Projektowanie Efektywnych Algorytmów

Projekt

20.12.2023

263900 Aleksandra Chrustek

(3) Tabu Search

1 Sformułowanie zadania

Zadanie polega na opracowaniu, napisaniu i zbadaniu problemu komiwojażera w wersji optymalizacyjnej algorytmem Tabu Search. Problem komiwojażera (eng. Travelling salesman problem, TSP) to zagadnienie polegające (w wersji optymalizacyjnej) na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym.

1. Graf pełny to zbiór wierzchołków, przy czym między każdymi dwoma wierchołkami istnieje krawędź je łącząca. [1]
2. Cykl Hamiltona to droga wiodąca przez wszystkie wierzchołki dokładnie raz, z wyjątkiem jednego wybranego, w którym cykl Hamiltona zaczyna się oraz kończy. [2]

Problem komiwojażera rozumiemy jako zadanie polegające na znalezienu najlepszej drogi dla podróżującego chcącego odwiedzić n miast i skończyć podróż w miejscu jej rozpoczęcia. Połączenie między każdym miastem ma swój koszt określający efektywnosć jej przebycia. Najlepsza droga to taka, której całkowity koszt (suma kosztów przebycia wszystkich połączeń między miastami na drodze) jest najmiejszy. Problem dzieli się na symetryczny i asymetryczny. Pierwszy polega na tym, że dla dowolnych miast A i B z danej instancji, koszt połączenia jest taki sam w przypadku przebycia połączenia z A do B, jak z B do A, czyli dane połączenie ma jeden koszt niezależnie od kierunku ruchu. W asymetrycznym problemie komiwojażera koszty te mogą być różne.

1. Sąsiedztwo rozwiązania - nazywamy tak zbiór permutacji otrzymanych poprzez wykonanie zaburzeń (np. ruchów typu swap(i,j) - zamiany miejscami elementu i z elementem j).
2. Lista Tabu - zbiór wykonanych zaburzeń (np. ruchów typu swap(i,j)), jest to kolejka typu LIFO, czyli Last In First Out - tzn. w momencie dodania nowego elementu, ostatni z nich jest usuwany. Może mieć dowolną ustaloną z góry długość.
3. Kryterium aspiracji - czasem ruchy nalezące do listy Tabu mogą okazać się dużo lepsze od dostępnych, przez co pominięcie ich może prowadzić do pominięcia rozwiązania optymalnego. W takim wypadku jest dozwolone wykorzystanie ruchu z listy Tabu. Kryterium aspiracji określa warunek, podczas którego jest to wykonywane.
4. Strategia dywersyfikacji - procedura przeszukiwania różnych obszarów lokalnych.
5. Długość kadencji - określa, ile iteracji dany ruch lub zamiana miast będzie uznawana za zakazaną na liście tabu.

2 Metoda

Algorytm Tabu Search, inaczej poszukiwanie z zakazami, należy do rodziny algorytmów typu local search (wyszukiwanie lokalne). Metoda oparta na iteracyjnym przeszukiwaniu przestrzeni rozwiązań, wykorzystując sąsiedztwo pewnych elementów tej przestrzeni oraz zapamiętując przy tym przeszukiwaniu ostatnie ruchy(transformacje rozwiązań) i częstość ich występowania, w celu unikania minimów lokalnych i poszukiwania rozwiązań globalnie optymalnych w rozsądnym czasie. Algorytm ten nie jest trudny w implementacji, jednakże nie gwarantuje znalezienia optymalnego rozwiązania, a w swojej najprostszej postaci generuje spory % błędu. Czas wykonania algorytmu zależny jest przede wszystkim od ustalonej liczby iteracji jakie ma wykonać oraz sposobu przeszukiwania sąsiadów obecnego rozwiązania. Zgodnie z tym w przypadku problemu komiwojażera liczba przeszukiwanych sąsiadów danego rozwiązania wynosić będzie *(n1)*(n2)/2*, gdzie n to liczba miast. Sąsiedzi dla danego rozwiązania generowani są za pomocą metody typu swap, która podmienia kolejno ze sobą wybrane miasta w drodze.
Schemat działania algorytmu przedstawia się następująco:

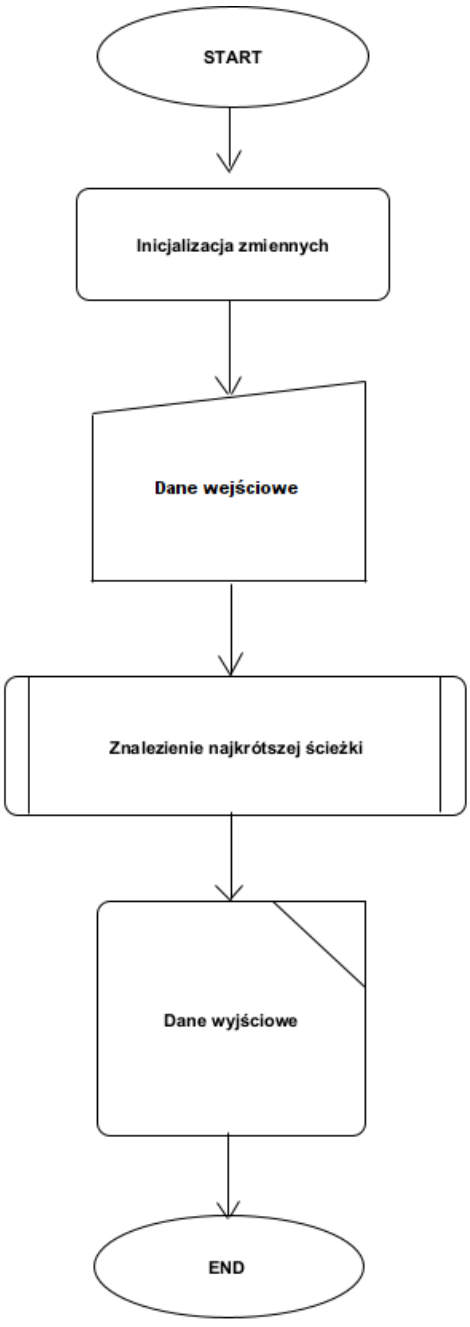
1. Wybieramy drogę początkową na sposób dowolny.
2. Wyliczamy jej koszt, tworzymy wyzerowaną listę tabu.
3. Przeszukujemy wszystkich sąsiadów, wybieramy najlepszego (chyba że wcześniej natkniemy się na większą poprawę obecnej drogi względem sąsiada od naszego kryterium aspiracji – wtedy przerywamy poszukiwania po założonej liczbie iteracji).
4. Modyfikujemy listę tabu, tak aby dana zamiana nie mogła wystąpić ponownie przez żądaną liczbę iteracji, kadencję poprzednich zmian zmniejszamy o 1.
5. Obecnie znalezioną drogę ustawiamy jako początkową oraz jako optymalną, jeżeli jest lepsza od optymalnej.
6. Powtarzamy kroki 3-5 dopóki nie wykonamy zadanej liczby iteracji.

Wybrana implementacja rozpoczyna algorytm od zainicjalizowania zmiennej *curr_sol* listą miast *cities*. Następnie używana jest funkcja *shuffle*, aby losowo permutować kolejność miast w tej liście. Funkcja *shuffle* korzysta z generatora liczb losowych, aby uzyskać losową permutację miast. W efekcie otrzymujemy losową początkową ścieżkę, która stanowi punkt startowy algorytmu Tabu Search. Zostały zaimplementowane dwa kryteria stopu algorytmu. Pierwsze to kryterium czasowe *stop* podawane przez użytkownika w sekundach lub ustawiane domyślnie na 5s. Kryterium jest weryfikowane pod koniec każdej iteracji algorytmu. Sprawdzane jest czy obecny czas wykonania nie przekracza maksymalnego czasu wykonania. Jeśli przekracza, to algorytm kończy zapętlenie. Kolejne kryterium to weryfikacja ilości iteracji bez zmiany najlepszej ścieżki *max_iterations*. Warunkiem wykonania pętli jest, aby dotychczasowa ilość iteracji bez poprawy *iterations_without_improvement* nie przekroczyła ilości podanej przez użytkownika. Jeśli nastąpi poprawa, to obecna ilości iteracji bez poprawy jest zerowana. Pary miast na aktualnej ścieżce są wymieniane przy pomocy funkcji *Swap*. Funkcja ta iteruje przez wszystkie możliwe pary miast w aktualnej ścieżce i dokonuje zamiany

