# Algorytm ewolucyjny dla symetrycznego problemu komiwojażera

### *Definicja symetrycznego, euklidesowego, dwuwymiarowego problemu komiwojażera*

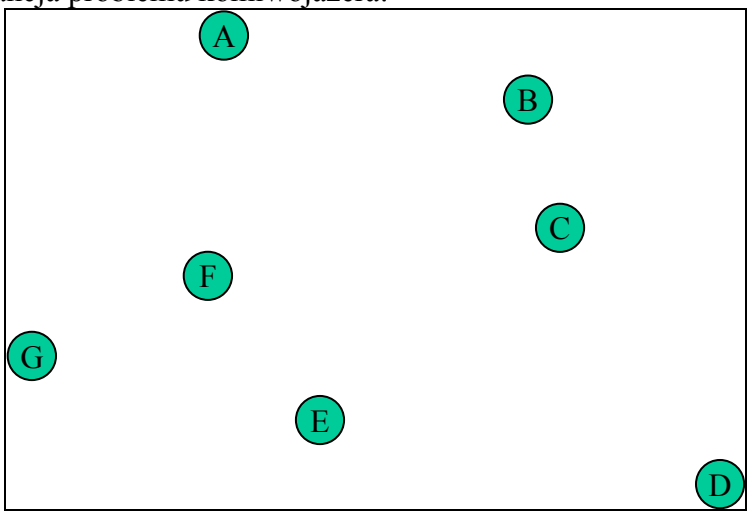
Dany jest zbiór *N* wierzchołków umieszczonych na płaszczyźnie dwuwymiarowej. Położenie każdego wierzchołka jest opisane przez dwie współrzędne *X* i *Y*. Odległość pomiędzy każdą parą wierzchołków jest odległością euklidesową zaokrąglona do liczby całkowitej w następujący sposób:

$$\text{floor}\ (0.5 + \text{sqrt}\ ((X1 - X2) * (X1 - X2) + (Y1 - Y2) * (Y1 - Y2)))$$

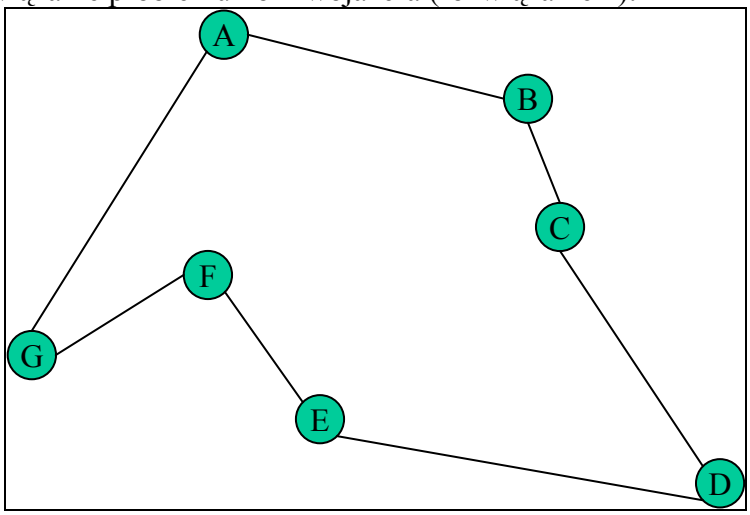
floor to funkcja zwracająca najbliższą liczbę całkowitą mniejszą lub równą niż podany parametr.

Rozwiązaniem problemu komiwojażera jest cykl Hamiltona, tj. ścieżka przechodząca przez wszystkie wierzchołki dokładnie raz. Celem jest znalezienie rozwiązania o jak najmniejszej długości.

