Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *PN 17:05*
Temat *Metoda podziału i ograniczeń i przegląd zupełny.*	**Problem** *TSP*
Skład grupy *241311 Dominik Maroszczyk*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *4 grudnia 2019*

1 Opis problemu

Problem komiwojażera (ang. Traveling Salesman Problem, TSP) jest jednym z najpopularniejszych oraz najczęściej rozważanych problemów optymalizacyjnych. Zadaniem komiwojażera jest znalezienie drogi o minimalnym koszcie, przemieszczając się z jednego miasta, odwiedzając wszystkie pozostałe tylko raz. Końcowym punktem musi być ten, z którego wyruszył. Z matematycznego punku widzenia problem sprowadza się do poszukiwania minimalnego cyklu Hamiltona w grafie pełnym, w którym każda krawędź ma określoną wagę. TPS należy do klasy problemów NP-trudnych. W realizowanym projekcie rozwiązujemy dwa warianty: problem symetryczny, problem asymetryczny. W pierwszym z nich odległość miedzy miastami (punktami) nie różni się od tego w którą stronę podróżujemy. Z miasta 1 -> 2 czy 2 -> 1. W problemie asymetrycznym ścieżki mogą mieć różne długości (wagi). Asymetryczność trasy w świecie rzeczywistym można przedstawić za pomocą drogi jednokierunkowej. Jest to bardzo prosty jak i intuicyjny sposób wyobrażenia problemu asymetrycznego.

2 Metoda rozwiązania

W pierwszym kroku, aby ułatwić pracę nad algorytmami zostały utworzone klasy *Matrix* oraz *Path*. Zaimplementowane metody m.in. wczytania problemu z pliku, tworzenie losowej ścieżki, liczenia dystansu między miastami.

```
1  class Matrix {
2  private:
3    std::string fileName = "";
4    std::string oldFileName = "";
5    int cost = 0;
6    int matrixSize = 0;
7    int **matrix = nullptr;
8
9  public:
10   ~Matrix();
11   void loadFromFile(std::string fileName);
12   void showMatrix();
13   void showMatrix(int **tab, int size);
14   int countPath(int tab[], int size);
15   int **getNewReducedMatrix(int **matrix, int matrixSize, int row =
        0,
16                             int col = 0);
17
18   inline int getSize() { return this->matrixSize; };
19   inline std::string getFileName() { return this->fileName; };
20   inline int **getMatrix() { return this->matrix; };
21   inline void setMatrix(int **tab) { this->matrix = tab; };
22 };
```

Listing 1: Klasa Matrix

Najdłuższa ścieżka przechodząca przez wszystkie miasta ma długość n. Komiwojażer musi powrócić do początkowego miasta, zatem aby policzyć jej długość należy dodać odległość miedzy ostatnim a pierwszym miastem.

```
int Matrix::countPath(int tab[], int size) {
  int sum = 0;
  for (size_t i = 0; i < size - 1; i++)
    sum += this->matrix[tab[i]][tab[i + 1]];

  if (size == this->matrixSize)
    sum += this->matrix[tab[size - 1]][tab[0]];

  return sum;
}
```

Listing 2: Metoda licząca odległość podanej ścieżki

```
int *Path::randomPath(int pathLength, int min, int max) {
  srand(unsigned(time(0)));
  if ((max - min) < pathLength && min >= max)
    throw std::length_error("Path length out of range");

  std::vector<int> temp;
  for (size_t i = min; i <= max; i++)
    temp.push_back(i);

  random_shuffle(temp.begin(), temp.end());
  pathWay = new int[pathLength];
  std::copy(temp.begin(), temp.begin() + pathLength, pathWay);
  return pathWay;
}
```

Listing 3: Metoda klasy Path zwracająca losową ścieżkę

Jedną z kluczowych struktur do wykonywania eksperymentów obliczeniowych jest klasa *TimeCounter*. Pozwala ona na obliczanie czasu wykonywania danej części kodu. Czas mierzony jest w sekundach.

```
#include <chrono>
#include <iostream>
using namespace std::chrono;

class TimeCounter {
private:
  time_point<high_resolution_clock> start;
  time_point<high_resolution_clock> stop;
public:
  inline void startTimer() {
```

```
11      this->start = high_resolution_clock::now();
12    }
13    inline void stopTimer() {
14      this->stop = high_resolution_clock::now();
15    }
16    inline double getElapsedTime() {
17      duration<double> elapsed = this->stop - this->start;
18      return elapsed.count();
19    }
20  };
```

