Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych

Współczesne techniki heurystyczne

SK6. Algorytm przeszukiwania z tabu dla problemu komiwojażera

Sprawozdanie 3

Autorzy:
Maciej Suchecki
Jacek Witkowski

Prowadzący:
dr inż. Sebastian Kozłowski

26 stycznia 2015

1 Część 1

1.1 Treść zadania

Tytuł SK6. Algorytm przeszukiwania z tabu dla problemu komiwojażera

Opis Należy zaimplementować algorytm przeszukiwania z tabu (wzbogacony w stosunku do wersji „klasycznej" o elementy wskazane przez prowadzącego), a następnie zastosować ten algorytm do rozwiązania problemu komiwojażera.

1.2 Implementacja

Język programowania Wybranym językiem programowania jest Java.

Założenia Zakładamy, że badane grafy są niekierowane.

Reprezentacja rozwiązania Rozwiązanie problemu będzie reprezentowane w postaci listy odwiedzanych miast (wierzchołków grafu).

Definicja sąsiedztwa Generowanie sąsiadów zostanie zrealizowane na wzór algorytmu *2-opt* [Cro58]. Rozwiązaniem B sąsiednim do rozwiązania A nazywamy rozwiązanie powstałe po odwróceniu kolejności dowolnej części listy odwiedzonych miast.

1.3 Element heurystyczny

Algorytm przeszukiwania z tabu [diSK14] jest nieefektywny w przypadkach, gdy ewaluacja sąsiednich rozwiązań jest czasochłonna. W takich przypadkach algorytm można usprawnić za pomocą metod heurystycznych. W tym wypadku chcemy zastosować metodę o nazwie „aspiration plus strategy" [GL98].

Aspiration plus strategy Strategia ta polega na heurystycznym wyborze sąsiada do dalszej ewaluacji. Do implementacji wybraliśmy strategię w wersji ze zmiennymi Min oraz Max. Zakłada ona ustalenie przed wykonaniem algorytmu trzech zmiennych: Plus, Min oraz Max. Oprócz tego należy ustalić próg aspiracji (threshold), który będzie minimum, przy którym algorytm powinien zaakceptować znalezione rozwiązanie. W naszym wypadku będzie on generowany dynamicznie w trakcie działania programu, na podstawie ewaluacji aktualnego najlepszego rozwiązania. Po ustaleniu ww. parametrów algorytm przeszukuje po kolei sąsiedztwo ostatniego rozwiązania, aż do znalezienia nowego rozwiązania z jakością powyżej ustalonego progu. Następnie sprawdzane jest Plus kolejnych rozwiązań. Dodatkowo przyjętym usprawnieniem jest założenie, że liczba sprawdzonych rozwiązań nie będzie mniejsza od parametru Min oraz większa od parametru Max. Ze wszystkich sprawdzonych rozwiązań wybierane jest następnie rozwiązanie najlepsze.

2 Część 2

2.1 Główny algorytm programu

2.1.1 Pseudokod algorytmu przeszukiwania z Tabu

```
// glowny algorytm
findSolution(graph) {
  aspiration = generateAspirationLevel(graph);
  currentSolution = generateInitialSolution(graph);
  previousSolution = currentSolution;
  bestSolution = currentSolution;
  iterationsWithoutImprovement = 0;

  while (iterationsWithoutImprovement < 5) {
    neighbourhood = generateNeighbourhood(currentSolution);
    currentSolution = getBestNeighbour(neighbourhood, aspiration, bestSolution);

    addToTabuList(move(currentSolution));

    iterationsWithoutImprovement++;
    if (distance(currentSolution) < distance(previousSolution)) {
      iterationsWithoutImprovement = 0;
    }
    if (distance(currentSolution) < distance(bestSolution)) {
      bestSolution = currentSolution;
    }

    aspiration = adjustAspiration(aspiration, distance(bestSolution));
  }

  return bestSolution;
}
```

Listing 1: Pseudokod algorytmu przeszukiwania z Tabu

2.1.2 Opis

Algorytm na początku inicjalizuje zmienne początkowe (na podstawie badanego grafu, przekazanego w zmiennej *graph*). Następnie w pętli generuje sąsiedztwo poprzednio wygenerowanego rozwiązania i za pomocą metody *getBestNeighbour* wyszukuje najlepsze rozwiązanie z sąsiedztwa. Jeśli znalezione rozwiązanie jest lepsze od dotychczas najlepszego, zostaje zapisane. Ponadto – ruch, który wygenerował z poprzednio sprawdzanego rozwiązania aktualne najlepsze, zostaje zapisany na liście Tabu. Wszystkie opisane kroki są następnie powtarzane, dopóki w pięciu kolejnych iteracjach nie uda nam się uzyskać poprawy rozwiązania. Dodatkowo, w każdym kroku algorytmu poziom aspiracji jest korygowany.