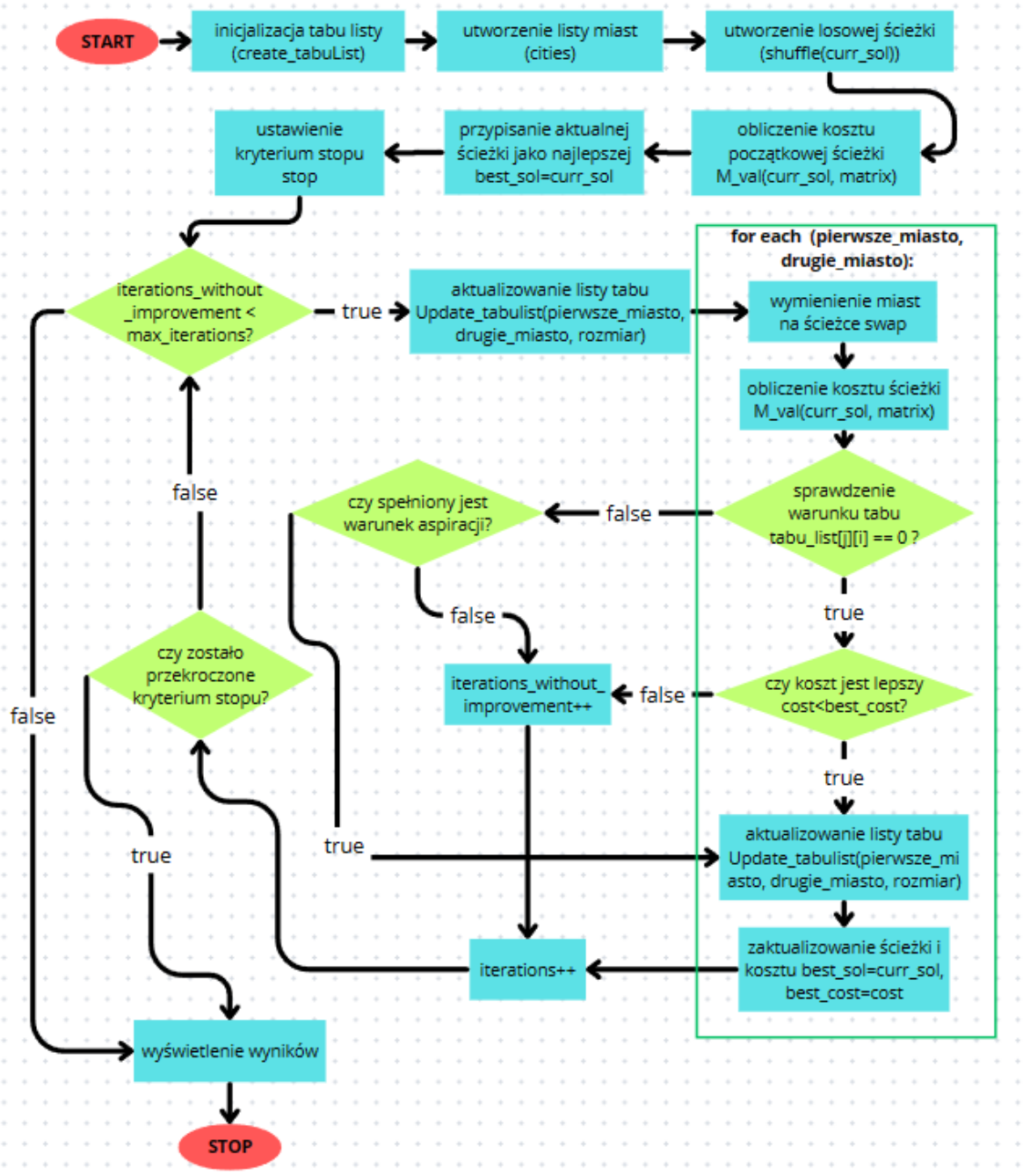
miejscami. Następnie sprawdza warunek tabu i aspiracji dla każdej zamiany. Jeśli warunek tabu jest spełniony (element tabu listy dla danej pary miast wynosi 0), a nowa ścieżka jest lepsza od dotychczas najlepszej, to zamiana jest akceptowana, a tabu lista oraz najlepsza ścieżka są aktualizowane. Jeśli warunek tabu nie jest spełniony, ale zamiana spełnia kryterium aspiracji (nowy koszt jest niższy od najlepszego kosztu), akceptowana jest zamiana miast, a tabu lista oraz najlepsza ścieżka są aktualizowane. Jeśli zamiana miast nie została zaakceptowana, inkrementowany jest licznik iteracji bez poprawy *iterations_ without_ improvement*. Rolę długości kadencji dla pary miast w implementacji pełni zmienna *rozmiar*. Kadencja ustawiana jest dla pary miast w funkcji *Add_ to_ tabulist*. Algorytm jest wykonywany, aż do przerwania pętli przez jedno z kryterium stopu.
Czynnikami jakie mogą wpływać na jakość rozwiązania jest ilość iteracji algorytmu, długość kadencji zamiany na liście tabu, wartość aspiracji (dla której ignorujemy zapis na liście tabu) oraz ilość iteracji po skorzystaniu z kryterium aspiracji.

3 Algorytm

Rozwiązanie zaimplementowane w programie obrazują następujące schematy.



Rysunek 1: Schemat blokowy programu



Rysunek 2: Schemat blokowy algorytmu opartego na metodzie Tabu Search

Opis algorytmu:

1. Tworzenie listy miast *cities*, a następnie generowanie losowej początkowej ścieżki *curr_sol* poprzez permutację miast.
2. Ustawienie aktualnej ścieżki jako najlepszej *best_sol* = *curr_sol*.
3. Ustawienie kryterium stopu dla algorytmu *stop*.
4. Inicjalizowanie tabu listy za pomocą funkcji *Create_tabuList*.
5. Aktualizowanie tabu listy za pomocą funkcji *Update_tabulist*.
6. Dla każdej pary miast (i, j) w aktualnej ścieżce:
   -Wymienienie miasta i i j.
   -Obliczenie kosztu nowej ścieżki *cost*.
   -Sprawdzenie warunku tabu i aspiracji: Jeśli warunek tabu jest spełniony (element tabu listy dla pary miast jest równy 0) lub warunek aspiracji jest spełniony (nowy koszt jest niższy od najlepszego kosztu), akceptowana jest zamiana miast.
   -Aktualizowanie tabu listy i najlepszej ścieżki w przypadku akceptacji zamiany.
   -Inkrementacja licznika iteracji bez poprawy, jeśli zamiana nie została zaakceptowana.
7. Inkrementacja licznika iteracji.
8. Sprawdzenie, czy algorytm osiągnął kryterium stopu. Jeśli tak, przerwanie pętli.

4 Dane testowe

Do sprawdzenia poprawności działania algorytmu i wykonania badań wybrano następujący zestaw instancji:
Dla problemu symetrycznego:
burma14.tsp 3323
gr21.tsp 2707
gr24.tsp 1272
gr202.tsp 40160
rbg323.tsp 1326
Dla problemu asymetrycznego:
p43.atsp 5620
ft70.atsp 38673
rbg358.atsp 1163
rbg443.atsp 2720
Instancje zostały pobrane ze zbioru TSPLIB[3].

