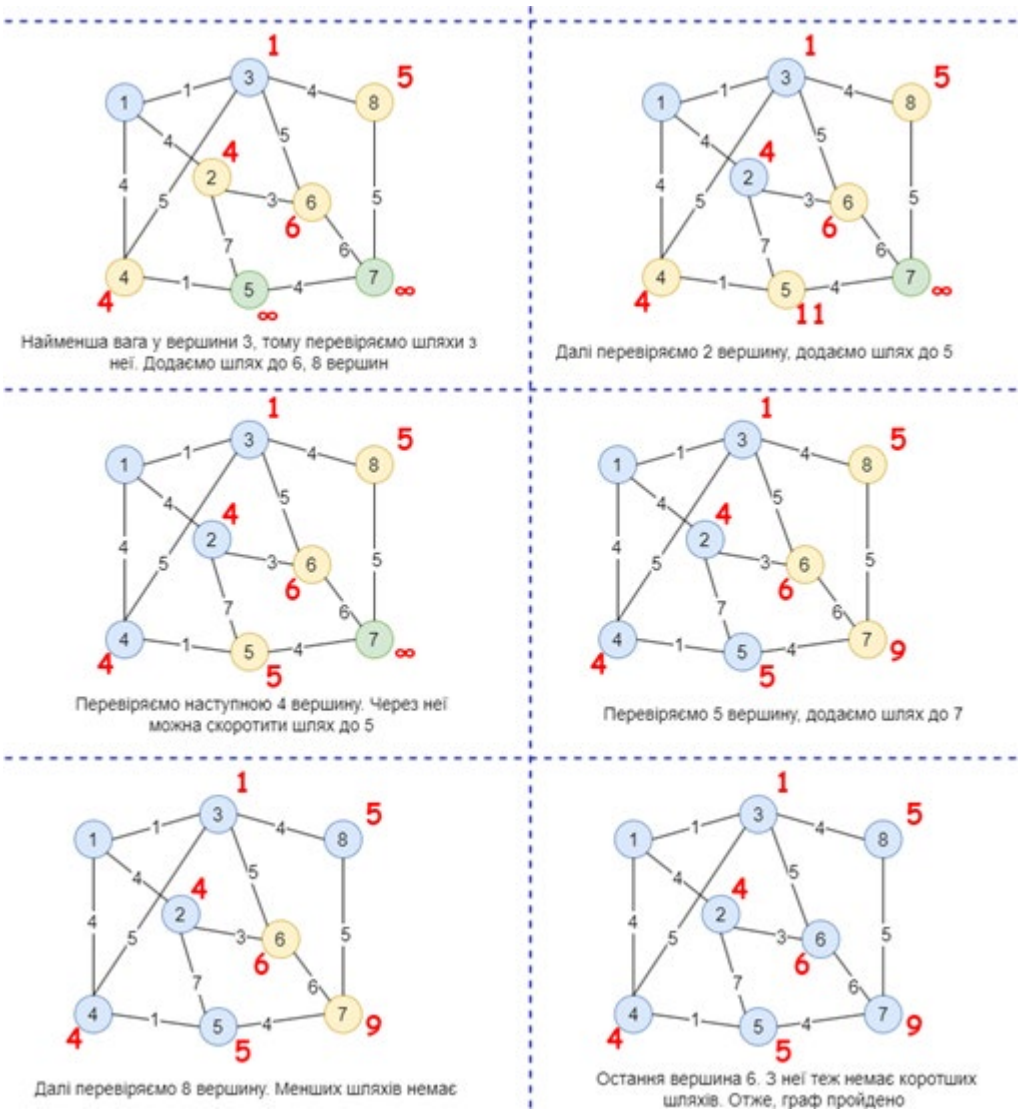

Рисунок 2.2. – Приклад алгоритму Дейкстри

Часова складність:

- Часова складність алгоритму Дейкстри залежить від способу реалізації, але в загальному випадку вона складається O(V^2) або O(V*log(V)), де V - кількість вершин у графі.

Переваги:

- Дейкстрин алгоритм надзвичайно ефективний на графах з невеликою кількістю вершин.

- Він гарантує знаходження найкоротшого шляху в графі з невід'ємними вагами.

Недоліки:

- Алгоритм Дейкстри не працює на графах з вагами ребер, які можуть бути від'ємними.

- Він може бути неефективним на графах з великою кількістю вершин через свою квадратичну часову складність.

- Вимагає реалізації пріоритетної черги для оптимізації роботи на великих графах.

Алгоритм Флойда-Уоршелла

Алгоритм Флойда-Уоршелла - це алгоритм для пошуку найкоротших шляхів між усіма парами вершин у графі. Ось загальні відомості про цей алгоритм:

Огляд:

- Тип алгоритму: Алгоритм Флойда-Уоршелла відноситься до класу алгоритмів пошуку найкоротших шляхів.

- Застосування: Цей алгоритм застосовується для знаходження найкоротших шляхів між всіма парами вершин у графі. Використовується в задачах маршрутизації мереж, аналізі транспортних систем, оптимізації маршрутів та інших областях.

- Вимоги до графа: Граф може бути орієнтованим або неорієнтованим, з вагами ребер, які можуть бути від'ємними або нульовими. Він може містити цикли, включаючи цикли від'ємної ваги.

Опис алгоритму:

1. Ініціалізація: Створюється матриця відстаней, де кожний елемент [i][j] представляє вагу ребра між вершинами i і j. Якщо вершини не мають прямого зв'язку, то відстань встановлюється на нескінченність (або велике число). Ваги діагональних елементів (відстань від вершини до себе) встановлюються на нуль.

2. Оновлення відстаней: Для кожної пари вершин (i, j) розглядається можливий шлях від вершини i до вершини j через проміжну вершину k (де k -

всі можливі проміжні вершини). Якщо такий шлях коротший, ніж поточна відстань між вершинами i та j, то відстань оновлюється.

3. Повторення: Шаг 2 повторюється для всіх можливих проміжних вершин k. Потім процедура повторюється для всіх пар вершин (i, j).

4. Кінцевий результат: Після завершення алгоритму, матриця відстаней містить найкоротші відстані між усіма парами вершин у графі. Також можливо відновити оптимальні шляхи для будь-якої пари вершин.

Наприклад розглянемо алгоритм Флойда для знаходження мінімальних шляхів для графа між двома точками (рис.2.3.)