Opis zmiennych

- *aspiration* – poziom aspiracji.
- *currentSolution, previousSolution* – aktualnie oraz poprzednio sprawdzane rozwiązanie.
- *bestSolution* – dotychczas najlepsze rozwiązanie.
- *iterationsWithoutImprovement* – liczba iteracji bez poprawy.
- *neighbourhood* – sąsiedztwo rozwiązania.

Opis funkcji

- *generateAspirationLevel* – generuje początkowy poziom aspiracji – patrz sekcja 2.4.
- *generateInitialSolution* – generuje początkowe rozwiązanie (listę wierzchołków grafu).
- *generateNeighbourhood* – generuje sąsiedztwo danego rozwiązania – patrz sekcja 2.3.
- *getBestNeigbour* – zwraca najlepszego sąsiada danego rozwiązania – patrz sekcja 2.2.
- *adjustAspiration* – zwraca nową wartość aspiracji – patrz sekcja 2.4.
- *distance* – zwraca długość ścieżki dla danego rozwiązania.
- *move* – zwraca ruch, który został wykonany do wygenerowania danego rozwiązania.
- *addToTabuList* – dodaje ruch do listy Tabu – patrz sekcja 2.5.

2.2 Algorytm wyszukiwania najlepszego rozwiązania w sąsiedztwie

2.2.1 Pseudokod algorytmu wyszukiwania najlepszego rozwiązania w sąsiedztwie

```
// wyszukiwanie najlepszego rozwiazania za pomoca Aspiration Plus Strategy
getBestNeighbour(neighbourhood, aspiration, bestSolution) {
  bestNeighbour = neighbourhood[0];
  neighboursChecked = 0;
  neighboursSinceAspirationSatisfied = 0;
  aspirationThresholdReached = false;

  for (every neighbour in neighbourhood) {
    if (moveShouldBeDone(neighbour, bestSolution, bestNeighbour)) {
      bestNeighbour = neighbour;
    }

    if(!aspirationThresholdReached) {
      aspirationThresholdReached = (distance(neighbour) < aspiration);
    }
    if (aspirationThresholdReached) {
      ++neighboursSinceAspirationSatisfied;
    }

    ++neighboursChecked;
    if (enoughSolutionsChecked()) break;
  }

  return bestNeighbour;
}

// sprawdzanie, czy powinno sie wykonac ruch
moveShouldBeDone(neighbour, bestSolution, bestNeighbour) {
  if (distance(neighbour) > distance(bestNeighbour) {
    return false;
  } else if (isTabu(move(neighbour))) {
    return distance(neighbour) < distance(bestSolution);
  }
  return true;
}
```

Listing 2: Pseudokod algorytmu wyszukiwania najlepszego rozwiązania w sąsiedztwie

2.2.2 Opis

Algorytm wyszukuje najlepsze rozwiązanie z danego sąsiedztwa wykorzystując Aspiration Plus Strategy. Na początku odbywa się inicjalizacja zmiennych początkowych, opisanych poniżej. Następnie w pętli odbywa się sprawdzanie kolejnych rozwiązań, dopóki nie zostanie spełnione jedno z poniższych warunków:

1. sprawdzono całe sąsiedztwo
2. sprawdzono przynajmniej tyle rozwiązań, ile wynosi parametr *max*
3. ogólnie sprawdzono przynajmniej *min* rozwiązań oraz od czasu spełnienia aspiracji sprawdzono przynajmniej *plus* parametrów

Powyższe warunki są sprawdzane przez funkcję *enoughSolutionsChecked()*.

Ponadto, sprawdzany sąsiad jest przypisany do zmiennej *bestNeighbour* tylko i wyłącznie wtedy, gdy jest lepszy od poprzedniego *bestNeighbour* oraz spełnia warunek aspiracji lub nie znajduje się na liście Tabu.

Na końcu działania algorytmu zmienna *bestNeighbour* jest zwracana jako reprezentująca najlepsze rozwiązanie.

