Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Poniedziałek 17:05*
Temat *Algorytm genetyczny i mrówkowy*	**Problem** *TSP*
Skład grupy ****** *Piotr Kozerski*	**Nr grupy** -
Prowadzący	**data** *25 stycznia 2020*

Spis treści

1 Wstęp **3**

2 Problem komiwojażera **3**

3 Algorytm genetyczny **3**
3.1 Sposób działania 3
3.2 Sposób implementacji 3
3.3 Analiza działania 5
3.3.1 Wynik działania algorytmu a wartości optymalne 5
3.3.2 Wynik działania w zależności od wielkości populacji 6
3.3.3 Wynik działania w zależności od prawdopodobieństwa mutacji 6
3.3.4 Wynik działania w zależności od ilości krzyżowań na iterację 7

4 Algorytm mrówkowy **8**
4.1 Sposób działania 8
4.2 Sposób implementacji 8
4.3 Analiza działania 10

5 Podsumowanie **11**

1 Wstęp

Dokument zawiera podsumowanie implementacji oraz analizy heurystyk w ramach trzeciego etapu projektu w ramach kursu *Projektowanie Efektywnych Algorytmów*. W ramach etapu ukończono implementację algorytmu genetycznego oraz mrówkowego.

2 Problem komiwojażera

Problem komiwojażera jest zagadnieniem optymalizacyjnym polegającym na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Należy znaleźć taką ścieżkę spośród węzłów podanego grafu, która uwzględnia każdy węzeł dokładnie raz jednocześnie zachowując minimalną wartość kosztu przejścia przez łączące je krawędzie. Celem projektu było zaprojektowanie algorytmów rozwiązujących ten problem na kilka możliwych sposobów.

Etap projektu obejmuje implementację algorytmów opartych na tzw. heurystyce. Algorytmy te nie dają gwarancji znalezienie optymalnego rozwiązania, są jednak wyraźnie szybsze niż algorytmy dokładne dając jednocześnie zadowalające w wielu przypadkach przybliżenie wyniku.

3 Algorytm genetyczny

3.1 Sposób działania

Algorytm wzorowany jest na zjawisku ewolucji zachodzącej w środowisku naturalnym. Populacją jest w tym przypadku zbiór potencjalnych rozwiązań problemu TSP. Przystosowanie każdego osobnika do środowiska w którym się znajduje określone jest funkcją obliczającą długość znalezionej drogi poprzez wszystkie węzły grafu. Osobnik o najmniejszym koszcie przejścia ścieżki jest więc przystosowany najlepiej. Jedna iteracja algorytmu odpowiada jednemu pokoleniu populacji. Każda iteracja rozpoczyna się od mutacji genomu. Następnie następuje krzyżowanie generujące nowe osobniki. Cała populacja zostaje w kolejnym kroku poddana selekcji metodą rank (n najlepszych osobników pozostaje w populacji, reszta jest odrzucona). Wynikiem działania algorytmu jest przystosowanie osobnika, który uzyskał najlepszy wynik w którejkolwiek z iteracji.

3.2 Sposób implementacji

W listingu 1 umieszczono główną funkcję algorytmu, której wywołanie rozpoczyna proces obliczeń. Przyjmuje ona jako argumenty parametry pozwalające dostosować zachowanie algorytmu.

- matrix - przechowuje informacje nt. grafu dla którego rozwiązywany jest problem TSP
- maxIterations - maksymalna liczba iteracji jaką może wykonać algorytm
- populationSize - ilość osobników w populacji
- selectionType - obiekt typu *enum* pozwalający określić sposób selekcji
- crossingStrategy - obiekt typu *enum* pozwalający określić sposób krzyżowania osobników
- mutationProbability - prawdopodobieństwo dokonania mutacji na pojedynczym fragmencie genomu. Warto zwrócić uwagę, że prawdopodobieństwo mutacji jednego osobnika jest więc większa n razy, gdzie n to długość genomu osobnika.
- crossesPerGeneration - ilość krzyżowań w trakcie jednej iteracji

- verbose - wartość równa "0"wyłącza dodatkowe informacje nt. przebiegu algorytmu, tj. wyświetla jedynie wynik. Wartość "1"pozwala na śledzenie postępów krok po kroku poprzez podanie ich na strumieniu wyjściowym.

