Kamil Kamyszek, 234932

WT-7.30

Wrocław, dn. 3 grudnia 2018

prowadzący: Dr inż. Jarosław Mierzwa

Projektowanie Efektywnych Algorytmów

Tabu Search

1 Wstęp Teoretyczny

1.1 Definicja Tabu Search

Przeszukiwanie tabu (Tabu search, TS) – **metaheurystyka** (algorytm) stosowana do rozwiązywania problemów optymalizacyjnych. Wykorzystywana do otrzymywania rozwiązań optymalnych lub niewiele różniących się od niego dla problemów z różnych dziedzin (np. planowanie, planowanie zadań). Twórcą algorytmu jest **Fred Glover**.

Podstawową ideą algorytmu jest przeszukiwanie przestrzeni, stworzonej ze wszystkich możliwych rozwiązań, za pomocą sekwencji ruchów. W sekwencji ruchów istnieją ruchy niedozwolone, **ruchy tabu**. Algorytm unika oscylacji wokół optimum lokalnego dzięki przechowywaniu informacji o sprawdzonych już rozwiązaniach w postaci listy tabu (TL).

1.2 Schemat Ogólny Algorytmu

```
//Absurdalne rozwiązanie startowe (np. 99999999)
S=best=99999999
//Inicjowanie listy tabu
t-[]
//Pętla trwa dopóki nie zostaną spełnione warunki końcowe (zadany czas)
WHILE (czasWykonaniaAlgorytmu<CzasZadanyPrzezUżytkownika)
{
//Wykonanie kroku
S = SELECT (sąsiedztwo(s), t)
//aktualizowanie listy tabu
T = UPDATE_TABU (s,t)
//Najlepsze rozwiązanie zapamiętujemy
If (f(najlepsze)<f(s)
Best = s
}
Return best;
```

1.3 Struktury sąsiedztwa

Ze względu na przestrzeń rozwiązań należy zdefiniować relację sąsiedztwa na parach elementów tej przestrzeni, która obejmuje całą dziedzinę przestrzeni przeszukiwań. Definicja relacji zależy oczywiście od przestrzeni i formatu rozwiązań (np. rozwiązaniami mogą być wektory binarne, wektory liczb rzeczywistych, permutacje zbioru liczb naturalnych itp...).

W wykonanym algorytmie uwzględniam trzy różne listy sąsiedztwa:

- insert(i,j) – przeniesienia j-tego elementu na pozycję i-tą

- swap(i,j) – zamiana miejscami i-tego elementu z j-tym

- invert(i,j) – odwrócenie kolejności w podciągu zaczynającym się na i-tej pozycji i kończącym na pozycji j-tej

1.4 Lista Tabu

Lista Tabu jest to lista ruchów zakazanych, które mają określoną kadencje bycia zakazanymi (w moim programie jest to czas równy ilości miast np. jeśli mamy 47 miast to tabu będzie trwało 47 iteracji).

1.5 Dywersyfikacja

Jest to procedura pozwalająca na przeszukiwanie różnych obszarów przestrzeni stanów. W moim programie wykorzystałem strategię dywersyfikacji nazywaną *metodą zdarzeń krytycznych.* Funkcja **CiriticalEvent()** sprawdza przez ile kolejnych iteracji nie zostało znalezione lepsze rozwiązanie, a następnie jeśli ta ilość przekroczyła ilość krytyczną (w moim przypadku jest to dwukrotność ilości miast.) to generuje nowe rozwiązanie początkowe. Algorytm ponownie rozpoczyna działanie od wygenerowanego rozwiązania.