Opis zmiennych

- *neighbourhood* – sąsiedztwo rozwiązania
- *aspiration* – poziom aspiracji
- *bestNeighbour* – dotychczas najlepszy sąsiad
- *neighbour* – aktualnie sprawdzany sąsiad
- *neighboursChecked* – liczba sprawdzonych jak dotąd sąsiadów
- *neighboursSinceAspirationSatisfied* – liczba sprawdzonych sąsiadów od kiedy poziom aspiracji jest spełniony
- *aspirationThresholdReached* – zmienna mówiąca, czy próg aspiracji został już osiągnięty.

Opis funkcji

- *distance* – zwraca długość ścieżki dla danego rozwiązania.
- *move* – zwraca ruch, który został wykonany do wygenerowania danego rozwiązania.
- *isTabu* – zwraca *true*, jeśli ruch należy do listy Tabu – patrz sekcja 2.5.
- *moveShouldBeDone* – sprawdza, czy powinno się wykonać ruch do podanego sąsiada.

2.3 Generowanie sąsiedztwa

2.3.1 Opis

Generowanie sąsiedztwa odbywa się za pomocą metody 2-opt. Metoda ta zakłada generowanie sąsiadów rozwiązania za pomocą odwrócenia kolejności miast w dowolnej części ścieżki. Za pomocą dwóch indeksów oznacza się początek i koniec odwracanego ciągu. Po wykonaniu takiej zamiany dla wszystkich możliwych indeksów i oraz k, otrzymujemy całe sąsiedztwo rozwiązania.

2.3.2 Przykład

W przykładzie wygenerujemy sąsiada ścieżki A → B → C → D → E → F → G → H → A, za pomocą indeksów: i = 4, k = 7. Nowa ścieżka zostanie wygenerowana w trzech krokach:

1. (A → B → C) – dodanie początkowego fragmentu w kolejności
2. A → B → C → (G → F → E → D) – dodanie środkowego fragmentu w odwrotnej kolejności
3. A → B → C → G → F → E → D (→ H → A) – dodanie końcowego fragmentu w kolejności

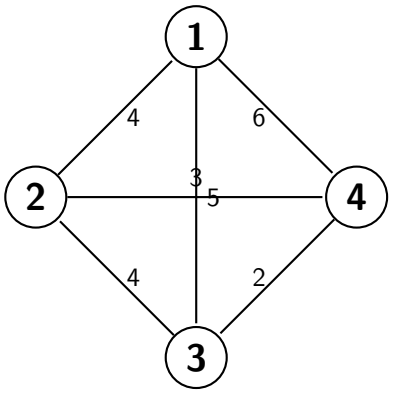
2.4 Ustalanie progu aspiracji

2.4.1 Opis

Próg aspiracji jest na początku ustalany na podstawie właściwości grafu wejściowego, a następnie dynamicznie korygowany. Wartość początkowa ustawiana jest na średnią wagę krawędzi pomnożoną przez liczbę wierzchołków grafu (patrz przykład poniżej).
Dodatkowo, w trakcie działania programu próg jest weryfikowany – jeśli zostanie znaleziona droga, która ma długość mniejszą niż aktualny próg aspiracji, próg zostaje zaktualizowany. Jego nowa wartość wynosi średnią arytmetyczną ze starej wartości progu aspiracji oraz najkrótszej jak dotąd znalezionej ścieżki.

2.4.2 Przykład

Dla pełnego grafu zdefiniowanego poniżej:



Uzyskamy następującą początkową wartość poziomu aspiracji:
$a = v * \frac{\sum w_e}{e} = 4 * \frac{4+6+3+5+4+2}{6} = 4 * \frac{24}{6} = 16$

2.5 Obsługa listy Tabu

2.5.1 Opis

TabuList jest kontenerem przechowującym listę ruchów zabronionych do wykonania w poszukiwaniu lepszych rozwiązań. Ruch określony jest poprzez dwa punkty podziału rozwiązania (generowanego metodą 2-opt).

Kontener zapewnia szybkie sprawdzanie (w czasie stałym), czy dany ruch jest zabroniony (tj. znajduje się w kontenerze), jak również szybkie dodawanie nowego ruchu (również w czasie stałym). Dodatkowo kontener „pamięta" kolejność dodawania ruchów do listy tabu. Dzięki temu w przypadku przekroczenia maksymalnej liczby ruchów przechowywanych w kontenerze, możliwe jest usunięcie najdawniej dodanego ruchu do kontenera (operacja wykonywana automatycznie).