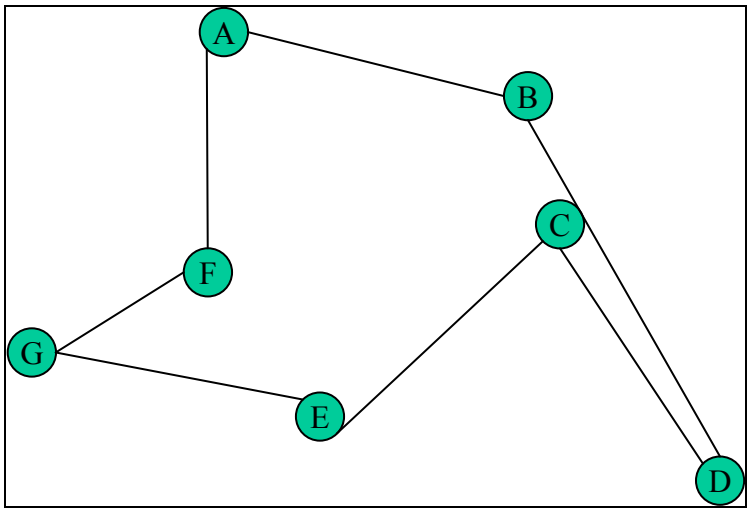
Przykładowa instancja problemu komiwojażera:




Przykładowe rozwiązanie problemu komiwojażera (rozwiązanie 1):




Przykładowe rozwiązanie problemu komiwojażera (rozwiązanie 2):


A
B
C
F
G
E
D

*Laboratorium Inżynierii Oprogramowania*

# Standard kodowania w języku C++

## *Nazwy plików*

- Wszystkie nazwy powinny być unikalne, nie powinno się używać znaków specjalnych takich jak np. '#', '~', '\t', or '\n', oraz znaków narodowych.
- Pliki nie powinny nosić nazw zarezerwowanych dla urządzeń I/O lub portów: lpt1, com4, irq8.

## *Deklaracje i definicje klas*

Każda klasa jest opisana w dwóch plikach:

- Pliku nagłówkowym z rozszerzeniem *.h- zawiera deklarację klasy;
- Pliku implementacyjnym z rozszerzeniem *.cpp- zawiera definicję klasy.

## *Pliki nagłówkowe*

Pliku nagłówkowy jest podzielony na 4 części:

- *Sekcja include*,
- *Sekcja definicji stałych symbolicznych,*
- *Sekcja opisów klas*,
- *Deklaracje klas*.

W **sekcji include** należy zwrócić uwagę na to, aby nie umieszczać ścieżki bezwględnej do pliku a jedynie jego nazwę lub ścieżkę względną.

Przykład:

```
#include ”pliczek.h”                    //! poprawne
#include ”..\katalog\pliczek.h”          //! poprawne
#include ”katalog\include\naglowek.h”    //! niepoprawne !!!
```

W **sekcji definicji stałych symbolicznych** nazwy powinny być pisane dużymi literami tak, aby łatwo było je odróżnić od pól.

**Sekcja opisu klasy** Przed deklaracją klasy powinien znaleźć się krótki opis.

Przykład:

```
/**  Opis: */
```

Komentarz może być podzielony na ogólny opis zawarty w pierwszej linii i szczegółowy opis zawarty w kolejnych liniach oddzielonych od pierwszej jedną linią pustą.

Przykład:

```
/** Opis gólny
*
*   Opis szczegółowy
*   Opis szczegółowy */
```

Natomiast **deklaracja klasy** składa się z 4 elementów:

- Definicje pól

  Pola tego samego typu nie powinny być wymienione po przecinku. Każde pole powinno być umieszczona w nowej linii, przed którą (patrz poniżej) powinien znaleźć się krótki komentarz.

  Powinno się unikać definiowania pól publicznych o ile nie ma to istotnego uzasadnienia.

