POLITECHNIKA WROCŁAWSKA
WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI

KIERUNEK: INFORMATYKA (INF)
SPECJALNOŚĆ: INŻYNIERIA SYSTEMÓW INFORMATYCZNYCH (INS)

ZARZĄDZANIE W SYSTEMACH I SIECIACH KOMPUTEROWYCH
PROJEKT

Implementacja wielowątkowego algorytmu B&B
do znajdowania najkrótszej ścieżki w grafie

AUTORZY:

Paweł Maciończyk, 248837
Artur Sołtys, 248854

PROWADZĄCY PRACĘ:
dr inż. Robert Wójcik, $K_{30}W_{04N}D_{03}$

OCENA PRACY:

WROCŁAW, 2021

Spis treści

Spis rysunków

Spis listingów

Rozdział 1

Wstęp

1.1. Cel projektu

Celem projektu jest zbadanie wpływu liczby uruchomionych wątków na czas rozwiązywania danej ilości problemów.

1.2. Zakres projektu

Projekt będzie obejmować:

- zaprojektowanie i zaimplementowanie algorytmu Branch&Bound pozwalającego na poprawne rozwiązanie problemu komiwojażera,
- zrównoleglenie ww. algorytmu,
- pomiar czasów znalezienia rozwiązań w zależności od ilości problemów i liczby wątków użytych do poszukiwań,
- przygotowanie wykresów oraz analizę wyników,
- stworzenie dokumentacji projektu.

Rozdział 2

Sformułowanie problemu

2.1. Podstawowe założenia

Problem komiwojażera [1] (ang. Travelling Salesman Problem) to zadanie polegające na znalezieniu najkrótszej i najefektywniejszej trasy (o jak najmniejszym koszcie). Założenia są następujące:

- dany jest graf, w którym każdy wierzchołek połączony jest z pozostałymi wierzchołkami krawędzią o ustalonej wadze,
- należy odwiedzić każdy wierzchołek (miasto) dokładnie raz,
- warunkiem końcowym jest powrót do wierzchołka startowego.

2.2. Zastosowane algorytmy

Problem optymalizacyjny będzie rozwiązywany za pomocą metody podziału i ograniczeń (ang. *Branch and Bound*) [2]. Jest to metoda, która dzieli problem na kilka podproblemów. Opiera się ona na wykorzystaniu strategii przeszukiwania drzewa, aby niejawnie wyliczyć wszystkie możliwe rozwiązania danego problemu. Ponadto reguła obcinania pozwala na wyeliminowanie pewnych obszarów rozwiązań, które na pewno nie uzyskają lepszego kosztu ścieżki.

2.3. Analiza złożoności obliczeniowej algorytmów

Problem komiwojażera jest zaliczany do klasy problemów NP-trudnych [4]. Problemami klasy NP-trudnej nazywamy problemy obliczeniowe, dla których znalezienie rozwiązania problemu nie jest możliwe ze złożonością wielomianową oraz sprawdzenie rozwiązania problemu jest co najmniej tak trudne jak każdego innego problemu NP. Problemy NP-trudne obejmują zarówno problemy decyzyjne jak również problemy przeszukiwania, czy też problemy optymalizacyjne.
Natomiast algorytm Branch&Bound w literaturze [3] cechuje się następującą złożonością obliczeniową: $O(n^3 * ln^2(n))$.

Rozdział 3

Projekt aplikacji

3.1. Wykorzystane technologie i narzędzia projektowania

- Java - do zaimplementowania algorytmu użyto języka programowania Java w wersji 11.0.10.
- Intellij IDEA Ultimate - za środowisko programistyczne posłużyło oprogramowanie firmy JetBrains w wersji 2021.2.2.
- Maven - w celu automatyzacji budowy oprogramowania oraz pobrania biblioteki OpenCSV skorzystano z narzędzia Maven w wersji 3.6.3.
- OpenCSV - aby usprawnić proces przetwarzania danych pomiarowych skorzystano z biblioteki OpenCSV w wersji 5.5.2, która pozwoliła na zapisywanie wyników pomiarów w do plików .csv.
- Microsoft Excel - do opracowania wyników pomiarów oraz przygotowania wykresów użyto arkuszy kalkulacyjnych Microsoft Excel.
- LaTeX - do stworzenia dokumentacji użyto systemu do zautomatyzowanego składu tekstu LaTeX w przeglądarkowym edytorze Overleaf.