Listing 1: Implementacja algorytmu genetycznego

```
1  Path TS_Genetic::TS_UseGenetic(AdjacencyMatrix* matrix, int maxIterations,
       int populationSize, SelectionType selectionType, CrossingStrategy
       crossingStrategy, float mutationProbability, int crossesPerGeneration,
       int verbose)
2  {
3      TS_Genetic instance = TS_Genetic(matrix, populationSize);
4      Path* bestPath = new Path(matrix);
5      int costOfBestPath = INT_MAX;
6
7      instance.generateRandomPopulation();
8
9      for (int i = 0; i < maxIterations; i++) {
10         // wygeneruj losowe mutacje
11         instance.randomlySwapMutatePopulation(mutationProbability);
12
13         // utworz nowe osobniki
14         instance.newGeneration_crossBestRandomly(populationSize,
               crossesPerGeneration, crossingStrategy);
15
16         // wybierz osobniki do nastepej iteracji
17         switch (selectionType) {
18         case SelectionType::rank:
19             instance.makeSelection_rank();
20             break;
21         default:
22             if (verbose) {
23                 std::cout << "Nie wybrano metody selekcji" << std::endl;
24             }
25         }
26
27         // wyswietl postep
28         if (verbose) {
29             instance.sortPopulationByPathCost();
30             std::cout << "Pokolenie "
31                 << i << ". Wynik: ";
32         }
33
34         if (verbose) {
35             for (int q = 0; q < min(10, populationSize); q++) {
36                 std::cout << instance.population[q].CalculateCost() << "...
                       ";
37             }
38             std::cout << std::endl;
39         }
40
```

```
41            int proposedBest = instance.population[0].CalculateCost();
42
43            if (proposedBest < costOfBestPath) {
44                costOfBestPath = proposedBest;
45                bestPath = new Path(matrix);
46                bestPath->ReplaceWithOtherInstance(instance.population[0]);
47            }
48        }
49
50        return *bestPath;
51    }
```

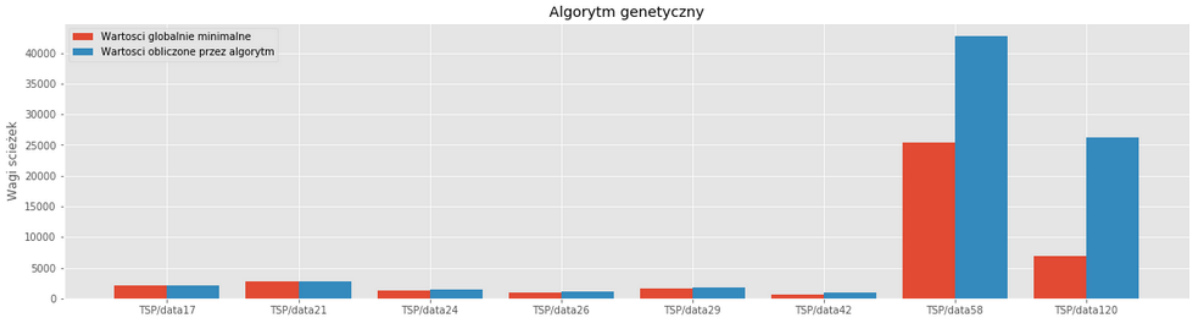
3.3 Analiza działania

Analizie została poddana dokładność wyników algorytmu w porównaniu do rozwiązań optymalnych, a także wpływ wartości poszczególnych parametrów na wartość rozwiązania.