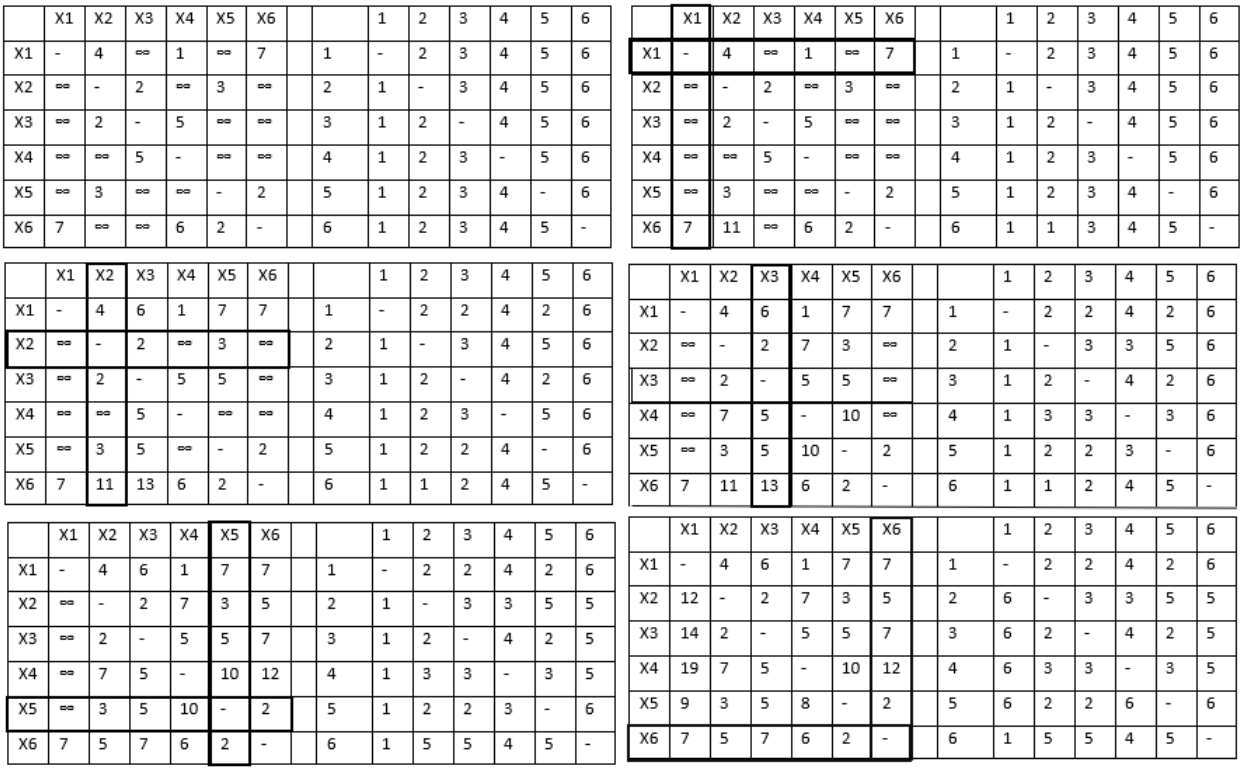

Рисунок 2.3. – Приклад алгоритму Флойда-Уоршелла

Часова складність:

- Часова складність алгоритму Флойда-Уоршелла становить O(V^3), де V - кількість вершин у графі. Цей алгоритм має кубічну складність і може бути неефективним для графів з великою кількістю вершин.

Переваги:

- Алгоритм Флойда-Уоршелла знаходить найкоротші шляхи між усіма парами вершин, що є його основною перевагою.

- Він працює з графами з вагами ребер будь-якого знаку, включаючи ваги, які можуть бути від'ємними.

- Дозволяє знаходити оптимальні шляхи для всіх пар вершин після одного виконання.

Недоліки:

- Алгоритм має кубічну часову складність, що робить його неефективним для великих графів.

- Велика кількість обчислень для всіх пар вершин може зробити алгоритм неефективним у великих мережах.

3. Програмна реалізація

Розглянемо програмну реалізацію алгоритмів на мові Java. Наведений код реалізує алгоритми для пошуку найкоротшого шляху (вони реалізовані як окремі класи) між станціями (між вершинами графа) , користувач вводить початкову та кінцеву станцію на виході отримує при алгоритмі Дейкстри шлях та дистанцію між станціями , та Флойда дистанцію між станціями, так як через особливості алгоритму не можливо реалізувати вивід шляху. Представлено код тільки для самих алгоритмів , але для візуальної реалізації було вирішено використовувати бібліотеку Java FX (з повним кодом програми можна ознайомитися за посиланням).Також програма виміряє час виконання кожного алгоритму та виводить його у консоль.

