2021

Projektowanie Efektywnych Algorytmów

PROJEKT 2

KAROL WAŻNY (252716)

1. Polecenie

W ramach projektu należało zaimplementować oraz dokonać analizy efektywności algorytmu Tabu Search (TS) oraz Symulowanego Wyżarzania (SA) dla asymetrycznego problemu komiwojażera (ATSP).

1.1. Dodatkowe wymagania

- używane struktury danych powinny być alokowane dynamicznie (w zależności od aktualnego rozmiaru problemu),
- program powinien umożliwić weryfikację poprawności działania algorytmu. W tym celu powinna istnieć możliwość wczytania danych wejściowych z pliku tekstowego,
- po zaimplementowaniu i sprawdzeniu poprawności działania algorytmu należy dokonać pomiaru czasu jego działania w zależności od rozmiaru problemu N (badania należy wykonać dla minimum 7 różnych reprezentatywnych wartości N),
- dla każdej wartości N należy wygenerować po 100 losowych instancji problemu (w sprawozdaniu należy umieścić tylko wyniki uśrednione),
- implementacje algorytmów należy dokonać zgodnie z obiektowym paradygmatem programowania.

2. Wstęp teoretyczny

Asymetryczny problem komiwojażera (ATSP – Asymmetric Travelling Salesman Problem) zdefiniowany jest jako zadanie znalezienia dla danego skierowanego grafu ważonego minimalnego cyklu przechodzącego przez wszystkie wierzchołki (minimalnego cyklu Hamiltona).

ATSP należy do klasy problemów NP.-trudnych, co oznacza, że nie ma algorytmu, który daje gwarancję obliczenia wyniku optymalnego w czasie wielomianowym.

Prezentowany program został wzbogacony o implementacje dwóch algorytmów opartych o strategię poszukiwania lokalnego: Tabu Search i symulowane wyżarzanie.

2.1. Poszukiwanie lokalne

Są dwa pojęcia istotne dla zrozumienia strategii poszukiwania lokalnego: sąsiedztwo rozwiązania i ruch.

Ruch to operacja (lub operacje) wykonywane w celu przejścia z jednego rozwiązania dopuszczalnego do następnego. Najczęściej na ruch nałożone są pewne ograniczenia, aby nie można było w jednym ruchu przejść z dowolnego punktu przestrzeni rozwiązań do dowolnego innego. Ruchem może być zamiana dwóch dowolnych wierzchołków w ścieżce albo zamiana dwóch sąsiadujących wierzchołków w ścieżce.

Sąsiedztwo rozwiązania to zbiór rozwiązań, do których z danego rozwiązania można przejść w jednym ruchu.

Szukanie rozwiązania optymalnego przy użyciu poszukiwań lokalnych polega na rozpoczęciu z dowolnego (losowego) punktu przestrzeni rozwiązań i wykonywaniu ruchów w celu przejścia do kolejnych rozwiązań. Największa trudność przy poszukiwaniu lokalnym polega na skonstruowaniu odpowiednio efektywnej metody wyboru wierzchołka z sąsiedztwa, do którego algorytm przejdzie w kolejnym ruchu.

Co istotne, algorytmy poszukiwania lokalnego wykonują się dużo szybciej niż algorytmy dokładne (opisane w Sprawozdaniu 1.), jednak nie dają gwarancji rozwiązania optymalnego – algorytmy te zwracają rozwiązania wystarczająco dobre. Jest to korzystne dla dużych instancji problemu, gdzie koszt uzyskania rozwiązania metodami dokładnymi byłby większy niż wartość rozwiązania.

W prezentowanych przykładach dopuszczalne rozwiązania to po prostu cykle Hamiltona w rozważanym grafie.

Za ruch uznajemy zamianę miejscami dwóch dowolnych wierzchołków w ścieżce:

$$0 -> 1 -> 2 -> 3 -> 4 -> 0$$

I po wykonaniu ruchu:

$$0 -> 3 -> 2 -> 1 -> 4 -> 0$$

Zatem sąsiedztwo to zbiór rozwiązań, które z danego rozwiązania można uzyskać zamieniając miejscami dwa wierzchołki.

Koszt rozwiązania to oczywiście koszt ścieżki, a celem jest znalezienie ścieżki o minimalnym koszcie.

