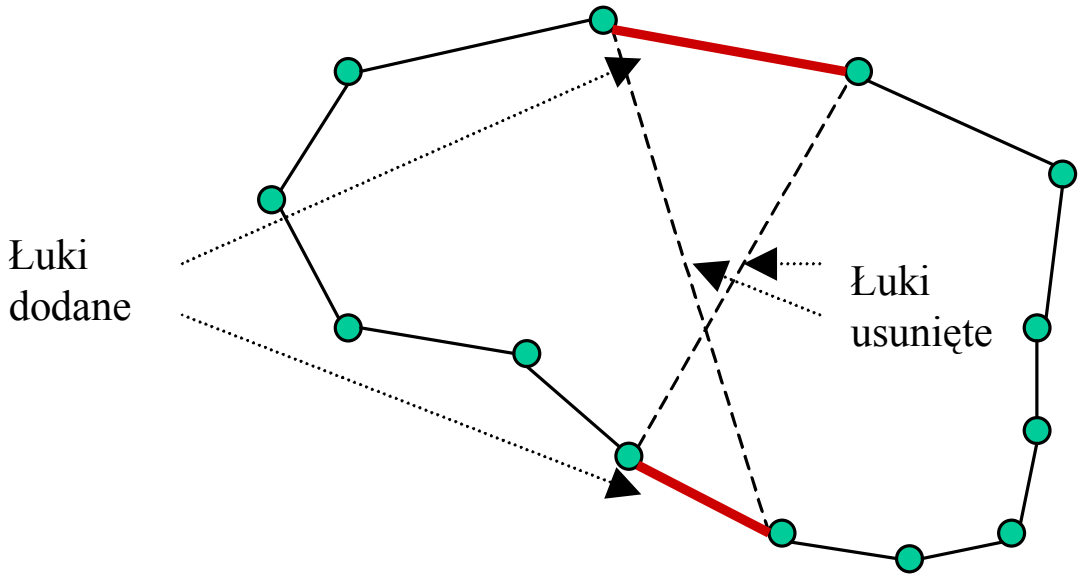
Lokalne przeszukiwanie dla problemu komiwojażera

Operator (lokalny ruch) wymiany dwóch łuków

Operator (lokalny ruch) polega na usunięciu dwóch nie sąsiadujących łuków i naprawie rozwiązania poprzez zamknięcie ścieżki dwoma innymi łukami (istnieje tylko jedna możliwość wstawienia nowych łuków).
Operator ilustruje poniższy rysunek.
Uwaga. W implementacji komputerowej zachodzi potrzeba odwrócenia kolejności miast na jednej z podścieżek powstałych po usunięciu łuków.



Lokalne przeszukiwanie

Stosowane jest lokalne przeszukiwanie w wersji zachłannej (*greedy*) (nie mylić z heurystyką zachłanną, która służy do konstrukcji rozwiązania od podstaw).
Lokalne przeszukiwanie rozpoczyna się od istniejącego, poprawnego rozwiązania. W każdej iteracji lokalnego przeszukiwania przeglądane są pary łuków i oceniane ruchy polegające na ich usunięciu. Przeglądanie rozpoczyna się od losowo wybranych łuków, co pozwala uniknąć preferencji dla pewnych ruchów. Pierwszy napotkany ruch przynoszący poprawę funkcji celu (długości ścieżki) jest wykonywany i rozpoczynana jest kolejna iteracja lokalnego przeszukiwania. Lokalne przeszukiwanie kończy się, jeżeli po przejrzeniu wszystkich par łuków (a więc po ocenieniu wszystkich możliwych ruchów) nie znaleziono ruchu przynoszącego poprawę wartości funkcji celu.

```
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 unsigned int uint;
15 typedef vector <uint> TUIntVector;
16
17 //------------------------------------------------------------------------------
18 /** Dane definiujące instancję problemu komiwojażera */
19
20 /** Dwuwymiarowa tablica odległości pomiędzy miastami */
21 vector <TUIntVector> Distances;
22
23 /** Liczba miast/wierzcholkow (nodes) */
24 uint NumberOfNodes;
25
26 /** Polozenie miasta (node) w dwuwymiarowej przestrzeni */
27 typedef struct {
28   int x, y;
29 } TPoint;
30
31
32 //------------------------------------------------------------------------------
33 /** Dane opisujące rozwiązanie problemu komiwojażera */
34
35 /** Tablica przechowująca rozwiązanie
36 *
37 *  Nextnodes [a] == b ozancza, że z po mieście (wierzchołku) a
38 *  następuje miasto b.
39 *  Warunki poprawności:
40 *  NextNodes [a] == b <=> PreviousNodes [b] == a
41 *  0 <= NextNodes [...] < NumberOfNodes
42 *  Opisywana ścieżka przechodzi przez wszystkie miasta dokładnie raz
43 */
44 vector <uint> NextNodes;
45
46 /** Tablica przechowująca rozwiązanie
47 *
48 *  Nextnodes [a] == b ozancza, że z po mieście (wierzchołku) a
49 *  następuje miasto b.
50 *  Uwaga tablica ta jest nadmarowa w stosunku do NextNodes, ale
51 *  jej użycie zwiększa efektywność.
52 *  Warunki poprawności:
53 *  NextNodes [a] == b <=> PreviousNodes [b] == a
54 *  0 <= PreviousNodes [...] < NumberOfNodes
55 *  Opisywana ścieżka przechodzi przez wszystkie miasta dokładnie raz
56 */
57 vector <uint> PreviousNodes;
```