3.3.1 Wynik działania algorytmu a wartości optymalne

Wykonano 100 iteracji na populacji o rozmiarze 100 osobników. Prawdopodobieństwo mutacji wynosiło 0.01. Jedno pokolenie generowało 50 nowych osobników.

Zastosowana konfiguracja okazała się mało skuteczna dla dużych grafów generując błąd rzędu nawet kilkuset procent. Biorąc pod uwagę także czas wykonania obliczeń, algorytm genetyczny w obecnej konfiguracji znacząco ustępuje algorytmowi mrówkowemu, opisanemu w dalszej części sprawozdania.




```
avgErr 52.820995257725784
```

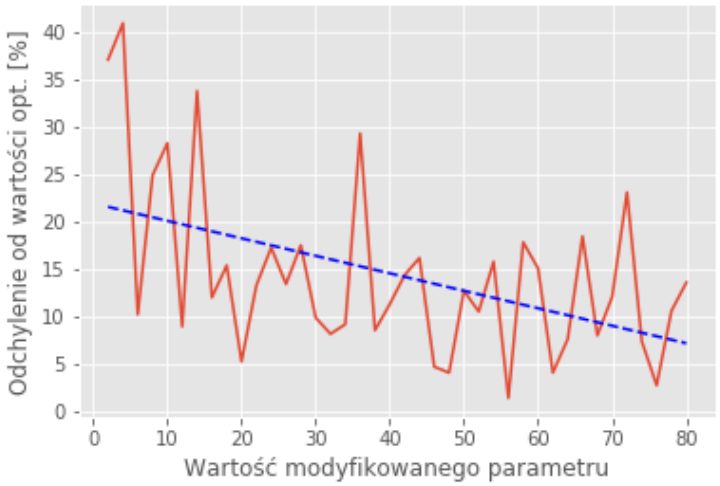
Rysunek 1: Wynik a wartości optymalne - wykres

	baseFile	actual_results	expected_results	error
0	TSP/data17	2085.0	2085	0.000000
1	TSP/data21	2707.0	2707	0.000000
2	TSP/data24	1469.0	1272	15.487421
3	TSP/data26	1071.0	937	14.300961
4	TSP/data29	1731.0	1610	7.515528
5	TSP/data42	966.0	699	38.197425
6	TSP/data58	42758.0	25395	68.371727
7	TSP/data120	26289.0	6942	278.694901

Rysunek 2: Wynik a wartości optymalne - tabela

3.3.2 Wynik działania w zależności od wielkości populacji

Zwiększenie populacji nieznacznie poprawia jakość wyniku. Wiąże się to między innymi z większą pulą losowych rozwiązań na starcie, a więc przeszukaniem większego fragmentu zbioru rozwiązań już przed pierwszą iteracją. Krzyżowanie odbywa się ponadto w większej puli osobników, co zmniejsza ryzyko przedwczesnego wpadnięcie w ekstremum lokalne.



