Arkadiusz Kozak 226122 **30.11.2017**

Prowadzący: **dr Inż. Dariusz Banasiak**

Algorytm komiwojażera oparty na metodzie symulowanego wyżarzania

1.Opis problemu

Zadanie polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Danych mamy ***N*** miast które komiwojażer ma odwiedzić oraz odległości między nimi. Celem jest znalezienie najkrótszej ścieżki łączącej wszystkie miasta i kończącej się w określonym punkcie.

2. Opis algorytmu

Algorytm symulowanego wyżarzania jest algorytmem wyszukującym przybliżone rozwiązanie konkretnej instancji problemu NP-trudnego. Jest to algorytm z dziedziny heurystyki, dzięki któremu jesteśmy w stanie znaleźć rozwiązanie które nie zawsze będzie najlepszym, jednak będzie w większości przypadków wystarczająco dobrym, a do tego rozwiązane rozstanie w rozsądnym czasie. Algorytmu metaheurystyczne działają za każdym razem inaczej, gdyż w swoim działaniu opierają się one na losowości. W symulowanym wyżarzaniu zezwalamy na zmianę rozwiązania z pewnym prawdopodobieństwem, nawet wtedy gdy jest ono gorsze, ponieważ możliwym jest że w sąsiedztwie gorszego będą lepsze niż te które mamy w aktualnym sąsiedztwie. Dzięki temu, możemy „unikać” problemów minimum lokalnego. Algorytm opiera się czterema głównymi parametrami :

- Temperaturą początkową **T**
- Temperaturą końcową T_{min}
- Funkcją prawdopodobieństwa **P**
- Funkcją obniżania temperatury **F**

Funkcja prawdopodobieństwa określa, z jakim prawdopodobieństwem będziemy zmieniać rozwiązania na gorsze. Jest ona zależna od aktualnego rozwiązania, wylosowanego nowego rozwiązania oraz temperatury. Jest to analogia do procesu metalurgicznego – dopóki temperatura jest wysoka, akceptujemy zmiany na rozwiązania początkowo gorsze, ale potencjalnie lepsze. Zakładamy że w miarę postępowania algorytmu i obniżania temperatury dojdziemy do rozwiązania „rozsądnego", więc wszelkie „ryzykowne" zamiany będą już zbędne. Z tego tez wynika funkcja obniżania temperatury. Przyjęte parametry poszczególnych funkcji:

$$P = e^{\frac{-(\Delta s)}{T}}$$

Gdzie:

e - stała Eulera

Δs – różnica drogi między aktualnym a wylosowanym rozwiązaniem

$$F = k * T_i$$

Gdzie:

k – pewna, ustalona odgórnie stała

T_i – aktualna temperatura

3. Przykład praktyczny

Tabela miast dla naszego przykładu:

\	0	1	2	3
0	∞	3	2	1
1	3	∞	5	2
2	3	1	∞	5
3	7	4	2	∞

Warunki poczatkowe:

Temperatura początkowa : 10

Temperatura końcowa : 2.5

Funkcja obniżania temperatury: 0.6 * T_i

K – aktualny najlepszy koszt

Losujemy rozwiązanie początkowe : 0-1-2-3-0 o koszcie 20.

*Podczas losowania nie zmieniamy położenia miasta początkowego

I iteracja programu: (T=10, K=20)

Losujemy nowe rozwiązanie, poprzez zamianę ze sobą dwóch losowych miast. Nasze nowe rozwiązanie: 0-1-3-2-0 koszt:10

Nowe rozwiązanie ma koszt mniejszy niż nasze aktualne rozwiązanie, zatem zamieniamy je. Zmniejszamy również temperaturę.

II iteracja programu (T=6, K=10)

Losujemy nowe rozwiązanie, poprzez zamianę ze sobą dwóch losowych miast. Nasze nowe rozwiązanie: 0-2-3-1-0 koszt:14

Nowe rozwiązanie ma większy koszt, sprawdzamy więc naszą funkcją prawdopodobieństwa. Wynosi ona w zaokrągleniu 0.51, sprawdzamy zatem czy wartość tej funkcji przewyższy losową liczbę z zakresu (0,1). Losowa liczba była mniejsza, a zatem przyjmujemy że rozwiązanie to jest perspektywiczne i przechodzimy do niego. Zmniejszamy temperaturę.