5 Procedura badawcza

Zbadana została zależność czasu rozwiązania problemu od wielkości instancji, jak również wpływ na rozwiązanie ilości iteracji. Obliczony został błąd względny rozwiązania według wzoru: $(|x - y|/x) * 100\%$, gdzie x to koszt optymalny, a y to koszt obliczony. Procedura badawcza polegała na uruchomieniu programu sterowanego plikiem inicjującym *conf.ini*
(format pliku: *nazwa_instancji liczba_wykonań nazwa_pliku_wyjściowego*).
Każda z instancji rozwiązywana była 10 razy dla poszczególnych liczb iteracji: 100, 500, 1000, 2000, 3000. Do pliku wyjściowego csv zapisywane były informacje o instancji: jej nazwa, liczba wykonań algorytmu, kryterium stopu w sekundach, maksymalna liczba iteracji bez polepszenia najlepszej ścieżki. Następnie zapisywane były czasy wykonań algorytmu oraz obliczony koszt dla każdego testu tej instancji. Plik wyjściowy zapisywany był w formacie csv. Poniżej przedstawiono zawartość przykładowego pliku wyjściowego dla instancji burma14.tsp przy 100 iteracjach.

burma14.tsp 10 1000 100
4;4732
3;3370
3;3367
3;3615
3;3367
2;3627
2;3503
3;3735
3;3629
2;4056
testburma14.csv

Pomiary zostały wykonane na platformie sprzętowej:
procesor: Intel® Core™ i5-01400F CPU @ 2.90GHz × 6
pamięć operacyjna: 16 GiB
system operacyjny: Windows 11 64-bit

W programie czas jest mierzony za pomocą funkcji *read_QPC()*, która wykorzystuje API systemu Windows do odczytywania licznika wydajności procesora (QPC - QueryPerformanceCounter).

```
long long int read_QPC()
{
    LARGE_INTEGER count;
    QueryPerformanceCounter(&count);
    return((long long int)count.QuadPart);
}
```

Opis działania:

1. *QueryPerformanceCounter(&count)* pobiera aktualną wartość licznika wydajności procesora i zapisuje ją w strukturze *count*.
2. *count.QuadPart* zawiera odczytaną wartość, która jest zwracana jako long long int.
3. W funkcji *choose_ option()* jest wykorzystywana funkcja *QueryPerformanceFrequency*, która zwraca liczbę tików licznika wydajności na sekundę. Jest to wykorzystywane do przekształcenia różnicy czasu na milisekundy:

```
long long int frequency, start, elapsed;
QueryPerformanceFrequency((LARGE_INTEGER *)&frequency);
```

Cały proces mierzenia czasu wygląda więc następująco:

1. Pobierana jest częstotliwość licznika wydajności za pomocą *QueryPerformance-Frequency* i zapisywana w zmiennej *frequency*.
2. Przed wykonaniem operacji, której czas chcemy zbadać, wykonujemy *start = read_ QPC()* aby zapisać aktualną wartość licznika wydajności.
3. Po wykonaniu operacji, ponownie wywołujemy *read_ QPC()* i odejmujemy *start* od odczytanej wartości, aby uzyskać czas, jaki upłynął.
4. Następnie przekształcamy różnicę czasu na milisekundy - używając (1000.0 * elapsed) / frequency).

Taki sposób mierzenia czasu opiera się na tym, że licznik wydajności procesora działa z bardzo wysoką precyzją, co pozwala na dokładne mierzenie czasu wykonania operacji.

Po każdym powtórzeniu wykonania algorytmu dla danej instancji, program informuje o znalezionej ścieżce oraz jej koszcie. Wyniki zostały opracowane w programie Microsoft Excel.

6 Wyniki

Dla problemu symetrycznego:

Tabela 1: Tabela wyników dla instancji n=14

ilość iteracji	*średni czas t(s)*	*średni błąd(%)*
100	0,0028	11,35
500	0,0082	10,92
1000	0,0149	17,18
2000	0,0288	13,87
3000	0,0425	19,43

Tabela 2: Tabela wyników dla instancji n=21

ilość iteracji	*średni czas t(s)*	*średni błąd(%)*
100	0,0065	24,04
500	0,0233	31,28
1000	0,0461	27,52
2000	0,0866	22,63
3000	0,1286	32,45

Tabela 3: Tabela wyników dla instancji n=24

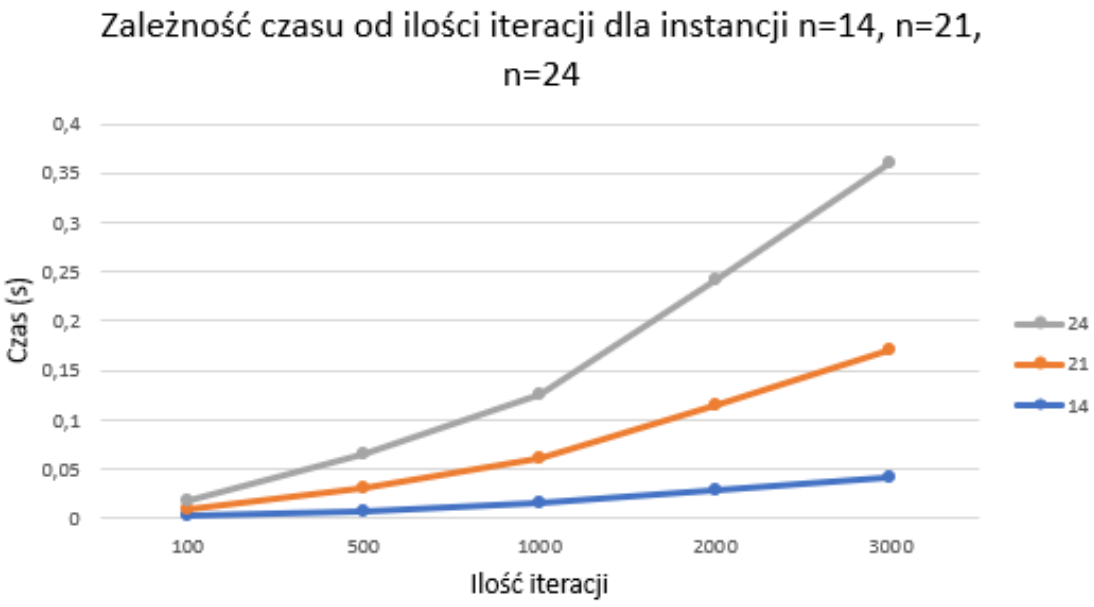
ilość iteracji	*średni czas t(s)*	*średni błąd(%)*
100	0,0093	24,18
500	0,0353	24,18
1000	0,0651	27,07
2000	0,1269	26,89
3000	0,1888	26,67

Tabela 4: Tabela wyników dla instancji n=202

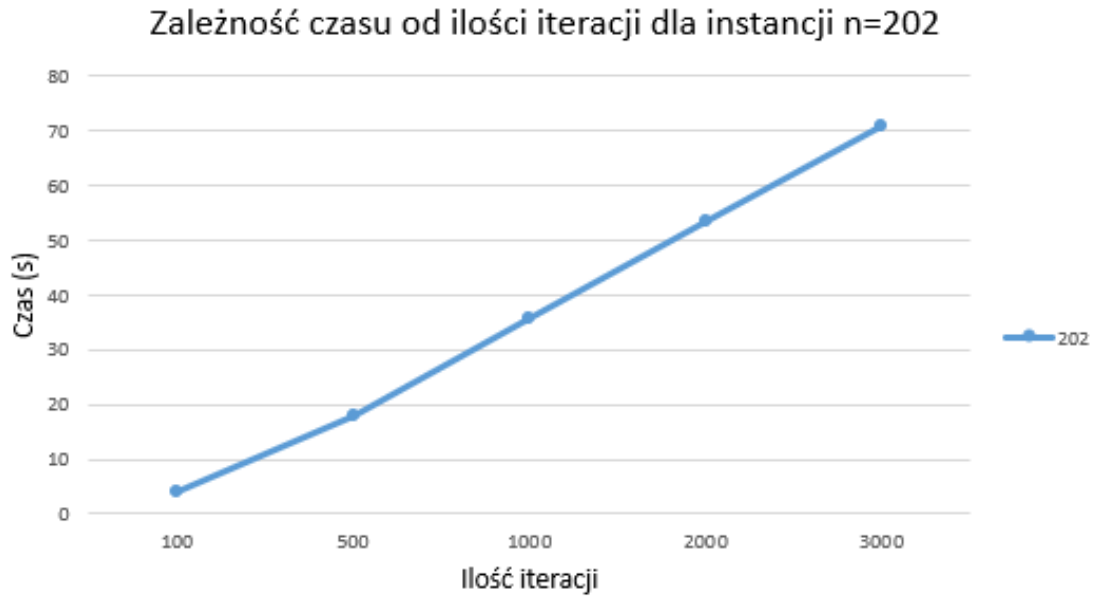
ilość iteracji	*średni czas t(s)*	*średni błąd(%)*
100	3,9553	38,39
500	17,8805	28,29
1000	35,5816	28,17
2000	53,3724	36,03
3000	70,9101	46,34