3.2. Struktura danych wejściowych

Dane do rozwiązania problemu komiwojażera wczytywane były z plików tekstowych. Pliki te w pierwszej linijce zawierały liczbę wierzchołków grafu n. Następne linijki zawierały macierz $n \times n$ w następującej postaci: w i-tym wierszu na j-tej pozycji przechowywana była waga krawędzi między wierzchołkami i, j. Wagi krawędzi były rozdzielone pojedynczym odstępem, a przekątna grafu wypełniona była zerami. Przykładową strukturę grafu 20-wierzchołkowego pokazano na rysunku 3.1.

```
20
0 30 56 67 64 22 4 57 5 20 47 54 26 63 41 34 21 0 52 59
39 0 50 11 86 81 19 16 19 53 60 26 97 83 22 14 89 66 98 87
98 73 0 87 8 22 86 73 50 76 40 85 10 90 44 95 97 92 98 27
54 9 97 0 28 71 16 77 54 91 63 83 10 82 15 77 19 58 44 79
98 61 7 46 0 70 72 28 59 61 91 6 71 46 13 23 10 96 26 69
28 98 70 35 87 0 20 69 88 5 97 68 41 37 14 10 56 94 51 27
52 94 36 40 47 49 0 90 33 54 52 94 27 12 79 18 44 45 35 66
39 56 6 38 55 16 25 0 9 53 90 8 28 70 63 73 56 2 72 67
28 65 18 47 76 57 72 63 0 86 46 62 82 66 41 4 24 36 79 58
74 24 65 42 44 25 96 32 42 0 65 83 61 9 49 55 82 72 49 38
2 46 54 21 9 4 19 68 96 89 0 88 68 6 57 37 47 36 99 18
8 36 43 96 68 77 92 47 21 27 86 0 1 32 49 51 7 96 75 58
93 77 42 43 11 38 38 56 17 40 51 85 0 52 63 66 16 43 77 21
95 4 48 28 1 80 84 46 70 99 54 44 2 0 96 47 77 21 40 84
42 85 65 89 76 60 81 93 75 23 68 21 32 42 0 87 68 2 14 62
90 1 56 83 10 83 71 32 13 99 56 34 40 47 28 0 87 25 86 6
11 78 34 50 1 90 99 90 3 17 40 88 87 77 45 76 0 76 17 69
50 41 54 41 38 55 96 89 75 25 95 34 74 0 41 28 68 0 40 34
17 50 10 22 99 75 43 27 94 58 44 59 65 53 15 39 34 9 0 58
5 43 32 13 38 39 68 50 12 29 26 99 94 35 94 87 53 39 96 0
```

Rys. 3.1: Zrzut ekranu tekstowej reprezentacji grafu o 20 wierzchołkach

Rozdział 4

Implementacja systemu

4.1. Wybrane klasy

Poniżej przedstawiono i omówiono klasy języka Java wykorzystane podczas implementacji algorytmu Branch&Bound.

4.1.1. Klasa Graph

Abstrakcyjna klasa `Graph` posiada zmienną `vertexAmount` przechowującą liczbę wierzchołków grafu oraz statyczną zmienną pomocniczną `INFINITY` przechowującą największą dodatnią liczbę całkowitą. Ponadto, klasa ta posiada jeszcze metodę `generateRandomFullGraph(...)`, która generuje pełny graf z krawędziami o losowych wagach obiektowi typu `ArrayGraph` przekazanemu jako parametr.