III iteracja programu (T=3.6, K=14)

Losujemy nowe rozwiązanie, poprzez zamianę ze sobą dwóch losowych miast. Nasze nowe rozwiązanie: 0-2-1-3-0 koszt:12

Nowe rozwiązanie ma koszt mniejszy niż nasze aktualne rozwiązanie, zatem zamieniamy je. Zmniejszamy również temperaturę.

W tym momencie aktualna temperatura spadła poniżej temperatury końcowej, a zatem kończymy algorytm. Finalnie naszą najlepszą ścieżką jest ścieżka 0-2-1-3-0 z kosztem 12. Nie jest to jednak optymalne rozwiązanie, gdyż najlepsze możliwe to ścieżka: 0-3-2-1-0 o koszcie 7.

4. Opis implementacji programu

Cały algorytm opiera się na przedstawionych poniżej klasach oraz ich polach i metodach:

Klasa reprezentująca wierzchołek:

```
public class Vertex
{
    public Vertex()
    {
        this.neighbors = new List<Edge>();
    }
    public short index { get; set; }
    public List<Edge> neighbors { get; set; }
}
```

Klasa ta posiada pole *Index* typu Short Integer, które określa nr wierzchołka oraz listę elementów typu *Edge*, która to lista zawiera krawędzie z wagami, po których możemy dostać się do sąsiednich wierzchołków. Posiada także domyślny konstruktor inicjalizujący listę krawędzi.

Klasa reprezentująca krawędź:

```
public class Edge
{
    public Edge()
    {
        this.vertex1 = new Vertex();
        this.vertex2 = new Vertex();
    }
    public short distance { get; set; }
    public Vertex vertex1 { get; set; }
    public Vertex vertex2 { get; set; }
}
```

Klasa ta posiada pole typu Short Integer odpowiadające za odległość między dwoma wierzchołkami (vertex1 oraz vertex2, które to oznaczają wierzchołek początkowy oraz końcowy danej krawędzi).

Klasa reprezentująca graf:

```
public class Graph
    {
        public Graph()
        {
            this.vertices = new List<Vertex>();
            this.edges = new List<Edge>();
        }
        public List<Vertex> vertices { get; set; }
        public List<Edge> edges { get; set; }
    }
```

Klasa ta reprezentuje w algorytmie naszą macierz wierzchołków oraz krawędzi. Posiada listę wierzchołków oraz listę krawędzi które łączą dane wierzchołki. Zaimplementowany został również domyślny konstruktor, inicjalizujący obie te listy.

Zaimplementowane metody:

```
public static Graph RandGraph(int size, int minDisctance, int maxDistance)
```

Powyższa metoda służy do generowania losowego grafu pełnego-symetrycznego. Możemy określić jego rozmiar oraz minimalną jak i maksymalną odległość między dwoma losowymi miastami.

```
public static int LoadSize(string path)
```

Powyższa metoda pozwala na zwrócenie rozmiaru grafu, do którego ścieżkę podajemy.

```
public static Graph CreateGraph(int size, string path)
```

Powyższa metoda służy do wygenerowania grafu, do którego ścieżkę podajemy. Kolejne krawędzie są wczytywane linia po linii, dla każdego wierzchołka.

```
public static int NearestNeighbour(Graph graph, int size)
```

Powyższa metoda odpowiada za znalezienie najkrótszej ścieżki we wskazanym grafie metodą najbliższego sąsiada i zwrócenie wartości tej ścieżki.