2.2. Symulowane wyżarzanie (SA)

2.2.1. Opis

W symulowanym wyżarzaniu kolejne rozwiązanie jest wybierane losowo z sąsiedztwa. Jeżeli nowe rozwiązanie jest lepsze od aktualnego, jest przyjmowane (algorytm wykonuje ruch do tego rozwiązania).

Jeżeli nowe rozwiązanie jest gorsze od aktualnego, jest przyjmowane z pewnym prawdopodobieństwem zależnym od różnicy kosztów aktualnego i nowego oraz od dodatkowego parametru nazywanego **temperaturą**.

Prawdopodobieństwo przyjęcia nowego rozwiązania opisane jest następującym równaniem:

$$P = \exp\left(-\frac{\Delta C}{T}\right)$$

Gdzie:

- P to prawdopodobieństwo przyjęcia gorszego rozwiązania,
- ΔC to różnica kosztów (przyjmuje tylko wartości dodatnie),
- T to wartość temperatury.

Z równania wynika, że wyższa temperatura sprzyja przyjmowaniu rozwiązań gorszych, a nawet dużo gorszych (w skrajnym przypadku zwiedzanie przestrzeni przyjmuje formę błądzenia losowego.

Z drugiej strony temperatura niższa sprzyja odrzucaniu rozwiązań gorszych (w skrajnym przypadku zwiedzanie przestrzeni przyjmuje postać algorytmu zachłannego i staje się bardzo podatne na utknięcie w minimum lokalnym).

Najczęściej algorytm zaczyna od temperatury wysokiej, a następnie w trakcie wykonywania obniża ją.

W prezentowanej wersji algorytm wykonuje k prób (restartów). W ramach restartu algorytm startuje z losowego rozwiązania i zadanej temperatury startowej po czym zmniejsza temperaturę co rundę, mnożąc ją przez zadaną stałą. Algorytm wykonuje się do momentu osiągnięcia zadanej temperatury minimalnej, po czym następuje restart.

W ramach jednej rundy (pomiędzy kolejnymi zmniejszeniami temperatury) algorytm wykonuje m ruchów (losowanie i przyjęcie bądź odrzucenie rozwiązania).

Algorytm przyjmuje zatem 5 parametrów: temperatura początkowa, temperatura końcowa, stała zmniejszania temperatury, liczba restartów i liczba ruchów pomiędzy zmniejszeniami temperatury. Największa trudność polega na wyborze odpowiednich wartości parametrów.

2.2.2. Złożoność obliczeniowa

Algorytm na początku musi obliczyć koszt pierwszego rozwiązania – odbywa się to w czasie liniowym. Następnie wykonuje się w czasie zależnym tylko od parametrów. Zatem oczekujemy zależności liniowej od rozmiaru instancji dla stałych parametrów.

2.2.3. Szczegóły implementacji

Algorytm w logice programu reprezentuje klasa `TSPSimulatedAnnealingSolver`. Klasa ta posiada metodę publiczną `solveFor()` przyjmującą jako parametr obiekt klasy `TSPInputMatrix` i zwracającą obiekt klasy `TSPSolution`.

Rozwiązania w trakcie wykonywania algorytmu przechowywane są jako tablica zawierająca indeksy wierzchołków. Tablica nie zawiera wierzchołka o indeksie 0 (to znaczy początkowego i końcowego).

Kolejność wierzchołków w tablicy odpowiada kolejności w ścieżce. Taka reprezentacja pozwala dokonać zamiany dwóch dowolnych wierzchołków w czasie $O(1)$.

W każdej chwili przechowywane jest rozwiązanie aktualne i rozwiązanie najlepsze znalezione do tej pory. Rozwiązanie najlepsze do tej pory jest zwracane po zakończeniu działania algorytmu jako rozwiązanie ostateczne.

Koszt kolejnego rozwiązania jest obliczany jako koszt aktualnego rozwiązania zmniejszony o wartość krawędzi, które znikają po wykonaniu ruchu i powiększony o wartość krawędzi, które się pojawiają (metoda `TSPStochasticSolver::costAfterSwapping`). Uwzględniono przy tym szczególny przypadek, gdy zamieniane wierzchołki znajdują się obok siebie w ścieżce. Takie podejście pozwala zaoszczędzić dużo czasu na obliczaniu i wyznaczać koszty w czasie niezależnym od rozmiaru instancji.

