Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *PN 17:05*
Temat *Algorytmy lokalnego przeszukiwania*	**Problem** *TSP*
Skład grupy *241311 Dominik Maroszczyk*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *15 stycznia 2020*

1 Opis problemu

Rozwiązując problem komiwojażera (ang Travelling Salesman Problem - TSP) w poprzednim etapie zaimplementowaliśmy algorytmy dokładne rozwiązujące problem. W kolejnym etapie omawianymi oraz implementowanymi algorytmami są algorytmy przeszukiwania lokalnego. Jak i poprzednio rozwiązujemy problem komiwojażera, czyli znalezienie najkrótszej ścieżki w grafie pełnym. Algorytmy przeszukiwania lokalnego bazują na zbiorze wszystkich możliwych rozwiązań, aby znaleźć optymalne wykonuję się sekwencje ruchów. Są one określone i wykonywane według przyjętych zasad. Algorytmy te ogranicza liczba iteracji oraz parametry algorytmów, dzięki odpowiedniemu dobraniu parametrów rozwiązanie może być dokładniejsze. Ponieważ liczba iteracji jest jednym z ograniczeń, to my decydujemy jak długo ma się wykonywać algorytm i jak dokładny ma być.

2 Metoda rozwiązania

2.1 Algorytm przeszukiwań z zabronieniami Tabu Search

Algorytm Tabu Search bazuje na przestrzeni, stworzonej z wszystkich możliwych rozwiązań dla podanej instancji problemu. Za pomocą określonej sekwencji ruchów przeszukujemy w poszukiwaniu potencjalnego rozwiązania. Przeszukiwanie polega na sprawdzaniu sąsiednich ścieżek aktualnej bazowej ścieżki.

a) Typy sąsiadów

Sąsiednie ścieżki nazywamy permutacje wykonaną jedną z operacji: *swap(a,b) insert(a,b) invert(a,b)*. Definiując permutacje (1) *n*-elementową (reprezentowana jako ścieżka o długości *n*):

$$\pi =< \pi(0), \pi(1), \pi(2), \pi(3), \pi(4), \pi(5), \pi(n-1) > \tag{1}$$

- Sąsiedztwo typu świap":
  Definiowane jako zmiana dwóch elementów ("miast")

$$swap(1,4) \quad \pi^* =< \pi(0), \mathbf{\pi(4)}, \pi(2), \pi(3), \mathbf{\pi(1)}, \pi(5), \pi(n-1) > \tag{2}$$

- Sąsiedztwo typu "insert":
  Wstawienie przed jeden z elementów drugi

$$insert(1,4) \quad \pi^* =< \pi(0), \mathbf{\pi(4)}, \mathbf{\pi(1)}, \pi(2), \pi(3), \pi(5), \pi(n-1) > \tag{3}$$

- Sąsiedztwo typu "invert":
  Zmiana poprzez odwrócenie kolejności wybranych elementów

$$invert(1,4) \quad \pi^* =< \pi(0), \mathbf{\pi(4)}, \mathbf{\pi(3)}, \mathbf{\pi(2)}, \mathbf{\pi(1)}, \pi(5), \pi(n-1) > \tag{4}$$

Definicje sąsiadów z listingów (2)(3)(4) zaimplementowane zostały przy użyciu instrukcji *switch ... case* w zależony od wybranej opcji tworzenia sąsiadów oraz odpowiednich funkcji realizujących zamiany. Na listingu (1) *temp[]* reprezentuje obecną główną ścieżkę.

```
switch (neighbourMode) {
    case 0:
      std::swap(temp[i], temp[j]); break;
    case 1:
      int val = temp[j];
      temp.erase(temp.begin() + j);
      temp.insert(temp.begin() + i, val);
      break;
    case 2:
      std::reverse(temp.begin() + i, temp.begin() + j + 1); break;
}
```

Listing 1: Instukcje realizujące tworzenie sąsiadów

Wytwarzaniem kolejnych kombinacji wykorzystuje zagnieżdżone pętle *for()* (listing 2). Aby nie powtarzać tych samych kombinacji wewnętrzna pętla zawsze zaczyna od wartości o 1 większej nic aktualna w zwnętrznej pętli.

```
for (size_t i = 0; i < matrixSize; i++) {
    for (size_t j = i + 1; j < matrixSize; j++) {
    }
}
```