2 Opis najważniejszych funkcji w projekcie

//Główna funkcja iterująca

```
public void Solution(){
java.util.Collections.shuffle(Path);                          //Wymieszanie listy z miastami
ArrayList<Integer>temporaryNewSolution;                       //Lista przechowująca tymczasowe wymieszane miasta
theBestSolutionCost =pathCostList(Path);                      //Przypisanie najlepszego kosztu
ArrayList<Integer> BestSolution= new ArrayList<>();           //Lista z drogą o najniższym obliczonym koszcie
timeCounterStart=System.currentTimeMillis();                  //Rozpoczęcie liczenia wykonywania algorytmu
while(timeCounterStop-timeCounterStart<=(Integer.parseInt(algorithmWorkTimeTextField.getText())*60000))

//Jeśli czasWykonania<czasPodany to wykonuj pętle:
{
   int city_1=0;                                              //2 miasta których droga będzie wpisywana do listy tabu
   int city_2=0;
   int bestCost=999999;                                       //startowy absurdalny koszt drogi
   for(int j=1;j<m_size;j++){                                 //Pierwsze miasto jest miastem startowym dlatego zaczynamy od 1
      for(int k=2;k<m_size;k++){                              //rozpoczęte od 2 a nie od 1 bo od nie można isc do siebie samego
         if(j!=k){                                            //warunek pozwalający nie chodzić do tych siebie samego w przyszlości
            temporaryNewSolution= new ArrayList<>();          //Tworzenie nowej listy z miastami przypisanie im wymieszanej listy miast
            for(int m=0;m<Path.size();m++){
               temporaryNewSolution.add(Path.get(m));
            }                                                 //W zależności od wybranej definicji sasiędztwa wykonuje się inny warunek
            if(swapRadioButton.isSelected())
            swap(temporaryNewSolution,j,k);                   //SWAP
            if(insertRadioButton.isSelected())
            insert(temporaryNewSolution,j,k);                 //INSERT
```

```
            if(invertRadioButton.isSelected())
            invert(temporaryNewSolution,j,k);                    //INVERT
            int newCost=pathCostList(temporaryNewSolution);      //Nowy koszt dla obliczonej drogi

            if(newCost<bestCost && tabuList[j][k]==0){          //Jeśli nowy koszt mniejszy niż poprzedni to przypisanie go jako najlepszy
              BestSolution= new ArrayList<>();
              for(int m=0;m<Path.size();m++){                    //Przypisanie jednocześnie nowej najlepszej drogi
                 BestSolution.add(temporaryNewSolution.get(m));
              }
              city_1=j;                                          //Miasta z nowego najlepszego kosztu wpisane będą do Listy Tabu
              city_2=k;
              bestCost=newCost;
            }
          }
        }
      }
      if(city_1 !=0){
        decrementation();                                       //Funkcja dekrementująca kadencje miast w liście tabu
        addTabu(city_1,city_2);                                 //Dodanie miast do listy tabu
      }
      if(theBestSolutionCost>bestCost){                         //Ustalenie najlepszego kosztu z najlepszych
        timeOfFoundTheBestPath=System.currentTimeMillis();
        theBestSolutionCost=bestCost;
        theBestSolutionPath= new ArrayList<>();
        for(int m=0;m<BestSolution.size();m++){
          theBestSolutionPath.add(BestSolution.get(m));      //Lista zawierająca najlepszy koszt
        }
       FinalTime=timeOfFoundTheBestPath-timeCounterStart;     //Czas odkrycia najlepszego kosztu
      }                                                         //przejscie do funkcji obliczającej nową definicje sąsiedztwa dla kolejnej iteracji
      if(swapRadioButton.isSelected())
        swap(Path,city_1,city_2);
      if(insertRadioButton.isSelected())
        insert(Path,city_1,city_2);
      if(invertRadioButton.isSelected())
        invert(Path,city_1,city_2);
      criticalEvent(bestCost,theBestSolutionPath);              //Kryterium dywersyfikacji: critivalEvent()
      timeCounterStop=System.currentTimeMillis();               //Licznik czasu

    }
}
```

//Funkcja obliczająca koszt drogi w danej liscie miast

```
public int pathCostList(ArrayList<Integer> Path){
  cost =0;                                                      //Startowy koszt = 0;
  for(int i=0;i<Path.size()-1;i++){                             //iteracja po kolejnych elementach listy
    cost=cost+cities[Path.get(i)][Path.get(i+1)];               //dodawanie kosztu drogi danego miasta
  }
  cost=cost+cities[Path.get(Path.size()-1)][Path.get(0)];       //koszt powrotu do miasta początkowego
  return cost;
}
```

//Funkcja zamieniająca i-te miasto z j-tym

```
public void swap (ArrayList<Integer>Path, int i ,int j){
  int tmp = Path.get(i);                                        //Zmienna pomocnicza
  Path.set(i,Path.get(j));                                  //Zamiana
  Path.set(j,tmp);
}
```

//Funkcja wstawiajaca j-ty element na miejsce i-tego

```
public void insert(ArrayList<Integer>Path, int i ,int j){
   int pos = Path.indexOf(Path.get(i));                    //pozycja i-tego elementu
   int pos2=Path.indexOf(Path.get(j));                     //pozycja j-tego elementu
   int temporary = Path.get(j);                            //pomocnicza zmienna
   Path.remove(pos2);                                              //usuniecie j-tego elementu

//wstawienie j-tego elementu na miejsce i-tego i przesuniecie listy o jeden
   Path.add(pos,temporary);
}
```

//Funkcja odwracająca kolejność w podciągu zaczynającym się na i-tej pozycji i kończącym na pozycji j-tej

```
public void invert(ArrayList<Integer>Path, int i ,int j){
     ArrayList<Integer> temporary_1 = new ArrayList<>();          //Lista pomocnicza
     ArrayList<Integer> temporaryReverse=new ArrayList<>();       //Lista pomocnicza do odwrócenia
     int pos1 = Path.indexOf(Path.get(i));                        //wartość i
     int pos2=Path.indexOf(Path.get(j));                                  //wartość j
     if(i<j){                                                             //Jeśli i<j
       for(int h=0; h<pos1;h++){                          //dopóki nie dojdzie do i→ dodawaj do listy
         temporary_1.add(Path.get(h));
       }
       for(int h=pos1;h<pos2;h++){                                //odwrócenie wartości w przedziale od i do j
         temporaryReverse.add(Path.get(h));
       }Collections.reverse(temporaryReverse);
       for(int h=0; h<temporaryReverse.size();h++){
         temporary_1.add(temporaryReverse.get(h));
       }
       for(int h=pos2; h<Path.size();h++){                //dopisanie wartości za J-tym elementem
         temporary_1.add(Path.get(h));
       }
     }
   if(i>j){
     for(int h=0; h<pos2;h++){
       temporary_1.add(Path.get(h));
     }
     for(int h=pos2;h<pos1;h++){
       temporaryReverse.add(Path.get(h));
     }Collections.reverse(temporaryReverse);
     for(int h=0; h<temporaryReverse.size();h++){
       temporary_1.add(temporaryReverse.get(h));
     }
     for(int h=pos1; h<Path.size();h++){
       temporary_1.add(Path.get(h));
     }
   }

Path.clear();                                                         //Czyszczenie głownej listy dróg
for(int h=0;h<temporary_1.size();h++)                    //Przypisanie nowej kolejności miast do listy
{
Path.add(temporary_1.get(h));
}
```