Tabela 5: Tabela wyników dla instancji n=323

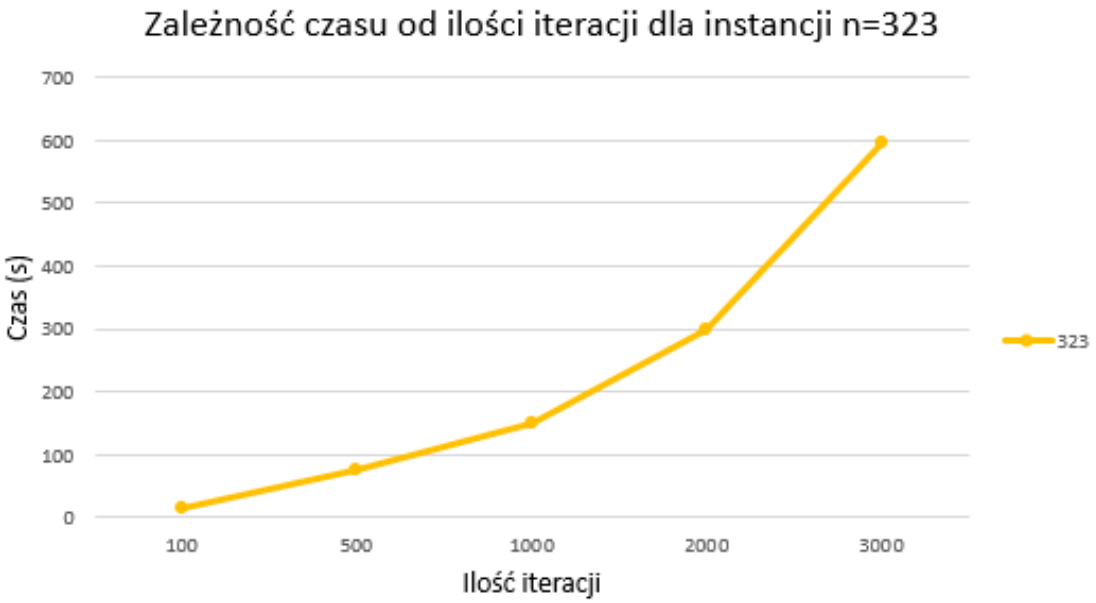
ilość iteracji	*średni czas t(s)*	*średni błąd(%)*
100	15,785	29,41
500	75,296	28,07
1000	150,862	28,27
2000	298,963	29,81
3000	597,921	24,80



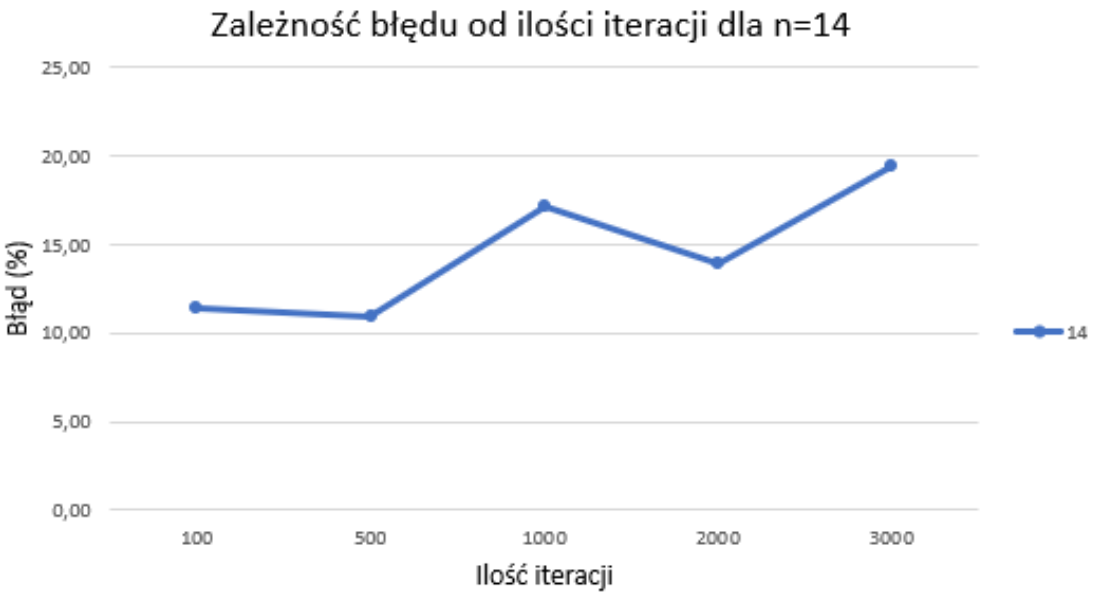
Rysunek 3: Zależność czasu od ilości iteracji dla instancji n=14, n=21, n=24



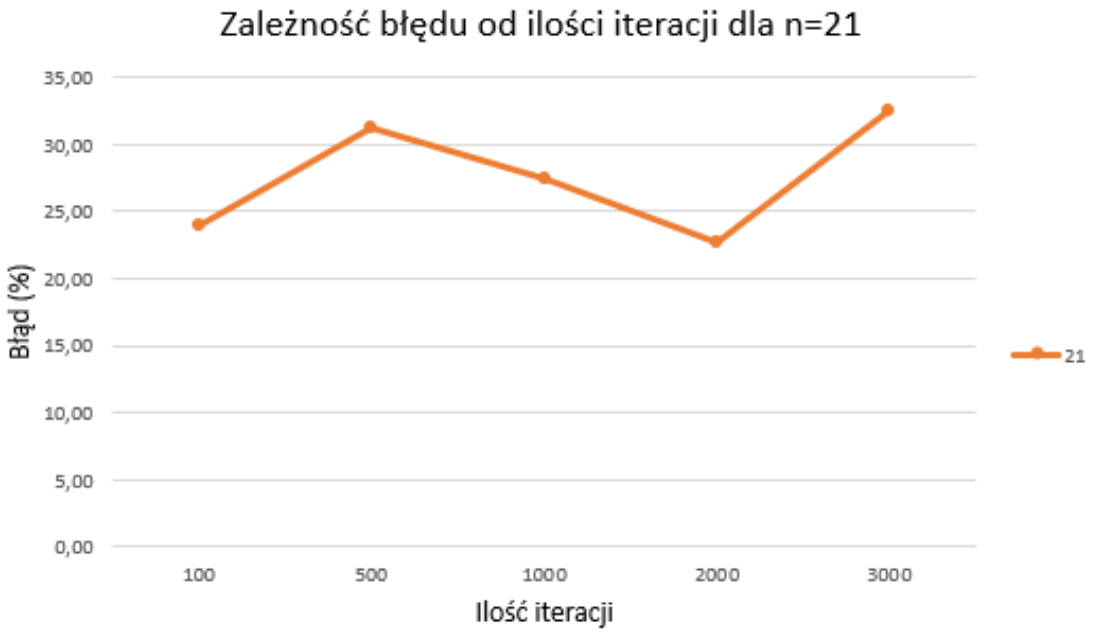
Rysunek 4: Zależność czasu od ilości iteracji dla instancji n=202



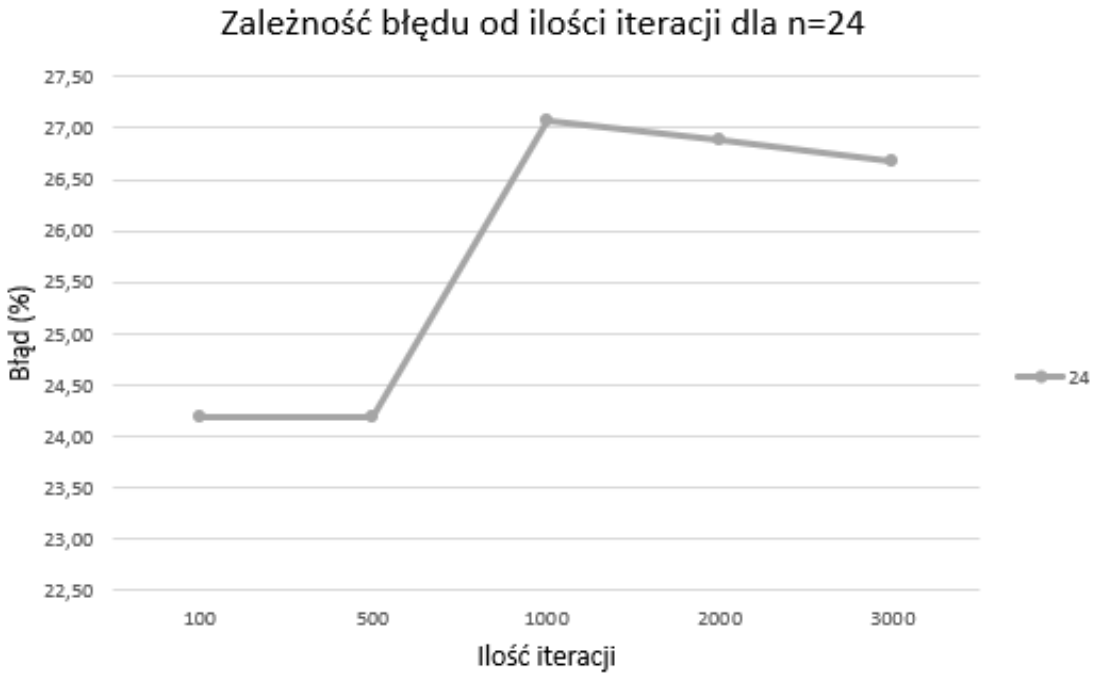
Rysunek 5: Zależność czasu od ilości iteracji dla instancji n=323