- Konstruktory i destruktory
- Deklaracje funkcji
- operatory

Każde pole powinno być poprzedzone komentarzem. Komentarz powinien podawać interpretację pola, jednostki i wszelkie ograniczenia, które muszą spełniać wartości pola. Np.:

```
/** Wartość funkcji celu - długość ścieżki.
*
*       Ograniczenia:
*       Objective > 0
*       Objective = Suma po a=0,...,TSPProblem.NumberOfNodes - 1 z
*               TSPProblem.Distance (a, NextNodes [a]). */
int Objective;
```

Deklaracja każdej metody powinna być poprzedzona komentarzem. Komentarz powinien podawać cel metody, parametry, zwracany wynik, ograniczenia – początkowe i końcowe warunki poprawności, wyjątki jakie mogą się pojawić podczas wykonywania metody. Np.:

```
/** Zwraca odległość pomiędzy dwoma miastami
*       Parametry:
*       n1 – pierwsze miasto
*       n2 – drugie miasto
*       Wynik – odległość pomiędzy miastami n1 i n2
*       Ograniczenia 0 <= n1, n2 <= NumberOfNodes */.
int Distance (int n1, int n2);
```

## *Zmienne*

Deklaracja każdej zmiennej powinna być poprzedzone komentarzem:

```
/** Opis: */
```

Komentarz powinien podawać interpretację pola, jednostki i wszelkie ograniczenia, które muszą spełniać wartości pola. Np.:

```
/** Wartość funkcji celu - długość ścieżki.
*
*       Ograniczenia:
*       Objective > 0
*       Objective = Suma po a=0,...,TSPProblem.NumberOfNodes - 1 z
*               TSPProblem.Distance (a, NextNodes [a]). */
int Objective;
```

Należy unikać stosowanie zmiennych globalnych, jeżeli mogą być one zastąpione zmiennymi lokalnymi lub polami w klasie.

Zmienne lokalne należy deklarować jak najbliżej miejsca ich użycia, najlepiej równocześnie z ich zainicjowaniem. Np.:

```
int Objective = 0;
```

## *Funckje*

Deklaracja (o ile znajduje się w pliku nagłówkowym) i definicja każdej funkcji powinna być poprzedzona komentarzem. Komentarz powinien podawać cel funkcji, parametry, zwracany wynik, ograniczenia – początkowe i końcowe warunki poprawności, wyjątki jakie mogą się pojawić podczas wykonywania metody. Np.:

```
/** Zwraca odległość pomiędzy dwoma miastami
*       Parametry:
*       n1 – pierwsze miasto
*       n2 – drugie miasto
*       Wynik – odległość pomiędzy miastami n1 i n2
*       Ograniczenia 0 <= n1, n2 <= NumberOfNodes */.
int Distance (int n1, int n2) {
…
```

## *Długość linii i szerokość tabulacji*

Linie programu nie powinny być dłuższe niż około 80 znaków. Jeżeli dana instrukcja nie mieści się w jednej linii należy kolejne linie tej samej instrukcji wcinać w stosunku do pierwszej, np.

```
Distances [iNode][iNode2] = floor (0.5 + sqrt ((double)(NodesPositions [iNode].x –
        NodesPositions [iNode2].x) * (NodesPositions [iNode].x - NodesPositions [iNode2].x) +
```

```
(double)(NodesPositions [iNode].y - NodesPositions [iNode2].y) *

(NodesPositions [iNode].y - NodesPositions [iNode2].y)));
```

Dla czytelności wyświetlanego kodu powinny być stosowane akapity i wcięcia przy użyciu tabulatorów. Wielkość tabulacji może być dowolna. Należy jednak pamiętać żeby konsekwentnie trzymać się w całym programie tej samej konwencji.

Akapity i wcięcia są używane wewnątrz:

- Deklaracji funkcji
- warunków
- pętli
- instrukcji ‘switch' , ‘case' i ‘typedef'
- bloków pomiędzy nawiasami klamrowymi

Każdy nawias klamrowy powinien być umieszczany w oddzielnej linii lub za instrukcją warunkową/pętli. W tym pierwszym przypadku pary odpowiadających sobie nawiasów klamrowych powinny znajdować się w tej samej kolumnie (być równooddalone od lewej krawędzi tekstu). W drugim przypadku nawias zamykający powinien się znaleźć w kolumnie, w której rozpoczyna się instrukcja warunkowa/petli. Otwarcie nawiasu powinno pociągać za sobą napisanie nawiasu zamykającego klamrę tuż pod danym nawiasem, dopiero w dalszej kolejności powinien powstawać kod wypełniający wnętrze nawiasów klamrowych.