Алгоритм Дейкстри:

```
public class Dijkstra {
   private final List<String> shortestPath; // Список для зберігання точок найкоротшого шляху
   private int shortestDistance; // Змінна для зберігання довжини найкоротшого шляху

   public Dijkstra() {
```

```
        shortestPath = new ArrayList<>();
        shortestDistance = 0;
    }
    // Метод для знаходження найкоротшого шляху між двома станціями
    public int findShortestDistance(Metro metro, String startStation, String endStation) {

        Map<String, Integer> shortestDistances = new HashMap<>();
        Set<String> unvisitedStations = new HashSet<>(metro.getStations());// невідвідані станції

        // Ініціалізація всіх станцій з нескінченною довжиною шляху
        for (String station : metro.getStations()) {
            shortestDistances.put(station, Integer.MAX_VALUE);
        }

        // Встановлення початкової станції з довжиною шляху 0
        shortestDistances.put(startStation, 0);

        while (!unvisitedStations.isEmpty()) {
            // Вибір станції з найменшою відомою довжиною шляху
            String currentStation = null;
            for (String station : unvisitedStations) {
                if (currentStation == null || shortestDistances.get(station) <
shortestDistances.get(currentStation))
                    currentStation = station;
}

            if (currentStation == null || shortestDistances.get(currentStation) ==
Integer.MAX_VALUE)
                break;

            // Видалення поточної станції зі списку невідвіданих
            unvisitedStations.remove(currentStation);

            // Оновлення відомих довжин шляху до сусідніх станцій
            for (String neighbor : metro.getConnectedStations(currentStation)) {
```

```
            int distanceToNeighbor = metro.getDistanceBetweenStations(currentStation,
neighbor);
            int totalDistance = shortestDistances.get(currentStation) + distanceToNeighbor;

            // Оновлення довжини шляху, якщо новий шлях є коротшим
            if (totalDistance < shortestDistances.get(neighbor))
                shortestDistances.put(neighbor, totalDistance);
        }
    }

    // Отримання найкоротшого шляху та його довжини
    findShortestPath(metro, startStation, endStation, shortestDistances);
    return shortestDistance;
  }

  // Метод для знаходження точок найкоротшого шляху та його довжини
  public void findShortestPath(Metro metro, String startStation, String endStation, Map<String,
Integer> shortestDistances) {
    shortestPath.clear();
    shortestDistance = shortestDistances.get(endStation);

    // Відновлення найкоротшого шляху, починаючи з кінцевої станції
    String currentStation = endStation;
    while (!currentStation.equals(startStation)) {
      shortestPath.add(currentStation);
      for (String neighbor : metro.getConnectedStations(currentStation)) {
        int distanceToNeighbor = metro.getDistanceBetweenStations(currentStation,
neighbor);
        int totalDistance = shortestDistances.get(currentStation) - distanceToNeighbor;

        // Перехід до попередньої станції в шляху
        if (totalDistance == shortestDistances.get(neighbor)) {
          currentStation = neighbor;
          break;
        }
```

```
            }
        }
        shortestPath.add(startStation);

        // Перевернення списку, щоб мати правильний порядок від початкової до кінцевої станції
        Collections.reverse(shortestPath);
    }

    // Метод для отримання списку точок найкоротшого шляху
    public List<String> getShortestPath() {
        return shortestPath;
    }
}
```

Алгоритм Флойда:

```
public class FloydWarshall {
    // Метод для знаходження найкоротшої відстані між двома станціями за допомогою алгоритму Флойда-Уоршалла
    public int findShortestDistance(Metro metro, String startStation, String endStation) {
        int numStations = metro.getStations().size();
        // Матриця для зберігання довжин найкоротших шляхів між станціями
        int[][] distances = new int[numStations][numStations];
        // Ініціалізація початкових відстаней
        for (int i = 0; i < numStations; i++) {
            for (int j = 0; j < numStations; j++) {
                // Встановлення відстані між станціями на нескінченність, крім випадку, коли це одна і та ж станція
                if (i == j) {
                    distances[i][j] = 0;
                } else {
                    distances[i][j] = Integer.MAX_VALUE;
                }
            }
        }
```

```
        // Заповнення початкових відстаней на основі інформації з метро
        for (String station : metro.getStations()) {
            int stationIndex = metro.getStations().indexOf(station);
            for (String neighbor : metro.getConnectedStations(station)) {
                int neighborIndex = metro.getStations().indexOf(neighbor);
                int distance = metro.getDistanceBetweenStations(station, neighbor);
                distances[stationIndex][neighborIndex] = distance;
                distances[neighborIndex][stationIndex] = distance;
            }
        }

        // Виконання алгоритму Флойда-Уоршалла для знаходження найкоротших шляхів
        for (int k = 0; k < numStations; k++) {
            for (int i = 0; i < numStations; i++) {
                for (int j = 0; j < numStations; j++) {
                    if (distances[i][k] != Integer.MAX_VALUE && distances[k][j] !=
Integer.MAX_VALUE
                            && distances[i][k] + distances[k][j] < distances[i][j]) {
                        distances[i][j] = distances[i][k] + distances[k][j];
                    }
                }
            }
        }

        // Отримання індексів початкової та кінцевої станцій
        int startIndex = metro.getStations().indexOf(startStation);
        int endIndex = metro.getStations().indexOf(endStation);

        // Повернення найкоротшої відстані між початковою та кінцевою станціями
        return distances[startIndex][endIndex];
    }
}
```

4. Приклади та візуалізація вхідних даних

Для реалізації функціоналу було взято за основу реальні карти метро, для кращої візуалізації щоб зрозуміти де на практиці можуть знадобитися алгоритми , програмний застосунок на своєму початку (рис.4.1.) запитує яку карту метро користувач хоче використати (в наявності дві карти : метро Києва – так як ми живем в Україні та метро Лондона – так як в ній знаходиться приблизно 300 станцій (вершин) і це буде краще для перевірки та зрівняння алгоритмів).