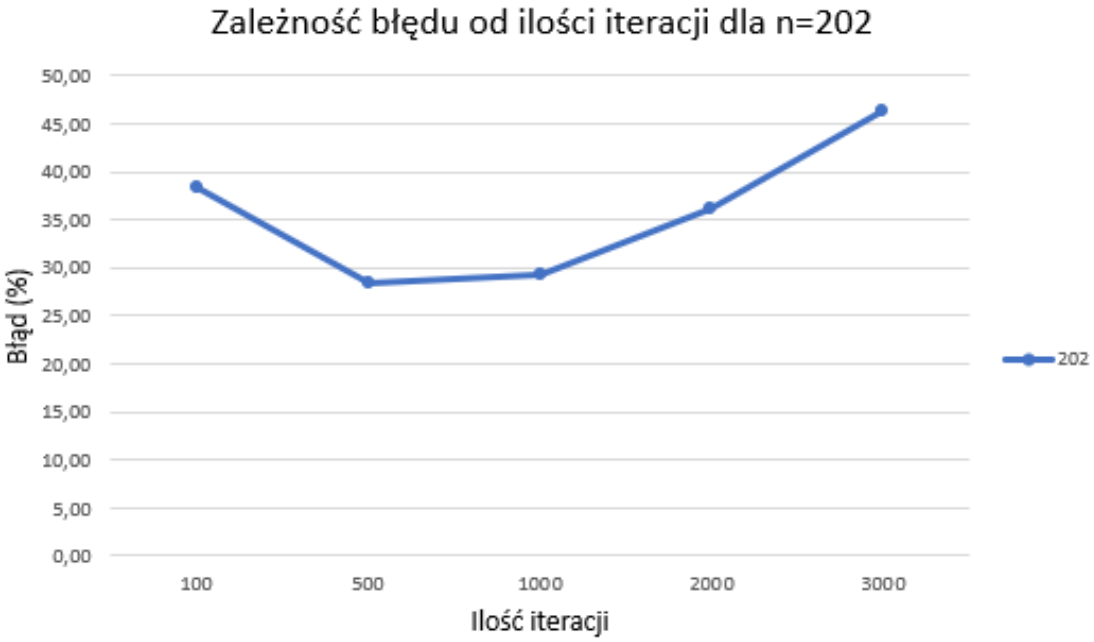
Rysunek 6: Zależność błędu od ilości iteracji dla n=14



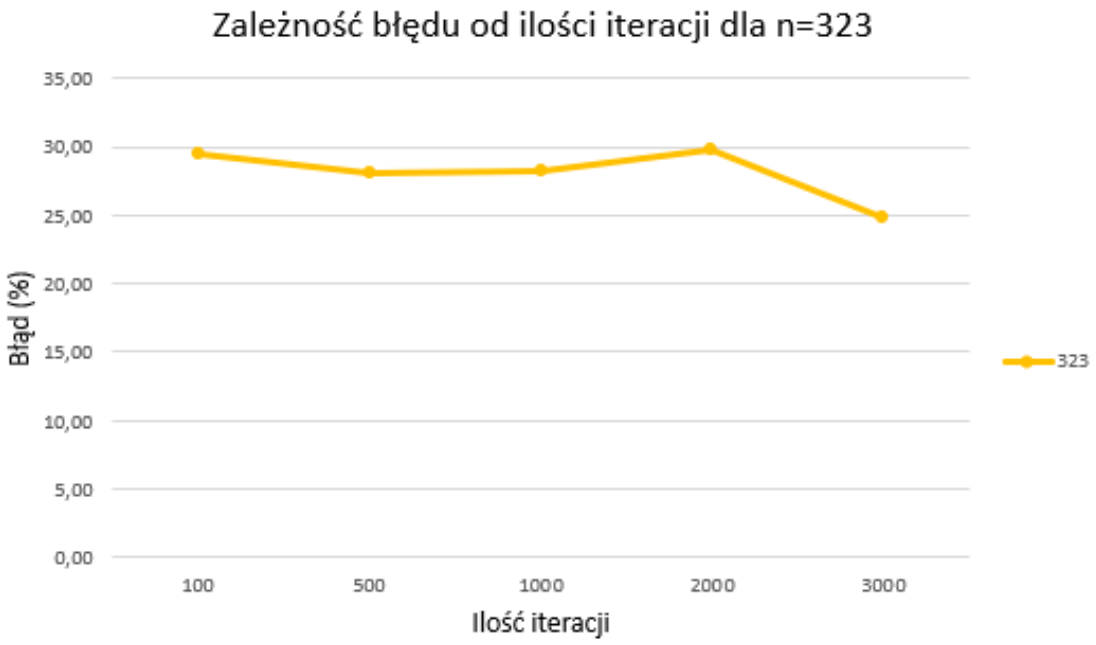
Rysunek 7: Zależność błędu od ilości iteracji dla n=21



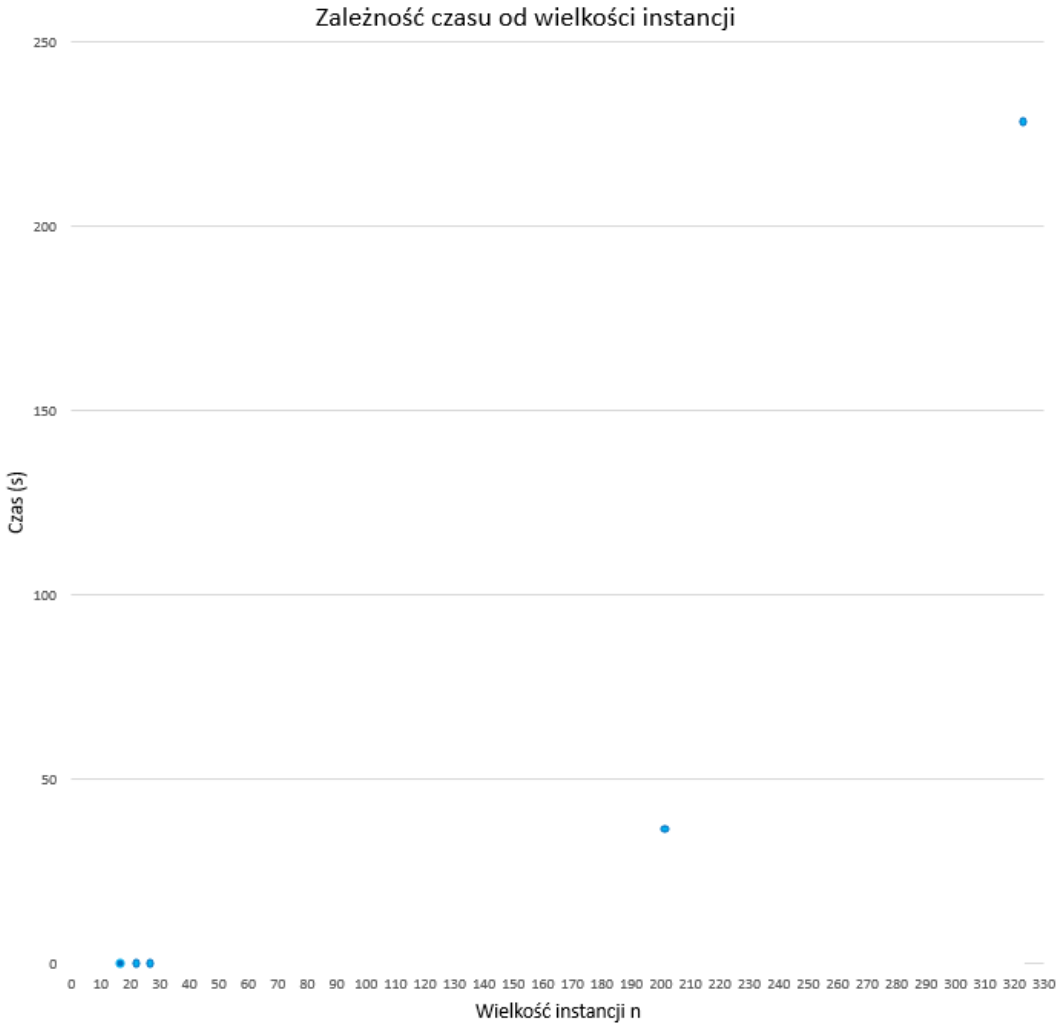
Rysunek 8: Zależność błędu od ilości iteracji dla n=24