Listing 4.1: Fragment kodu źródłowego klasy Graph

```
public abstract class Graph {
    protected int vertexAmount;
    private static int INFINITY = Integer.MAX_VALUE;

    public int getVertexAmount() {
        return vertexAmount;
    }

    public void setVertexAmount(int vertexAmount) {
        this.vertexAmount = vertexAmount;
    }

    static void generateRandomFullGraph(ArrayGraph graph, int maxWeight) {
        Random random = new Random();
        for (int i = 0; i < graph.vertexAmount; i++) {
            for (int j = 0; j < graph.vertexAmount; j++) {
                if (i != j) {
                    int randomWeight = ((random.nextInt() % maxWeight) +
                        ↪ maxWeight + 1) % maxWeight;
```

```
                    graph.addEdge(i, j, randomWeight);
                }
            }
        }
    }
    ...
}
```

4.1.2. Klasa ArrayGraph

Klasa `ArrayGraph` rozszerza klasę `Graph`. Posiada ona zmienną o nazwie `neighborhoodMatrix`, która przechowuje macierz sąsiedztwa w postaci dwuwymiarowej tablicy z liczbami całkowitymi. Co więcej, klasa ta posiada metody do ustalania wag, pobierania wagi krawędzi oraz do wypisywania macierzy sąsiedztwa w konsoli.

Listing 4.2: Kod źródłowy klasy ArrayGraph

```
public class ArrayGraph extends Graph{
    private int[][] neighborhoodMatrix;

    public ArrayGraph(int vertexAmount) {
        this.vertexAmount = vertexAmount;
        this.neighborhoodMatrix = new int[vertexAmount][vertexAmount];

        for (int i = 0; i < vertexAmount; i++) {
            for (int j = 0; j < vertexAmount; j++) {
                    this.neighborhoodMatrix[i][j] = 0;
            }
        }
    }

    public boolean addEdge(int v, int w, int weight) {
        if (this.neighborhoodMatrix[v][w] > 0)
            return false;
        else {
            this.neighborhoodMatrix[v][w] = weight;
            return true;
        }
    }

    public int getWeight(int v, int w) {
        return neighborhoodMatrix[v][w];
    }

    public void displayGraph() {
        for (int i = 0; i < vertexAmount; i++) {
            for (int j = 0; j < vertexAmount; j++) {
                System.out.printf("%d ", this.neighborhoodMatrix[i][j]);
```

```
            }
            System.out.println();
        }
    }
}
```

4.1.3. Klasa ArrayListComparator

Do porównywania list przechowujących obiecujące rozwiązania, które są trzymane w kolejce priorytetowej utworzono klasę `ArrayListComparator` implementującą interfejs `Comparator`. Dzięki metodzie `compare(...)` kolejka priorytetowa będzie nadawała wyższy priorytet tym trasom, których koszt jest mniejszy.

Listing 4.3: Kod źródłowy klasy ArrayListComparator

```
public class ArrayListComparator implements Comparator<ArrayList<Integer>>
    ↪ {

    @Override
    public int compare(ArrayList<Integer> o1, ArrayList<Integer> o2) {
            return Integer.compare(o1.get(0), o2.get(0));
    }
}
```

4.2. Realizacja algorytmu

Algorytm widoczny na listingu 4.4 został zaimplementowany w oparciu o strategię najpierw najlepszy. Podstawowa różnica w stosunku do strategii przeszukiwania wszerz polega na obecności kolejki priorytetowej, w której priorytetem jest wartość granicy wierzchołka. Dzięki tej kolejce zawsze przeglądane będą w pierwszej kolejności węzły z najniższą granicą. Po inicjalizacji kolejki oraz pomocniczych zmiennych algorytm przegląda kolejne węzły o najlepszych granicach. Dla każdego odwiedzanego węzła sprawdzane jest początkowo, czy wciąż jest obiecujący. Zapobiega to sytuacji wydłużania obliczeń w algorytmie typu przeszukiwania wszerz, gdyż jeśli węzeł przestanie być obiecujący to nie będzie on dalej rozwijany. Wymusza to jednak konieczność zapamiętania wartości granicy dla każdego węzła wrzuconego do kolejki. W dalszej części pętli następuje przeglądanie potomków odwiedzanego wierzchołka, aktualizując w razie potrzeby informacje o najlepszym rozwiązaniu i dodając obiecujących potomków do kolejki priorytetowej[5].