Do wyznaczania wartości pseudolosowych używane są klasy `RealRandom<double>` (do liczb zmiennoprzecinkowych) oraz `Randomizer` (do liczb całkowitych). Obie klasy są napisanymi na potrzeby projektu wrapperami na natywne generatory pseudolosowe C++. `Randomizer` jest wykorzystywany do wybierania kolejnego rozwiązania z sąsiedztwa oraz do wyznaczenia rozwiązań startowych. `RealRandom` jest wykorzystywany do losowego rozpatrywania przyjęcia bądź odrzucenia rozwiązania gorszego.

Parametry są reprezentowane jako obiekt klasy wewnętrznej `Parameters`.

Do wyznaczenia losowego rozwiązania początkowego używany jest algorytm tasowania Fishera-Yatesa.

2.3. Tabu search (TS)

2.3.1. Opis

W tym algorytmie wybór kolejnego rozwiązania polega na przejrzeniu sąsiedztwa i wybraniu rozwiązania o najmniejszym koszcie (strategia zachłanna). Jednak za każdym razem, gdy wykonywany jest ruch, zapamiętywany jest sam ruch i moment jego wykonania (ruch jest dodawany do listy Tabu). W kolejnych iteracjach algorytmu nie wolno wykonać ruchu znajdującego się na liście Tabu. Czas (liczba iteracji), przez jaki ruch jest Tabu jest parametrem algorytmu.

Pozostałe parametry algorytmu to całkowita liczba iteracji i liczba restartów.

2.3.1.1. Kryterium aspiracji

Jeżeli dany ruch jest objęty tabu, ale prowadziłby do rozwiązania lepszego niż dowolne znalezione do tej pory, można go wykonać (trochę w myśl zasady, że jak nie wolno, ale się bardzo chce to wolno).

2.3.2. Złożoność obliczeniowa

Czas wykonania algorytmu zależy od parametrów (głównie zadanej liczby iteracji oraz liczby restartów).

Jednak dla stałych parametrów należy obliczyć koszt rozwiązania początkowego (złożoność liniowa), a następnie w każdej iteracji przejrzeć całe sąsiedztwo, żeby wyznaczyć kolejny ruch (złożoność kwadratowa). Wprawdzie dla każdego z przeglądanych rozwiązań z sąsiedztwa należy wyznaczyć jego koszt, ale ze względu na zastosowaną implementację dzieje się to w czasie $O(1)$ (dokładniej opisane w podpunkcie „Szczegóły implementacji”).

2.3.3. Przykład praktyczny

Rozważmy poniższą macierz incydencji (wierzchołki są numerowane od 0):

—	81	50	18	75
81	—	76	21	37
50	76	—	24	14
18	21	24	—	19
75	37	14	19	—

Przyjmijmy, że parametry algorytmu to:

- Długość listy tabu: 4
- Liczba iteracji: 10
- Liczba startów: 1

Aby rozwiązać problem komiwojażera dla grafu reprezentowanego przez tę macierz, zaczniemy od losowej permutacji wierzchołków:

$$0 -> 3 -> 2 -> 4 -> 1 -> 0$$

Zatem tablica zawierająca aktualne rozwiązanie wygląda następująco:

$$[3, 2, 4, 1]$$

Koszt tego rozwiązania to:

$$Cost = 174$$

Początkowa macierz tabu wygląda następująco:

-	-5	-5	-5
-	-	-5	-5
-	-	-	-5
-	-	-	-

Przebieg algorytmu:

2.3.3.1. Iteracja 0:

Koszty po wykonaniu ruchu:

WIERZCHOŁEK	**3**	**2**	**4**	**1**
3	-	211	215	208
2	-	-	208	140
4	-	-	-	230
1	-	-	-	-

Najlepszy jest ruch polegający na zamianie wierzchołków 1 i 2. Dodatkowo nie jest to ruch objęty tabu i jest to ruch prowadzący do rozwiązania lepszego niż najlepsze dotychczas znalezione. Zatem aktualizujemy najlepsze dotychczas rozwiązanie i aktualne rozwiązanie:

$$Cost, BestCost = 140$$

$$Current, Best = [3, 1, 4, 2]$$

Lista tabu:

WIERZCHOŁEK	3	1	4	2
3	-	-5	-5	-5
1	-	-	-5	0
4	-	-	-	-5
2	-	-	-	-

2.3.3.2. Iteracja 1:

Przeglądanie sąsiedztwa:

WIERZCHOŁEK	3	1	4	2
3	-	185	207	200
1	-	-	200	174
4	-	-	-	204
2	-	-	-	-

Najkorzystniejsza jest zamiana wierzchołków 1 i 2, ale ten ruch jest objęty tabu. Dodatkowo, nie jest spełnione kryterium aspiracji (bo nowe rozwiązanie nie jest lepsze niż najlepsze znalezione dotychczas). Dlatego wykonujemy najkorzystniejszy ruch nieobjęty tabu: zamiana wierzchołków 1 i 3.

