Kamil Pawelski 263795 Wałbrzych, 23.10.2023

Sprawozdanie 1
Projektowanie Efektywnych Algorytmów

1. Wstęp teoretyczny

Problem komiwojażera, problem w którym celem jest znalezienie jak najkrótszej drogi pomiędzy wszystkimi miastami. Każde miasto musi być odwiedzone dokładnie jeden raz oraz musimy powrócić do miasta, z którego rozpoczęła się podróż. Inaczej mówiąc problem polega na znalezieniu najmniejszego cyklu Hamiltona w grafie. W projekcie rozważany jest asymetryczny problem komiwojażera (ATSP), który polega na tym że droga z miasta A do miasta B jest innej długości niż droga z miasta B do miasta A. W symetrycznym obie te drogi będą takie same.

2. Złożoność obliczeniowa

Pierwszy stosowany algorytm przegląd zupełny będzie sprawdzał każdą dostępna możliwość. Będziemy mieli cztery miasta musimy z któregoś zacząć. Wybieramy jeden wybraliśmy z pośród tych czterech. Teraz z tego miasta mamy trzy różne drogi do wyboru. Znowu zostanie wybrana najkrótsza. I tak dalej, aż do powrotu. Zauważyć się da, że pierw mamy cztery drogi do wybory, następnie trzy, dwie i jedną. W takim razie złożoność obliczeniowa danego algorytmu to O(n!) gdzie n jest liczbą miast. Łatwo sobie wyobrazić, że dla coraz większej ilości miast, przegląd zupełny będzie coraz mniej efektywny, a nawet można powiedzieć, że od pewnej ilości będą one nierozwiązywalne.

3. Opis działania algorytmu

W programie zostały zastosowane cztery klasy: Menu, BruteForceSearch, RandomOrFileMatrix i Matrix. Klasa Matrix reprezentuje graf w postaci macierzy. Klasa RandomOrFileMatrix umożliwia nam utworzenie obiektu Matrix na dwa sposoby. Pierwszy sposób czyli wygenerowanie własnej macierzy o wybranym rozmiarze. Drugi to wczytanie jest ze specjalnie przygotowanych plików. Klasa BruteForceSearch z niej znajduję się cały algorytm przeglądu zupełnego. Główne metody wykonujące algorytm w programie to

Listing 3.1: Metoda Search

```
var stopwatch = new Stopwatch();
stopwatch.Start();
do
{
  int currentPathLen = CalculatePathLength();

  if (currentPathLen < _minPathLength)
  {
    _minPathLength = currentPathLen;
   Array.Copy(_permutation,
_bestPermutation,
    _permutation.Length);
  }

  if (IsLastPermutation()) break;
    GenerateNextPermutation();
} while (true);
stopowatch.Stop();
```

Listing 3.2: Metoda GenerateNextPermutation

```
int swapStart = -1;
for (int i = _permutation.Length - 2; i >= 0; i--)
{
  if (_permutation[i] < _permutation[i + 1])
  {
    swapStart = i;
    break;
  }
}
if (swapStart == -1) return;

int swapWith = -1;
for (int i = _permutation.Length - 1; i > swapStart; i--
)
{
  if (_permutation[i] > _permutation[swapStart])
  {
    swapWith = i;
    break;
  }
}
Swap(ref _permutation[swpaStart], ref
_permutation[swapWith])
Array.Reverse(_permutation, swapStart + 1,
permutation.Length – swapStart – 1);
```

Search (Listing 3.1) oraz GenerateNextPermutation (Listing 3.2). Są również dwie metody IsLastPermutation sprawdza czy to już ostatnia permutacja oraz CalculatePathLength, ale skupię się na tych dwóch. Na listingu 3.1 został pokazany fragment metody Search. Algorytm rozpoczyna się od obliczenia długości ścieżki. Jeśli ścieżka będzie krótsza to nastąpi przekopiowanie z tablicy _permutation do _bestPermutation oraz weźmie obecną ścieżkę za najkrótszą. Następnie sprawdzi czy jest to ostatnia permutacja jeśli nie zacznie generować kolejną. Przechodzimy więc do listingu 3.2 metody GenerateNextPermutation. W pierwszej pętli szukamy elementu iterując od końca, który będzie mniejszy od następnego. Po znalezieniu takiego elementu, szukać będziemy elementu większego od znalezionego wcześniej. Po odnalezieniu zamieniane są miejscami. Na sam koniec zamienia kolejność elementów po znalezionym pierwszym elemencie. Tak ogólnie mówiąc działa przedstawiony algorytm w programie. Ostatnia klasa Menu pozwala nam poruszać się w prosty sposób po programie wybierając wybrane opcję.

4. Plan eksperymentu

Do wykonania eksperymentu zastosujemy różne wymiary macierz, aby sprawdzić wydajność algorytmu przeglądu zupełnego. Każdy test zostanie przeprowadzony 100 razy, a ich czas zostanie uśredniony. Macierze, które zostaną wykorzystane:

- 6x6
- 7x7
- 8x8
- 9x9
- 10x10
- 11x11
- 12x12

Dane generowane są przy pomocy metody GenerateRandomMatrix (Listing 4.1).