Listing 4.4: Fragment kodu źródłowego implementacji algorytmu Branch&Bound

```
static ArrayList<Integer> ATSPBranchAndBound(ArrayGraph graph) {
    PriorityQueue<ArrayList<Integer>> priorityQueue = new PriorityQueue<>(
        ↪ new ArrayListComparator());

    ArrayList<Integer> optimalRoute = new ArrayList<Integer>();
    int optimalRouteLength = INFINITY;


    ArrayList<Integer> currentRoute = new ArrayList<Integer>();
    currentRoute.add(0);
    currentRoute.add(0);
    priorityQueue.add(currentRoute);

    while (!priorityQueue.isEmpty()) {
        currentRoute = priorityQueue.poll();

        if (optimalRouteLength == INFINITY || currentRoute.get(0) <
            ↪ optimalRouteLength) {
            for (int i = 0; i < graph.vertexAmount; i++) {

                ...

                ArrayList<Integer> nextRoute = new ArrayList<>(currentRoute
                    ↪ );
                nextRoute.add(i);

                if (nextRoute.size() > graph.vertexAmount) {
                    nextRoute.add(0);
                    nextRoute.set(0, 0);

                    for (int j = 1; j < nextRoute.size() - 1; j++) {
                        nextRoute.set(0, nextRoute.get(0) + graph.getWeight
                            ↪ (nextRoute.get(j), nextRoute.get(j + 1)));
                    }
                    if (optimalRouteLength == INFINITY || nextRoute.get(0)
                        ↪ < optimalRouteLength) {
                        optimalRouteLength = nextRoute.get(0);
                        nextRoute.remove(0);
                        optimalRoute = nextRoute;
                    }
                } else {
                    nextRoute.set(0, 0);
                    for (int j = 1; j < nextRoute.size() - 1; j++) {
                        nextRoute.set(0, nextRoute.get(0) + graph.getWeight
                            ↪ (nextRoute.get(j), nextRoute.get(j + 1)));
                    }
                    for (int j = 1; j < graph.vertexAmount; j++) {
```

```
                        ...

                        int minEdge = -1;
                        for (int k = 0; k < graph.vertexAmount; k++) {
                            if (j == i && k == 0)
                                continue;

                            ...

                            int consideredLength = graph.getWeight(j, k);

                            if (minEdge == -1)
                                minEdge = consideredLength;
                            else if (minEdge > consideredLength)
                                minEdge = consideredLength;
                        }
                        nextRoute.set(0, nextRoute.get(0) + minEdge);
                    }

                    if (optimalRouteLength == INFINITY || nextRoute.get(0)
                        ↪ < optimalRouteLength) {
                        priorityQueue.add(nextRoute);
                    }
                }
            }
        } else {
            break;
        }
    }
    return optimalRoute;
}
```

4.3. Metoda odczytu danych wejściowych

Implementacja metody pozwalającej wczytać macierz sąsiedztwa do programu na podstawie odpowiednio przygotowanego pliku *.txt* przedstawiono na listingu 4.5:

Listing 4.5: Metoda pozwalająca na wczytanie grafu z pliku *.txt*

```
private static ArrayGraph getArrayGraph(String filename) throws
    ↪ FileNotFoundException {

    File file = new File(filename);
    Scanner scanner = new Scanner(file);
    int firstLine = scanner.nextInt();
    ArrayGraph graph = new ArrayGraph(firstLine);
```

```
    for (int i = 0; i < firstLine; i++) {
        for (int j = 0; j < firstLine; j++) {
            graph.addEdge(i, j, scanner.nextInt());
        }
    }
    return graph;
}
```

4.4. Metoda prezentacji i zapisu wyników

Wyniki otrzymywane podczas wykonywania się algorytmu były gromadzone w plikach *.csv* w postaci czasu znajdywania rozwiązania problemów wyrażonego w *ms*.

Do zaimplementowania metody zapisującej wyniki wykorzystano klasy z biblioteki OpenCSV, które pozwalają na szybką serializację i generowanie plików o rozszerzeniu *.csv*.