Stan aktualny:

$$Cost = 185$$

$$Current = [1, 3, 4, 2]$$

Lista tabu:

WIERZCHOŁEK	1	3	4	2
1	-	1	-5	-5
3	-	-	-5	0
4	-	-	-	-5
2	-	-	-	-

2.3.3.3. Iteracja 2:

Przeglądanie sąsiedztwa:

WIERZCHOŁEK	1	3	4	2
1	-	140	241	211
3	-	-	211	208
4	-	-	-	215
2	-	-	-	-

Najkorzystniejsza jest zamiana wierzchołków 1 i 3, ale ten ruch jest objęty tabu. Dodatkowo, nie jest spełnione kryterium aspiracji (bo nowe rozwiązanie nie jest lepsze niż najlepsze znalezione dotychczas). Dlatego wykonujemy najkorzystniejszy ruch nieobjęty tabu: zamiana wierzchołków 1 i 2.

Stan aktualny:

$$Cost = 211$$

$$Current = [2, 3, 4, 1]$$

Lista tabu:

WIERZCHOŁEK	2	3	4	1
2	-	1	-5	2
3	-	-	-5	0
4	-	-	-	-5
1	-	-	-	-

2.3.3.4. Iteracja 3:

Przeglądanie sąsiedztwa:

WIERZCHOŁEK	2	3	4	1
2	-	174	275	185
3	-	-	185	200
4	-	-	-	207
1	-	-	-	-

Najkorzystniejsza jest zamiana wierzchołków 2 i 3, ale ten ruch jest objęty tabu. Dodatkowo, nie jest spełnione kryterium aspiracji (bo nowe rozwiązanie nie jest lepsze niż najlepsze znalezione dotychczas). Dlatego wykonujemy najkorzystniejszy ruch nieobjęty tabu: zamiana wierzchołków 3 i 4.

Stan aktualny:

$$Cost = 185$$

$$Current = [2, 4, 3, 1]$$

Lista tabu:

WIERZCHOŁEK	2	4	3	1
2	-	1	-5	2
4	-	-	3	0
3	-	-	-	-5
1	-	-	-	-

2.3.3.5. Iteracja 4:

Przeglądanie sąsiedztwa:

WIERZCHOŁEK	2	4	3	1
2	-	215	208	211
4	-	-	211	241
3	-	-	-	140
1	-	-	-	-

Najkorzystniejsza jest zamiana wierzchołków 1 i 3. Jest to ruch nieobjęty tabu, więc go wykonujemy. Nie aktualizujemy jednak najlepszego znalezionego do tej pory rozwiązania.

Stan aktualny:

$$Cost = 140$$

$$Current = [2, 4, 1, 3]$$

Lista tabu:

WIERZCHOŁEK	2	4	1	3
2	-	1	-5	2
4	-	-	3	0
1	-	-	-	4
3	-	-	-	-

2.3.3.6. Iteracja 5:

Przeglądanie sąsiedztwa:

WIERZCHOŁEK	2	4	1	3
2	-	204	174	200
4	-	-	200	207
1	-	-	-	185
3	-	-	-	-

Najkorzystniejsza jest zamiana wierzchołków 1 i 2. Jest to ruch nieobjęty tabu, więc go wykonujemy. Nie aktualizujemy jednak najlepszego znalezionego do tej pory rozwiązania.

Stan aktualny:

$$Cost = 174$$

$$Current = [1, 4, 2, 3]$$

Lista tabu:

WIERZCHOŁEK	1	4	2	3
1	-	1	5	2
4	-	-	3	0
2	-	-	-	4
3	-	-	-	-

2.3.3.7. Iteracja 6:

Przeglądanie sąsiedztwa:

WIERZCHOŁEK	1	4	2	3
1	-	230	140	208
4	-	-	208	215
2	-	-	-	211
3	-	-	-	-

Najkorzystniejsza jest zamiana wierzchołków 1 i 2, ale ten ruch jest objęty tabu. Dodatkowo, nie jest spełnione kryterium aspiracji (bo nowe rozwiązanie nie jest lepsze niż najlepsze znalezione dotychczas). Dlatego wykonujemy najkorzystniejszy ruch nieobjęty tabu: zamiana wierzchołków 3 i 4.