Listing 2: Wytwarzanie kolejnych permutacji

b) Lista tabu

Nowo utworzony sąsiad zostaje sprawdzony, jeśli uzyskany koszt jest mniejszy, zostaje on przypisany jako tymczasowy najlepszy ruch. Po zakończeniu pętli zewnętrznej najlepszy z nich zostaje główną ścieżką. Na tym etapie bardzo łatwo aby algorytm znalazł się w lokalnym minimum. Następne wyszukiwanie najlepszego sąsiada mogło by powrócić do poprzedniego rozwiązania. W algorytmie Tabu Search jednym z rozwiązań tego problemu i jednym z głównych założeń algorytmu jest utworzenie listu tabu. Listy zawierającej wszystkie ostatnie najlepsze ruchy (sąsiadów). W kolejnych iteracjach i poszukiwaniach nie sprawdzamy ich. W tym celu zaimplementowana została vector struktur (listing 3) *std::vector<Tabu> tabuList* oraz jaka druga możliwość tablica dwuwymiarowa *int **tabuArray*, w której *tabuArray[a][b]* a i b oznaczają zakazany ruch (a,b).

```
struct Tabu {
    int move[2];
    int cadence;

    Tabu(int move[2], int life) {
      this->move[0] = move[0];
      this->move[1] = move[1];
      this->cadence = life;
    }
};
```

Listing 3: Struktura listy tabu

Mechanizm sprawdzania, czy aktualnych ruch należy do listu tabu prezentuje listing (4). Zawarte są obie implementacje, zarówno lista struktur jak i dwuwymiarowa tablica.

```
if (representationMode == 0) {
    for (Tabu tl : tabuList)
        if ((tl.move[0] == i && tl.move[1] == j) ||
           (tl.move[0] == j && tl.move[1] == i))
            forbiddenMove = true;
} else {
    for (size_t k = 0; k < matrixSize; k++)
        for (size_t l = k + 1; l < matrixSize; l++)
            if (tabuArray[k][l] == 0 || tabuArray[k][l] == 0)
                forbiddenMove = true;
}
```

Listing 4: Sprawdzanie ruchów zakazanych

Gdy ruch znajduje się na liście, ustawiona zostaje zmienna typu bool *forbiddenMove = true* na wartość true. Następnie pomijany jest etap tworzenia sąsidniej ścieżki prezentowanej na listingu (1).
W definicji struktury pojawia się pojęcie kadencji (ang cadence). Jest to liczba oznaczająca długość, liczbę iteracji poszukiwania sąsiadów, w jakiej dany ruch pozostaje na liście tabu. W strukturze jest to jeden z atrybutów *int cadence*, natomiast w tablicy pole o indeksach zakazanego ruchu przyjmuje wartość kadencji. Po znalezieniu najlepszej sąsiedniej ścieżki wszystkie kadencje maleją o 1, i dodawany jest nowy zakazany ruch. Zmniejszanie i usuwanie ruchów których kadencja już minęła prezentuje listing (5), dla listy tabu jako lista struktur *Tabu*. Dla tablicy operacje są analogiczne.

```
for (auto it = tabuList.begin(); it != tabuList.end();) {
    it->cadence -= 1;
    if (it->cadence == 0)
        it = tabuList.erase(it);
    else
        it++;
}
Tabu t = Tabu(tempTabuMove, this->cadence);
tabuList.push_back(t);
```

Listing 5: Zmniejszanie kadencji i usuwanie ruchów

c) Zaimplementowane ograniczenia

Kolejnym ograniczeniem algorytmu jest liczba gorszych rozwiązań. Jeśli rozwiązanie, czyli aktualny najlepszy ruch jest gorszy od najmniejszego kosztu globalnego liczba zwiększana jest o 1. Przed rozpoczęciem algorytmu podajemy maksymalną wartość, jej przekroczenie wywołuje przypisanie aktualnej ścieżki jako losową. Takie rozwiązanie zmniejsza możliwość zapętlania się algorytmu w minimach lokalnych.

```
if (tempMin < result) {
    result = tempMin;
    bestPath = mainPath;
    worseResults = 0;
} else if (worseResults == worseResultsLimit) {
    worseResults = 0;
    tabuList.clear();
    mainPath = p.randomVectorPath(matrixSize, 0, matrixSize - 1);
} else if (tempMin >= result) {
    worseResults++;
}
```

Listing 6: Sprawdzanie aktualnego rozwiązania z globalnym minimum

Dodatkową opcją zaimplementowanego algorytmu jest możliwość wyboru pierwszej głównej ścieżki wejściowej. Jedną jest pierwsza permutacja dla problemu zgodnie z definicją (5). Drugą możliwością jest losowa ścieżka generowana przez pomocniczą klasę *Path*. Metodą (7) tej klasy jest *std::vector<int> randomVectorPath(int pathLength, int min, int max)* zwracana losowa ścieżka reprezentowana jest jako wektor.

$$firstPerm(n) \quad \pi =< \pi(0), \pi(1), \pi(2), \pi(3), \pi(4), ..., \pi(n-2), \pi(n-1) > \tag{5}$$

```
srand(unsigned(time(0)));
if ((max - min) < pathLength && min >= max)
    throw std::length_error("Path length out of range");

std::vector<int> temp;
for (size_t i = min; i <= max; i++)
    temp.push_back(i);
random_shuffle(temp.begin(), temp.end());
std::vector<int> newRandom(temp.begin(), temp.begin() + pathLength);
return newRandom;
```