Przykład:

```
if (iIteracja == 0) {
    GenerujRozwiazaniePoczatkowe ();
}
```

lub

```
if (iIteracja == 0)
{
    GenerujRozwiazaniaPoczatkowe ();
}
```

## *Odstępy*

Po każdym średniku powinna wystąpić co najmniej jedna spacja.

Przykład:

```
for(iIteracja = 0; iIteracja <= iGornaGranica; iIteracja++)
{
        statements
}
```

Jedna spacja obowiązuje również po przecinku przy wymienianiu parametrów wywołania funkcji lub metody.

Przykład:

```
PobierzDane(iParamametr1, iParametr2, iParametr3);
```

Operatory powinny być z obu stron otoczone spacjami.

Przykład:

```
dZysk = dPrzychod – dKoszt;

Stream << Node << '\t';
```

## *Notacja węgierska*

Notacja ta zakłada, że każde pole składa się z identyfikatora typu i nazwy, tak że nie jest konieczne szukanie typu pola w sekcji deklaracji pól na początku programu, gdyż jej typ jest określony przedrostkiem nazwy. Propozycja użycia notacji węgierskiej dotyczy m. in. typów pól i zmiennych:

| Skrót typu | znaczenie |
|---|---|
| *c* | char |
| *b* | bool |
| *i* | int |
| *l* | long |
| f | float |
| d | double |

Przykład:

```
iNumer          - zmienna typu integer,
```

## *Nazwy pól, metod, klas*

Nazwy stosowane dla pól, metod, klas powinny być zrozumiałe. Nie należy stosować skrótów.

- stosowanie nazw wielowyrazowych bez stosowania odstępów i początkiem każdego członu od wielkiej litery.

  Przykład:

  ```
  int iLiczbaIteracji
  ```

- stosowanie prefiksu C w nazwie klasy

  Przykład:

```
class CProblem
{
        ...
}
```

## *Komentarze*

Zalecane jest stosowanie dwóch typów komentarzy:

- komentarze jednej linii

  Przykład:

  ```
  //! Krótki komentarz
  ```

- komentarze blokowe

  Przykład:

  ```
  /** Komentarz blokowy
  *    ... */
  ```

## *Return, break, continue, goto*

Każde wystąpienie *return* w miejscu poza końcem funkcji lub metody powinno być skomentowane.

Podobnie też każde wystąpienie instrukcji break i continue.

Używanie instrukcji goto jest niedozwolone

## *Ogólne zasady kodowania*

W jednej linii powinna znajdować się jedna instrukcja.

Przykład:

źle:

```
NodeToExchane1 = Node1; NodeToExchane2 = Node2; bImprovementFound = true;
```

dobrze:

```
NodeToExchane1 = Node1;
NodeToExchane2 = Node2;
bImprovementFound = true;
```

Należy stosować proste konstrukcje, przestrzegać zasady KISS (keep it simple stupid).

Linie kodu zawierające powiązane instrukcje powinny być łączone w bloki. Poszczególne bloki powinny być oddzielone wolnymi liniami.

Przykład:

```
WeightVector.Rescale (NondominatedSet.ApproximateIdealPoint, NondominatedSet.ApproximateNadirPoint);


TPoint TempReferencePoint = NondominatedSet.ApproximateIdealPoint;

TempReferencePoint.Augment (NondominatedSet.ApproximateIdealPoint,
NondominatedSet.ApproximateNadirPoint);


TempPopulation.clear ();

for (int isol = 0; isol < MainPopulation.size (); isol++) {

        UpdateTempPopulation (TempReferencePoint, MainPopulation [isol]);

}
```

## *Komentarze w kodzie metod i funkcji*

Komentarze w kodzie metod i funkcji powinny opisywać poszczególne bloki programu, pętle i instrukcje warunkowe. Można je pominąć, jeżeli kod sam się tłumaczy (ale kiedy **naprawdę** sam się tłumaczy).