//Funkcja odpowiedzialna za dywersyfikacje

```
public void criticalEvent(int bestCost,ArrayList<Integer> bestBestSol){
  if(theBestSolutionCost<bestCost){                    //Jesli kolejna iteracja nie dala lepszego kosztu
    critical++;
  }
  else{                                                                    //Jeśli znaleziono lepszy koszt
    critical=0;

  }

  if(critical>maxCritical){                           //Jeśli wartość krytyczna przekroczyła max
    ArrayList<Integer> tmp = new ArrayList<>();   //Tworzenie listy tymczasowej
    for(int n=0;n<Path.size();n++){
      tmp.add(Path.get(n));
    }
    for(int m=0;m<m_size;m++){
      java.util.Collections.shuffle(tmp);              //mieszanie listy tymczasowej
      if(pathCostList(Path)>pathCostList(tmp)){    //jeśli w tymczasowej znaleziono lepszą droge
        Path= new ArrayList<>();                       //to tworze nową listę z tą drogą
        for(int n=0;n<Path.size();n++){
          Path.add(tmp.get(n));
        }
        if(theBestSolutionCost>pathCostList(Path)){  //to samo z lepszym kosztem
          bestBestSol= new ArrayList<>();
          for(int n=0;n<Path.size();n++){
            bestBestSol.add(Path.get(n));
          }
        }
      }
    }
    for(int m=0;m<m_size;m++){                        //Czyszczenie tabu
      for(int n=0;n<m_size;n++){
        tabuList[m][n]=0;
      }
    }
    critical=0;
  }
}
```

//Funkcja dodająca do listy tabu nowe miasta

```
public void addTabu(int city1,int city2){
  tabuList[city1][city2]+= time;                  //przypisanie kadencji do miast
  tabuList[city2][city1]+= time;                //kadencje dostaje jednocześnie droga np. 4→3 i 3→4
}
```

//funkcja dekrementująca liste tabu

```
public void decrementation(){
  for(int i=0;i<m_size;i++){                                    //pętla przeszukująca liste tabu
    for(int j=0;j<m_size;j++){
      if(tabuList[i][j]>0){                                     //Jeśli kadencja > 0
        tabuList[i][j]--;                                              //Zmniejszenie kadencji
      }
    }
  }
}
```

3 Wyniki pomiarów

ftv47.atsp		Sąsiedztwo: SWAP	Czas: 2 minuty
Koszt znaleziony	Moment znalezienia (ms)	Koszt optymalny	Błąd względn
1856	96476	1776	0,045045045
1825	117871	1776	0,02759009
1879	55451	1776	0,057995495
1838	45718	1776	0,03490991
1840	38355	1776	0,036036036
1841	79116	1776	0,036599099
1836	114046	1776	0,033783784
1867	55112	1776	0,051238739
1855	30622	1776	0,044481982
1852	15804	1776	0,042792793
ftv47.atsp		Sąsiedztwo: INSERT	Czas: 2 minuty
Koszt znaleziony	Moment znalezienia (ms)	Koszt optymalny	Błąd względny
2044	359	1776	0,150900901
1957	727	1776	0,101914414
2034	211	1776	0,14527027
1856	2457	1776	0,045045045
1797	680	1776	0,011824324
1849	1085	1776	0,041103604
1818	1809	1776	0,023648649
2097	103	1776	0,180743243
1925	5507	1776	0,083896396
1840	3397	1776	0,036036036
ftv47.atsp		Sąsiedztwo: INVERT	Czas: 2 minuty
Koszt znaleziony	Moment znalezienia (ms)	Koszt optymalny	Błąd względny
3185	236	1776	0,793355856
3652	74	1776	1,056306306
3411	282	1776	0,920608108
3281	203	1776	0,84740991
3080	168	1776	0,734234234
3656	65	1776	1,058558559
3191	851	1776	0,796734234
3406	458	1776	0,917792793
4008	43	1776	1,256756757
3241	101	1776	0,824887387

Z przeprowadzonych wyżej pomiarów wynika, że przy zastosowaniu definicji sąsiedztwa SWAP, w której to zamieniamy i-ty element z j-tym, można uzyskać najczęściej koszt zbliżony do tego optymalnego. Na potwierdzenie tego stwierdzenia został obliczony błąd względny, który przy pierwszej metodzie jest najmniejszy. Przy definicji sąsiedztwa INSERT wyniki również zbytnio nie odbiegają od optymalnego rozwiązania. Sytuacja zmienia się podczas użycia sąsiedztwa INVERT,

której to wyniki odbiegają znacząco i koszty przejścia 47 miast są czasem nawet dwukrotnie wyższe od optymalnych. Zaciekawić może również moment znalezienia najlepszego rozwiązania. Dla metody SWAP oscyluje on w przedziale od 15 sekund do 2 minut (warto zaznaczyć, że algorytm miał ogranicznik czasowy 2 minut) co jest imponujące w porównaniu z dwiema kolejnymi definicjami sąsiedztwa, które wskazują na dużo mniejsze wyniki. Według przeprowadzonych pomiarów można dojść do konkluzji, iż definicja sąsiedztwa INSERT jest najlepsza, jeśli chcemy uzyskać, w krótkim czasie wynik mocno zbliżony do optymalnego, gdyż rozwiązanie to jest znajdowane w przedziale od 0.3 do 5.5 sekundy. Na dodatek przy tej metodzie został znaleziony koszt 1797, który jest najbliższy optymalnemu. Jednak, jeśli policzymy średni koszt i porównamy obie metody to dokładniejsza będzie definicja SWAP.