Stan aktualny:

$$Cost = 215$$

$$Current = [1, 3, 2, 4]$$

Lista tabu:

WIERZCHOŁEK	1	3	2	4
1	-	1	5	2
3	-	-	3	6
2	-	-	-	4
4	-	-	-	-

2.3.3.8. Iteracja 7:

Przeglądanie sąsiedztwa:

WIERZCHOŁEK	1	3	2	4
1	-	204	207	275
3	-	-	275	174
2	-	-	-	185
4	-	-	-	-

Najkorzystniejsza jest zamiana wierzchołków 3 i 4, ale ten ruch jest objęty tabu. Dodatkowo, nie jest spełnione kryterium aspiracji (bo nowe rozwiązanie nie jest lepsze niż najlepsze znalezione dotychczas). Dlatego wykonujemy najkorzystniejszy ruch nieobjęty tabu: zamiana wierzchołków 1 i 3.

Stan aktualny:

$$Cost = 204$$

$$Current = [3, 1, 2, 4]$$

Lista tabu:

WIERZCHOŁEK	3	1	2	4
3	-	7	5	2
1	-	-	3	6
2	-	-	-	4
4	-	-	-	-

2.3.3.9. Iteracja 8:

Przeglądanie sąsiedztwa:

WIERZCHOŁEK	3	1	2	4
3	-	215	241	230
1	-	-	230	208
2	-	-	-	140
4	-	-	-	-

Najkorzystniejsza jest zamiana wierzchołków 2 i 4, ale ten ruch jest objęty tabu. Dodatkowo, nie jest spełnione kryterium aspiracji (bo nowe rozwiązanie nie jest lepsze niż najlepsze znalezione dotychczas). Dlatego wykonujemy najkorzystniejszy ruch nieobjęty tabu: zamiana wierzchołków 3 i 4.

Stan aktualny:

$$Cost = 230$$

$$Current = [4, 1, 2, 3]$$

Lista tabu:

WIERZCHOŁEK	4	1	2	3
4	-	7	5	2
1	-	-	3	6
2	-	-	-	4
3	-	-	-	-

2.3.3.10. Iteracja 9:

Przeglądanie sąsiedztwa:

WIERZCHOŁEK	4	1	2	3
4	-	174	200	204
1	-	-	204	275
2	-	-	-	207
3	-	-	-	-

Najkorzystniejsza jest zamiana wierzchołków 1 i 4, ale ten ruch jest objęty tabu. Dodatkowo, nie jest spełnione kryterium aspiracji (bo nowe rozwiązanie nie jest lepsze niż najlepsze znalezione dotychczas). Dlatego wykonujemy najkorzystniejszy ruch nieobjęty tabu: zamiana wierzchołków 1 i 2.

Stan aktualny:

$$Cost = 204$$

$$Current = [4, 2, 1, 3]$$

Lista tabu:

WIERZCHOŁEK	4	2	1	3
4	-	7	5	2
2	-	-	3	6
1	-	-	-	4
3	-	-	-	-

Po dziesiątej iteracji ostatniego startu (w tym przypadku pierwszego) algorytm zwraca najlepsze znalezione dotąd rozwiązanie:

$$0 -> 3 -> 1 -> 4 -> 2 -> 0$$

I jego koszt:

$$Cost = 140$$

2.3.4. Szczegóły implementacji

Algorytm w logice programu reprezentuje klasa `TSPTabuSearchSolver`. Klasa ta ma podobną sygnaturę i wiele podobnych rozwiązań do `TSPSimulatedAnnealingSolver`. Wynika to z faktu, że opracowanie implementacji tych algorytmów w wielu miejscach wymagało rozwiązania podobnych problemów. Ze względu na to podobieństwo, elementy wspólne zostały wydzielone do wspólnej nadklasy `TSPStochasticSolver`.

Części implementacji współdzielone przez obydwa algorytmu to losowanie rozwiązania startowego, obliczenie rozwiązania początkowego oraz wyznaczenie kosztu rozwiązania następnego (opisane w sekcji poświęconej implementacji algorytmu symulowanego wyżarzania). Tak samo też zawsze przechowywana jest aktualne rozwiązanie i najlepsze znalezione do tej pory rozwiązanie.