Rozdział 5

Testowanie poprawności i ocena rozwiązań

5.1. Weryfikacja poprawności działania algorytmu

Po zaimplementowaniu algorytmu Branch&Bound przystąpiono do sprawdzenia poprawności jego działania. W tym celu postanowiono użyć danych wejściowych, dla których znane jest optymalne rozwiązanie.
Testy wykonano z wykorzystaniem zestawów danych, które zostały zaczerpnięte ze strony `https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html`. Za każdym razem otrzymano wyniki zgodne z wartościami oczekiwanymi.

5.2. Sposób testowania

Wykonując testy zaimplementowanego algorytmu kilkukrotnie uruchomiono funkcję odpowiadającą za generowanie wyników, zmieniając tylko jej parametr *filename* (nazwa pliku z danymi wejściowymi).
W tym celu przygotowano pliki z danymi wejściowymi dla każdego z wymienionego poniżej problemu:

- mały problem *(tsp_17.txt)* - rozwiązanie zajmuje ok. 0.3 sekundy,
- średni problem *(tsp_20.txt)* - rozwiązanie zajmuje ok. 8 sekund,
- duży problem *(tsp_23.txt)* - rozwiązanie zajmuje ok. 30 sekund.

Początkowo planowano dla każdego problemu przeprowadzić testy na liczbie wątków równej: 2, 3, 4, 5, 8, 10, 25, 50, 100. Ostatecznie udało się to osiągnąć tylko dla małego problemu. Wynika to z faktu, że dla większych problemów potrzebne były o wiele większe zasoby pamięci. Pomimo zmian konfiguracji programu za pomocą flagi `-Xmx4096m` większa liczba wątków wciąż potrzebowała jeszcze więcej pamięci.

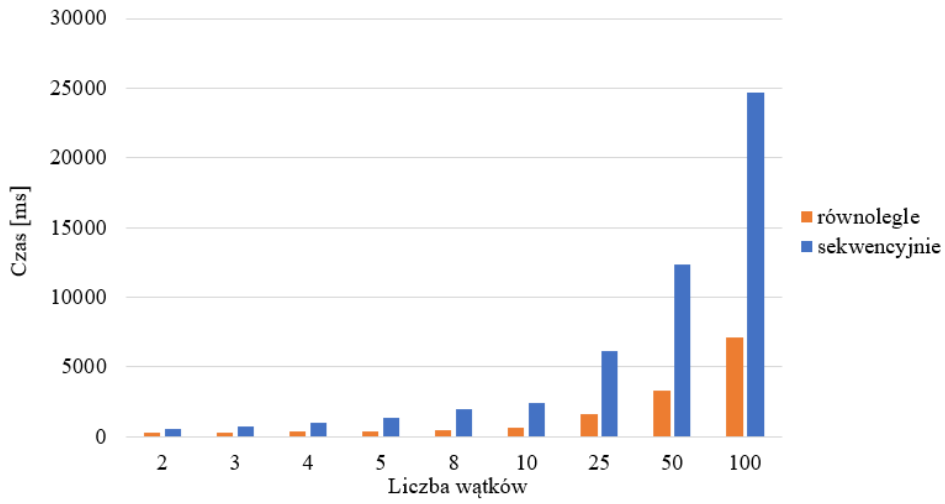
Każda konfiguracja poziomu trudności oraz liczby wątków została przetestowana zarówno w podejściu sekwencyjnym, jak i równoległym.

Ponadto w celu uśrednienia otrzymanych wyników, dla małych problemów przeprowadzono 10 powtórzeń, dla średnich - 5 oraz dla dużych - 2.