ftv170.atsp		Sąsiedztwo: SWAP	Czas: 4 minuty
Koszt znaleziony	Moment znalezienia (ms)	Koszt optymalny	Błąd względny
3950	175649	2755	0,433756806
4475	203215	2755	0,624319419
4165	215975	2755	0,511796733
4681	216780	2755	0,699092559
4397	206337	2755	0,59600726
4701	223019	2755	0,706352087
4162	228546	2755	0,510707804
4506	231547	2755	0,635571688
4314	226208	2755	0,565880218
4428	233788	2755	0,607259528
ftv170.atsp		Sąsiedztwo: INSERT	Czas: 4 minuty
Koszt znaleziony	Moment znalezienia (ms)	Koszt optymalny	Błąd względny
4553	82010	2755	0,652631579
4846	23718	2755	0,758983666
5182	108532	2755	0,880943739
4209	101488	2755	0,527767695
4550	18353	2755	0,65154265
4847	37906	2755	0,759346642
4843	96971	2755	0,757894737
5028	59068	2755	0,825045372
5328	20911	2755	0,933938294
4832	212922	2755	0,753901996

ftv170.atsp		Sąsiedztwo: INVERT	Czas: 4 minuty
Koszt znaleziony	Moment znalezienia (ms)	Koszt optymalny	Błąd względny
10317	43172	2755	2,744827586
10798	40407	2755	2,919419238
11554	31127	2755	3,193829401
10162	40874	2755	2,688566243
10511	36352	2755	2,815245009
11525	31930	2755	3,183303085
10868	99761	2755	2,944827586
10325	47712	2755	2,747731397
47712	50129	2755	16,31833031
10541	18103	2755	2,826134301

Wyniki pomiarów przy grupie 170 miast są bardzo podobne do poprzednich. Używając definicji sąsiedztwa SWAP możemy spodziewać się dokładniejszego wyniku znalezionego pod koniec działania algorytmu (4 minuty). W definicji INSERT wynik minimalnie bardziej odbiega od optymalnego jednak zostaje znaleziony dużo szybciej. Przy ostatniej metodzie niestety nie można spodziewać się wyniku nawet zbliżonego do tego optymalnego. Warto wspomnieć, że koszt najlepszy obliczony dla 170 miast w 4 minuty to 3950. Koszt optymalny natomiast to 2755 co daje błąd względny na poziomie 0.43. W porównaniu z poprzednimi obliczeniami dla 47 miast w 2 minuty, w których najmniejszy uzyskany błąd względny wyniósł 0.01 jest do zdecydowane pogorszenie. Przypuszczam, że poprawienie wyniku byłoby możliwe, gdyby czas algorytmu zwiększono.

rgb403.atsp		Sąsiedztwo: SWAP	Czas: 6 minut
Koszt znaleziony	Moment znalezienia (ms)	Koszt optymalny	Błąd względny
2676	356601	2465	0,085598377
2686	335180	2465	0,089655172
2677	357483	2465	0,086004057
2734	358287	2465	0,109127789
2710	329123	2465	0,099391481
2743	333814	2465	0,112778905
2708	360549	2465	0,098580122
2643	343535	2465	0,072210953
2682	343535	2465	0,088032454
2645	345382	2465	0,073022312

rgb403.atsp		Sąsiedztwo: INSERT	Czas: 6 minut
Koszt znaleziony	Moment znalezienia (ms)	Koszt optymalny	Błąd względny
2585	293373	2465	0,048681542
2589	297471	2465	0,05030426
2561	277172	2465	0,038945233
2524	280796	2465	0,023935091
2584	290688	2465	0,048275862
2555	294813	2465	0,036511156
2562	302543	2465	0,039350913
2542	301987	2465	0,031237323
2586	285379	2465	0,049087221
2578	296288	2465	0,045841785
rgb403.atsp		Sąsiedztwo: INVERT	Czas: 6 minut
Koszt znaleziony	Moment znalezienia (ms)	Koszt optymalny	Błąd względny
5289	325789	2465	1,145638945
5072	282642	2465	1,057606491
5148	286752	2465	1,088438134
5102	277382	2465	1,069776876
5194	256113	2465	1,107099391
5325	332615	2465	1,160243408
5228	330844	2465	1,120892495
5244	293092	2465	1,127383367
5131	361576	2465	1,081541582
5175	289942	2465	1,099391481

Przyglądając się wynikom uruchomionego algorytmu tabu search przez 6 minut dla 407 miast można dojść do wniosku, że definicja sąsiedztwa INSERT jest najlepsza. Znaleziony koszt drogi jest zbliżony do optymalnego (co pokazuje bardzo mały błąd względny) i jednocześnie koszt ten został znaleziony najszybciej z wszystkich trzech definicji. Porównywalna, jednak niewiele mniej dokładna definicja sąsiedztwa SWAP jest również dobrym wyborem w przypadku obliczania drogi. Niestety i tym razem metoda INVERT jest najmniej dokładna i generuje ponad dwukrotnie dłuższą drogę.