Listing 7: Generowanie losowej ścieżki

Znając wszystkie możliwości tworzymy obiekt klasy *TabuSearch*, korzystając z konstruktora. Zawiera on listę inicjalizacyjną.

```
TabuSearch::TabuSearch(int iterat, int repMode, int cadence, int startMode,
    int neighbourMode, int lim)
    : iterations(iterat), representationMode(repMode), cadence(cadence),
      startMode(startMode), neighbourMode(neighbourMode),
      worseResultsLimit(lim) {}
```

Listing 8: Konstruktor klasy *TabuSearch*

3 Eksperymenty obliczeniowe

Obliczenia zastały wykonane na komputerze klasy PC z procesorem AMD Ryzen 5 2600, kartą graficzną NVIDIA GeForce GTX 1060 6BG, 16GB RAM i DYSK SSD. W obliczeniach odchylenia standardowego wykorzystano wzór:

3.1 Szybkość obliczeń

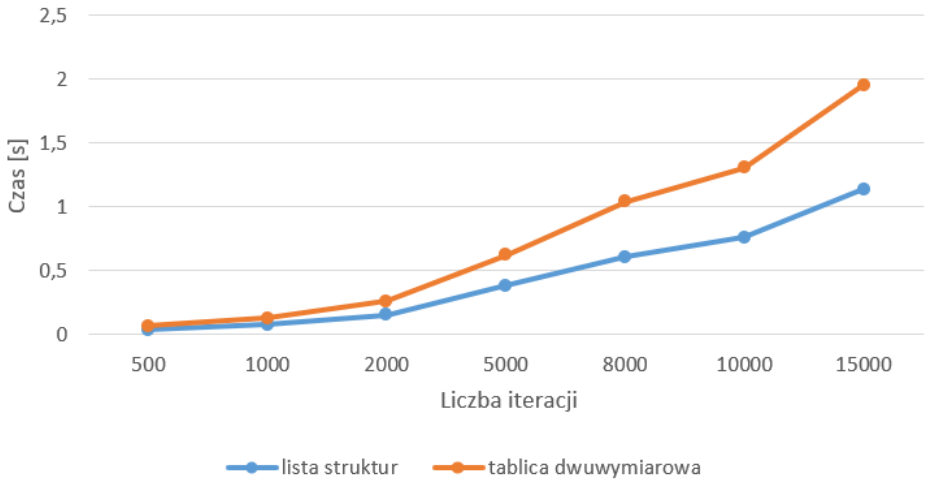
Pierwszym wykonanym eksperymentem jest pomiar czasu w zależności od reprezentacji listy Tabu. Testy wykonywane są na problemie wielkości 17 oraz 58. W obu przypadkach

- liczba testów dla każdej wartości iteracji jest równa 20
- wejściową ścieżką jest pierwsza permutacja
- sąsiedzi dobierani są metodą swap
- kadencja jest połową wielkości problemu
- limit gorszych rozwiązań jest równy wielkości problemu
- wynik dla poszczególnej ilości iteracji jest średnią 20 pomiarów

Tablica 1: Szybkość algorytmów w zależności od reprezentacji

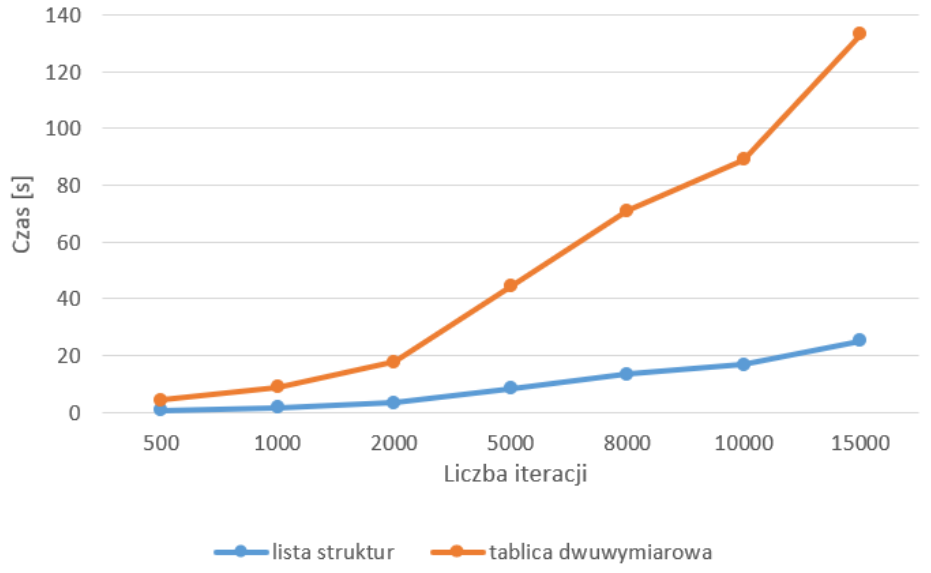
Wielkość problemu:	**n = 17**		**n = 58**	
Iteracje	**Struktura**	**Tablica**	**Struktura**	**Tablica**
500	0,038 s	0,065 s	0,870 s	4,431 s
1000	0,075 s	0,130 s	1,709 s	8,903 s
2000	0,152 s	0,260 s	3,390 s	17,826 s
5000	0,381 s	0,625 s	8,433 s	44,499 s
8000	0,608 s	1,040 s	13,492 s	71,121 s
10000	0,760 s	1,304 s	16,825 s	88,937 s
15000	1,137 s	1,953 s	25,267 s	132,992 s