Rysunek 3: Wynik a wielkość populacji

3.3.3 Wynik działania w zależności od prawdopodobieństwa mutacji

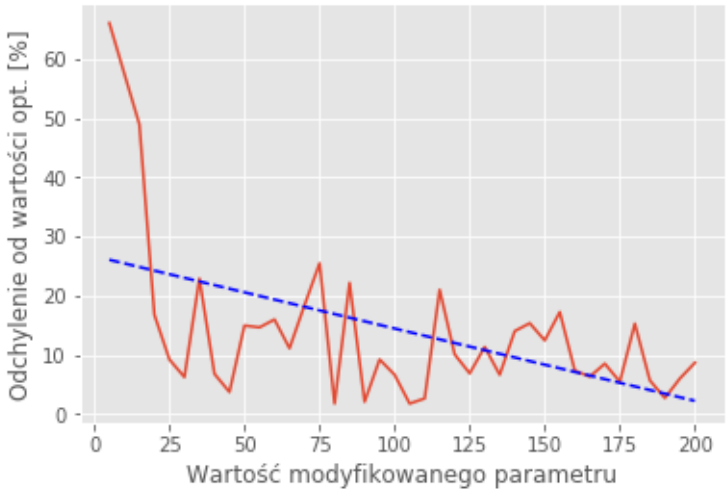
Na wykresie na rysunku 4 została ukazana zależność błędu pomiaru od wartości parametru mutacji. Analiza wyników pomiarów nasuwa wniosek, że prawdopodobieństwo mutacji należy zachować na niskim poziomie. Jej duże wartości mocno zaburzają ewolucję osobników niszcząc genom obiecujących jednostek. Warto zauważyć, że nie dotyczy to wartości parametru bliskich "1.0". Wynika to z konstrukcji algorytmu: jako że mutacja odbywa się na ruchach typu swap, zastosowanie tego ruchu dla wszystkich genów powoduje przeniesienie genu pierwszego na koniec ścieżki. Jako że ścieżka jest cyklem, nie wpływa to na koszt jej przejścia. Innymi słowy, mutacja z

parametrem ”1.0” nie narusza struktury ścieżki, a jedynie przenosi jej elementy na liście o jedno miejsce.

Rysunek 4: Wynik a prawdopodobieństwo mutacji

3.3.4 Wynik działania w zależności od ilości krzyżowań na iterację

Badanie wykazuje nieznaczne polepszenie osiągów algorytmu wraz ze wzrostem liczby krzyżowań. Należy jednak odpowiednio dostosować ilość krzyżowań do wielkości populacji, gdyż jej duża wartość przy niewielkiej liczbie osobników szybko zapełni populację podobnymi sobie genomami zawężając obszar poszukiwań.



Rysunek 5: Wynik a ilość krzyżowań

4 Algorytm mrówkowy

4.1 Sposób działania

Algorytm wzorowany jest na sposobie znajdowania optymalnej drogi przez populację mrówek. Rozwiązanie problemu opiera się na tzw. inteligencji rozproszonej, czyli wykorzystaniu dużej liczby niezależnych, niesterowanych odgórnie jednostek w celu znalezienia rozwiązania. Kluczowym pojęciem jest tutaj *feromon* czyli związek chemiczny pozostawiany przez mrówki w celu oznaczenia przebytej drogi. Duża ilość feromonu na drodze oznacza że pokonuje ją duża ilość zostawiających go mrówek, co z kolei oznacza potencjalną trasę do źródła pożywienia.

Każda z mrówek reprezentuje jedno potencjalne rozwiązanie problemu (jedną ścieżkę). W każdej iteracji umieszczana jest ona w losowym węźle, a następnie krok po kroku odwiedza wszystkie węzły grafu, tak aby spełnić założenia problemu TSP (utworzenie cyklu, każdy węzeł występuje raz).

Mrówka znajdując się w węźle X dokonuje wyboru kolejnego węzła (spośród jeszcze nieodwiedzonych) z prawdopodobieństwem określonym poniższym wzorem:

$$p_{ij} = \begin{cases} \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\Sigma_{c_{i,l} \in \Omega}(\tau_{ic})^{\alpha}(\tau_{ic})^{\beta}}, & \bigvee c_{i,l} \in \Omega \\ 0, & \bigvee c_{i,l} \notin \Omega \end{cases} \tag{1}$$

Gdzie Ω reprezentuje zbiór możliwych ruchów z obecnego węzła wyłączając węzły odwiedzone, τ_{ij} to wskaźnik feromonu na drodze z *i* do *j*, α to parametr określający wpływ feromonu na decyzję mrówki, η_{ij} to wartość lokalnej funkcji kryterium (np. odwrotność kosztu przejścia drogi z *i* do *j*), β to parametr regulujący wpływ tej funkcji.

Zwiększając parametr β zwiększamy prawdopodobieństwo wyboru przez mrówkę opcji zachłannej (lokalnie najmniej kosztowny ruch z perspektywy węzła w którym w danej chwili się znajduje). Zwiększając parametr α zwiększamy prawdopodobieństwo wyboru podyktowanego zostawionym przez inne mrówki feromonem.