Rysunek 9: Zależność błędu od ilości iteracji dla n=202



Rysunek 10: Zależność błędu od ilości iteracji dla n=323



Rysunek 11: Zależność czasu od wielkości instancji

Dla problemu asymetrycznego:

Tabela 6: Tabela wyników dla instancji n=43

ilość iteracji	*średni czas t(s)*	*średni błąd(%)*
100	0,0403	31,68
500	0,1766	26,47
1000	0,3460	22,00
2000	0,6890	31,64
3000	1,0276	37,77

Tabela 7: Tabela wyników dla instancji n=70

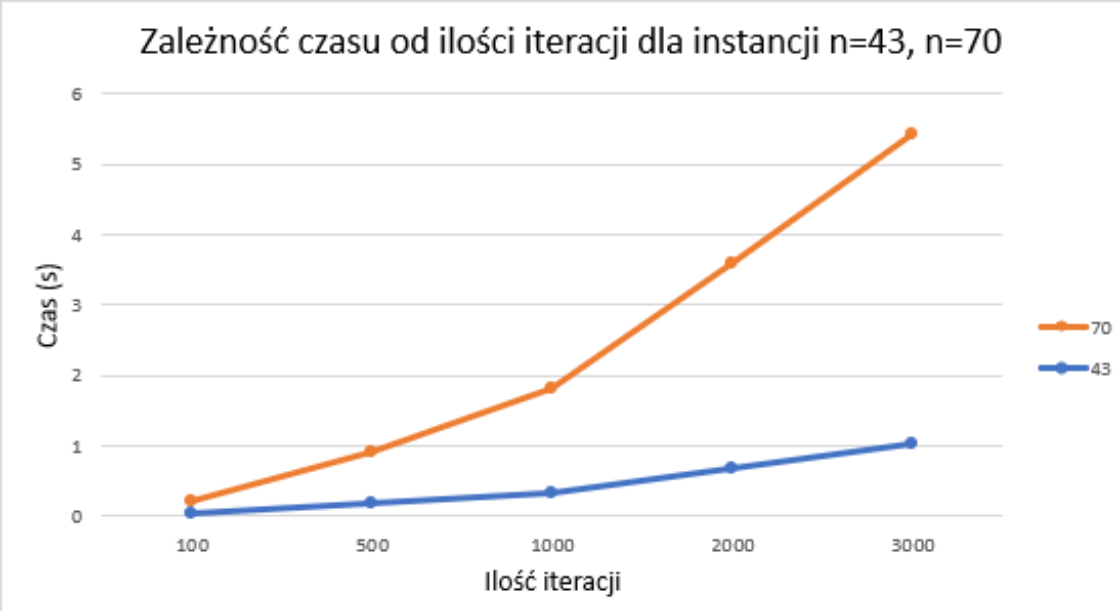
ilość iteracji	*średni czas t(s)*	*średni błąd(%)*
100	0,1624	18,18
500	0,7425	19,81
1000	1,4723	18,63
2000	2,9162	19,46
3000	4,3916	18,62

Tabela 8: Tabela wyników dla instancji n=358

ilość iteracji	*średni czas t(s)*	*średni błąd(%)*
100	21,588	39,84
500	103,415	40,29
1000	204,675	41,74
2000	410,098	39,96
3000	738,176	37,02

Tabela 9: Tabela wyników dla instancji n=443

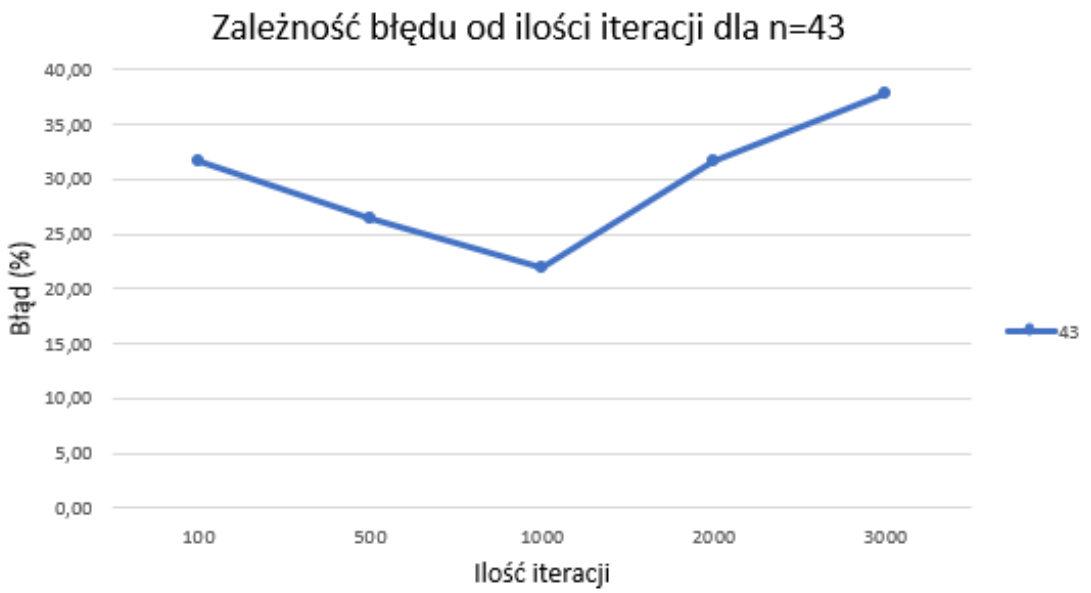
ilość iteracji	*średni czas t(s)*	*średni błąd(%)*
100	72,4011	34,16
500	86,8812	37,83
1000	147,6198	35,93
2000	295,3596	41,51
3000	502,1773	33,42



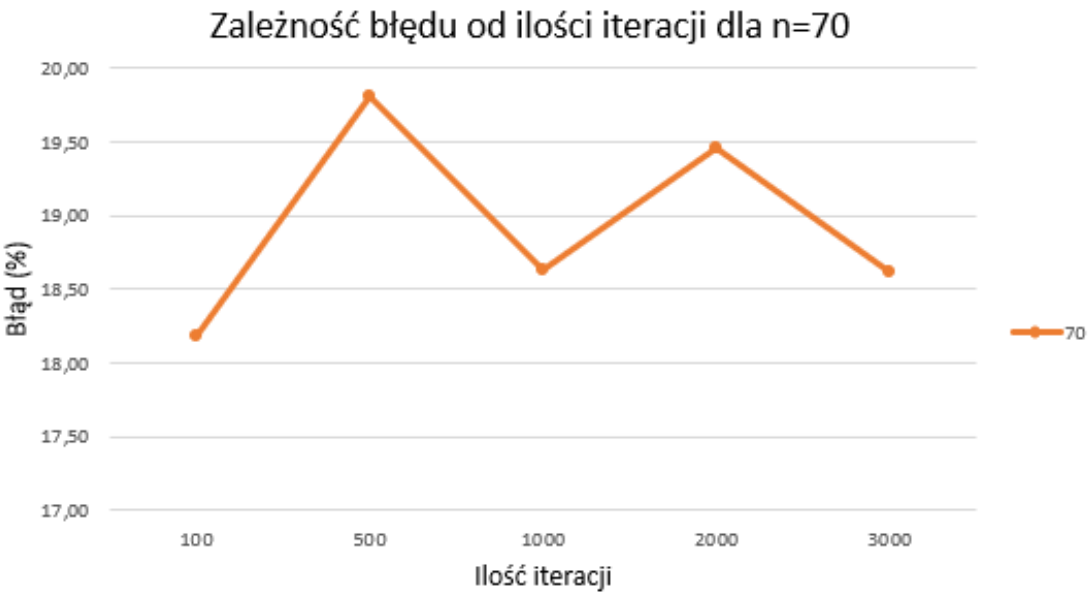
Rysunek 12: Zależność czasu od ilości iteracji dla instancji n=43, n=70