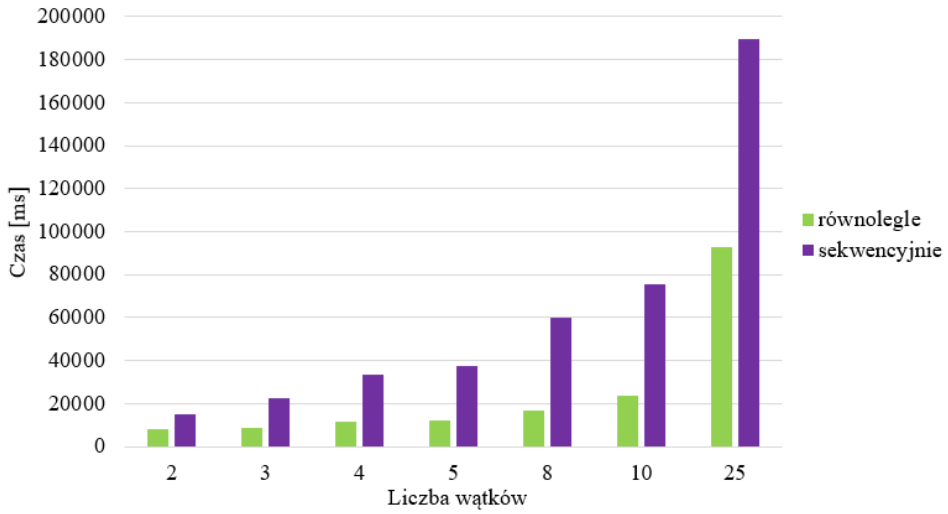
5.3. Wyniki testów

Na podstawie otrzymanych wyników przygotowano dwa rodzaje wykresów:

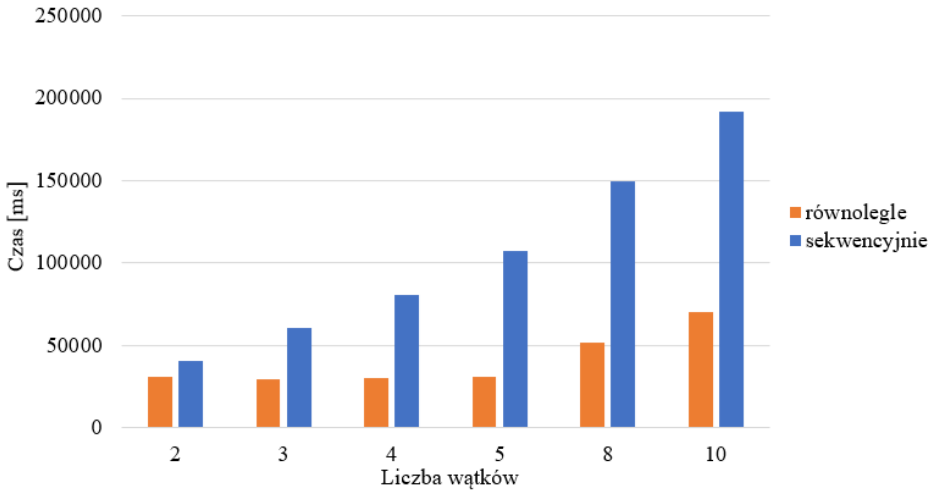
- Wykres zależności czasu (potrzebnego na rozwiązanie problemu o danej trudności) od liczby wątków - na wykresie jedna seria danych odpowiada szeregowemu wykonywaniu się zadań, natomiast druga - równoległemu;
- Wykres przedstawiający stosunek czasu rozwiązywania problemu o danej trudności szeregowo do czasu rozwiązywania go równolegle.



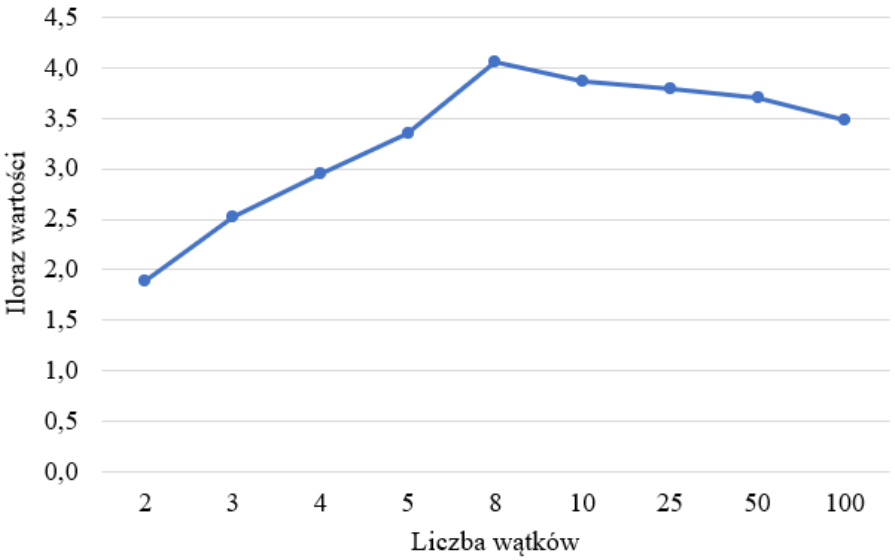
Rys. 5.1: Wykres czasu potrzebny na rozwiązanie problemów o małym rozmiarze w zależności od liczby wątków