By zapewnić szybkie sprawdzanie czy ruch znajduje się w kontenerze wykorzystano kontener *HashMap*. By pamiętać kolejność dodawanych ruchów wykorzystano listę typu *LinkedList*.

3 Część 3

3.1 Prezentacja wyników

3.1.1 Definicje

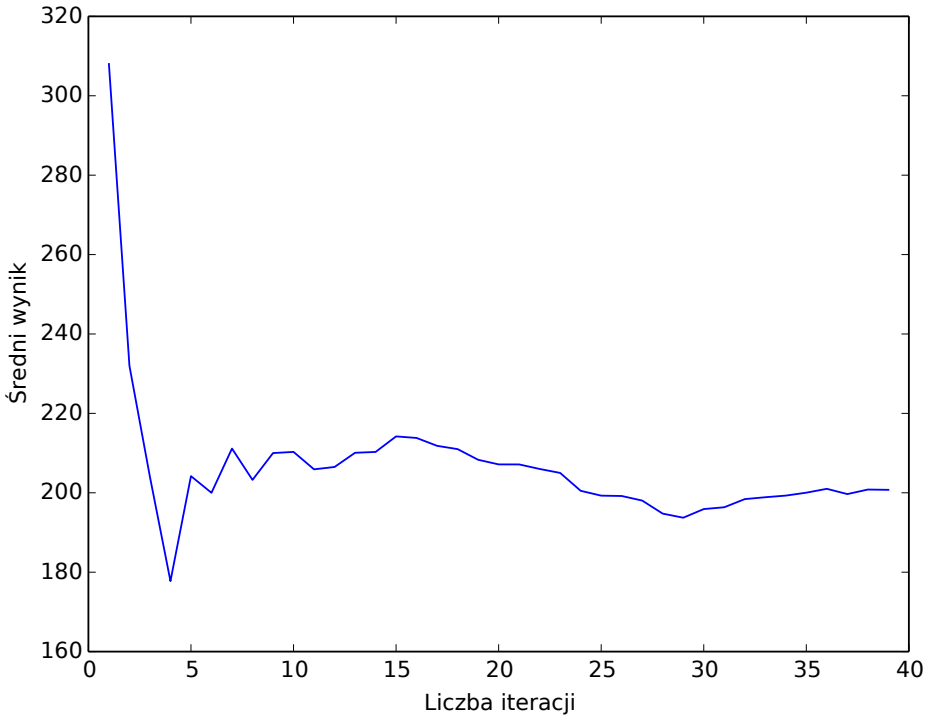
W kontekście testowania projektu definiujemy dwa parametry określające wydajność algorytmu:

- **wynik** – długość najkrótszej ścieżki znalezionej przez program (jakość rozwiązania),
- **czas działania programu** – czas, jaki upłynął pomiędzy wykonaniem programu (dla zadanych parametrów oraz grafu wejściowego), a jego zakończeniem i zwróceniem wyniku.

3.1.2 Uśrednianie wyników

Ze względu na fakt, iż zaimplementowany algorytm jest algorytmem heurystycznym, jakość otrzymywanych wyników zależy od właściwości grafu podanego na wejściu. Z tego powodu – dla zwiększenia czytelności wyników – każde uruchomienie programu jest powtarzane kilkukrotnie dla różnych grafów. Wynik oraz czas działania programu jest przyjmowany jako średnia z odpowiednich wartości w kolejnych uruchomieniach.

Liczba powtórzeń była wyznaczona doświadczalnie, za pomocą eksperymentu przedstawionego na wykresie poniżej:



Rysunek 1: Zależność średniego wyniku działania programu od liczby powtórzeń

W celu wyznaczenia liczby powtórzeń, wygenerowaliśmy 50 losowych grafów pełnych. Następnie uruchamialiśmy solver dla coraz większej liczby grafów, uśredniając przy tym otrzymany wynik. Na otrzymanym wykresie widać, że wartość 10 jest minimalną liczbą powtórzeń, dla których można już potwierdzić wiarygodność otrzymywanych wyników.

