Politechnika Wrocławska

Projekt

Zarządzanie w systemach i sieciach komputerowych

Analiza wpływu liczby procesów na czas działania algorytmów dla problemu komiwojażera

Authors:
Rafał Pieniążek
Jakub Pomykała

Supervisor:
Dr inż. Robert Wójcik

10 maja 2016

Spis treści

Spis rysunków

Spis tabel

1 Wstęp

1.1 Cel projektu

Celem projektu jest zbadanie wpływu zmiany ilości wątków na czas działania programów.

1.2 Zakres projektu

Projekt obejmuje implementację oraz zbadanie czasu działania algorytmów grafowych poszukiwania najkrótszej ścieżki pomiędzy wszystkimi wierzchołkami. Problem znany jest powszechnie jako problem komiwojażera.

2 Sformułowanie problemu

Problem komiwojażera jest problemem optymalizacyjnym, polegającym na znalezieniu ścieżki pomiędzy ustalonymi miastami dla następujących warunków:

- wszystkie miasta są odwiedzone dokładnie jeden raz
- rozpoczynamy i kończymy w tym samym mieście.
- koszt (suma wag krawędzi) jest najmniejszy z wszystkich możliwych

Oznacza to, że należy znaleźć taki cykl Hamiltona dla grafu reprezentującego zbiór miast, dla którego suma wag wybranych krawędzi jest najmniejsza.

2.1 Opis wariantów problemu

2.1.1 Przegląd zupełny

W algorytmie przeglądu zupełnego zastosowana jest taktyka brutalna, silna. Systematycznie sprawdzane są kolejne wartości funkcji długości drogi dla wszystkich możliwych permutacji wierzchołków. Zastosowany przez nas algorytm w sposób sprytny i zwinny, poprzez rekurencję zapewnia sprawdzenie wszystkich możliwości.

2.1.2 Branch&Bound

W rozwiązaniu powyższego problemu zastosowano metodę podziału i ograniczeń. Metoda ta jest metodą optymalizacji dyskretnej, opierając się na podejściu *dziel i zwyciężaj*. W każdym kroku algorytmu przeglądane jest drzewo potencjalnych rozwiązań. Jeżeli natrafimy na węzeł, który jest liściem, czyli można określić dla niego długość drogi komiwojażera,sprawdzamy, czy nowo znaleziona wartość nie jest lepsza od aktualnie zapisanej. Jeżeli tak jest, to zapamiętujemy nowe rozwiązanie. W przypadku, gdy dany węzeł nie jest liściem, tworzymy dla niego podproblemy. W tym celu odwiedzamy kolejne miasto starając się oszacować dolne ograniczenie kosztów całej trasy. W tym projekcie wykorzystano najprostszy sposób szacowania. Na początku z macierzy sąsiedztwa wycinana jest przekątna, następnie kolumny i wiersze miejsc już odwiedzonych. Następnie wartości z tak przygotowanej macierzy są sortowane w kolejności niemalejącej. Szacowanie polega na dodaniu do siebie tylu kolejnych wartości z listy, ile zostało miast do odwiedzenia. W niniejszym projekcie do przetrzymywania drzewa rozwiązań wykorzystano listę jednokierunkową. Wybór wynika z faktu, iż i tak należy przejrzeć wszystkie możliwe węzły, aby sprawdzić, czy ich ograniczenie nie jest większe niż aktualnie znalezione.

3 Projekt aplikacji

3.1 Wybrane klasy

3.1.1 klasa Matrix

Poniżej przedstawiono kod klasy matrix. Odzwierciedla ona macierz sąsiedztwa, w której przechowywane są informacje na temat grafu.

```
public class Matrix {

        private int[][] matrix;

        private int edgeCount;

        public Matrix(final int size) {
        edgeCount = size;
        matrix = new int[size][size];
        }

        public Matrix(int[][] matrix) {
        edgeCount = matrix.length;
        this.matrix = matrix;
        }

        public int getSize() {
        return edgeCount;
        }

}
```

3.1.2 klasa Edge

Klasa Edge pełni formę klasy pomocniczej podczas działania algorytmów. Przechowuje ona informację o konkretnej krawędzi, czyli wierzchołek początkowy i końcowy wraz z wagą.

```
public class Edge {

        public int startVertex;
        public int endVertex;
        public int weight = 0;

}
```