Rysunek 13: Zależność czasu od ilości iteracji dla instancji n=358

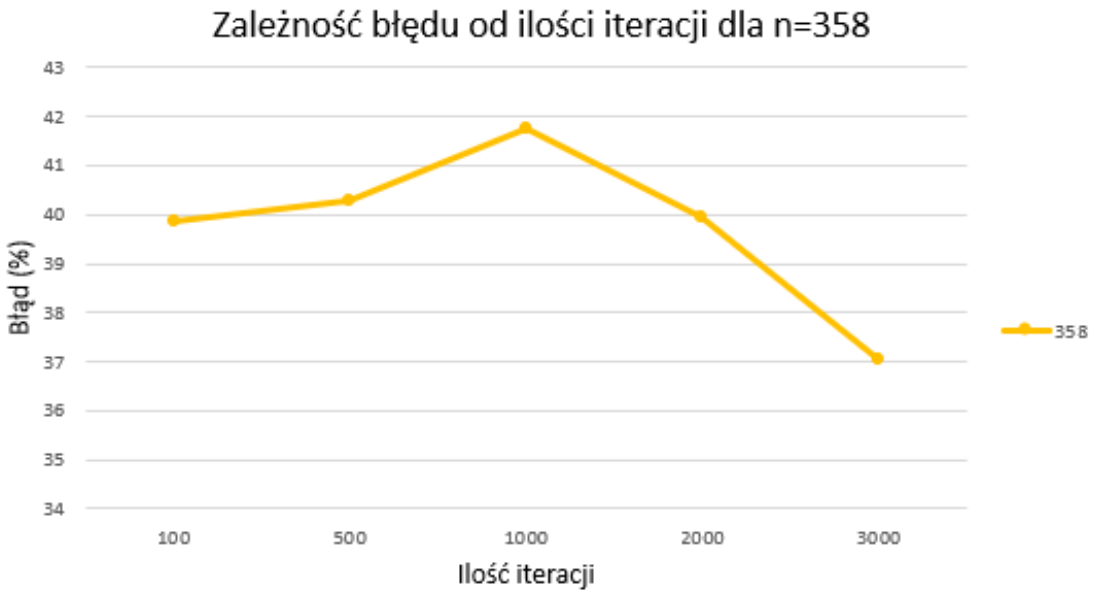
Rysunek 14: Zależność czasu od ilości iteracji dla instancji n=443



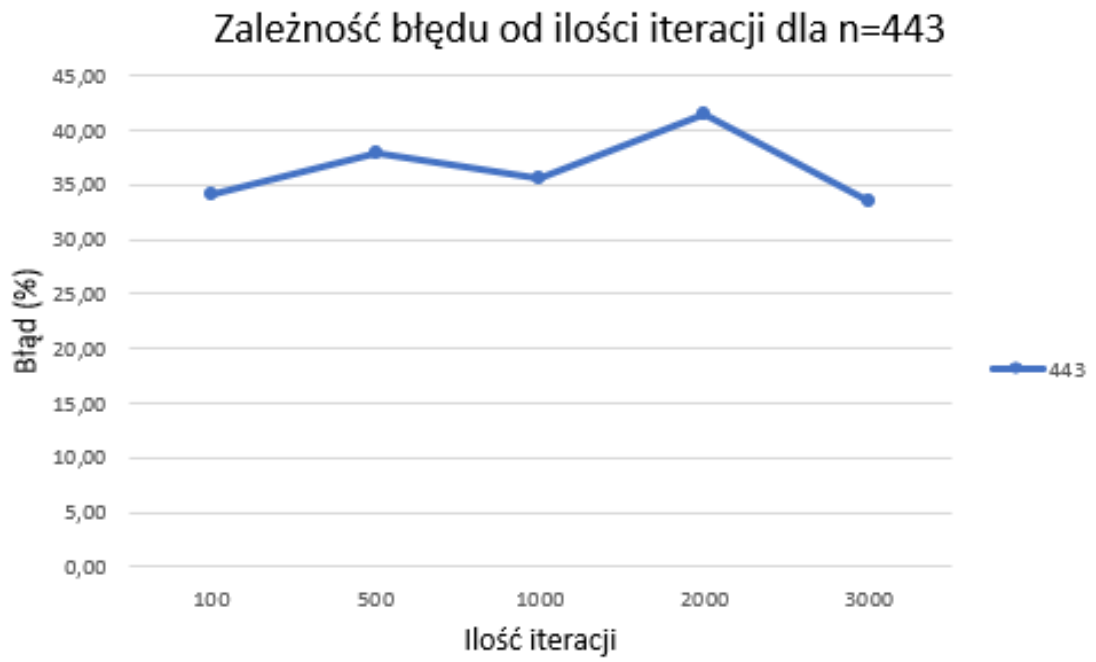
Rysunek 15: Zależność błędu od ilości iteracji dla n=43



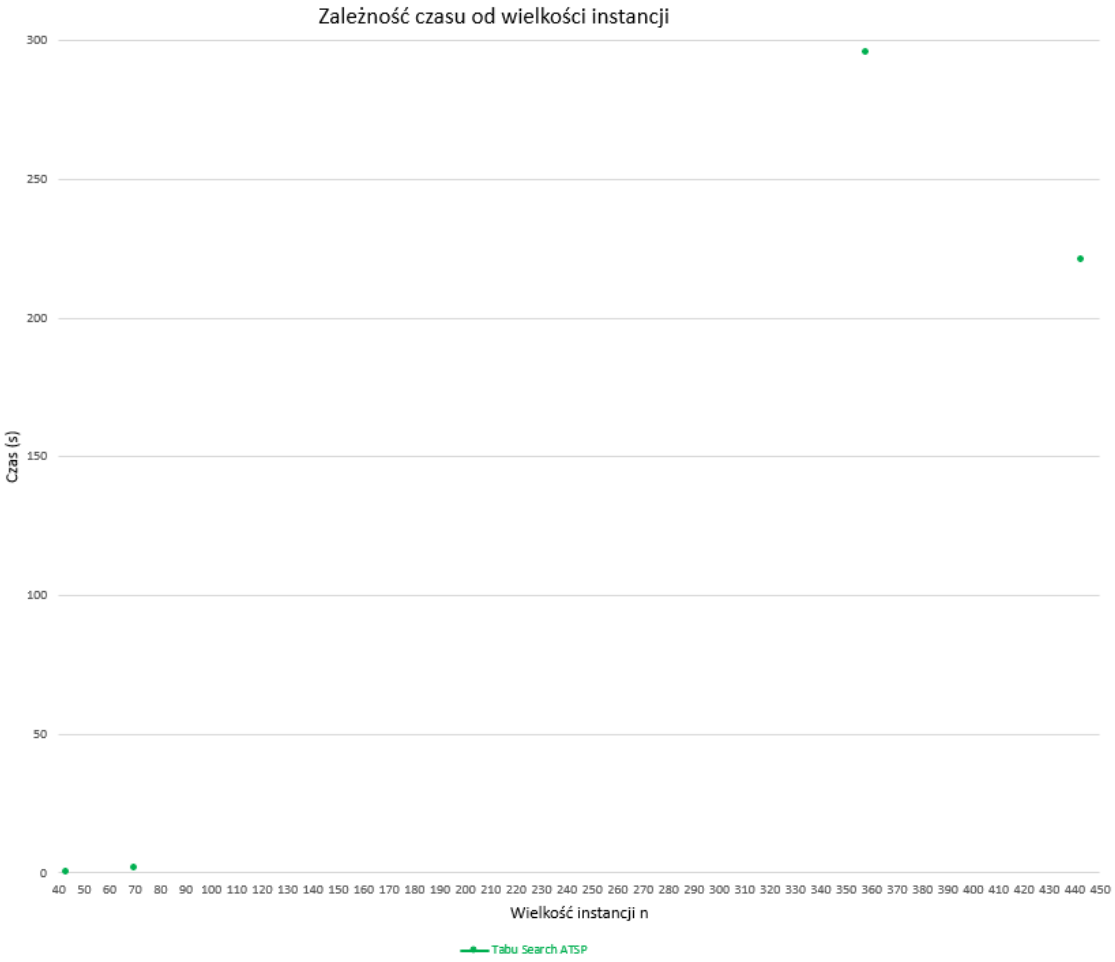
Rysunek 16: Zależność błędu od ilości iteracji dla n=70