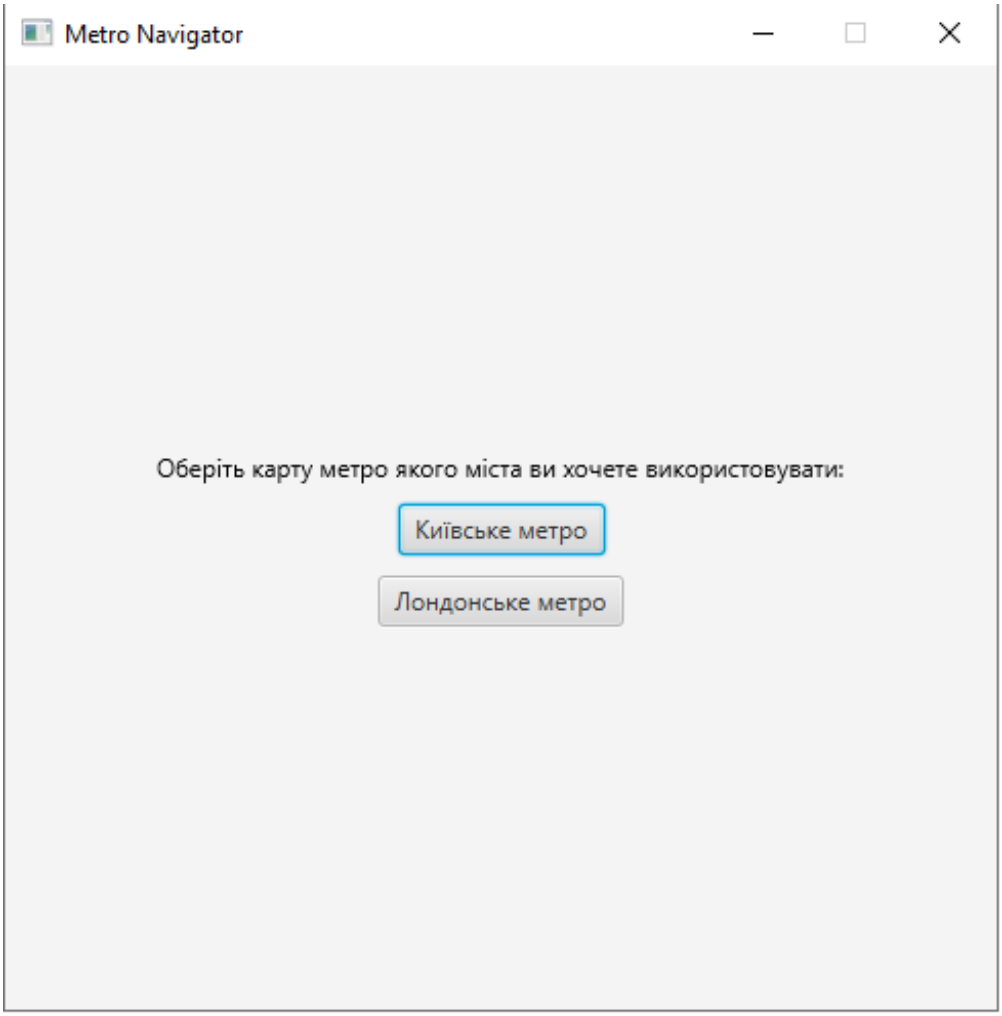

Рисунок 4.1. – Початковий екран програми

При натисканні на кнопку виводиться нове вікно з картою (рис.4.2. та рис.4.3.) та комірцями для вводу даних, кнопки для вибору за яким алгоритмом користувач хоче провести пошук.

Рисунок 4.2. – Карта метро Києва

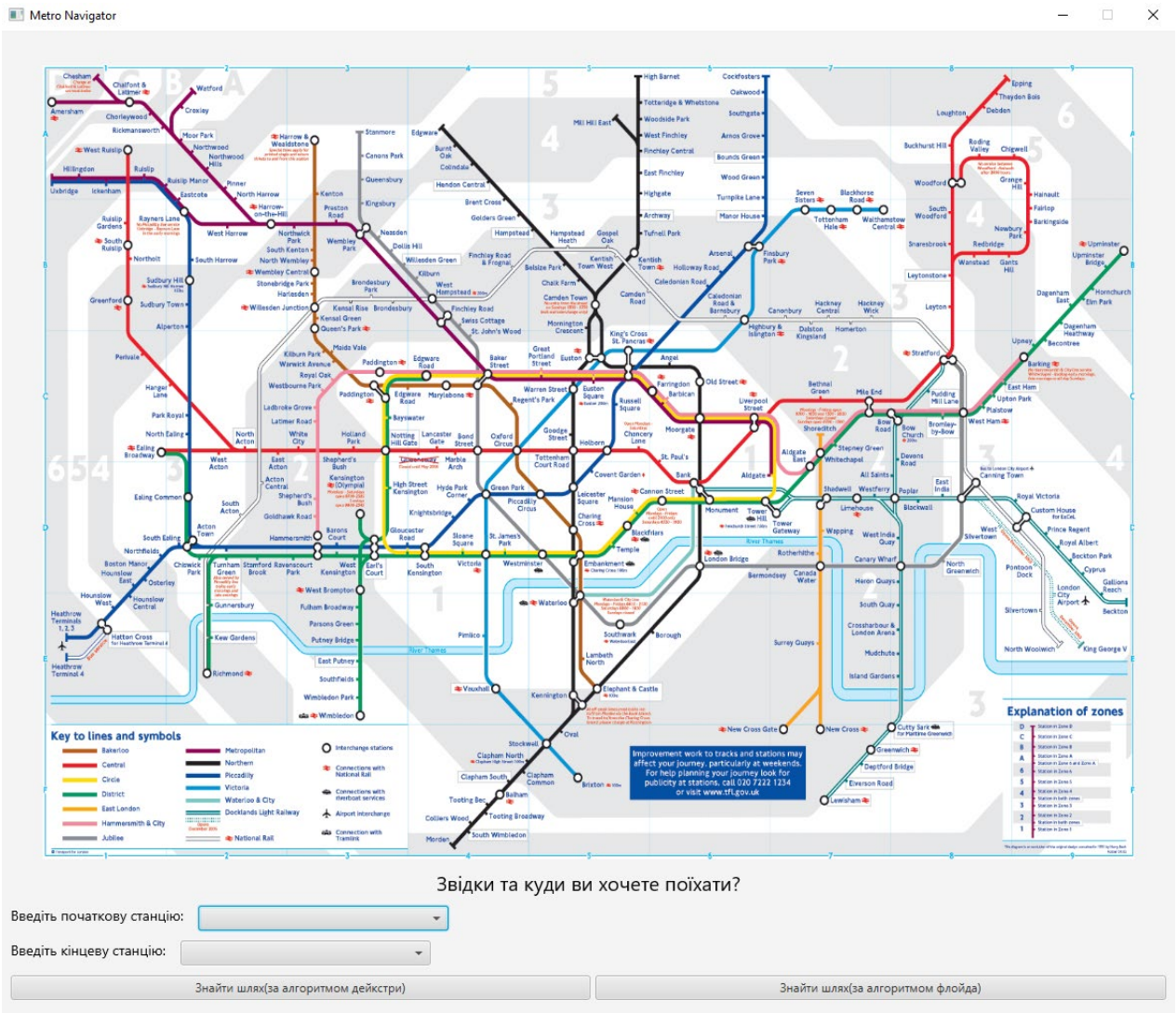

Рисунок 4.3. – Карта метро Лондона

Програма запитує за якими станціями виконати пошук , поля «Введіть початкову станцію» та «Введіть кінцеву станцію» надають можливість вибору станцій зі списку та кнопки надають можливість вибрати алгоритм для пошуку(рис.4.4. та рис.4.5.)