W momencie startu algorytmu, kiedy wartości feromonu są wszędzie takie same, mrówki dokonują jedynie wyborów zachłannych. Z biegiem czasu przyrost feromonu ogniskuje ruch populacji na zawężonej grupie ścieżek.

4.2 Sposób implementacji

W listingu 1 umieszczono główną funkcję algorytmu, której wywołanie rozpoczyna proces obliczeń. Przyjmuje ona jako argumenty parametry pozwalające dostosować zachowanie algorytmu.

- matrix - przechowuje informacje nt. grafu dla którego rozwiązywany jest problem TSP
- maxNumOfIterations - maksymalna liczba iteracji jaką może wykonać algorytm
- numberOfAnts - liczba mrówek równolegle przechodzących graf w każdej iteracji
- collectiveMemoryPower - parametr określający wpływ feromonu na decyzję pojedynczego osobnika. Im większy, tym większe prawdopodobieństwo, że mrówka podąży ścieżką często uczęszczaną.
- greedyChoicePower - parametr określający wpływ wyboru zachłannego na decyzję osobnika. Im większy, tym większe prawdopodobieństwo, że mrówka podąży ścieżką o lokalnie najmniejszym koszcie.
- evaporationRate - szybkość parowania feromonu
- pheromoneStartValue - wartość początkowa feromonu na ścieżkach grafu

- verbose - wartość równa "0"wyłącza dodatkowe informacje nt. przebiegu algorytmu, tj. wyświetla jedynie wynik. Wartość "1"pozwala na śledzenie postępów krok po kroku poprzez podanie ich na strumieniu wyjściowym.

Listing 2: Implementacja algorytmu genetycznego

```
1  Path TS_Ant_Colony::TS_UseAntColony(AdjacencyMatrix* matrix, int
       maxNumOfIterations, int numberOfAnts, float collectiveMemoryPower, float
        greedyChoicePower, float evaporationRate, float pheromoneStartValue,
       bool verbose)
2  {
3      TS_Ant_Colony environment = TS_Ant_Colony(matrix, numberOfAnts,
           collectiveMemoryPower, greedyChoicePower, evaporationRate,
           pheromoneStartValue);
4      Path* bestPath = new Path(matrix);
5      int costOfBestPath = INT_MAX;
6
7      for (int i = 0; i < maxNumOfIterations; i++) {
8          environment.SeedAnts(numberOfAnts);
9
10         for (int step = 0; step < matrix->GetSize(); step++) {
11             for (int ant = 0; ant < numberOfAnts; ant++) {
12                 environment.MoveAntLeavePheromone(&environment.ants[ant]);
13             }
14         }
15
16         environment.EvaporatePheromone(evaporationRate);
17
18         int currentIterationBest = INT_MAX;
19
20         int prevAntValue = -1;
21         bool valuesAreTheSame = true;
22         for (int ant = 0; ant < numberOfAnts; ant++) {
23             int proposedBest = environment.ants[ant].CalculateCost();
24
25             if (prevAntValue != proposedBest && prevAntValue != -1) {
26                 valuesAreTheSame = false;
27             }
28             prevAntValue = proposedBest;
29
30             if (currentIterationBest > proposedBest) {
31                 currentIterationBest = proposedBest;
32             }
33
34             if (costOfBestPath > proposedBest) {
35                 if (environment.ants[ant].Validate()) {
36                     costOfBestPath = proposedBest;
37                     bestPath = new Path(matrix);
38                     bestPath->ReplaceWithOtherInstance(environment.ants[ant
                           ]);
39                 }
```

```
40                      else {
41                          std::cout << "Found best but is invalid" << std::endl;
42                      }
43                  }
44              }
45
46              // usun populacje
47              environment.ants = vector<Path>();
48
49              if (verbose) {
50                  std::cout << "Iter. " << i << ": global best: " <<
                        costOfBestPath << " current iter: " << currentIterationBest
                        << std::endl;
51              }
52
53              if (valuesAreTheSame) {
54                  // wszystkie mrowki znalazly te sama droge, mozna przerwac
                        algorytm
55                  break;
56              }
57          }
58
59          return *bestPath;
60      }
```