Analizując tabele 1 można zauważyć różnicę w czasach wykonywania algorytmu. Implementacja bazująca na wektorze struktur wypada w tym teście zdecydowanie pomyślniej. Dla mniejszego problemu różnica jest niewielka, lecz gdy liczba miast do przebycia przez naszego podróżnika zwiększa się, różnica w czasie również.



Rysunek 1: Zależność czasu od wybranej reprezentacji dla problemu n=17.

Zobrazowane wyniki na wykresach 3.1 oraz 3.1 wyraźnie pokazują różnicę w prędkości między implementacjami. Podczas gdy, przy problemie wielkości 17 różnica czasu dla 15000 iteracji jest niemalże 2-krotna, już w kolejnym przypadku przekracza 5-krotność czasu implementacji wektora struktur.

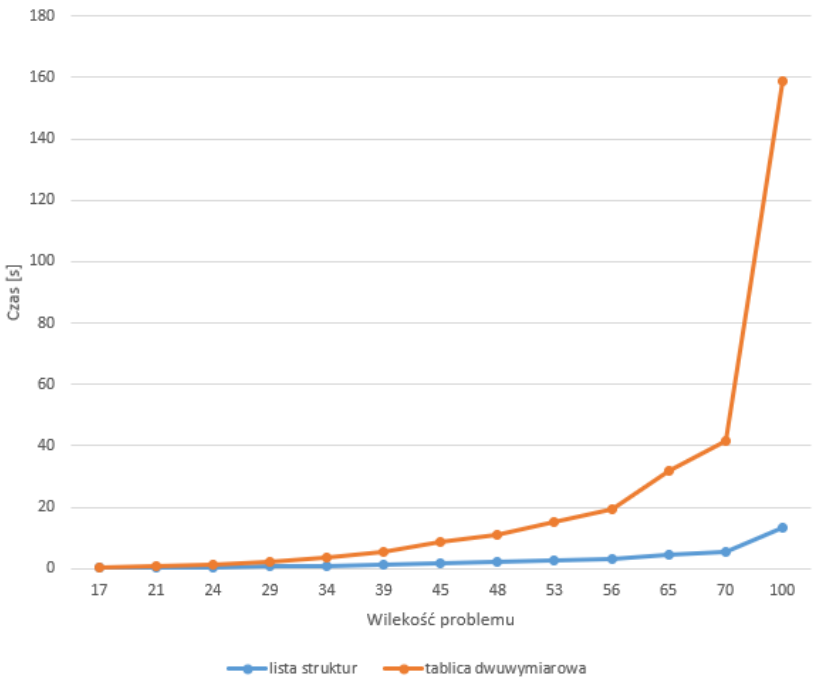


Rysunek 2: Zależność czasu od wybranej reprezentacji dla problemu n=58.

Szybkość wykonania algorytmu zależeć może również od wielkości problemu. Dla tych samych ustawień algorytmu, różnica jedynie w reprezentacji, przetestujemy szybkość otrzymywanego wyniku.

Tablica 2: Szybkość algorytmów w zależności od wielkości problemu

Wielkość problemu	**Struktura**	**Tablica**
17	0,182	0,245
21	0,293	0,493
24	0,397	0,869
29	0,612	1,459
34	0,895	2,811
39	1,238	4,157
45	1,737	6,996
48	2,063	8,890
53	2,609	12,787
56	2,992	16,573
65	4,449	27,461
70	5,352	36,374
100	13,258	145,283



Rysunek 3: Zależność czasu od wielkości problemu.

3.2 Dokładność wyników

a) Zależność od wielkości problemu

Pierwszym przykładem dla małych instancji problemu i małej liczby iteracji. Zostało wykonane 25 prób dla każdej wielkości problemu. Kadencja równa 20, a limit gorszych wyników 800. Sąsiedztwo typu świap". Ponieważ poszukujemy rozwiązania pierwszą główną ścieżką jest losowa permutacja, pozwoli to zwiększyć losowość każdej próby.