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

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

```
167     // Weź następne miasto
168     Node = NextNode;
169   }
170
171   // Dodaj łuk (Node, FirstNode)
172   NextNodes [Node] = FirstNode;
173   PreviousNodes [FirstNode] = Node;
174
175   // Oblicz wartość funkcji celu
176   Objective = 0;
177   for  (i = 0; i < NumberOfNodes; i++) {
178     Objective += Distance (i, NextNodes [i]);
179   }
180
181 }
182
183 void  FindGreedy ()
184 {
185   NextNodes.resize (NumberOfNodes);
186   PreviousNodes.resize (NumberOfNodes);
187
188   // Wylosuj pierwsze miasto
189   uint FirstNode = rand () % NumberOfNodes;
190   uint Node = FirstNode;
191
192   // Zaalokuj tablicę miast dodanych do ścieżki i wypełnij
193   // ją wartościami false
194   vector <bool > bNodeAdded;
195   bNodeAdded.resize (NumberOfNodes, false);
196
197   // Dodawaj losowo kolejne miasta do ścieżki
198   bNodeAdded [Node] = true;
199   for  (uint i = 0; i < NumberOfNodes; i++) {
200     // Znajdź najbliższe miasto, które nie zostało jeszcze dodane do ścieżki
201     uint iBestDistance;
202     uint iBestNode;
203     bool  bFirstIteration = true;
204     for  (uint j = 1; j < NumberOfNodes; j++) {
205       if  (! bNodeAdded [j]) {
206         if  (bFirstIteration) {
207           iBestDistance = Distance (Node, j);
208           iBestNode = j;
209           bFirstIteration = false;
210         }
211         else  {
212           if  (iBestDistance > Distance (Node, i)) {
213             iBestDistance = Distance (Node, i);
214             iBestNode = i;
215           }
216         }
217       }
218     }
219
220     // Dodaj łuk (Node, NextNode)
221     bNodeAdded [iBestNode] = true;
222     NextNodes [Node] = iBestNode;
223     PreviousNodes [iBestNode] = Node;
```

```
224
225     // Weź następne miasto
226     Node = iBestNode;
227   }
228
229   // Dodaj łuk (Node, FirstNode)
230   NextNodes [Node] = FirstNode;
231   PreviousNodes [FirstNode] = Node;
232
233   // Oblicz wartość funkcji celu
234   Objective = 0;
235   for  (i = NumberOfNodes - 1; i >= 0; i-- ) {
236     Objective += Distance (i, NextNodes [i]);
237   }
238 }
239
240 void  Save (ostream& Stream)
241 {
242   Stream << Objective << '\n';
243
244   // Wypisz permutację miast opisujących rozwiązanie
245   // rozpoczynając od miasta 0
246   uint Node = 0;
247   for  (uint i = 0; i < NumberOfNodes; i++) {
248     Stream << Node << '\t';
249     Node = NextNodes [Node];
250   }
251   Stream << '\n';
252 }
253
254 void  ExchangeArcs (uint Node1, uint Node2) {
255   // Uaktualnij wartość funkcji celu
256   // Łuki usuwane
257   Objective -= Distance (Node1, PreviousNodes [Node1]);
258   Objective -= Distance (Node2, PreviousNodes [Node2]);
259   // Łuki dodawane
260   Objective += Distance (Node1, Node2);
261   Objective += Distance (NextNodes [Node1], NextNodes [Node2]);
262
263   // Zapamietaj następniki Node1 i Node2
264   uint Next1 = NextNodes [Node1];
265   uint Next2 = NextNodes [Node2];
266
267   // Po wymianie łuków ścieżka od Node2 do Next1 będzie miała odrócony kierunek
268   // Należy wymienić wpisy w tablicach NextNodes i PreviousNodes
269
270   // Zaczynając od Node2
271   uint ArcStartingNode = Node2;
272   uint ArcEndingNode = PreviousNodes [Node2];