Рисунок 4.4. – Вибір вхідних даних для метро Києва

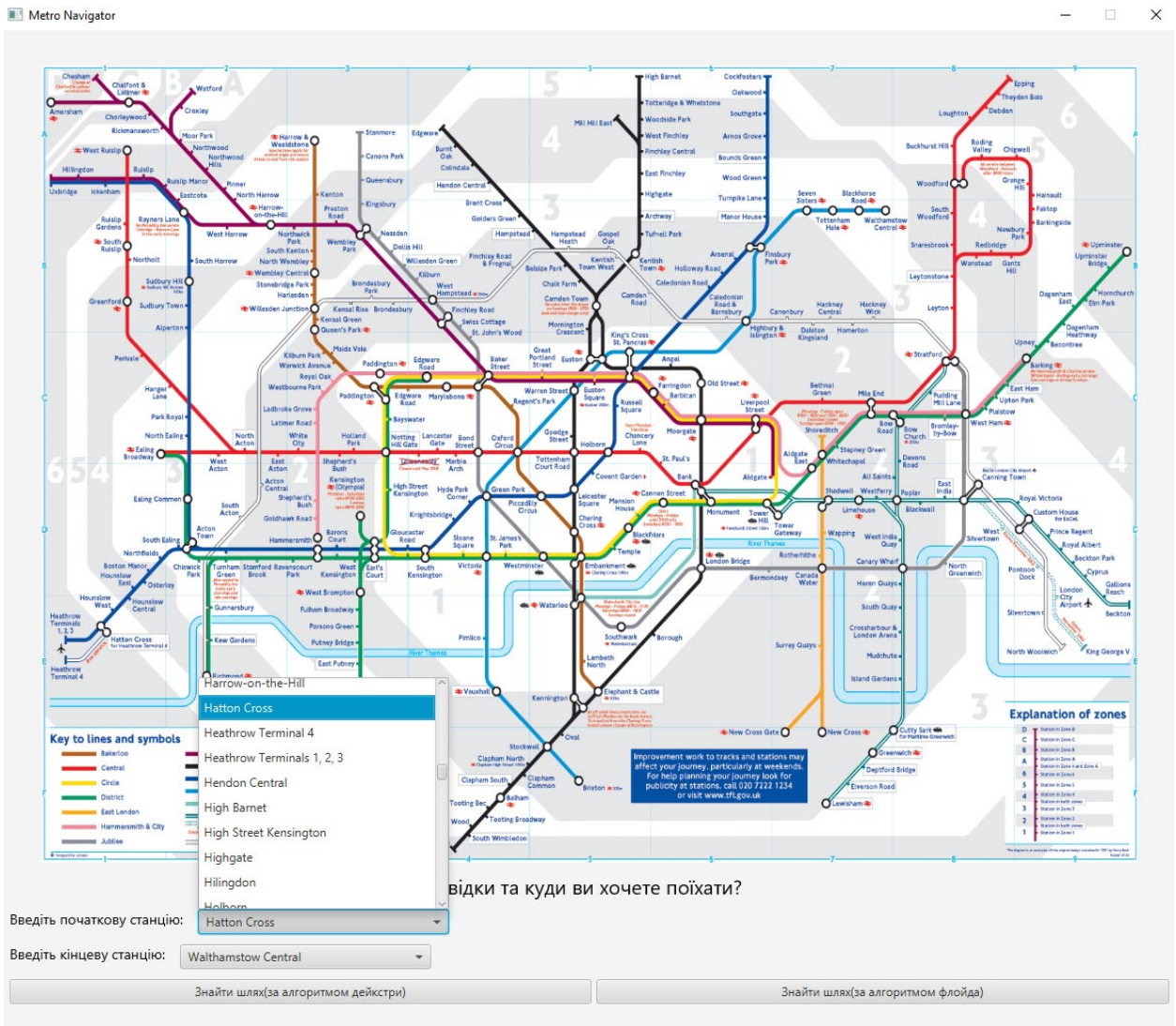

Рисунок 4.4. – Вибір вхідних даних для метро Лондона

5. Приклади та візуалізація вихідних даних

Після натискання на кнопку вибраного алгоритму відповідно виводяться дані про результат алгоритму Дейкстри (рис.5.1. та рис.5.2.) , Флойда (рис.5.3. та рис.5.4.), маємо напис з інформацією про найкоротшу дистанцію між вибраними станціями. Для виводу шляху маємо поле з прокруткою. Після кожного визову алгоритму в консоль виводився час виконання (рис.5.5., рис.5.6., рис.5.7., рис.5.8.).