Kod generujący powyższy wykres jest przedstawiony poniżej:

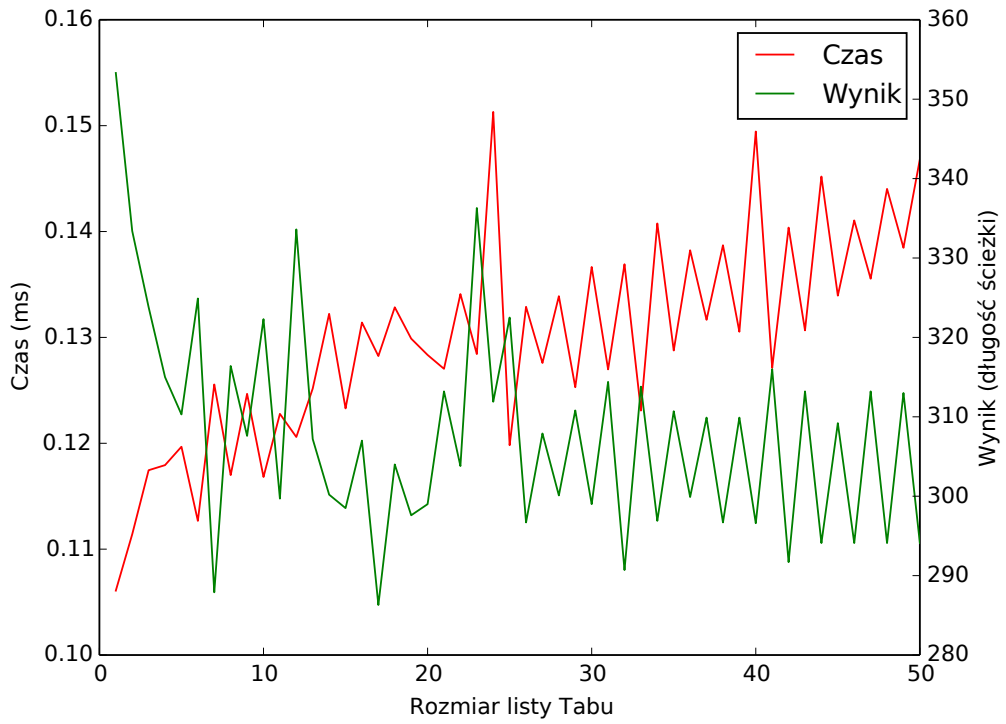
```
# parameters
maxIterations = 50
graphSize =  10

# generate graphs
filenames = graph.saveRandomGraphsToFiles(maxIterations, graphSize)

# collect data
averageResults = {}
for iterations in range(1, maxIterations):
  averageResult = 0
  for i in range(iterations):
    averageResult += runSolver(filenames[i])
  averageResults[iterations] = averageResult / iterations

plotGraph(averageResults, "Liczba iteracji", "Sredni wynik")
```

Listing 3: Fragment kodu programu w języku Python generującego powyższy wykres

3.1.3 Badanie wpływu parametrów na wydajność algorytmu

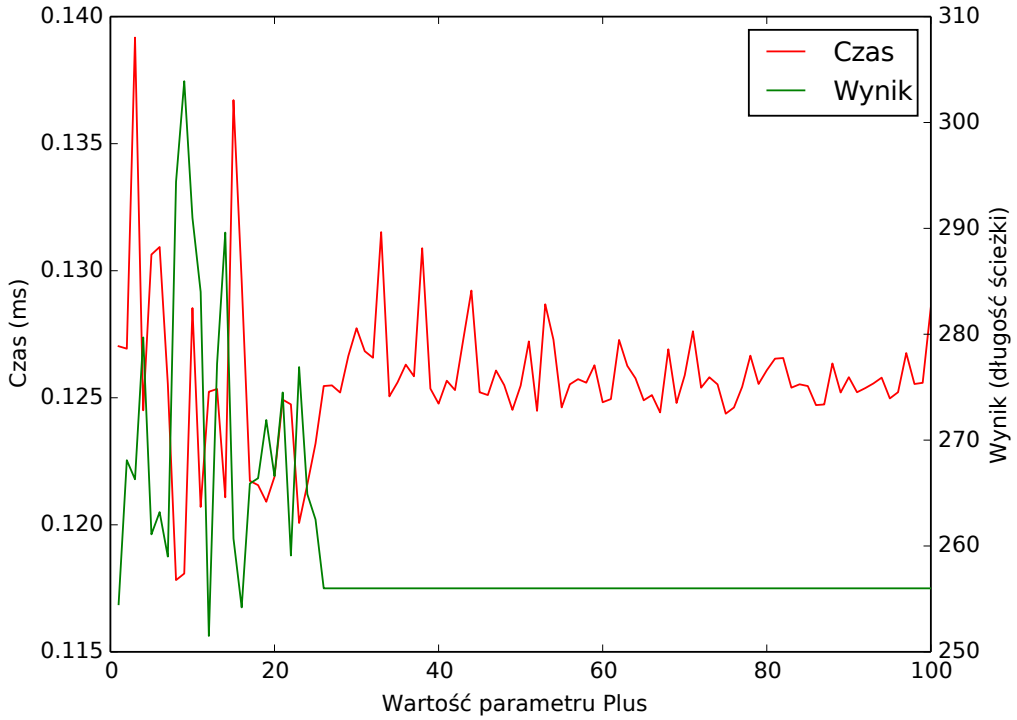
Po wyznaczeniu wymaganej liczby powtórzeń został utworzony skrypt testujący. Generuje on na początku 10 losowych grafów. Następnie uruchamia solver dla poszczególnych kombinacji parametrów. Uzyskane wyniki wraz z czasami wykonania programu są zapisywane w celu wykonania na ich podstawie wykresów, które zaprezentowane są poniżej.