Listing 4: Klasa TimeCounter

2.1 Brute Force

Algorytmem zaimplementowanym jako pierwszy jest Brute Force. Charakteryzuje się dużą złożonością obliczeniową, ponieważ jest to przegląd zupełny. Wszystkie możliwe kombinacje są sprawdzane, co wiąże się z dużą ilością operacji do wykonania i w ostateczności czas rozwiązania jest długi. Nie jest to wydajny algorytm. Dla dużych instancji problemu znalezienie rozwiązania jest pochłaniającym czas procesem.

```
1  void BruteForce::heapPermutation(int tab[], int k, int startSize) {
2    if (k == 1) {
3      // wykonywanie czynnosci liczace dlugosc sciezki
4      // oraz sprawdzajace czy jest to najkrotsza droga
5    }
6
7    for (int i = 0; i < k; i++) {
8      heapPermutation(tab, k - 1, startSize);
9
10     if (k % 2 == 1)
11       std::swap(tab[0], tab[k - 1]);
12     else
13       std::swap(tab[i], tab[k - 1]);
14   }
15 }
```

Listing 5: Metoda generująca wszytkie możliwe kombinacje ścieżki

Algorytm został zaimplementowany bazując na jednej z popularniejszych metod. Wykorzystywana jest funckja *std::swap()*, zamienia ona kolejnością dwie podane w parametrach dane.

2.2 Branch and Bound

Drugim algorytmem jest Branch And Bound. Jest rozwiązaniem, które jest ogólnie stosowane do rozwiązywania problemów optymalizacji kombinatorycznej. Problemy te są zwykle wykładnicze pod względem złożoności czasowej i mogą wymagać zbadania wszystkich możliwych permutacji w najgorszym przypadku jak w poprzednim przypadku. Technika algorytmu Branch And Bound rozwiązuje te problemy stosunkowo szybko. Rozwiązanie jest najczęściej przedstawiane za pomocą grafu lub drzewa. W metodzie rozwiązania dla bieżącego węzła w drzewie obliczamy granicę aby ograniczyć maksymalną wagę ścieżki którą możemy uzyskać. Jeśli wyliczony przez nasz koszt jest większy niż granica, wtedy ignorujemy dalszą cześć drzewa. Aby ułatwić implementacje algorytmu, dodana została klasa *Node*. W niej znajduje się metoda która tworzy nowe węzły drzewa, odwołując się do swojego rodzica.

```
1  Node::Node(Node *parent, int key) {
2    this->level = parent->level + 1;
3    this->key = key;
4    this->matrixSize = parent->matrixSize;
5
6    // Przypisywanie dzieci
7    this->children = new int[this->matrixSize - this->level - 1];
8    int temp = 0;
9    for (int i = 0; i < this->matrixSize - level; i++) {
10     if (parent->children[i] != key) {
11       children[temp] = parent->children[i];
12       temp++;
13     }
14   }
15
16   this->reducedMatrix = matrix.getNewReducedMatrix(
17       parent->reducedMatrix, parent->matrixSize, parent->key, this->
           key);
18
19   this->cost = this->reducedMatrix[0][0] +
20                parent->reducedMatrix[parent->key][key] + parent->cost
                    ;
21   this->reducedMatrix[0][0] = -1;
22
23   this->currentPath = new int[level + 1];
24   for (int i = 0; i <= parent->level; i++) {
25     this->currentPath[i] = parent->currentPath[i];
26   }
27   this->currentPath[level] = key;
28 }
```

Listing 6: Metoda klasy *Node* tworząca nowe węzły

'Sercem' algorytmu jest pętla *while()* oraz kolejka priorytetowa ustawiająca odpowiednia elementy klasy *Node*.

```
struct NodeComparison {
public:
  bool operator()(const Node *node1, Node *node2) const {
    return node1->cost > node2->cost;
  }
};
```