Przykład (komentarze opisujące bloki)

```
// Przeskaluj wektor wag

WeightVector.Rescale (NondominatedSet.ApproximateIdealPoint, NondominatedSet.ApproximateNadirPoint);


// Ustaw punkt odniesienia

TPoint TempReferencePoint = NondominatedSet.ApproximateIdealPoint;

TempReferencePoint.Augment (NondominatedSet.ApproximateIdealPoint,
NondominatedSet.ApproximateNadirPoint);


// Znajdź populację chwilową

TempPopulation.clear ();

for (int isol = 0; isol < MainPopulation.size (); isol++) {

        UpdateTempPopulation (TempReferencePoint, MainPopulation [isol]);

}
```

Przykład (komentarz opisujący pętlę)

```
// Przeglądaj miasta rozpoczynając od StartingNode1 aż do przejrzenia wszystkich miast lub znalezienia
ruchu
```

```
        // poprawiającego wartość funkcji celu
    for (int in1 = 0; (in1 < TSPProblem.NumberOfNodes) && !bImprovementFound; in1++) {
            int Node1 = (StartingNode1 + in1) % TSPProblem.NumberOfNodes;
```

Przykład (niepotrzebny komentarz – dzięki poprawnym nazwom zmiennych)

```
        // Jeżeli Liczba elementów jest mniejsza od Dostępnej liczby elementów
        if (iLiczbaElementow < iDostepnaLiczbaElementow)
```