Listing 4.1: Metoda GenerateRandomMatrix

```
public Matrix GenerateRandomMatrix(int dimension)
{
      int[,] matrix = new int[dimension, dimension];
       Random random = new Random();
       _fileMatrixName = "NAME: Wygenerowany";
       _type = "TYPE: ATSP";
       _comment = "COMMENT:  Asymmetric TSP (Fischetti) for DIMENSION " + dimension;
       _dimension = dimension;
       for (int i = 0; i < dimension; i++)
        {
          for (int j = 0; j < dimension; j++)
           {
              if (i == j)
              {
                 matrix[i, j] = -1;
              }
              else
              {
                 matrix[i, j] = random.Next(1, 1001);
               }
           }
       }
       return new Matrix(dimension, matrix);
}
```

Do wygenerowania danych używany jest objekt Random, który generuje nam losowo od 1 do 1000. W wypadku gdy trafią się dwa takie same wierzchołki ustawiane jest na -1.

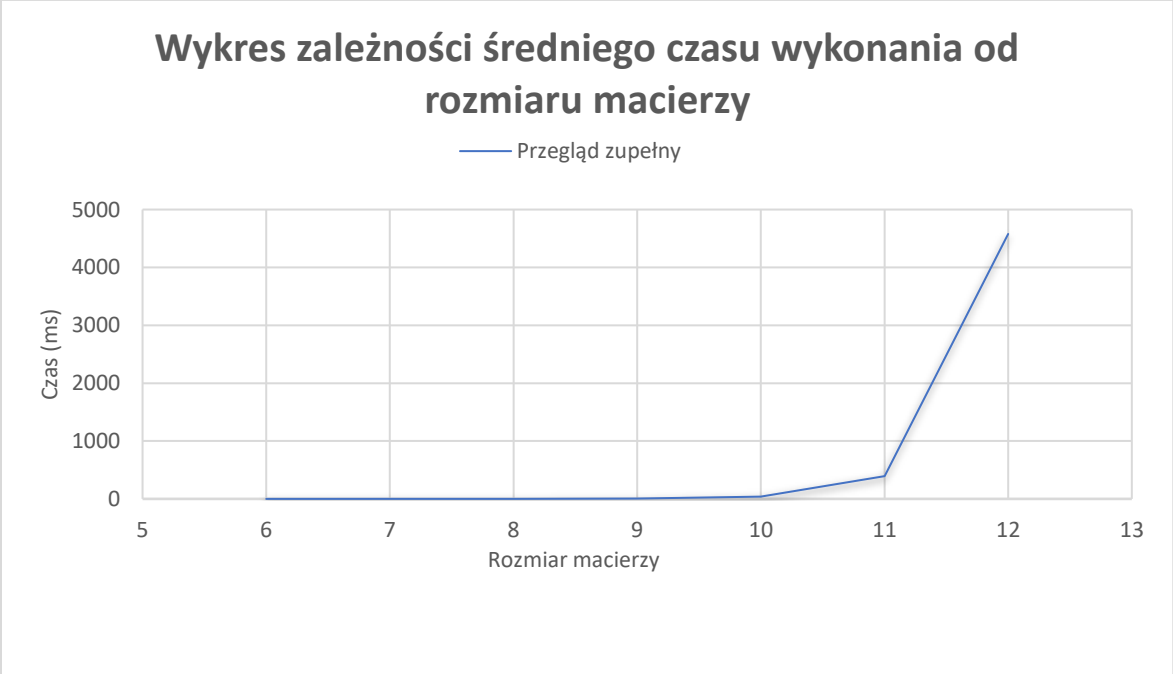
Do pomiaru czasu tworzymy obiekt Stopwatch z przestrzeni nazw System.Diagnostics. Obiekt inicjalizowany i uruchamiany jest przed rozpoczęciem pętli w metodzie Search (Listing 3.1) oraz kończy obliczanie czasu po zakończeniu tej pętli. Następnie wyniki zostają zapisane w pliku csv.

5. Wyniki eksperymentu

Tabela 5.1: Wyniki średnich czasów wykonania dla zadanych rozmiarów

Rozmiar macierzy	Średni czas wykonania (ms)
6x6	0,0134
7x7	0,0760
8x8	0,4971
9x9	4,1481
10x10	37,7291
11x11	393,8360
12x12	4578,4953

Wykres 5.2: Wykres zależności średniego czasu wykonania od rozmiaru macierzy.



6. Wnioski

Na podstawie wyników otrzymanych w wyniku testów oraz przedstawionych w tabeli 5.1 oraz na wykresie 5.2 można dojść do wniosków, że przy coraz większym rozmiarze macierzy czas zaczyna znacząco rosnąć. Zaczynaliśmy od macierzy 6x6, której czas wynosi 0,0134 ms. Dla ostatniego testu w przybliżeniu to już 4,6 sekundy. Potwierdza to, że dla coraz większych rozmiarów algorytm staję się bardzo nieefektywny. Potwierdza to tylko złożoność zastosowanego algorytmu O(n!). Można uznać, że algorytm nadaję się wyłącznie do małych rozmiarów.

7. Bibliografia

http://algorytmy.ency.pl/artykul/problem_komiwojazera

https://pl.wikipedia.org/wiki/Problem_komiwoja%C5%BCera