Listing 7: Struktura określająca metodę porównywania elementów klasy *Node*

Warunkiem kończoncym pętlę jest pusta kolejka. Pierwszą czynnością jest pobranie z kolejki pierwszego elementu. Następnie sprawdzany jest poziom, czyli długość ścieżki. Jeśli długość nie jest równa wielkości problemu dodawane są nowe dzieci do obecnego rodzica. W przeciwnym wypadku liczona jest waga aktualnej ściezki.

```
while (!isQueueEmpty()) {
    tempNode = getTop();
    if (this->nodeMode == 1)
      tempNode->show();
    queuePop();

    // Je li jeszcze nie koniec
    if (tempNode->level != matrixSize - 1) {
      // dodawanie dzieci
      for (size_t i = 0; i < (matrixSize - tempNode->level - 1); i++)
          {
        Node *newChild = new Node(tempNode, tempNode->children[i]);
        if (newChild->cost <= upperBound)
          queuePush(newChild);
      }
    } else {
      int *path = new int[matrixSize];
      for (int i = 0; i < matrixSize; i++)
        path[i] = tempNode->currentPath[i];

      cost = tempNode->cost;

      if (cost < upperBound) {
        minPaths.clear();
        upperBound = cost;
        minPaths.push_back(path);
      } else if (cost == upperBound)
        minPaths.push_back(path);
    }
  }
```

Listing 8: Pętla *while()* algorytmu Branch And Bound

3 Eksperymenty obliczeniowe

Obliczenia zastały wykonane na komputerze klasy PC z procesorem AMD Ryzen 5 2600, kartą graficzną NVIDIA GeForce GTX 1060 6BG, 16GB RAM i DYSK SSD. W obliczeniach odchylenia standardowego wykorzystano wzór:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{1}$$

Wszystkie wyniki dla Brute Force zebrano i przedstawiono w tabeli nr 3 gdzie:

- x - rozmiar problemu,
- n - ilość testów,
- $t_{BF}(s)$ - średni czas dla algorytmu Brute Force,
- σ_{BF} - odchylenie standardowe dla algorytmu Brute Force

x	n	$t_{BF}(s)$	σ_{BF}
5	50	$\sim$0.0000	$\sim$0.0000
7	50	$\sim$0.0000	$\sim$0.0000
8	50	0.0018	0.0004
9	50	0.0172	0.0004
10	50	0.1812	0.0014
11	30	2.1920	0.0041
12	15	26.2631	0.0298
13	5	357.0200	0.0907
14	1	4985.1000	0.0000

Tablica 1: Algorytm Brute Force obliczenia.

Wszystkie wyniki dla Branch and Bound zebrano i przedstawiono w tabeli nr 3 gdzie:

- x - rozmiar problemu,
- n - ilość testów,
- $t_{BF}(s)$ - średni czas dla algorytmu Branch And Bound - kolejka Best First,
- $t_{LIFO}(s)$ - średni czas dla algorytmu Branch And Bound - kolejka LIFO,
- $t_{FIFO}(s)$ - średni czas dla algorytmu Branch And Bound - kolejka FIFO,

Podczas testów kolejkowanie FIFO nie działało prawidłowo, ominięto pomiary.

x	n	$t_{BF}(s)$	$t_{LIFO}(s)$
9	50	0.0000	0.0030
10	50	0.0050	0.0038
11	50	0.0070	0.0040
12	50	0.0071	0.0290
13	50	0.0074	0.0234
14	25	0.0080	0.0390
15	25	0.1021	0.1371
16	10	0.3673	0.2856
17	10	0.4258	0.3684
18	10	0.5585	0.7666

Tablica 2: Czas obliczeń algorytmu Branch And Bound w zależności od używanej kolejki

4 Wnioski

Porównanie obydwu zaimplementowanych algorytmów jednoznacznie definiuje nam który z nich jest wydajniejszy pod względem złożoności czasowej. Różnice można zaobserwować już na niższych instancjach problemu. Algorytm Brute Force dla 14 miast wykonywał się około 83 min co w porównaniu do drugiego algorytmu jest ogromną różnicą. Dalsze próby ze względu na długi czas wykonywania zostały pominiętę. Podczas testów zaobserwowany został błąd kolejkowania elementów klasy *Node*, gdy opcja została ustawiona na kolejkę typu FIFO. Różnice między algorytmami zwiększają się wraz ze zwiększaniem wielkości problemu.