3.1.4 Wpływ rozmiaru listy Tabu na wydajność algorytmu



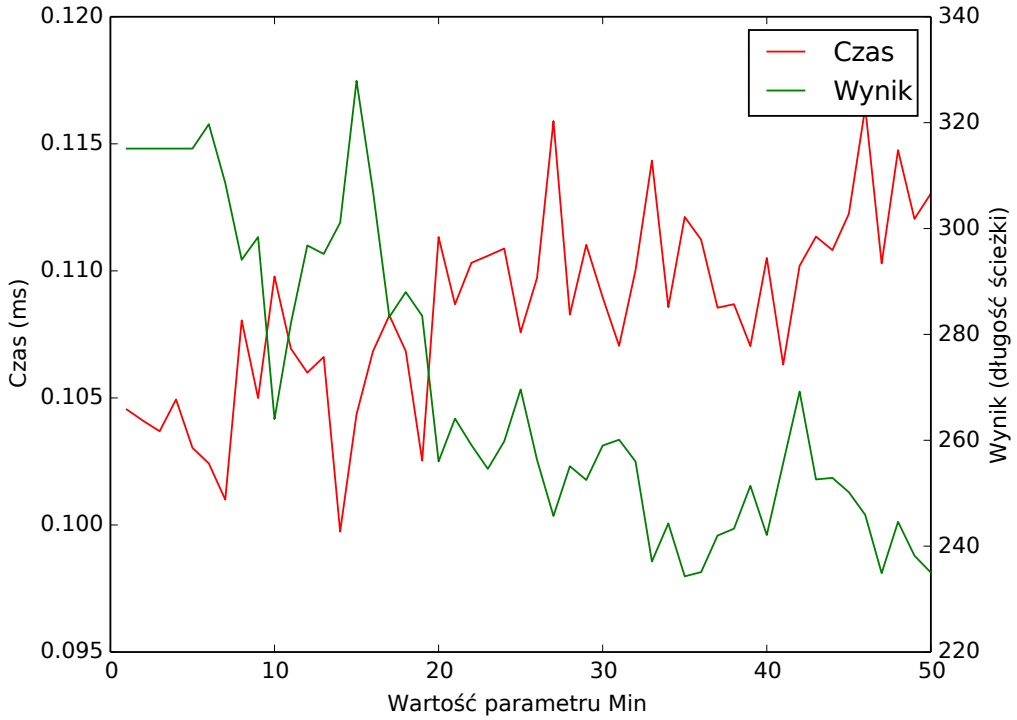
Rysunek 2: Wpływ rozmiaru listy Tabu na wydajność algorytmu.