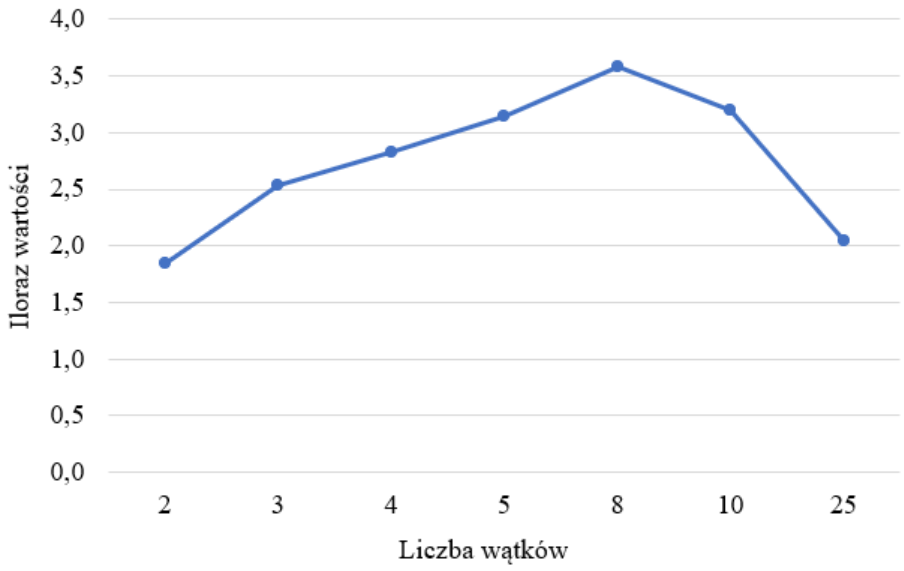
Rys. 5.2: Wykres czasu potrzebny na rozwiązanie problemów o średnim rozmiarze w zależności od liczby wątków



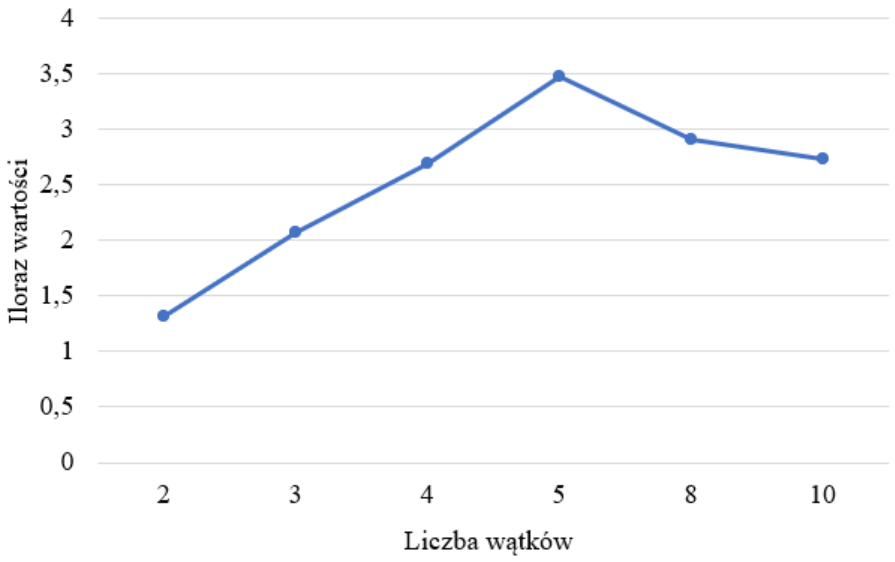
Rys. 5.3: Wykres czasu potrzebny na rozwiązanie problemów o dużym rozmiarze w zależności od liczby wątków