273   uint a = PreviousNodes [3];
274   // Dla wszystkich łuków aż do osiągnięcia Next1
275   do {
276     // Zapamiętaj koniec następnego łuku
277     uint NextEndingNode = PreviousNodes [ArcEndingNode];
278
279     // Zmień opis łuku (ArcStartingNode, ArcEndingNode)
280     NextNodes [ArcStartingNode] = ArcEndingNode;
```

```
281     PreviousNodes [ArcEndingNode] = ArcStartingNode;
282
283     // Przejdź do kolejnego łuku
284     ArcStartingNode = ArcEndingNode;
285     ArcEndingNode = NextEndingNode;
286
287   } while (ArcStartingNode != Next1);
288
289   // Wymień łuki
290   NextNodes [Node1] = Node2;
291   PreviousNodes [Node2] = Node1;
292
293   NextNodes [Next1] = Next2;
294   PreviousNodes [Next2] = Next1;
295
296 }
297
298 void GreedyLocalSearch ()
299 {
300   bool bImprovementFound;
301
302   // Wykonuj ruchy
303   do {
304     uint NodeToExchane1, NodeToExchane2;
305
306     double FunctionChange;
307
308     bImprovementFound = false;
309
310     // Wylosuj miasto, od którego rozpocznie się przegląd łuków
311     uint StartingNode1 = rand () % NumberOfNodes;
312
313     // Przeglądaj miasta rozpoczynając od StartingNode1
314     // aż do przejrzenia wszystkich miast lub znalezienia ruchu
315     // poprawiającego wartość funckji celu
316     for (uint in1 = 0; (in1 < NumberOfNodes) && !bImprovementFound; in1++) {
317       uint Node1 = (StartingNode1 + in1) % NumberOfNodes;
318
319       // Wylosuj miasto, od którego rozpocznie się przegląd łuków
320       uint StartingNode2 = rand () % NumberOfNodes;
321       // Przeglądaj miasta rozpoczynając od StartingNode2
322       // aż do przejrzenia wszystkich miast lub znalezienia ruchu
323       // poprawiającego wartość funckji celu
324       for (uint in2 = 0; (in2 < NumberOfNodes) && !bImprovementFound; in2++) {
325         uint Node2 = (StartingNode2 + in2) % NumberOfNodes;
326
327         // Jeżeli Node1 i Node2 są różne nie są sąsiadami
328         // i Node1 < Node2 (to pozwala uniknąć dwukrotnego sprawdzania tego
329         // samego ruchu
330         if ((NextNodes [Node2] != Node1) && (NextNodes [Node1] != Node2) &&
           (Node1 != Node2) && (Node1 < Node2)) {
331
332           // Oceń ruch polegający na usnięciu łuków rozpoczynających
333           // się w Node1 i Node2
334           FunctionChange = Distance (Node1, Node2) -
335             Distance (NextNodes [Node1], NextNodes [Node2]) +
336             Distance (Node1, NextNodes [Node1]) -
```

```
337             Distance (Node2, NextNodes [Node2]);
338
339           if  (FunctionChange < 0) {
340             NodeToExchane1 = Node1;
341             NodeToExchane2 = Node2;
342             bImprovementFound = true;
343           }
344         }
345       }
346     }
347     // Jeżeli znaleziono ruch poprawiający
348     if  (bImprovementFound) {
349       ExchangeArcs (NodeToExchane1, NodeToExchane2);
350
351     }
352   } while  (!bImprovementFound);
353 }
354
355 int  main()
356 {
357   srand ((unsigned )time (NULL));
358   if  (Load ("kroa100.txt")) {
359     FindRandom ();
360     Save (cout);
361     GreedyLocalSearch ();
362     Save (cout);
363
364     FindGreedy ();
365     Save (cout);
366     GreedyLocalSearch ();
367     Save (cout);
368   }
369   return  0;
370 }
371
```