3.1.3 klasa Node

Klasa Node jest również wierzchołkiem, lecz nie grafu, ale drzewa przeszukiwania w algorytmie podziału i ograniczeń. Obiekty tej klasy są fundamentalne dla działania całości algorytmu.

```
public class Node {

        public int[][] matrix;

        public Edge[] solution;
        public int[] endEdges;

        public int added = 0;
        public float lowerBound;

        public Node(final int[][] matrix) {
        this.matrix = matrix;
        lowerBound = 0;
        }
```

```
}
```

3.2 Realizacja algorytmów wyznaczania rozwiązań

3.2.1 Przegląd zupełny

W algorytmie przeglądu zupełnego systematycznie sprawdzane są kolejne wartości funkcji długości drogi dla wszystkich możliwych permutacji wierzchołków. Zastosowany przez nas algorytm w sposób sprytny i zwinny, poprzez rekurencję, zapewnia sprawdzenie wszystkich możliwości. Najpierw sprawdzany jest warunek stopu rekurencji,czyli fakt, ze rozważamy permutację zbioru wszystkich możliwych elementów. Następnie wyliczana zostaje długość ścieżki w następujący sposób:

```
int  travelCosts = 0;
for (int i = 1; i < route.length; i++) {
        travelCosts += matrix.getWeight(route[i - 1], route[i]);
}
//powrot do miejsca startu
int n = matrix.getEdgeCount();
travelCosts += matrix.getWeight(route[n - 1], route[0]);
```

3.2.2 Branch&Bound

Algorytm Branch and Bound został zaimplementowany na podstawie algorytmu Little'a. Sukcesywnie przeglądane jest drzewo przeszukiwań dla kolejnych węzłów. Dla każdego poddrzewa obliczona zostaje wartość dolnego ograniczenia. Jeżeli wstępna wartość jest większa od dotychczas znalezionej wycinane zostaje całe poddrzewo rozważanych rozwiązań. W naszym algorytmie oszacowanie odbywa sie na podstawie minimalnych możliwych dróg pozostałych do rozważenia, przy czym nie brana jest pod uwagę kolejność sumowanych dróg.

3.3 Odczyt i zapis danych

3.3.1 Wczytywanie z plików .tsp

W celu analizy danych przetwarzanych zaimplementowany został mechanizm wczytywania danych z plików o rozszerzeniu *.tsp*

```
public class TSPFileParserImpl implements TSPFileParser {
FileReader fr = null;
BufferedReader bfr;
String tspFileExtension = ".tsp";

public List<Edge> parseFile(String fileName) {
        List<Edge> edgeList;
        openFile(fileName);
        cityList = readData();
        closeFile();
        return edgeList;
}

private List<Edge> readData() {
List<Edge> edgeList = new ArrayList<>();

Optional<String> readedLine = Optional.ofNullable(readLine());
while (readedLine.isPresent()) {
                String dataCheck = readedLine.get();
        if (isLineContainsData(dataCheck)) {
                Edge edge = getEdgeFromData(dataCheck);
                edgeList.add(edge);
        }
        readedLine = Optional.ofNullable(readLine());
}
return edgeList;
```

```
}

private Edge getEdgeFromData(String line) {

        Iterable<String> splittedData = splitData(line);

        if (isValid(splittedData)) {
        Iterator<String> stringSlitted = splittedData.iterator();
        int index = Integer.parseInt(stringSlitted.next());
        double x = Double.parseDouble(stringSlitted.next());
        double y = Double.parseDouble(stringSlitted.next());
        return new Edge(x, y);
        }
return null;
}
}
```

3.3.2 Generowanie macierzy losowo

Generowanie macierzy odbywa sie w klasie MatrixGeneratorSingleton:

```
public class MatrixGeneratorSingleton {

        private final static int INF = Integer.MAX_VALUE;
        private static MatrixGeneratorSingleton instance;

        private Random random;
        private int maxWeight;

        public MatrixGeneratorSingleton() {
                random = new Random();
                maxWeight = 100;
        }

        public static MatrixGeneratorSingleton getInstance() {

                if (instance == null) {
                        instance = new MatrixGeneratorSingleton();
                }
                return instance;
        }

        public Matrix generate(int size) {
                int[][] result = new int[size][size];
                for (int row = 0; row < size; row++) {
                        for (int col = 0; col < size; col++) {
                                if (row != col) {
                                        int value = random.nextInt(maxWeight) + 1;
                                        result[row][col] = value;
                                        result[col][row] = value;
                                }
                        }
                }
                for (int i = 0; i < size; i++) {
                result[i][i] = INF;
                }
                return new Matrix(result);
        }
}
```