3.1 Ścieżki dla najlepszych uzyskanych rozwiązań

3.1.1 47 miast

SWAP (koszt:1825): 33 27 28 2 41 43 42 22 19 44 15 18 17 12 7 23 34 13 46 36 35 14 16 45 39 21 40 20 38 37 0 25 47 26 1 11 8 32 6 3 24 4 29 30 31 5 10 9 33

INSERT (koszt: 1797): 19 44 20 38 37 0 25 18 17 12 32 7 23 34 13 46 36 14 35 15 16 45 39 21 40 47 26 1 10 11 8 3 24 4 29 30 5 31 6 9 33 27 28 2 41 43 42 22 19

INVERT (koszt: 3080): 23 13 32 8 24 29 5 31 4 27 28 2 42 10 11 0 25 47 40 20 38 37 19 44 15 46 36 35 14 34 7 30 3 33 43 22 39 16 18 17 45 21 26 41 1 12 9 6 23

3.1.2 170 miast

SWAP (koszt: 3950): 162 122 128 132 133 6 10 76 74 75 11 12 20 21 22 23 24 15 25 149 148 136 130 131 164 127 116 117 118 119 124 135 7 159 16 32 158 36 31 30 28 29 27 26 150 161 160 151 152 142 134 126 125 129 146 145 144 143 147 137 138 139 140 141 8 14 13 17 18 19 37 38 35 157 33 34 156 45 49 73 77 110 109 108 166 154 89 90 91 85 86 83 84 71 60 39 40 155 41 42 54 58 62 63 57 61 68 167 70 92 93 94 96 165 163 102 120 121 103 88 153 87 69 67 66 65 64 56 55 46 47 48 52 53 43 44 51 59 50 170 168 72 78 82 79 80 81 0 1 2 3 4 9 5 169 111 112 113 115 104 114 107 106 105 97 99 98 95 100 101 123 162

INSERT (koszt: 4209): 60 50 51 52 53 43 55 54 58 59 68 67 167 70 87 86 93 107 106 105 97 165 104 114 109 108 83 84 71 49 73 1 2 3 9 4 7 8 14 13 17 21 20 37 38 39 40 47 48 170 77 0 111 112 164 130 135 134 6 152 151 142 141 131 113 127 126 125 129 146 145 144 143 137 138 139 140 115 116 117 118 119 120 121 124 136 147 148 149 161 160 150 25 26 27 28 30 31 33 34 156 44 45 46 168 72 78 82 79 80 81 110 166 92 94 96 99 100 102 103 163 95 91 154 88 153 69 66 63 64 56 57 62 61 85 75 11 10 76 74 12 18 19 32 158 36 41 155 42 65 89 90 98 101 123 162 122 128 132 35 157 29 22 23 24 15 159 16 5 133 169 60

INVERT (koszt: 10162): 127 126 117 122 130 138 144 25 26 11 50 51 165 163 90 88 70 86 45 78 12 18 19 170 71 77 128 3 5 134 169 110 99 100 104 98 101 123 118 121 145 139 32 33 34 155 39 44 58 49 82 79 0 1 4 14 150 152 7 8 10 159 13 28 41 46 60 67 65 89 153 166 92 94 103 119 141 131 125 146 6 151 15 17 21 27 23 20 158 55 112 132 108 93 97 91 85 63 57 69 168 35 22 24 9 80 111 107 106 102 162 120 124 129 147 148 149 113 114 164 116 133 142 143 140 2 81 72 37 16 30 36 157 47 61 52 53 48 73 76 74 75 29 31 43 54 154 135 136 115 68 66 64 56 62 167 87 105 95 96 137 109 83 84 38 161 160 40 156 42 59 127

3.1.3 403 miasta

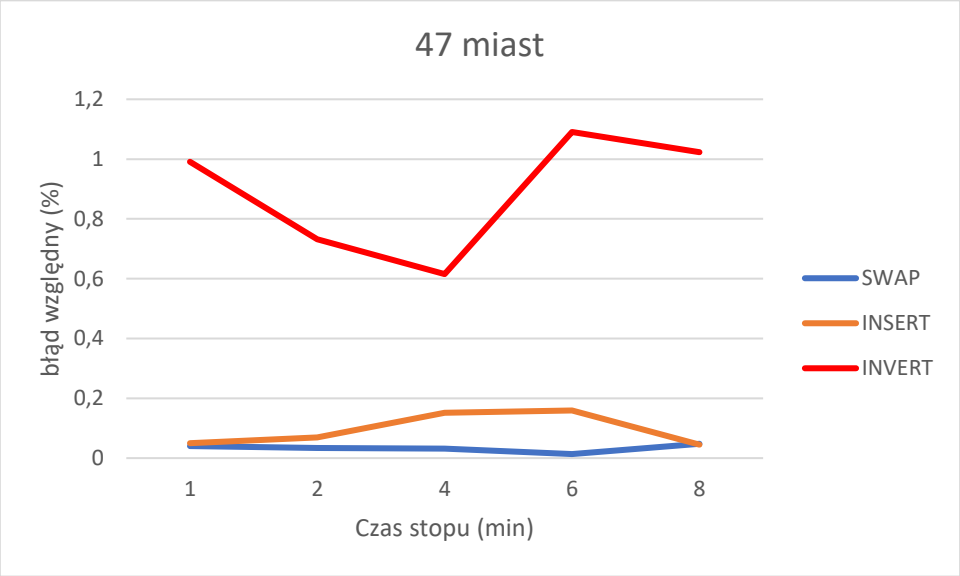
SWAP (koszt: 2643): 201 89 151 101 183 17 12 402 64 365 245 125 273 106 165 103 254 223 400 176 156 325 235 121 77 31 344 153 310 168 253 238 229 14 62 364 303 198 152 76 66 60 44 387 145 353 352 142 218 384 383 207 45 338 85 93 9 361 51 298 373 355 115 94 97 179 368 68 194 324 212 160 15 190 100 78 382 381 331 378 266 302 118 11 148 154 161 285 282 272 356 286 74 309 50 333 267 327 307 7 32 322 107 386 2 98 164 3 126 55 48 47 113 293 163 25 149 146 336 292 343 131 186 53 134 306 319 372 41 340 81 22 21 128 33 376 58 8 305 37 374 354 200 398 396 370 328 281 332 215 6 158 1 375 371 395 82 247 213 28 314 59 167 36 184 4 318 124 339 193 95 271 357 330 141 392 217 203 295 178 138 297 335 251 358 350 166 67 189 240 219 140 137 54 79 109 135 280 279 86 342 377 308 394 206 30 43 34 105 19 57 221 199 202 23 225 188 173 29 208 91 52 40 155 143 363 379 230 226 147 111 369 360 393 90 72 71 175 88 69 150 84 241 112 133 222 250 299 49 296 136 61 46 246 123 192 114 96 209 278 337 104 349 171 63 224 239 117 252 191 187 304 5 87 56 182 262 261 211 391 390 24 13 288 159 323 73 177 99 389 312 35 268 16 144 220 197 316 351 257 397 260 195 366 321 362 162 237 367 75 265 326 258 174 185 38 270 388 348 290 42 172 26 234 228 18 130 169 27 301 315 92 347 157 269 236 227 320 102 127 399 122 283 277 263 345 276 259 232 116 129 255 242 70 385 300 401 214 119 216 132 243 256 249 289 181 346 180 359 311 10 233 380 291 204 139 110 275 210 65 108 120 284 329 294 231 205 264 274 317 170 334 287 83 248 313 20 341 244 80 39 196 0 201