3.1.5 Wpływ parametru Plus na wydajność algorytmu



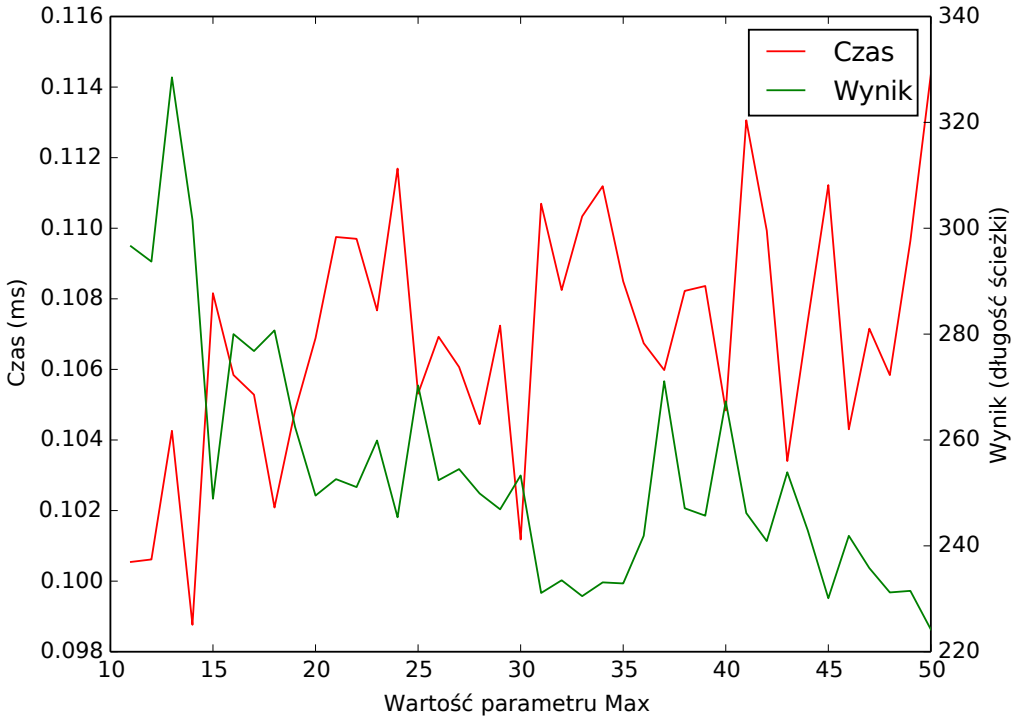
Rysunek 3: Wpływ parametru Plus na wydajność algorytmu.

3.1.6 Wpływ parametru Min na wydajność algorytmu



Rysunek 4: Wpływ parametru Min na wydajność algorytmu.

3.1.7 Wpływ parametru Max na wydajność algorytmu



Rysunek 5: Wpływ parametru Max na wydajność algorytmu.

3.1.8 Wnioski

Uzyskane wyniki są zgodne z naszymi oczekiwaniami. Po pierwsze, dla wszystkich parametrów widać wyraźny wpływ na czas działania algorytmu – zwiększanie wartości każdego z parametrów powoduje zasadniczo zwiększenie czasu wykonywania algorytmu. Jest to zgodne z intuicją, ponieważ zwiększenie każdego z parametrów powoduje zwiększenie liczby rozwiązań koniecznych do sprawdzenia, a co za tym idzie – liczby operacji do wykonania.

Z tego samego powodu na wykresach można wyraźnie zaobserwować polepszanie się znalezionego rozwiązania wraz ze zwiększaniem parametrów. Ciekawe są zwłaszcza pierwsze dwa wykresy. W przypadku listy Tabu, można zaobserwować oscylacje na końcu wykresu. Odnośnie parametru Plus, widać ustabilizowanie się wyniku dla wartości powyżej 25 – zwiększanie parametru nie daje poprawy ze względu na ograniczony rozmiar sąsiedztwa rozwiązania.

Na podstawie dokładniejszej analizy wykresów można wyznaczyć optymalne wartości parametrów dla tak zaimplementowanego algorytmu. Oczywiście zależą one od konkretnego grafu (w szczególności oczywiście od liczby wierzchołków) oraz naszych priorytetów (szybkość/jakość) aczkolwiek w ogólności prezentują się one według nas następująco:

- rozmiar listy Tabu: optymalna wartość zawiera się w przedziale (15, 25)
- parametr Plus: optymalna wartość zawiera się w przedziale (25, 35)
- parametr Min: optymalna wartość zawiera się w przedziale (20, 50)
- parametr Max: optymalna wartość zawiera się w przedziale (20, 50)

Literatura

[Cro58] G. A. Croes. A method for solving traveling-salesman problems. pages 791–812, 1958.

[diSK14] dr inż. Sebastian Kozłowski. Materiały do wykładu - contemporary heuristic techniques w wersji angielskiej, 2014.

[GL98] Fred W. Glover and Manuel Laguna. *Tabu Search*, volume 1. Springer, 1998.