3.3.3 Generwoanie plików cvs

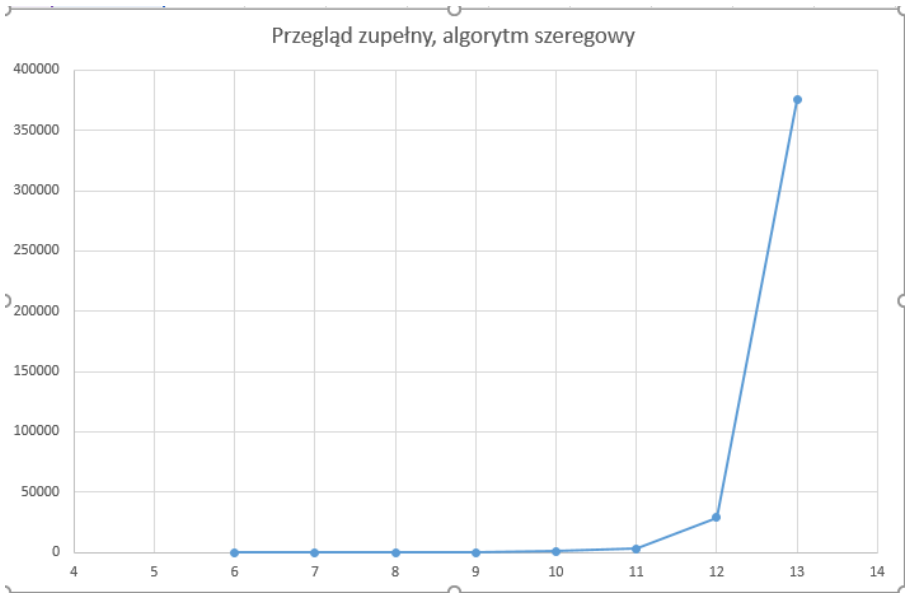
```
public class CSVGenerator {

String SEPARATOR = ";";

public String createFile(int generations, int population, boolean random) {
        String filename = getFilename(generations, population, random);
        try {
                FileWriter writer = new FileWriter(filename);

                writer.append("Nazwa instancji");
                writer.append(SEPARATOR);
                writer.append("Czas");
                writer.append(SEPARATOR);
                writer.append("Poczatkowa dlugosc");
                writer.append(SEPARATOR);
                writer.append("Koncowa dlugosc");
                writer.append('\n');

                writer.flush();
                writer.close();
        } catch (IOException e) {
                e.printStackTrace();
        }
        return filename;
}

public void addRecord(String outputfileName, String tspName, long time, int initialPath, in
try {
        FileWriter writer = new FileWriter(outputfileName, true);
        writer.append(tspName);
        writer.append(SEPARATOR);
        writer.append(String.valueOf(time));
        writer.append(SEPARATOR);
        writer.append(String.valueOf(initialPath));
        writer.append(SEPARATOR);
        writer.append(String.valueOf(finalPath));
        writer.append('\n');
        writer.flush();
        writer.close();
} catch (IOException e) {
        e.printStackTrace();
}
}
```

4 Testowanie wydajności

4.1 Czasy wykonania algorytmów

4.1.1 Czas działania algorytmu dla przeglądu zupełnego - wersja szeregowa

Pierwszym etapem było zbadanie czasu działania algorytmu podczas pracy w wersji szeregowej. Sukcesywnie uruchamiano algorytmy Na jednym wątku. Czas działania wzrastał wykładniczo. Efekt ten jest spodziewany, mianowicie przy każdym dodatkowym n- wierzchołku ilość obliczeń rośnie o n+1. Poniżej przedstawiona została tabela wraz z zebranymi wynikami:



Rysunek 1: Algorytm przeglądu zupełnego - szeregowo

Tabela 1: Uśrednione wyniki dla 8 różnych ilości wierzchołków

Ilość miast	czas działania algorytmu [ms]
6	0,08
7	0,5
8	3,2
9	25,2
10	765
11	2803
12	28798
13	375225

Czas działania algorytmu dla przeglądu zupełnego ma charakter wykładniczy. Jest to metoda z najgorszą możliwą złożonością, jednakże z pewnością daje dobre wyniki.