INSERT (koszt: 2524): 263 314 59 170 334 217 9 181 346 338 140 66 60 98 221 389 360 69 245 111 369 158 339 388 348 290 308 394 14 283 277 352 172 26 246 244 117 234 228 71 141 392 249 318 166 67 94 240 219 186 53 214 262 261 361 347 341 127 192 207 0 193 42 366 292 343 345 299 298 291 315 304 260 176 31 344 218 349 400 177 174 250 242 70 153 75 254 223 168 152 310 73 82 247 178 138 297 268 16 163 25 149 51 259 43 169 271 88 96 189 336 321 289 372 41 79 62 13 205 204 137 54 239 80 237 162 48 2 386 150 34 335 401 115 160 365 213 356 105 391 258 136 61 357 358 293 95 55 332 296 295 241 112 322 317 312 35 235 121 47 280 279 208 77 156 210 311 27 373 333 267 93 384 383 203 157 284 125 74 216 200 337 252 371 46 129 85 161 148 154 135 270 209 278 266 23 393 159 7 81 22 21 230 226 147 286 273 327 180 340 331 378 57 123 396 395 232 108 120 116 402 287 83 91 399 198 313 20 236 227 212 367 76 119 305 215 29 253 211 78 382 320 110 326 390 128 255 167 36 183 17 12 397 5 155 323 309 109 103 276 49 37 106 165 142 385 199 175 294 107 307 143 238 229 225 196 45 24 302 89 38 233 324 144 87 56 33 376 68 184 4 375 191 187 194 380 379 179 368 222 185 164 3 19 122 206 30 126 10 132 288 86 11 151 101 64 15 364 303 377 374 363 362 32 328 370 330 329 104 145 131 355 6 134 243 316 351 281 90 72 18 201 118 182 265 275 325 133 269 50 301 256 220 197 92 146 381 190 100 44 102 124 1 195 264 113 257 188 173 319 130 97 251 274 272 28 359 387 84 350 171 63 224 65 99 300 285 282 231 306 52 40 39 202 248 58 8 354 342 398 139 114 353 263

INVERT (koszt: 5072): 321 22 215 88 169 55 101 389 77 113 279 374 398 108 56 12 316 36 85 41 303 230 272 96 114 95 352 388 329 26 34 35 44 286 323 360 341 79 38 317 222 236 121 118 8 119 16 147 265 146 70 384 62 328 293 231 258 175 120 287 271 192 400 387 300 224 87 382 247 239 221 84 288 348 106 189 332 152 309 217 268 68 244 134 5 151 173 73 240 154 135 280 237 133 23 344 308 199 229 399 213 383 117 294 395 82 391 373 20 48 100 168 331 4 59 148 336 371 358 60 30 248 150 155 127 367 75 278 29 183 232 357 257 176 289 50 259 111 27 334 161 153 310 86 226 381 6 285 353 262 89 261 139 393 190 350 178 263 253 185 212 266 181 376 338 201 298 326 235 324 365 339 97 233 138 122 7 52 198 92 397 66 128 172 305 37 204 214 340 187 241 45 284 337 304 319 345 378 93 392 13 94 104 51 325 132 98 112 19 39 314 390 158 163 2 327 64 219 343 90 184 196 191 208 260 318 245 130 354 43 197 249 157 193 267 270 346 69 33 282 243 379 375 368 385 143 171 264 205 362 18 162 380 320 149 322 364 126 21 109 275 159 359 160 170 211 227 312 295 81 209 225 142 174 99 369 165 32 386 47 57 218 124 372 186 110 351 107 61 46 311 10 299 347 102 342 166 9 3 292 202 269 182 206 283 17 203 145 76 63 307 28 177 281 140 402 167 302 91 355 164 137 306 78 228 220 276 254 313 277 401 207 0 136 210 65 361 116 58 53 24 356 54 131 1 25 11 195 156 83 125 250 291 330 333 252 129 255 290 366 396 15 194 144 251 80 363 315 123 179 297 246 301 394 14 180 49 74 141 349 40 377 105 234 273 370 256 71 188 296 72 242 335 103 274 31 216 200 223 115 42 238 67 321