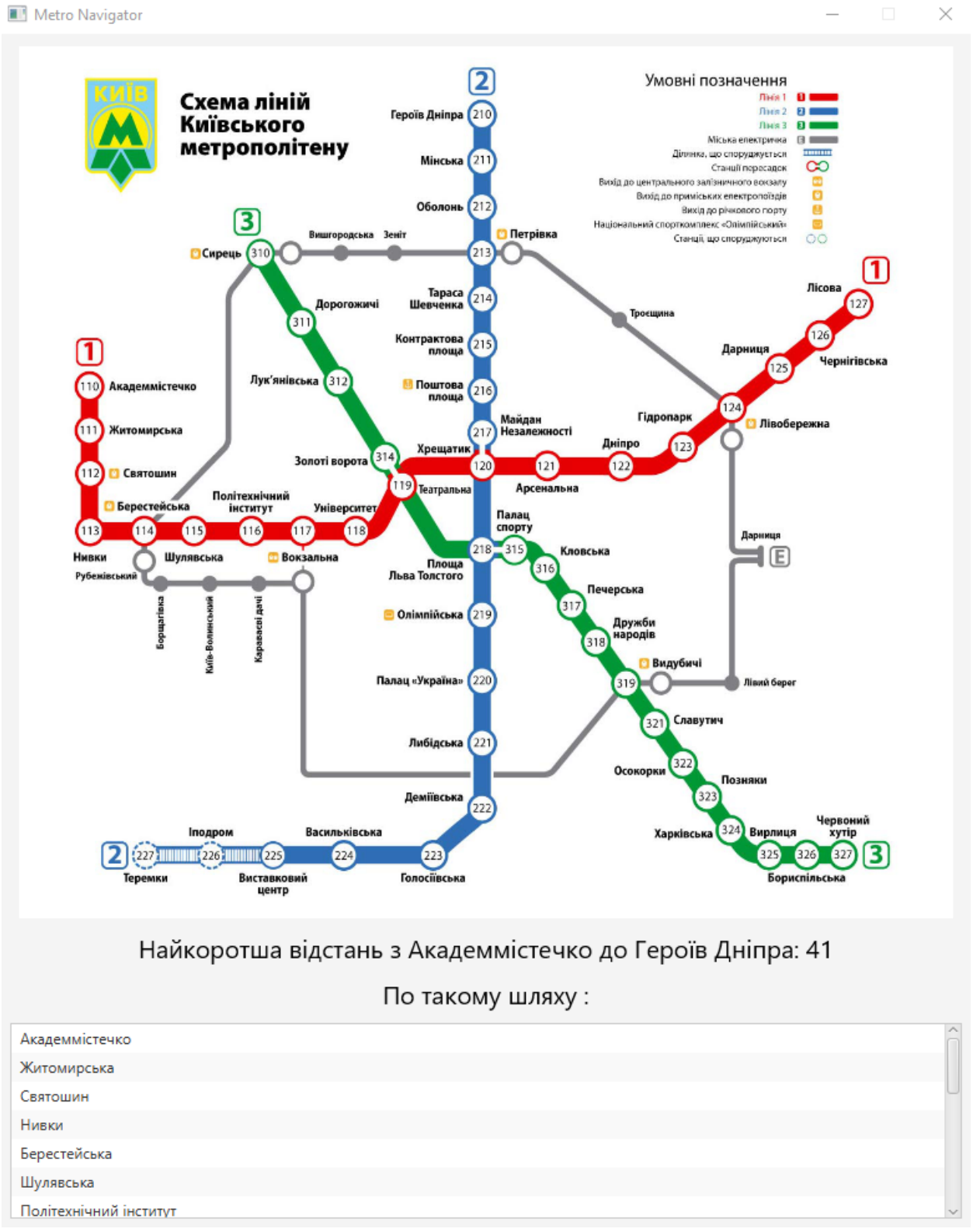

Рисунок 5.1. – Вивід вихідних даних для алгоритма Дейкстри для метро Києва

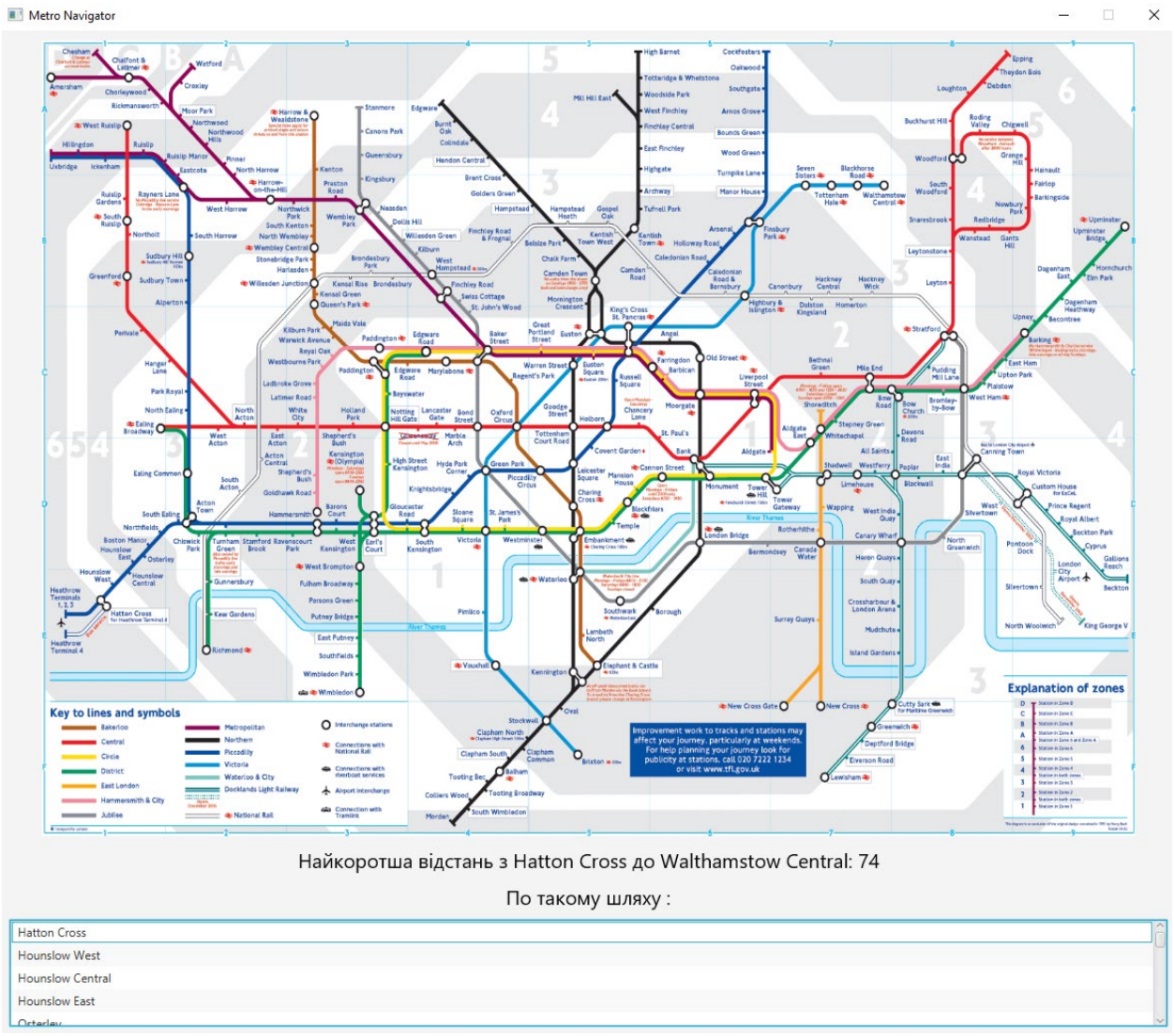

Рисунок 5.2. – Вивід вихідних даних для алгоритма Дейкстри для метро Лондона

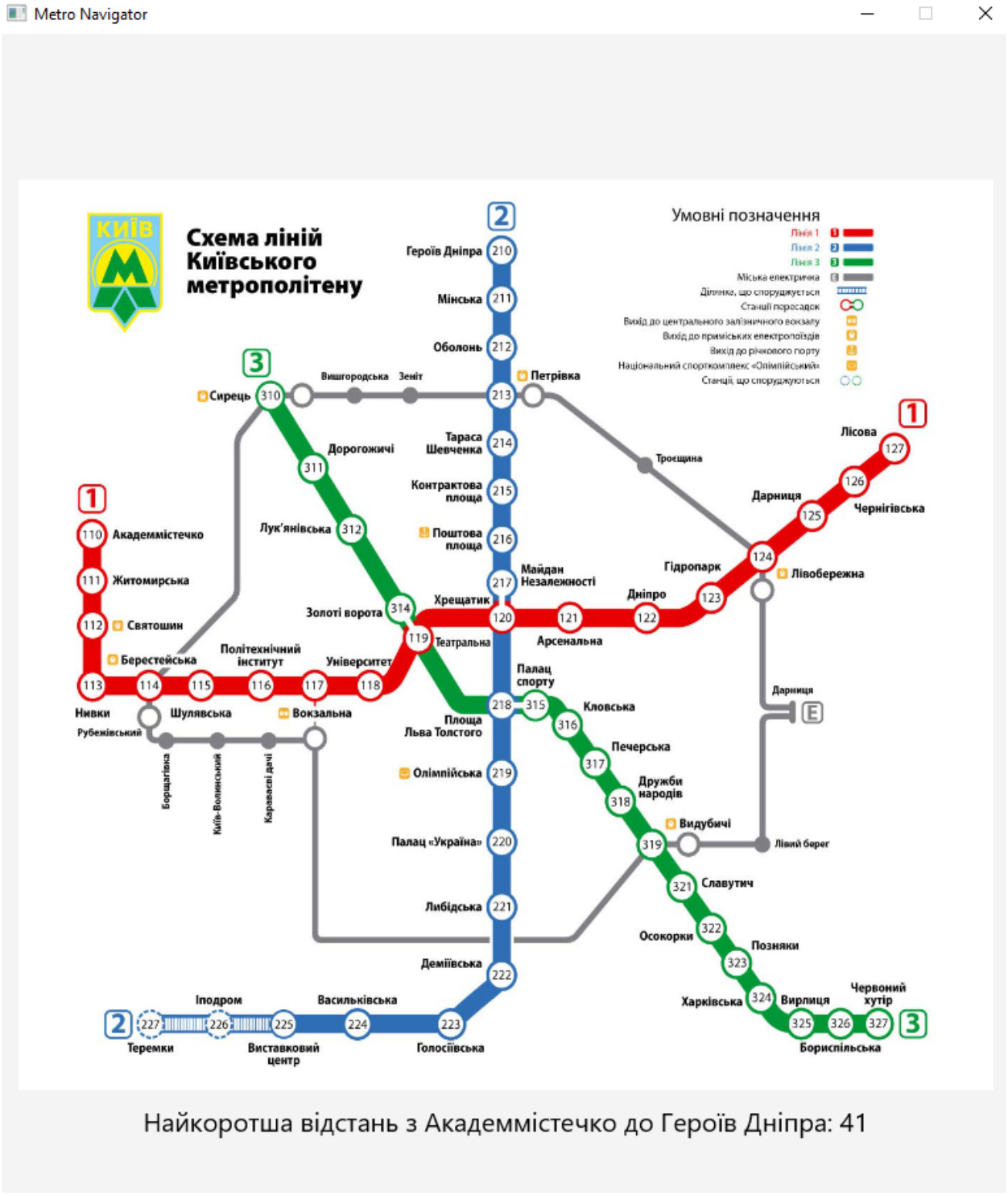

Рисунок 5.3. – Вивід вихідних даних для алгоритма Флойда для метро Києва

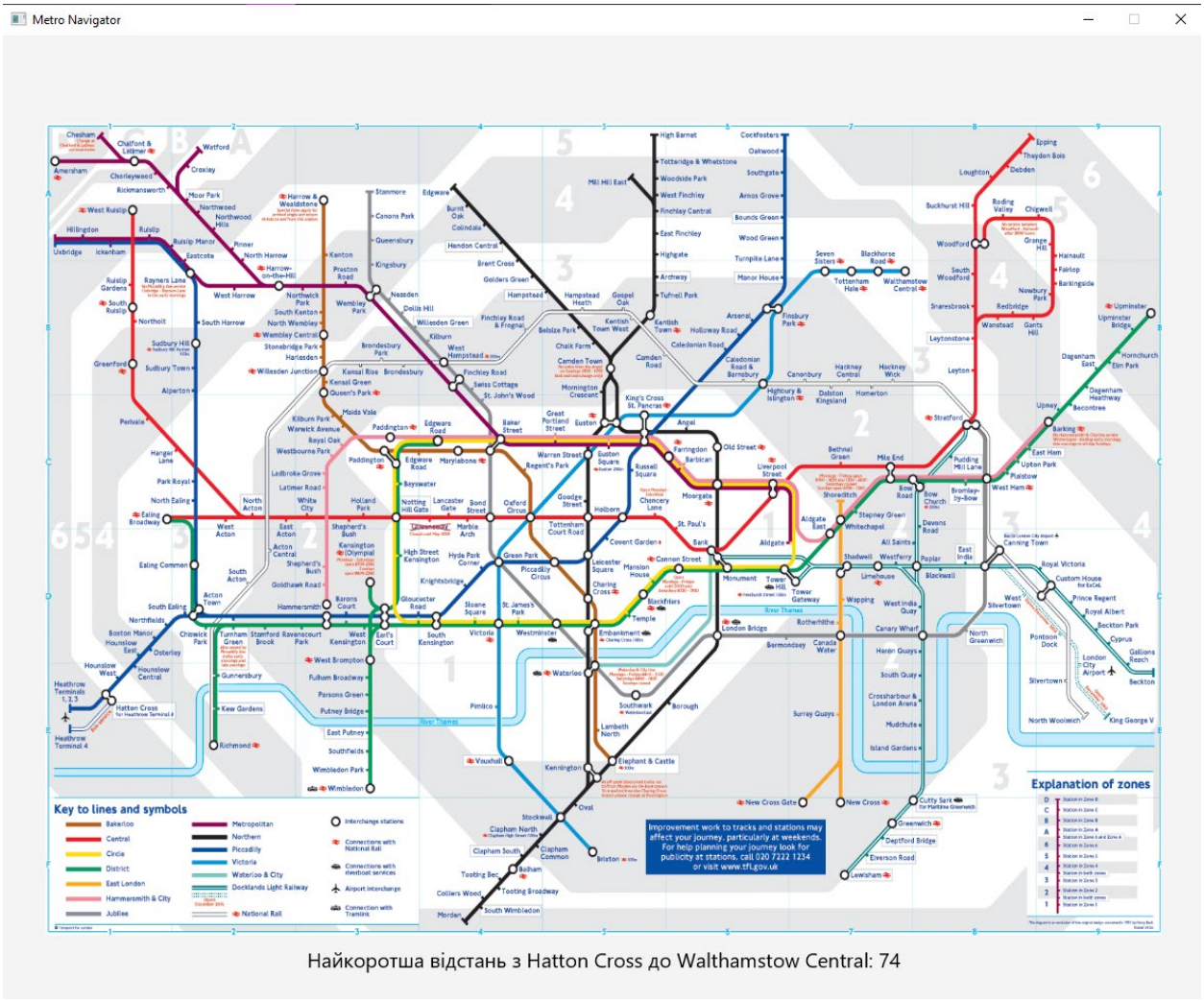

Рисунок 5.4. – Вивід вихідних даних для алгоритма Флойда для метро Лондона

Рисунок 5.5. – Вивід часу роботи для алгоритма Дейкстри для метро Києва

Рисунок 5.6. – Вивід часу роботи для алгоритма Дейкстри для метро Лондона

Рисунок 5.7. – Вивід часу роботи для алгоритма Флойда для метро Києва

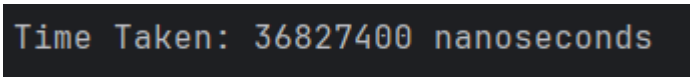

Рисунок 5.8. – Вивід часу роботи для алгоритма Флойда для метро Лондона

6. Аналіз отриманих результатів та висновки щодо досягнення мети РГР.

Виходячи з тестування двох алгоритмів маємо таку інформацію:

1. Час виконання.

- Найкоротша відстань з Ealing Broadway до Stratford: 65

Час затрачений алгоритмом Дейкстри становить: 7688500 наносекунд.

Час затрачений алгоритмом Флойда становить: 33232600 наносекунд.

- Найкоротша відстань з High Barnet до North Greenwich: 72

Час затрачений алгоритмом Дейкстри становить: 7370200 наносекунд.

Час затрачений алгоритмом Флойда становить: 35185300 наносекунд.

- Найкоротша відстань з Chesman до Baker Street: 50

Час затрачений алгоритмом Дейкстри становить: 7366900 наносекунд.

Час затрачений алгоритмом Флойда становить: 35942500 наносекунд.

- Найкоротша відстань з Alperton до Tottenham Cour Road: 51

Час затрачений алгоритмом Дейкстри становить: 8048500 наносекунд.

Час затрачений алгоритмом Флойда становить: 32672300 наносекунд.