4.1.2 Czas działania algorytmu dla przeglądu zupełnego - wersja równoległa

Kolejnym krokiem było uruchomienie algorytmu przeglądu zupełnego na wielu wątkach. Dla każdego rozmiaru problemu został uruchomiony osobny wątek. Z racji charakteru tego rozwiązania niemożliwe jest badanie czasu dla osobnych problemów. Dlatego, zbadany czas jest wynikiem sumarycznym pracy algorytmu dla rozmiarów *10, 11, 12*. Czas działania wyniósł *29117ms*.

4.1.3 Porównanie czasów wykonywania algorytmu przeglądu zupełnego

Wykres przedstawiony poniżej pozwala zaobserwować różnice pomiędzy czasem wykonywania algorytmu zarówno dla metody szeregowej jak i równoległej.

Tabela 2: Porównanie czasu wykonywania algorytmów w wersji szeregowej i równoległej

Rodzaj algorytmu	czas działania [ms]
szeregowo	32366
równolegle	29117

Poniżej przedstawiono wykres ukazujący różnicę w czasach działania algorytmu. Warto zauważyć, że czas algorytmu uruchomionego równolegle jest zbliżony do pojedynczego uruchomienia algorytmu dla przypadku szeregowego w rozmiarze 12 wierzchołków. Oznacza to, że w czasie jaki algorytm szeregowy potrzebuje na obliczenie jedynie problemu dla 12 wierzchołków, wersja równoległa zdołała obliczyć rozwiązanie dla rozmiaru 10 oraz 11.

Rysunek 2: Przegląd zupełny-porównianie uruchomienia szeregowego i równoległego

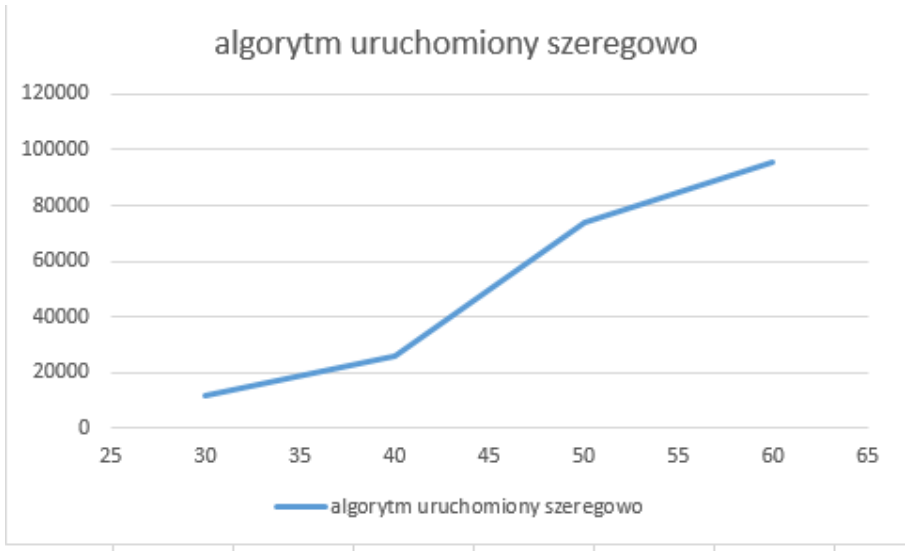
4.1.4 Czas działania algorytmu metodą podziału i ograniczeń - wersja szeregowa

Czas działania algorytmu w tym przypadku jest dużo bardziej przystępny. W poniższym zestawieniu zebrane są testy czasów dla algorytmu uruchomionego sekwencyjnie, na jednym wątku. Charakterystyka algorytmu pozwala, a właściwie zmusza do badania algorytmu dla większych danych testowych.

Tabela 3: Uśrednione wyniki dla różnych ilości wierzchołków

Ilość miast	czas działania algorytmu [ms]
30	11541
40	25827
50	73618
60	95695

Poniższy wykres przedstawia czas działania algorytmu dla metody podziału i ograniczeń. Jest to wykres zależności czasu działania algorytmu od ilość wierzchołków.