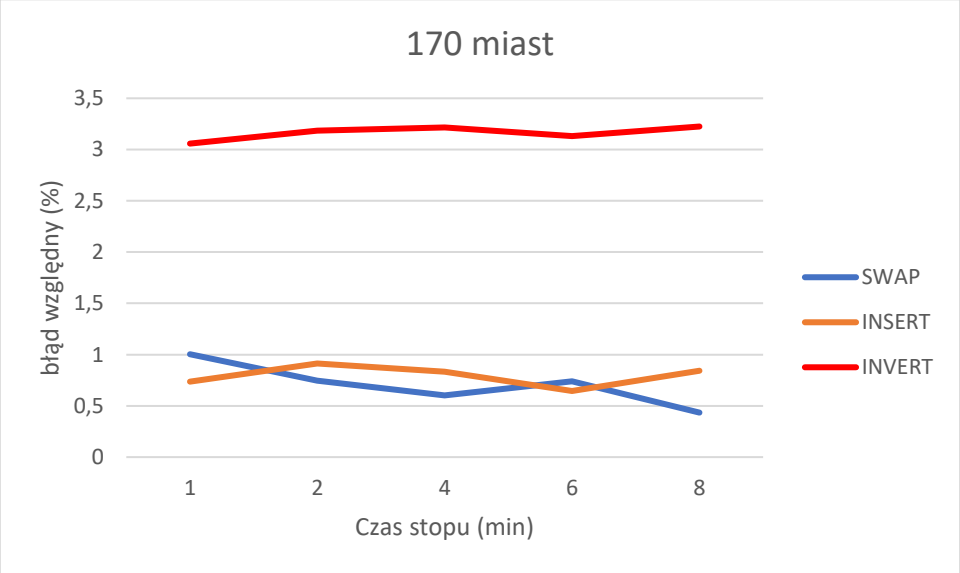
3.2 Wykres błędu w funkcji czasu

ftv47.atsp		Sąsiedztwo: SWAP	
Koszt znaleziony	Koszt optymalny	Błąd względny	Czas (min)
1848	1776	0,040540541	1
1836	1776	0,033783784	2
1832	1776	0,031531532	4
1800	1776	0,013513514	6
1861	1776	0,04786036	8
ftv47.atsp		Sąsiedztwo: INSERT	
Koszt znaleziony	Koszt optymalny	Błąd względny	Czas (min)
1864	1776	0,04954955	1
1899	1776	0,069256757	2
2045	1776	0,151463964	4
2059	1776	0,159346847	6
1856	1776	0,045045045	8
ftv47.atsp		Sąsiedztwo: INVERT	
Koszt znaleziony	Koszt optymalny	Błąd względny	Czas (min)
3536	1776	0,990990991	1
3075	1776	0,731418919	2
2869	1776	0,615427928	4
3713	1776	1,090653153	6
3593	1776	1,023085586	8



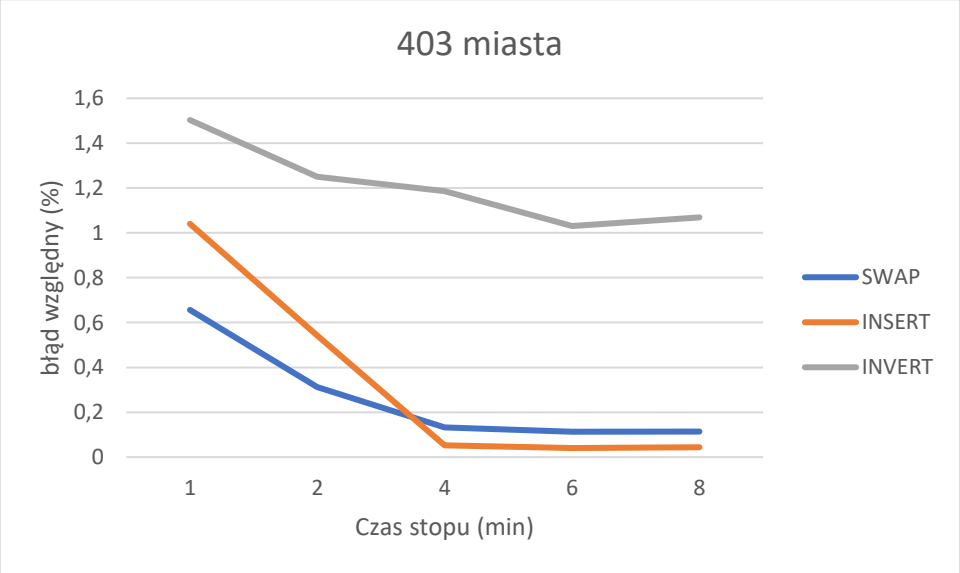
Po wykresie można wywnioskować, że błąd względny jest najmniejszy dla dwóch definicji sąsiedztwa: SWAP i INSERT co pokrywa się z pomiarami dokonanymi wyżej dla 10 instancji w czasie 2 minut. Sąsiedztwo INVERT jest obarczone największym błędem.

ftv170.atsp		Sąsiedztwo: SWAP	
Koszt znaleziony	Koszt optymalny	Błąd względny	Czas (min)
5520	2755	1,003629764	1
4811	2755	0,746279492	2
4415	2755	0,602540835	4
4796	2755	0,740834846	6
3951	2755	0,434119782	8
ftv170.atsp		Sąsiedztwo: INSERT	
Koszt znaleziony	Koszt optymalny	Błąd względny	Czas (min)
4786	2755	0,737205082	1
5271	2755	0,913248639	2
5051	2755	0,833393829	4
4532	2755	0,645009074	6
5072	2755	0,841016334	8
ftv170.atsp		Sąsiedztwo: INVERT	
Koszt znaleziony	Koszt optymalny	Błąd względny	Czas (min)
11180	2755	3,058076225	1
11530	2755	3,185117967	2
11616	2755	3,216333938	4
11385	2755	3,132486388	6
11639	2755	3,224682396	8



W przypadku błędu względnego dla 170 miast oscyluje on bliżej 0.5% jednak ze zwiększającym się czasem powoli maleje. Poza tą zmianą algorytmy uruchomione z podanymi definicjami sąsiedztwa zachowują się podobnie jak dla 47 miast. Wyniki są zgodne jednocześnie z tymi dla 10 instancji 170 miast z czasem stopu 4 minuty.