Lista tabu zrealizowana została jako tablica tablic (wykorzystana klasa `ffarray<>`) odpowiadająca połowie macierzy sąsiedztwa:

```
1.   - [ ][ ][ ][ ]
2.   -  - [ ][ ][ ]
3.   -  -  - [ ][ ]
4.   -  -  -  - [ ]
5.   -  -  -  -  -
```

Element i,j w macierzy odpowiada zamianie miejscami wierzchołków znajdujących się pod indeksami i oraz j w ścieżce – nie zamianie wierzchołków o indeksach i oraz j.

Ze względu na oszczędność miejsca i symetrię zamiany (zamiana i z j jest równoważna zamianie j z i) przechowujemy tylko połowę macierzy tabu. Macierz tabu przechowuje indeks ostatniej iteracji algorytmu, przy której doszło do zamiany miejscami wierzchołków znajdujących się pod indeksami i oraz j w tablicy reprezentującej ścieżkę. Aby sprawdzić, czy dany ruch jest tabu, algorytm odejmuje indeks aktualnej iteracji od wartości w tablicy tabu; jeżeli uzyskana różnica jest mniejsza niż zadana długość tabu, ruch jest tabu, w przeciwnym razie – nie jest.

Elementy macierzy tabu są inicjalizowane wartością $-tabuLength - 1$, żeby w pierwszej iteracji żaden ruch nie był objęty tabu.

Przy wyznaczaniu następnego ruchu jest przechowywany tylko najlepszy znaleziony ruch i odpowiadający mu nowy koszt, a nie wszystkie ruchy i odpowiadające im nowe koszty.

3. Pomiary czasu

3.1. Plan

Eksperyment polega na pomiarze czasu dla każdego z algorytmów dla dziesięciu różnych wielkości instancji: 4, 6, 8, 10, 12, 15, 20, 30, 50 i 100 węzłów (dla przeglądu zupełnego i nie wykonano pomiarów dla więcej niż 12 węzłów, dla BnB nie wykonano pomiaru dla więcej niż 15 węzłów, a dla programowania dynamicznego dla więcej niż 20 węzłów; algorytmy te dla większej ilości węzłów dawały wyniki w czasie tak długim, że pomiar czasu byłby niepraktyczny).

Dla każdego rozmiaru wygenerowano 100 instancji problemu, po czym zmierzono czas obliczenia rozwiązania przez każdy z algorytmów. Czasy zsumowano, aby uzyskać wartość średnią.

3.2. Generowanie instancji problemu

Aby wygenerować instancję problemu, generowano N losowych punktów na płaskiej powierzchni (dokładnie o rozmiarze 300x300 jednostek), po czym wyznaczano odległości między każdymi dwoma punktami. Następnie zinterpretowano punkty jako wierzchołki grafu, a odległości między nimi jako wagi przypisane do poszczególnych krawędzi.

Jest to zmiana względem poprzedniego etapu, gdzie generowano macierz losowych parami niezależnych wartości o rozmiarze N. Metodę zmieniono, aby rozważane instancje były bardziej realne.

Oczywiście zmiana sposobu generowania danych pociąga za sobą konieczność powtórzenia pomiarów dla algorytmów z poprzedniego punktu. Pomiary powtórzono.

3.3. Metoda pomiaru czasu

Do pomiaru czasu użyto klasy `StopWatch`. Klasa ta ma cztery metody: jedna do rozpoczęcia pomiaru, jedna do zakończenia pomiaru i dwie do odczytania czasu od rozpoczęcia do zakończenia pomiaru (jedna zwraca wynik jako liczbę całkowitą, druga jako liczbę zmiennoprzecinkową). W tym przypadku użyto metody zwracającej liczbę zmiennoprzecinkową, ponieważ liczba całkowita (zależnie od zastosowanego mnożnika) albo nie dawała zadowalającej precyzji dla małych instancji, albo dochodziło do przekroczenia zakresu – liczba całkowita się „przekręcała” i wynik wychodził nieprawidłowy.

Aby zmierzyć czas, tworzono instancję problemu, instancję klasy reprezentującej algorytm oraz obiekt klasy `StopWatch`, po czym wywoływano metodę `start()` stopera, przekazywano instancję problemu do metody `solveFor()` obiektu algorytmu (co bezpośrednio prowadziło do rozwiązania problemu) i zatrzymywano stoper metodą `stop()`. Następnie odczytywano zmierzoną wartość czasu.