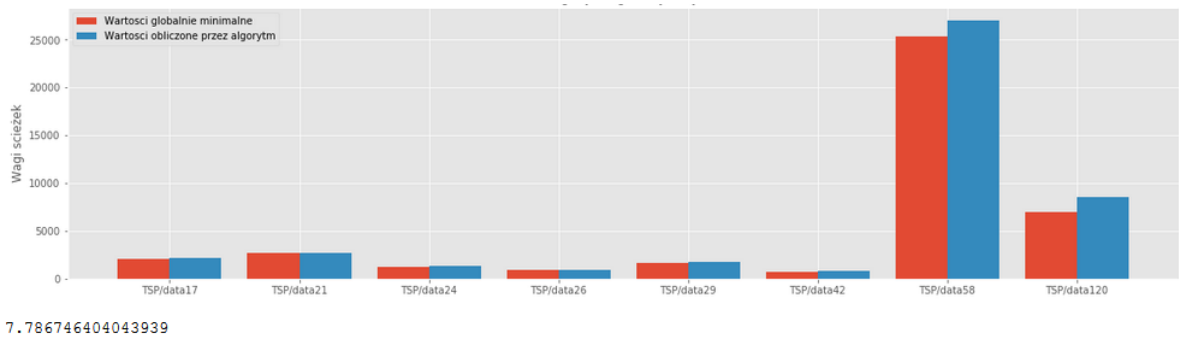
Wartość zwracana przez funkcję to najlepszy wynik uzyskany przez któregokolwiek osobnika z populacji mrówek przez cały okres działania algorytmu. Algorytm może zakończyć się szybciej niż po wykonaniu wszystkich założonych w parametrze maxNumOfIterations, jeżeli wystąpi sytuacja w której wszystkie mrówki uzyskają ten sam wynik.

4.3 Analiza działania

Poniżej zamieszczono parametry użyte podczas testowania na przykładowych zestawach danych:

- ilość iteracji: 500
- liczba mrówek: 10
- siła feromonu: 1
- siła wyboru zachłannego: 3
- szybkość parowania feromonu: 0.5
- wartość początkowa feromonu: 1

Błąd w wynikach algorytmu jest najmniejszy spośród wszystkich testowanych heurystyk. Jest on w stanie najmniejszym nakładem czasowym uzyskać najbliższy rzeczywistemu wynik. Ponadto, już kilka pierwszych iteracji wystarcza aby uzyskać dobre przybliżenie wyniku, nawet w przypadku dużych grafów.




```
7.786746404043939
```

Rysunek 6: Wynik a wartości optymalne - wykres

	baseFile	actual_results	expected_results	error
0	TSP/data17	2158.0	2085	3.501199
1	TSP/data21	2707.0	2707	0.000000
2	TSP/data24	1328.0	1272	4.402516
3	TSP/data26	955.0	937	1.921025
4	TSP/data29	1764.0	1610	9.565217
5	TSP/data42	793.0	699	13.447783
6	TSP/data58	27026.0	25395	6.422524
7	TSP/data120	8541.0	6942	23.033708

Rysunek 7: Wynik a wartości optymalne - tabela

5 Podsumowanie

W ramach projektu zademonstrowano działanie niektórych heurystyk w celu przybliżenia rozwiązania problemu komiwojażera. Każdy z nich został zaimplementowany w podstawowym stopniu i istnieje duże pole do optymalizacji przybliżenia wyniku oraz czasu wykonywania. Najmniejszy błąd w stosunku do wartości optymalnych okazał się generować algorytm mrówkowy.