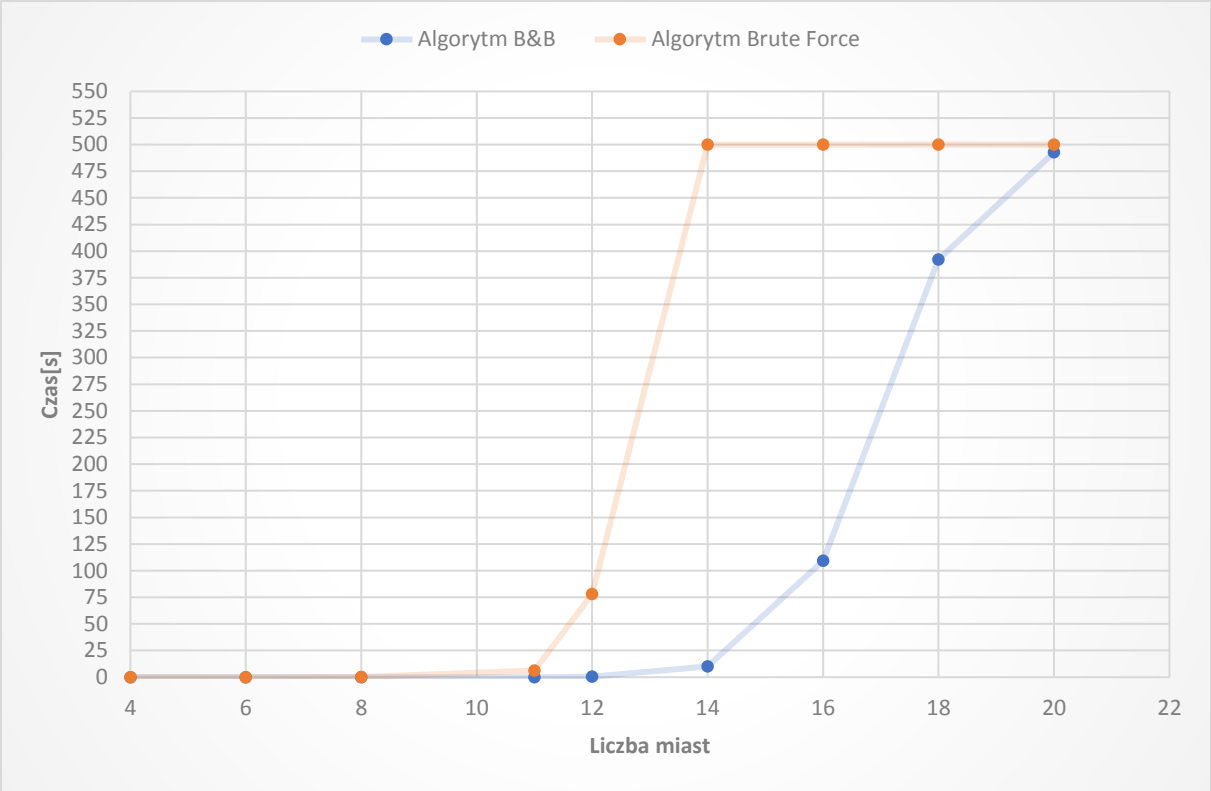
```
void BruteFRecurring(Vertex vertex)
```

Powyższa metoda to główny punkt całego algorytmu. Odpowiada ona za przejście grafu przeglądając kolejne ścieżki, jednocześnie obcinając nieperspektywiczne gałęzie. Wyznaczamy również stos wierzchołków które zawierać będą się w rozwiązaniu, jak i całkowitą długość tej ścieżki.

5.Badania efektywności algorytmu

Każda z poniższych instancji została wykonana w pętli 20 razy, a wynik został uśredniony. Wagi krawędzi wyznaczone zostały w sposób losowy i zawierały się w przedziale <10,100>. Wierzchołkiem startowym zawsze był wierzchołek o indeksie 0. W przypadku gdy algorytm wykonuje się w czasie przekraczającym 500 sekund, jest on anulowany a jego wynik ustawiany jako max(500s).

Czas[s]		Ilość miast
Metoda Brute Force	Metoda B&B	
0,0001	0,0001	4
0,0013	0,0001	6
0,0516	0,0015	8
6,1025	0,0162	11
78,129	0,3985	12
X	9,9725	14
X	109,312	16
X	392,129	18
X	488,722	20



6. Wnioski

Jak widać na przedstawionych tabelach, algorytm B&B jest średnio o znacząco wydajniejszy od algorytmu typu Brute Force. Problemem jednak jest fakt, iż dla pewnych nawet nie dużych instancji może się on zachować tak samo jak przegląd zupełny, co już przy 14 miastach wyklucza otrzymanie wyniku w rozsądnym czasie. Algorytm B&B może być z powodzeniem stosowany dla większych ilości miast, jednak musimy liczyć się z faktem że nie jesteśmy w stanie przewidzieć w jakim czasie otrzymamy wynik gdyż jest to mocno zależne od danej instancji miast. Znaczenie również ma wybranie metody wyznaczania pierwszej wartości ścieżki, którą porównujemy. Im bliższe bowiem wynikowi ograniczenie znajdziemy, tym więcej gałęzi będzie mogło być obciętych a co za tym idzie nasze wyniki czasowe mogą ulec znacznej poprawie.