Rys. 5.4: Stosunek czasu rozwiązania małego problemu sekwencyjnie do czasu rozwiązywania go równolegle



Rys. 5.5: Stosunek czasu rozwiązania średniego problemu sekwencyjnie do czasu rozwiązywania go równolegle



Rys. 5.6: Stosunek czasu rozwiązania dużego problemu sekwencyjnie do czasu rozwiązywania go równolegle

5.4. Wnioski z testów i badań

Na podstawie wykresów przedstawionych na rysunkach [5.1-5.3] można zauważyć, że wprowadzenie zrównoleglenia wykonywanych algorytmów za każdym razem (niezależnie od liczby wątków/problemów) powodowało zmniejszenie czasu poszukiwania rozwiązania.

Natomiast analizując dane zamieszczone na wykresach [5.4-5.6] można dostrzec, jaką wartość ma rzeczywiste przyspieszenie związane z zastosowaniem podejścia równoległego zamiast sekwencyjnego. W idealnych warunkach taki iloraz powinien być równy liczbie wątków, na których uruchomiony został zaimplementowany algorytm. W rzeczywistości, uwzględniając czas potrzebny na przełączanie się wątków, zasoby pamięci oraz uruchomione na komputerze różne inne programy, widoczny rezultat jest mniejszy.

Ponadto zauważyć można, że ciągłe zwiększanie liczby wątków nie powoduje wzrostu wartości przyspieszenia działania programu. Najbardziej optymalną liczbą wątków wydaje się być 8 - dla problemów małych i średnich oraz 5 - dla problemów dużych. Zwiększanie liczby wątków ponad te wartości powoduje jedynie zmniejszenie wartości ilorazu czasu rozwiązywania danego problemu sekwencyjnie do czasu rozwiązywania go równolegle. Występowanie takiego efektu wynika z częstej potrzeby przełączania się i przydzielania zasobów procesora dla liczby wątków większej od liczby jednostek logicznych.

Rozdział 6

Podsumowanie

Celem projektu było zbadanie wpływu liczby uruchomionych wątków na czas rozwiązywania danej ilości problemów komiwojażera. W ramach projektu zaimplementowany został algorytm Branch&Bound do znajdowania najkrótszej ścieżki w grafie. Początkowo planowano zrównoleglić poszczególne kroki algorytmu (pętle, generowanie przestrzeni rozwiązań). Ostatecznie nie udało się tego zrealizować wewnątrz samego algorytmu, więc utworzono funkcje realizujące algorytm w sposób sekwencyjny oraz równoległy.

Przeprowadzone testy porównujące czasy działania algorytmu wykonywanego sekwencyjnie oraz równolegle pozwoliły zrozumieć, jak w prosty sposób można osiągnąć widoczne przyspieszenie działania programów. Nabyta w tym zakresie wiedza pozwoliła autorom zastosować podział na wątki w aplikacjach realizowanych w ramach pracy dyplomowej.

Literatura

[1] Problem komiwojażera. http://algorytmy.ency.pl/artykul/problem_komiwojazera. Dostęp: 09.12.2021 r.

[2] S. J. Radosław Grymin. Fast Branch and Bound Algorithm for the Travelling Salesman Problem. http://radoslaw.grymin.staff.iiar.pwr.edu.pl/publications/BandB.pdf. Dostęp: 09.12.2021 r.

[3] D. R. Smith. On the Computational Complexity of Branch and Bound Search Strategies. https://apps.dtic.mil/dtic/tr/fulltext/u2/a081608.pdf, listopad 1979. Dostęp: 09.12.2021 r.

[4] P. Szawdyński. Klasy problemów NP. https://cpp0x.pl/kursy/Teoria-w-Informatyce/Zlozonosc-obliczeniowa/Klasy-problemow-NP/430. Dostęp: 09.12.2021 r.

[5] M. Łyczek. Metoda podziału i ograniczeń. https://www.ii.uni.wroc.pl/ prz/2011lato/ah/ opracowania/met_podz_ogr.opr.pdf, 16 marca 2011 r. Dostęp: 09.12.2021 r.