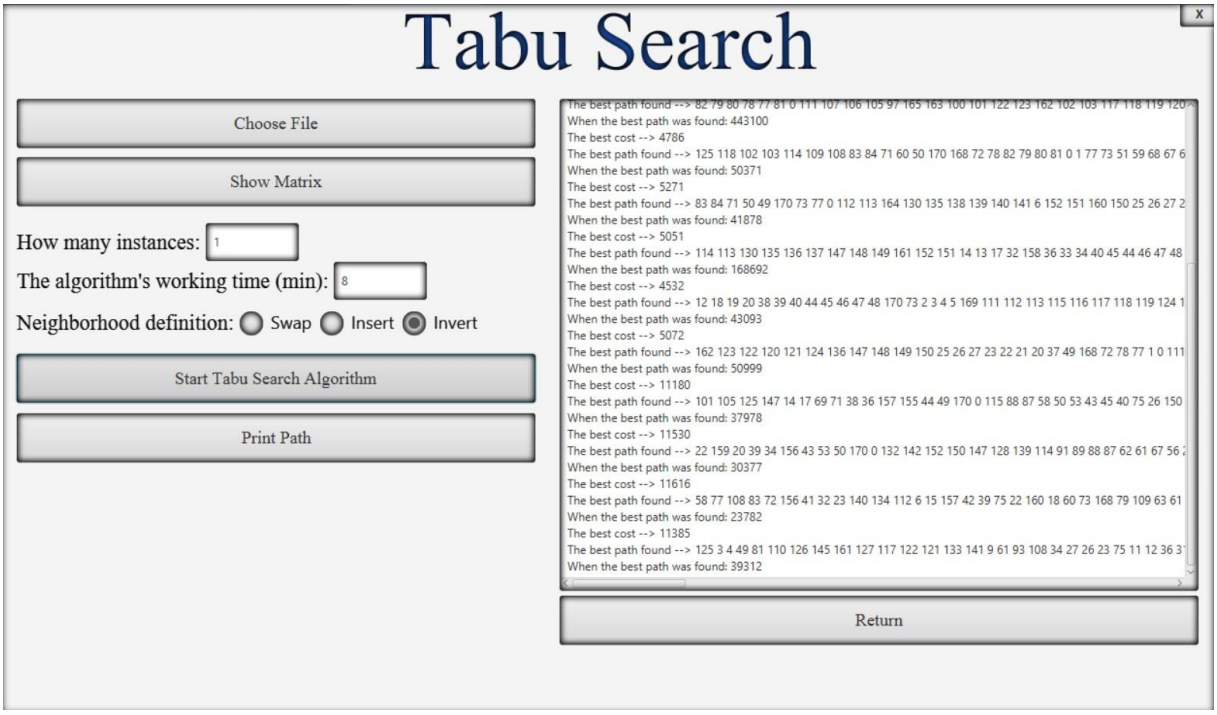
rgb403.atsp		Sąsiedztwo: SWAP	
Koszt znaleziony	Koszt optymalny	Błąd względny	Czas (min)
4083	2465	0,656389452	1
3234	2465	0,311967546	2
2791	2465	0,132251521	4
2744	2465	0,113184584	6
2747	2465	0,114401623	8
rgb403.atsp		Sąsiedztwo: INSERT	
Koszt znaleziony	Koszt optymalny	Błąd względny	Czas (min)
5029	2465	1,040162272	1
3800	2465	0,54158215	2
2596	2465	0,053144016	4
2565	2465	0,040567951	6
2574	2465	0,044219067	8
rgb403.atsp		Sąsiedztwo: INVERT	
Koszt znaleziony	Koszt optymalny	Błąd względny	Czas (min)
6170	2465	1,503042596	1
5547	2465	1,25030426	2
5388	2465	1,185801217	4
5005	2465	1,030425963	6
5099	2465	1,068559838	8



Patrząc na powyższy wykres można dojść do stwierdzenia, że dla 403 miast im więcej czasu damy na obliczenie tym szybciej błąd względny będzie mniejszy, a co za tym idzie znaleziony koszt drogi bardziej zbliżony do optymalnego. Jednak i w tym przypadku definicja sąsiedztwa INVERT jest najmniej efektywna.

4 Implementacja Graficzna

W celu łatwiejszego dokonywania pomiarów stworzony został przyjazny użytkownikowi interfejs graficzny wykonany w technologii JavaFX.



5 Źródła

[1] https://cs.pwr.edu.pl/zielinski/lectures/om/localsearch.pdf?fbclid=IwAR2Y_K00u4gSZNwiRLIKqn-anqQPZcPuuUZyuCjXWIZeQO74AIm_-icy_R8

[2] http://www.zio.iiar.pwr.wroc.pl/pea/w5_ts.pdf

[3] http://www.cs.put.poznan.pl/mhapke/TO-TS2.pdf

[4] https://pl.wikipedia.org