## *Warunki poprawności*

Warunki poprawności należy sprawdzać za pomocą makra assert. Wszystkie linie wprowadzone w celu sprawdzenia warunku poprawności należy oznaczać za pomocą komentarza znajdującego się w tej samej linii: //### Warunek nr. Np.:

```
            int OldObjective = Objective; //### Warunek 1


            ExchangeArcs (NodeToExchane1, NodeToExchane2);


            // Nowa wartość funkcji celu musi być lepsza od poprzedniej
            assert (Objective < OldObjective); //### Warunek 1
```

```
1   // TSP Simple.cpp : Defines the entry point for the console application.
2   //
3
4   #include "stdafx.h"
5
6   #include <fstream>
7   #include <iostream>
8   #include <vector>
9   #include <cmath>
10  #include <ctime>
11
12  using   namespace std;
13
14  typedef   vector <int > TIntVector;
15
16  //------------------------------------------------------------------------------
17  /** Dane definiujące instancję problemu komiwojażera */
18
19  /** Dwuwymiarowa tablica odległości pomiędzy miastami */
20  vector <TIntVector> Distances;
21
22  /** Liczba miast/wierzcholkow (nodes) */
23  int   NumberOfNodes;
24
25  /** Polozenie miasta (node) w dwuwymiarowej przestrzeni */
26  typedef   struct   {
27    int   x, y;
28  }  TPoint;
29
30
31  //------------------------------------------------------------------------------
32  /** Dane opisujące rozwiązanie problemu komiwojażera */
33
34  /** Tablica przechowująca rozwiązanie
35  *
36  *  Nextnodes [a] == b ozancza, że z po mieście (wierzchołku) a
37  *  następuje miasto b.
38  *  Warunki poprawności:
39  *  NextNodes [a] == b <=> PreviousNodes [b] == a
40  *  0 <= NextNodes [...] < NumberOfNodes
41  *  Opisywana ścieżka przechodzi przez wszystkie miasta dokładnie raz
42  */
43  vector <int > NextNodes;
44
45  /** Tablica przechowująca rozwiązanie
46  *
47  *  Nextnodes [a] == b ozancza, że z po mieście (wierzchołku) a
48  *  następuje miasto b.
49  *  Uwaga tablica ta jest nadmarowa w stosunku do NextNodes, ale
50  *  jej użycie zwiększa efektywność.
51  *  Warunki poprawności:
52  *  NextNodes [a] == b <=> PreviousNodes [b] == a
53  *  0 <= PreviousNodes [...] < NumberOfNodes
54  *  Opisywana ścieżka przechodzi przez wszystkie miasta dokładnie raz
55  */
56  vector <int > PreviousNodes;
57
```

```
58  /** Wartość funckji celu - długość ścieżki.
59  *
60  *  Warunki poprawności:
61  *  Objective > 0 (teoretycznie wartość 0 jest dopuszczalna, w praktycznych
    instancjach nie może się pojawić)
62  *  Objective = Suma po a=0,...,NumberOfNodes - 1 z
63  *      Distance (a, NextNodes [a]).
64  */
65  int  Objective;
66
67  //---------------------------------------------------------------------------
68  /** Zwraca odległość pomiędzy dwoma miastami */
69  int  Distance (int  n1, int  n2){
70    return  Distances [n1][n2];
71  }
72
73  /** Wczytuje problem z pliku.
74  *
75  *  Zwraca false jeżeli odczyt się nie powiódł */
76  bool  Load(char *  FileName)
77  {
78    // Tablica położen miast
79    vector <TPoint> NodesPositions;
80
81    fstream Stream (FileName, ios::in);
82    if  (Stream.rdstate () != ios::goodbit) {
83      cout << "Cannot open " << FileName << '\n';
84      Stream.close ();
85      return  false;
86    }
87
88    // Wczytaj liczbę miast/wierzchołków
89    Stream >> NumberOfNodes;
90    if  (Stream.rdstate () != ios::goodbit) {
91      cout << "Error reading " << FileName << '\n';
92      Stream.close ();
93      return  false;
94    }
95    if  (NumberOfNodes <= 1) {
96      cout << "Error reading " << FileName << ". Number of nodes <= 1" << '\n';
97      Stream.close ();
98      return  false;
99    }
100
101   // Zaalokuj tablicę położeń miast
102   NodesPositions.resize (NumberOfNodes);
103
104   // Wczytaj pozycje miast w przestrzeni dwuwymiarowej
105   for  (int  iNode = 0; iNode < NumberOfNodes; iNode++) {
106     Stream >> NodesPositions [iNode].x;
107     Stream >> NodesPositions [iNode].y;
108     char  c;
109     do {
110       Stream.get (c);
111     } while  (c != '\n');
112     if  (Stream.rdstate () != ios::goodbit) {