- Найкоротша відстань з Bounds Green до Parsons Green: 57

Час затрачений алгоритмом Дейкстри становить: 7567200 наносекунд.

Час затрачений алгоритмом Флойда становить: 35584700 наносекунд.

- Найкоротша відстань з Wimbledon до Angel: 45

Час затрачений алгоритмом Дейкстри становить: 8059000 наносекунд.

Час затрачений алгоритмом Флойда становить: 30870200 наносекунд.

- Найкоротша відстань з Chigwell до Acton Town: 76

Час затрачений алгоритмом Дейкстри становить: 8415300 наносекунд.

Час затрачений алгоритмом Флойда становить: 30961100 наносекунд.

Отже , в середньому алгоритм Дейкстри в 4.3 рази швидше працює ніж алгоритм Флойда. Це викликано тим що алгоритму Флойда приходиться будувати матрицю для всіх станцій (на карті метро Лондона це приблизно 300 на 300), тому алгоритм Флойда виходить не ефективним по часу для знаходження найкоротшого шляху для пасажира метрополітену.

2. Використання пам’яті.

Алгоритм Флойда, хоча відомий своєю надійністю в знаходженні найкоротших шляхів, вимагає квадратичний обсяг пам'яті через зберігання усіх можливих шляхів між усіма парами вершин у графі. Це особливо стає проблемою при опрацюванні великих графів з значною кількістю вершин, що

може обмежити його застосування в сучасних системах з обмеженим обсягом пам'яті.

Навпаки, алгоритм Дейкстри, бувши оптимізованим, використовує лише лінійний обсяг пам'яті. Він зберігає лише інформацію про поточні найдешевші шляхи до кожної вершини, що робить його більш ефективним для великих графів і витісняє обмеження щодо обсягу пам'яті, яке може виникнути при використанні алгоритму Флойда.

3. Ефективність та зручність.