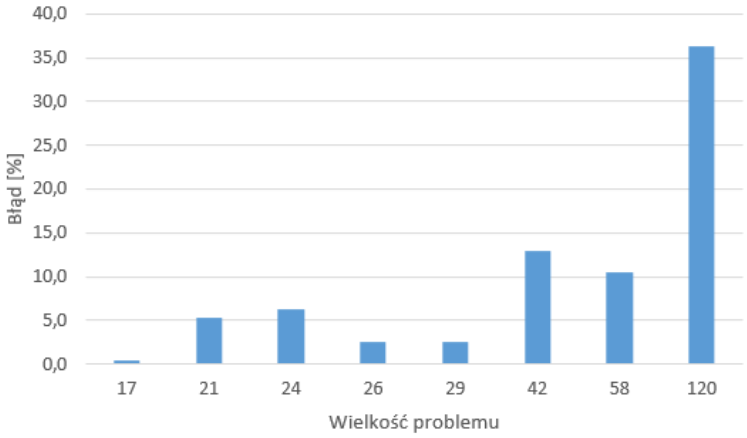
Tablica 3: Uzyskany wynik dla małych instancji problemu

Wielkość problemu	**Wartość najniższa**	**Wynik pomiarów**	**Błąd**
10	212	212	0,0%
11	202	202	0,0%
12	264	264	0,0%
13	269	269	0,0%
14	125	125	0,0%
15	291	291	0,0%
16	156	156	0,0%
17	??	2085	??%
18	187	187	0,0%

Dla problemów o większej ilości wierzchołków. Dokładność algorytmu zależy od jego ustawień. Możemy uzyskać w pewnych warunkach przegląd zupełny, dlatego ustawienia algorytmu zostały zmienione dla zróżnicowania wyników. Algorytm wykonuje się w czasie <1s, a jego wyniki prezentuje tabele 4.

Tablica 4: Uzyskany wynik dla małych instancji problemu

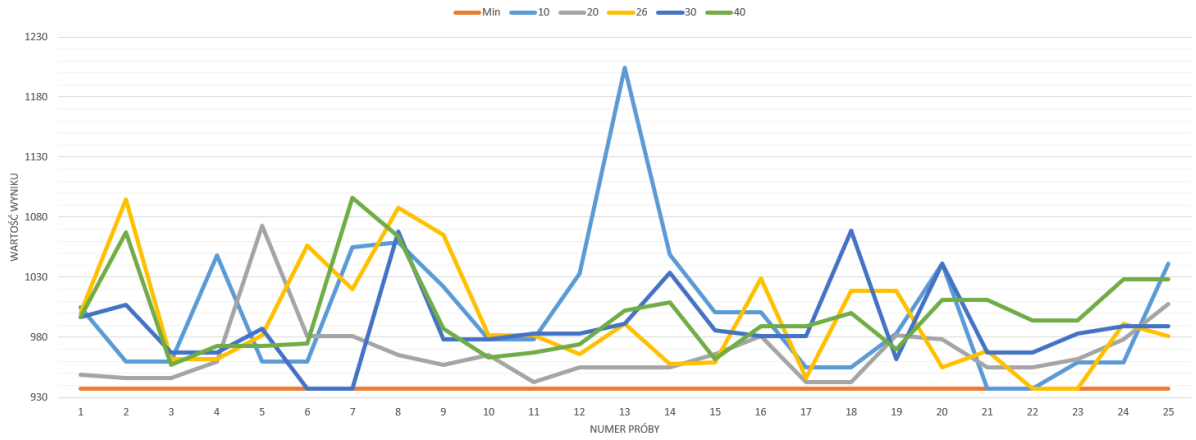
Wielkość problemu	**Wartość najniższa**	**Wynik pomiarów**	**Błąd**
17	2085	2095	0,5%
21	2707	2852	5,4%
24	1272	1351	6,2%
26	937	961	2,6%
29	1610	1651	2,5%
42	699	789	12,9%
58	25395	28075	10,6%
120	6942	9459	36,3%



Rysunek 4: Błędy pomiarów w zależności od wielkości problemu.

b) Wpływ kadencji na wynik

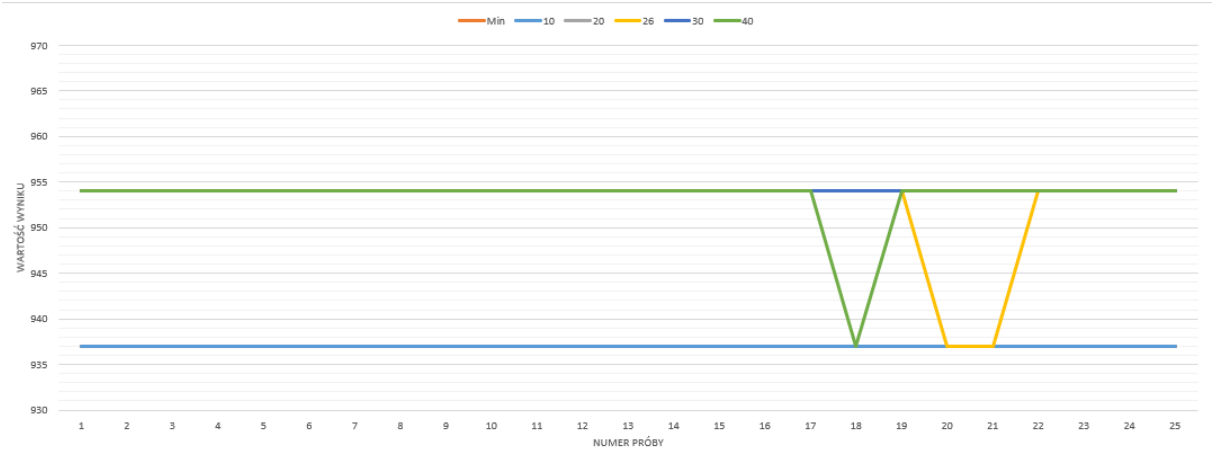
Jednym z ograniczeń algorytmu jest kadencja. Określa liczbę jak i długość listy tabu, zawierającej zakazane ruchy. W tej części przetestujemy algorytm zmieniając jedynie kadencję. Test wykonamy na problemie o wielkości 26. Początkową ścieżką jest permutacja losowa.