113       cout << "Error reading " << FileName << '\n';
```

```
114       Stream.close ();
115       return  false;
116     }
117   }
118
119   // Zaalokuj tablicę odległości pomiędzy miastami
120   //    Distances.resize (NumberOfNodes);
121   for  (iNode = 0; iNode < NumberOfNodes; iNode++) {
122     TIntVector IntVector;
123     IntVector.resize (NumberOfNodes);
124     //      Distances [iNode].resize (NumberOfNodes);
125     Distances.push_back (IntVector);
126   }
127
128   // Oblicz odległość euklidesową pomiędzy miastami (zgodnie z zasadmi
      opisanymi w TSPLib
129   for  (iNode = 0; iNode < NumberOfNodes; iNode++) {
130     for  (int  iNode2 = 0; iNode2 < NumberOfNodes; iNode2++) {
131       Distances [iNode][iNode2] = floor (0.5 + sqrt ((double )(NodesPositions
          [iNode].x - NodesPositions [iNode2].x) * (NodesPositions [iNode].x -
          NodesPositions [iNode2].x) +
132         (double )(NodesPositions [iNode].y - NodesPositions [iNode2].y) *
          (NodesPositions [iNode].y - NodesPositions [iNode2].y)));
133       Distances [iNode2][iNode] = Distances [iNode][iNode2];
134     }
135   }
136
137   Stream.close ();
138   return  true;
139 }
140
141   /** Konstruuje poczatkowe losowe rozwiązanie, odpowiadające losowej
142   *  permutacji miast */
143 void  FindRandom ()
144 {
145   NextNodes.resize (NumberOfNodes);
146   PreviousNodes.resize (NumberOfNodes);
147
148   // Wylosuj pierwsze miasto
149   int  FirstNode = rand () % NumberOfNodes;
150   int  Node = FirstNode;
151
152   // Zaalokuj tablicę miast dodanych do ścieżki i wypełnij
153   // ją wartościami false
154   vector <bool > bNodeAdded;
155   bNodeAdded.resize (NumberOfNodes, false);
156
157   // Dodawaj losowo kolejne miasta do ścieżki
158   bNodeAdded [Node] = true;
159   for  (int  i = 0; i < NumberOfNodes - 1; i++) {
160     bNodeAdded [Node] = true;
161     // Wylosuj miasto, które nie zostało jeszcze dodane do ścieżki
162     int  NextNode = rand () % NumberOfNodes;
163     while  (bNodeAdded [NextNode]) {
164       NextNode = (NextNode + 1) % NumberOfNodes;
165     }
166
```

```
167     // Dodaj łuk (Node, NextNode)
168     NextNodes [Node] = NextNode;
169     PreviousNodes [NextNode] = Node;
170
171     // Weź następne miasto
172     Node = NextNode;
173   }
174
175   // Dodaj łuk (Node, FirstNode)
176   NextNodes [Node] = FirstNode;
177   PreviousNodes [FirstNode] = Node;
178
179   // Oblicz wartość funkcji celu
180   Objective = 0;
181   for  (i = 0; i < NumberOfNodes; i++) {
182     Objective += Distance (i, NextNodes [i]);
183   }
184
185 }
186
187 void  FG ()
188 {
189   NextNodes.resize (NumberOfNodes);
190   PreviousNodes.resize (NumberOfNodes);
191   int  FN = rand () % NumberOfNodes;
192   int  N = FN;
193   vector <bool > NA;
194   NA.resize (NumberOfNodes, false);
195   NA [N] = true;
196   for  (int  i=0;i<NumberOfNodes- 1;i++) {
197     int  BD, BN;
198     bool  bFirstIteration = true;
199     for  (int  j=0;j<NumberOfNodes;j++)
200       if  (! NA [j])
201         if  (bFirstIteration) {BD = Distance (N, j); BN = j; bFirstIteration =
          false;}
202         else
203           if  (BD > Distance (N, j)) {BD = Distance (N, j);BN = j;}
204     NA [BN] = true;
205     NextNodes [N] = BN;
206     PreviousNodes [BN] = N;
207     N = BN;
208   }
209   NextNodes [N] = FN;
210   PreviousNodes [FN] = N;
211   Objective = 0;
212   for  (i = 0; i < NumberOfNodes; Objective += Distance (i, NextNodes [i]),i++);
213 }
214
215 /** Zapisuje rozwiązanie do podanego strumienia */
216 void  Save (ostream& Stream)
217 {
218   Stream << Objective << '\n';
219
220   // Wypisz permutację miast opisujących rozwiązanie
221   // rozpoczynając od miasta 0
222   int  Node = 0;
```

```
223   for  (int  i = 0; i < NumberOfNodes; i++) {
224     Stream << Node << '\t';
225     Node = NextNodes [Node];
226   }
227   Stream << '\n';
228 }
229
230 int  _tmain(int  argc, _TCHAR* argv[])
231 {
232   srand ((unsigned )time (NULL));
233   if  (Load ("kroa100.txt")) {
234     FindRandom ();
235     Save (cout);
236     FG ();
237     Save (cout);
238   }
239   return  0;
240 }
241
```