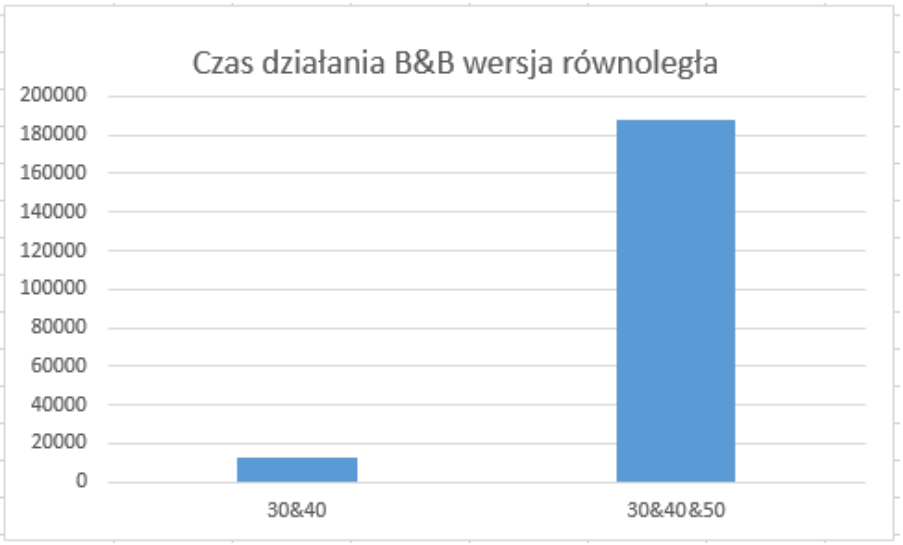
4.1.5 Czas działania algorytmu metodą podziału i ograniczeń - wersja równoległa

Podobnie jak w przypadku algorytmu przeglądu zupełnego tak i w tym przypadku charakterystyka sposobu wywoływania algorytmu nie pozwala na rozróżnienie czasu działania dla pojedynczych rozmiarów problemu. W tym przypadku algorytm został uruchomiony na kilku wątkach, po jednym dla każdego rozmiaru. Po zakończeniu działania wszystkich wątków został zmierzony czas. Sprawdzono czas działania algorytmu dla 30 i 40

wątków. Następnie dołożony został kolejny wątek liczący rozwiązanie dla 30, 40 i 50 wierzchołków.

Tabela 4: wyniki działania równoległego algorytmu komiwojażera

Ilość miast	czas działania algorytmu [ms]
30,40	12925
30,40,50	188394



4.1.6 Porównanie czasu działania algorytmu metodą podziału i ograniczeń

Poniższy wykres przedstawia różnicę pomiędzy czasami wykonywania dla algorytmu podziału i ograniczeń.

4.2 Analiza i ocena jakości

Przeanalizowanie jakości jest zadaniem nietrywialnym, lecz możliwym. Algorytm przeglądu zupełnego daje rozwiązanie optymalne zawsze. Kosztowne jest jednak sprawdzenie wszystkich możliwych rozwiązań. W wykresów przestawionych w tym dokumencie można z łatwością wywnioskować, że czas działania rośnie w sposób wykładniczy.Przy 15 wierzchołkach problem jest praktycznie nierozwiązywalny przez domowe komputery. Drugie podejście analizowanie w tym projekcie pozwala na znaczne zredukowanie czasu działania. Jednakże odbywa się to kosztem pamięci. Jest to spowodowane faktem przechowywania w pamięci drzewa przeszukiwań, które podczas pracy algorytmu rozrasta się bardzo szybko. Podczas badań czasu pracy algorytmu Java zwróciła błąd przepełnienia pamięci już dla 75 wierzchołków. Nie jest to wartość deterministyczna, ponieważ zależy od wygenerowanych danych wejściowych. Może okazać się, ze szybko znajdziemy minimum lokalne funkcji drogi i odrzucone zostaną rozwiązania na pewno nie optymalne.

4.3 Wnioski z testów i badań

Największą trudnością podczas badań było wstępne oszacowanie czasu pracy algorytmów. Jeżeli rozmiar problemu wejściowego był zbyt duży, czas oczekiwania na wyniki stawał się zbyt duże.

5 Podsumowanie

Wykonanie projektu pozwoliło uzmysłowić nam istotę dobrego planowania pracy podczas operacji na wielu wątkach. Najistotniejszymi problemami jest synchronizowany dostęp do zasobów podczas pracy wielu wątków. Zmierzyliśmy się z tym problem podczas mierzenia czasu pracy algorytmów. Zastosowanie odpowiednich technik i badań operacyjnych umożliwiło implementację algorytmów. Algorytmy rozwiązujące optymalizacyjny problem komiwojażera są tematem rozległym i nietrywialnym.

6 Bibliografia

References

[1] Thomas Cormen, *Wstęp do algorytmów*, Leiserson,Rivest, Stein 7nd edition, 2005.

[2] John Little, *An algorithm for the traveling salesman problem*, Little, John 1963.