Rysunek 5: Wpływ kadencji na wynik.

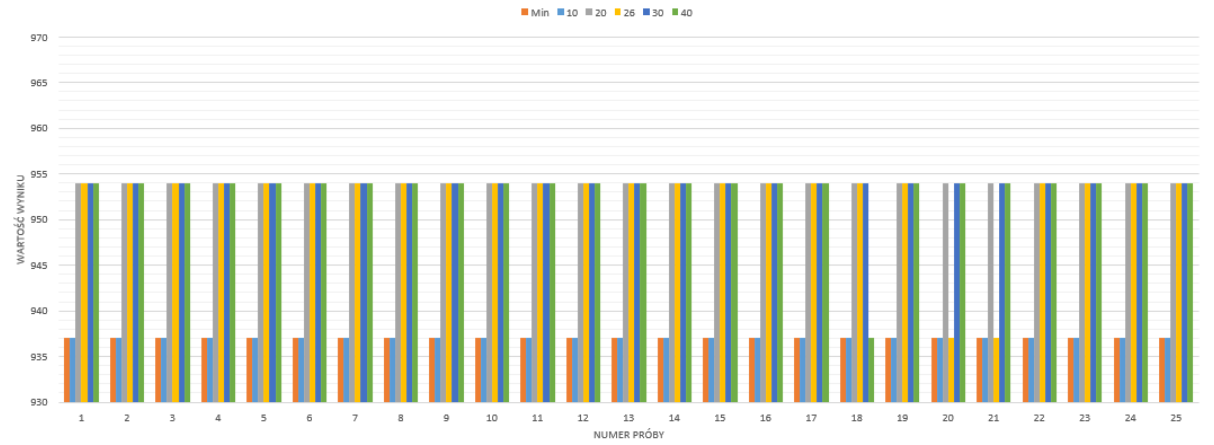
Wykresy 5, 6 jak i 7 przedstawią jako *Min* minimalne rozwiązanie tego problemu. Kolejno liczby 10, 20, 26, 30, 40 to długość listy tabu, kadencja. Wyniki uzyskane na wykresach 6 oraz 7 różnią się znacząco, ponieważ ścieżką wejściową była pierwsza permutacja.

Ponieważ na wykresie 6 wartości pokrywają się, przedstawione na wykresie słupkowym (7) są znacznie czytelniejsze lecz nie oddają różnicy. Jak możemy zaobserwować kadencja nie ma bezpośredniego wpływu na wynik. Składa się na to ustawienie ścieżki wejściowej, gdy jest ona losowa również wyniki uzyskiwane przy różnych ustawieniach kadencji są bardzo losowe. Ustawienie jej na pierwszą permutację diametralnie zmieniają wyniki algorytmu, stają się one być niezmienne. Porównując oba wykresy



Rysunek 6: Wpływ kadencji na wynik.

wydaję się, że kadencja nie ma wpływu na wynik. Lecz nie jest to jedyne ograniczenie algorytmu i w połączeniu z innymi daje zupełnie inne wyniki.



Rysunek 7: Wpływ kadencji na wynik.

c) Ograniczanie liczby gorszych wyników

Jednym z zaimplementowanych ograniczeń jest limit gorszych wyników porównując to minimum globalnego. Testowany algorytm na ustawieniach:

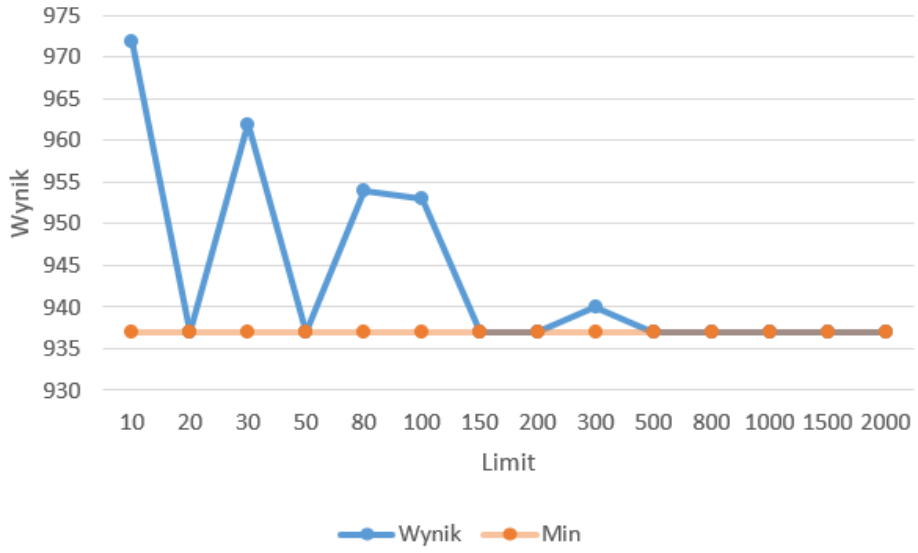
- Liczba iteracji równa 4000
- Reprezentacja listy tabu jako wektory struktur
- Kadencja równa 20
- Początkowa ścieżka - losowa

Wartości limitu prezentuje tabela 5, oraz wartości minimalne osiągnięte przez algorytm w danych 25 próbach dla konkretnej wartości limitu.

Tablica 5: Uzyskane wyniki dla danych limitów

Limit	**Wynik pomiarów**
10	972
20	937
30	962
50	937
80	954
100	953
150	937
200	937
300	940
500	937
800	937
1000	937
1500	937
2000	937

Stosując ograniczenie w postaci limitu gorszych rozwiązań, należy pamiętać aby limit nie był zbyt mały. Algorytm nie może należć minimalnego rozwiązania gdy go zbyt ograniczamy.



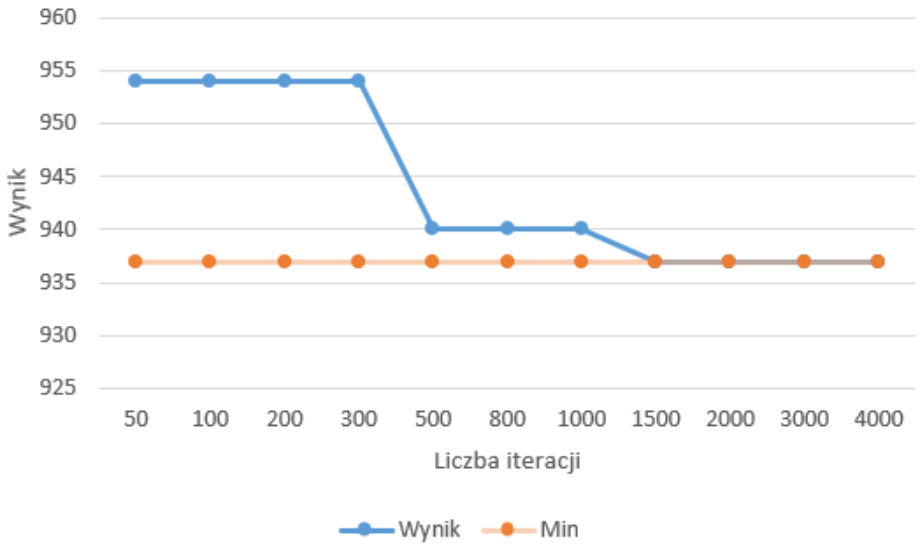
Rysunek 8: Wpływ limitu na wynik.

d) Wpływ liczby iteracji na wynik

Jak już wiemy dużą rolę w końcowym wyniku ma odpowiednie ustawienie limitu. W tym teście mamy na celu skupienie się nad liczbą iteracji zatem limit ustawimy na 1000. Na początku ścieżkę startową ustawimy na pierwszą permutację.

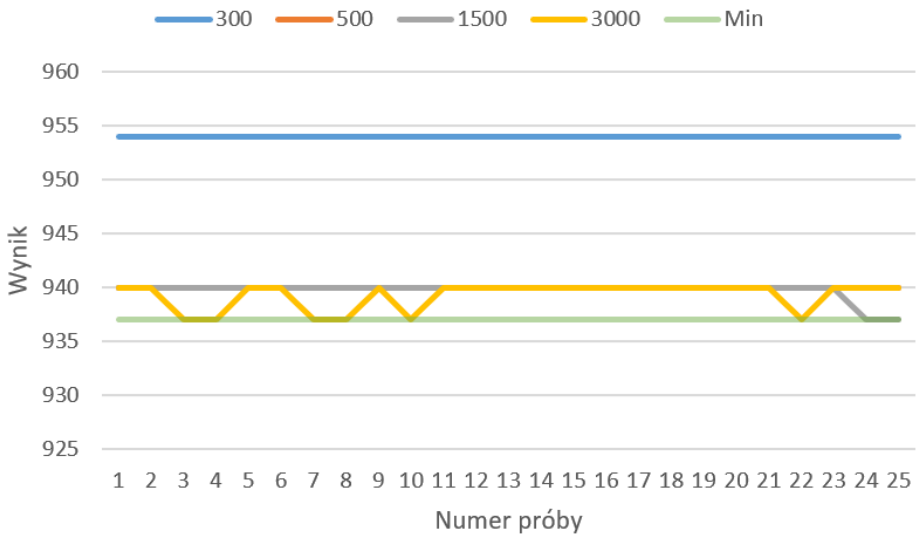
Tablica 6: Wyniki dla różnej ilości iteracji