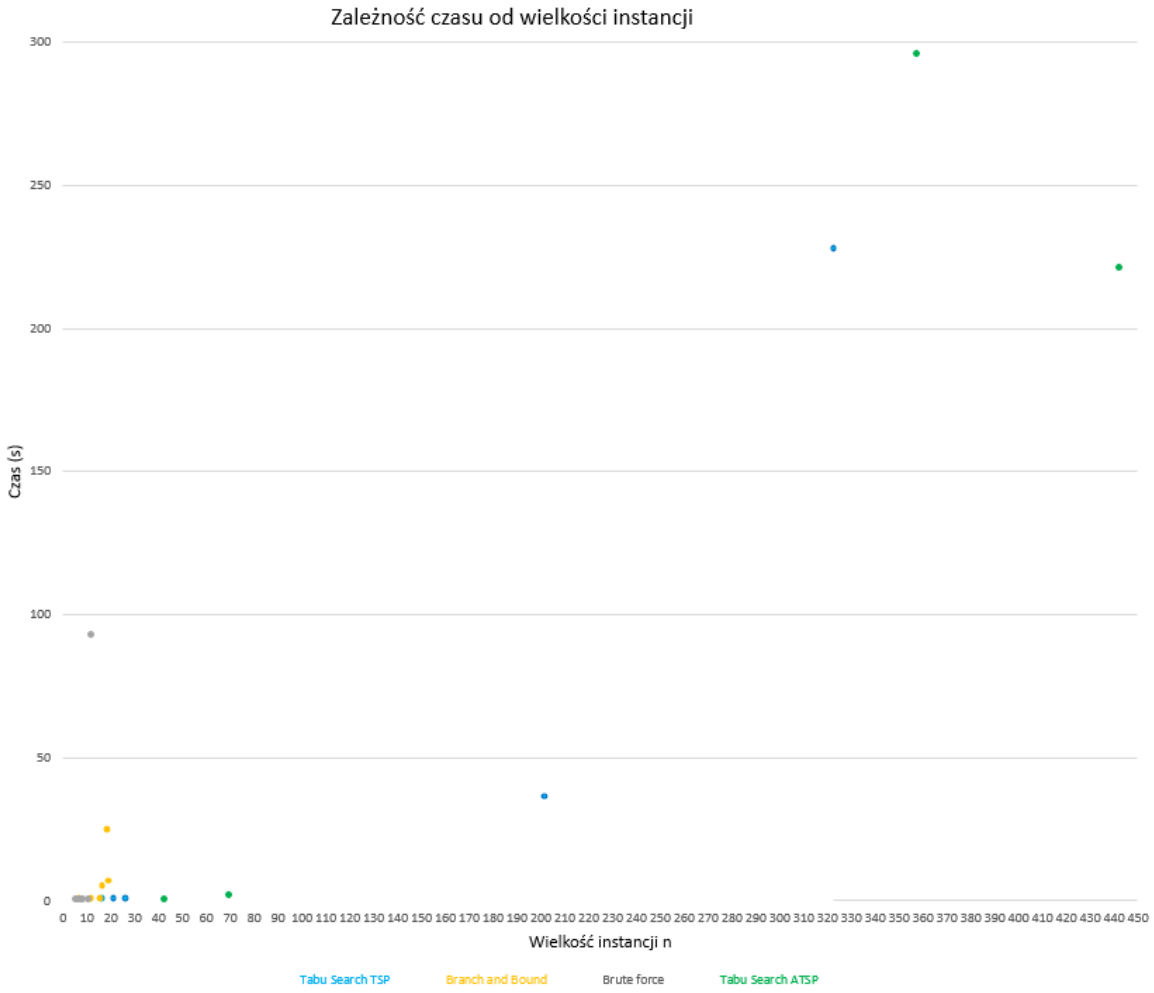
Rysunek 17: Zależność błędu od ilości iteracji dla n=358



Rysunek 18: Zależność błędu od ilości iteracji dla n=443



Rysunek 19: Zależność czasu od wielkości instancji



Rysunek 20: Wpływ wielkości instancji n na czas uzyskania rozwiązania problemu komiwojażera metodą Brute force, Branch and Bound oraz Tabu Search

7 Analiza wyników i wnioski

Badany czas rozwiązania problemu TSP rośnie wraz z wielkością instancji. Dla instancji n=14, 21, 24, 43, 70 są to czasy rzędu maksymalnie kilku sekund, natomiast warto zauważyć, że dla instancji n=323, 358, 443 czas rozwiązania problemu TSP sięga zdecydowanie większych wartości rzędu 800 sekund. Spowodowane jest to faktem, że dla większej liczby miast problem staje się znacząco bardziej skomplikowany.

Algorytm ten w wersji podstawowej charakteryzuje się generowaniem dużego procentu błędu. Dla tej implementacji wszystkie błędy znajdują się w przedziale 10%-41%. Dla poszczególnych instancji wahania błędu nie przekraczają 10%. Dla problemu symetrycznego można zauważyć, że wraz z wielkością instancji rośnie także procent błędu. Dla problemu asymetrycznego odbiega od tej reguły instancja n=70, dla której błąd jest mniejszy niż dla instancji n=43. Może to być spowodowane poziomem komplikacji instancji. Instancje reagują bardzo niestandardowo na zmiany ilości iteracji - nie da się jednoznacznie określić zależności ilości iteracji od generowanego błędu. Można zauważyć, że instancje osiągają mniejszy procent błędu w okolicach pewnej ilości iteracji, jednak jest to zróżnicowane w zależności od instancji. Może to wskazywać na to, że dobrane parametry algorytmu, które były odpowiednie dla niektórych przypadków, w innych przypadkach nie sprawdzały się równie dobrze. Przykładowo instancja n=14 generuje najmniejszy procent błędu dla ilości iteracji 100 i 500, natomiast instancja n=21 dla ilości iteracji 2000. Dla instancji n=323 i n=70 ilość iteracji nie wpływała znacząco na procent błędu - dla n=70 błąd waha się w zakresie 2%.

Można zauważyć, że dla problemu symetrycznego badany czas był mniejszy niż dla problemu asymetrycznego. Dla wybranych do badań instancji również generowany procent błędu rozwiązania jest mniejszy w przypadku instancji symetrycznych. Nie można jednak porównać dokładnie zbadanego problemu symetrycznego z asymetrycznym, ponieważ dla każdego z nich badane były inne zestawy danych.

Dalsza optymalizacja parametrów algorytmu może pomóc w osiągnięciu lepszych wyników dla różnych instancji problemu. Eksperymenty z różnymi kombinacjami parametrów, takimi jak długość listy Tabu, kryteria aspiracji czy dywersyfikacja, najprawdopodobniej pozwoliłyby na zmiejszenie generowanego procentu błędu. W celu bardzo dokładnego zweryfikowania skuteczności algorytmu można także przeprowadzić ewaluację na różnorodnych przypadkach zestawów danych.

Algorytm Tabu Search dla problemu komiwojażera osiąga zdecydowanie mniejsze czasu wykonania niż algorytmy Brute Force i Branch and Bound. Możliwe jest również badanie zdecydowanie większych instancji niż dla wspomnianych algorytmów. Algorytm Brute Force już dla n=13 osiągał wartości większe niż Tabu Search dla n=358. Podobnie w przypadku Branch and Bound - czas dla n=20 jest większy niż w przypadku Tabu Search dla n=70. Porównując więc te trzy algorytmy, zdecydowanie najlepiej pod względem czasu wykonania prezentuje się Tabu Search. Pozwala również na operowanie na zdecydowanie większych instancjach niż Brute force i Branch and Bound. Dużą

jego wadą jest jednak duży procent generowanego błędu. W celu osiągania małych czasów obliczeń, operowania na dużych instancjach i otrzymania rozwiązania z małym procentem błędu, konieczne może być rozważenie alternatywnych strategii rozwiązania problemu TSP.

Źródła:

[1] https://pl.wikipedia.org/wiki/Graf_pe%C5%82ny
[2] https://pl.wikipedia.org/wiki/Cykl_Hamiltona
[3] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Spis rysunków

Spis tabel