Алгоритми Флойда та Дейкстри є двома різними підходами до визначення найкоротшого шляху в графі, і кожен з них має свої переваги та недоліки, які залежать від конкретних умов застосування. В нашому випадку саме для визначення найкоротшого шляху для пасажира метрополітену, тобто в графі з великою кількістю вершин та ребер але без відє'мних ваг.

Алгоритм Дейкстри має в загальному квадратичну часову складність O(V^2), він працює надзвичайно ефективно на графах з малою та середньою кількістю вершин, та в графах з великою кількістю вершин в порівнянні з алгоритмом Флойда, який має кубічну часову складність O(V^3). До того ж через особливості алгоритму Флойда неможливо реалізувати вивід пройденого шляху , як це представлено в алгоритмі Дейскстри. Це є достатньо вагомими плюсом так як для пасажира метрополітену важливо знати по яким саме станціям потрібно пройти шлях.

Висновок: в результаті проведених тестуваннь та досліджень алгоритмів Дейкстри та Флойда для визначення найкоротших шляхів у метрополітені було розроблено програму для візуалізації даних та реалізовано програмно ці алгоритми на мові Java. Для даної теми алгоритм Дейкстри виявився більш ефективнішим по часовій складності, практично доведено що алгоритм Дейкстри працює швидше ніж алгоритм Флойда та вирачає менше пам'яті. Також Дейкстри надає змогу реалізувати вивід пройденого шляху по

станціях , що є вагомим плюсом. Але, слід зауважити , що алгоритм Флойда надає можливість працювати з графами з вагами ребер будь-якого знаку, включаючи ваги, які можуть бути від'ємними та Дозволяє знаходити оптимальні шляхи для всіх пар вершин після одного виконання, що надає практичну можливість знаходити зразу декілька найкоротших дистанцій між станціями. Після виконання РГР поставлена мета була досягнена.