Liczba iteracji	Wynik pomiarów
50	954
100	954
200	954
300	954
500	940
800	940
1000	940
1500	937
2000	937
3000	937
4000	937



Rysunek 9: Iteracje a wynik

Ilość iteracji ma bezpośredni wpływ na otrzymany wynik. Zwiększając ich liczbę wydłużamy czas oraz zwiększamy dokładność. Wykres 11 dla wybranych serii ukazuje zmianę w częstości uzyskiwanych minimów rozpatrywanego problemu. Dla większej liczby iteracji, najlepszy wynik występuje częściej.



Rysunek 10: Częstość występowania minimum

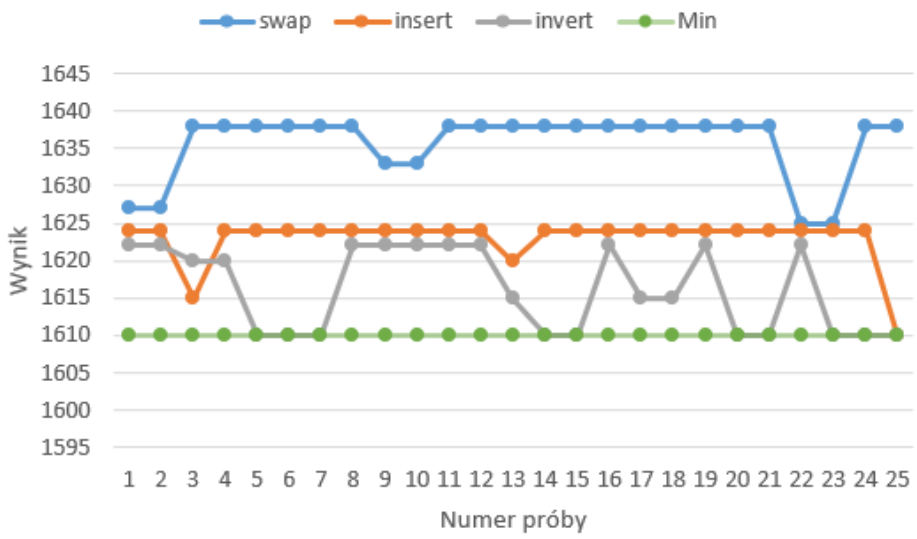
e) Różne typy sąsiedztw, a wynik

Testowane typy sąsiedztw zostały wyżej opisane (punkt 2.1a))

Tablica 7: Wyniki dla różnej ilości iteracji

Wielkość problemu	**Min**	**swap**	**insert**	**invert**
21	2707	2758	2707	2707
24	1272	1330	1272	1272
29	1610	1625	1610	1610
42	699	700	699	699

Podczas testów dobierane były takie wartości iteracji, aby dało się zauważyć różnice w otrzymanych wynikach dla różnych typów sąsiednich ścieżek.



Rysunek 11: Częstość występowania minimum dla wielkości problemu 29

4 Wnioski

Algorytm Tabu Search wykonuje się w bardzo krótkim czasie. Dla małych problemów, wielkości nawet do 50 wierzchołków (miast), możemy znaleźć minimalne rozwiązanie dla danego problemu w mniej niż sekundę. Czas w jakim otrzymujemy wynik jest imponujący, lecz sam wynik nie zawsze musi być najmniejszy z możliwych. Ideą algorytmu są jego ograniczenia. Jednym z najważniejszych jest lista tabu, której długość możemy nadawać. Wpływ jaki ma na algorytm zależy od jego innych ograniczeń i ich ustawień. Z wykonanych eksperymentów wiemy, że sama czasami jedna blokada nie zmieni znacznie działanie algorytmu. Zauważalne różnice zarówno jak i w czasie wykonywania algorytmu jak i w jego dokładności ma liczba iteracji. Czym więcej iteracji tym dłużej jak i dokładniej. Kolejnym czynnikiem wpływającym na wynik jest limit gorszych rozwiązań. Zbyt mały może zakłócić prawidłową pracę algorytmu. Algorytm Tabu Search jest algorytmem, który w zależności od ustawień wykonywać kilka operacji i zwracać wynik bliski najlepszemu, lecz także wykonywać się dłużej, a wynik z większym prawdopodobieństwem będzie równy minimum. Należy jednak pamiętać, że nie jest to algorytm dokładny, więc nie możemy mieć pewności co do minimalnego wyniku.