3.4. Parametry

Przy pomiarach zastosowano parametry domyślne.

3.4.1. SA

Parametr	Wartość
Temperatura początkowa	5000
Stała chłodzenia	0.999
Ruchy pomiędzy obniżeniami temperatury	20
Temperatura końcowa	50
Starty	1

3.4.2. TS

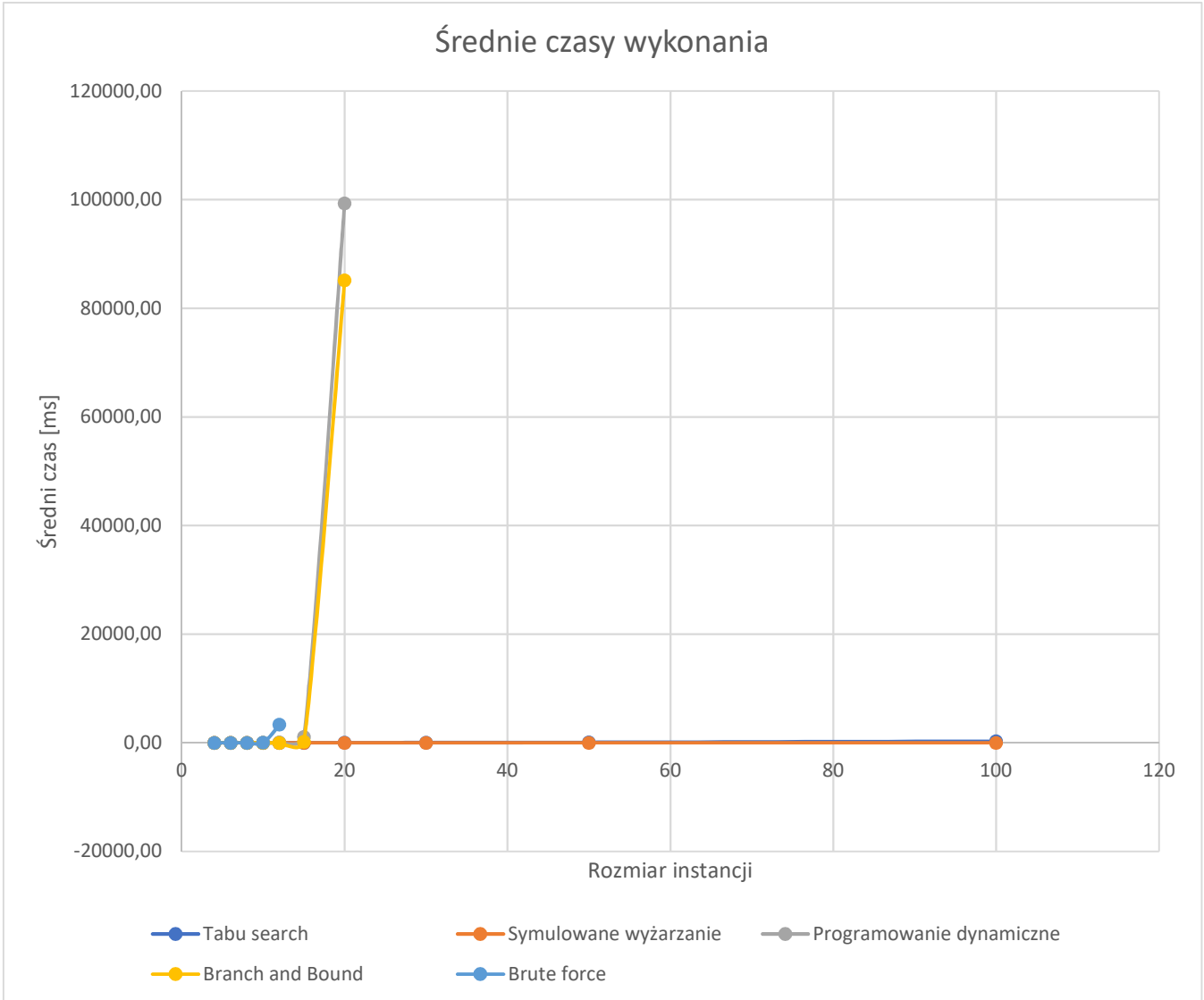
Parametr	Wartość
Długość listy tabu	5
Liczba iteracji	1000
Starty	1

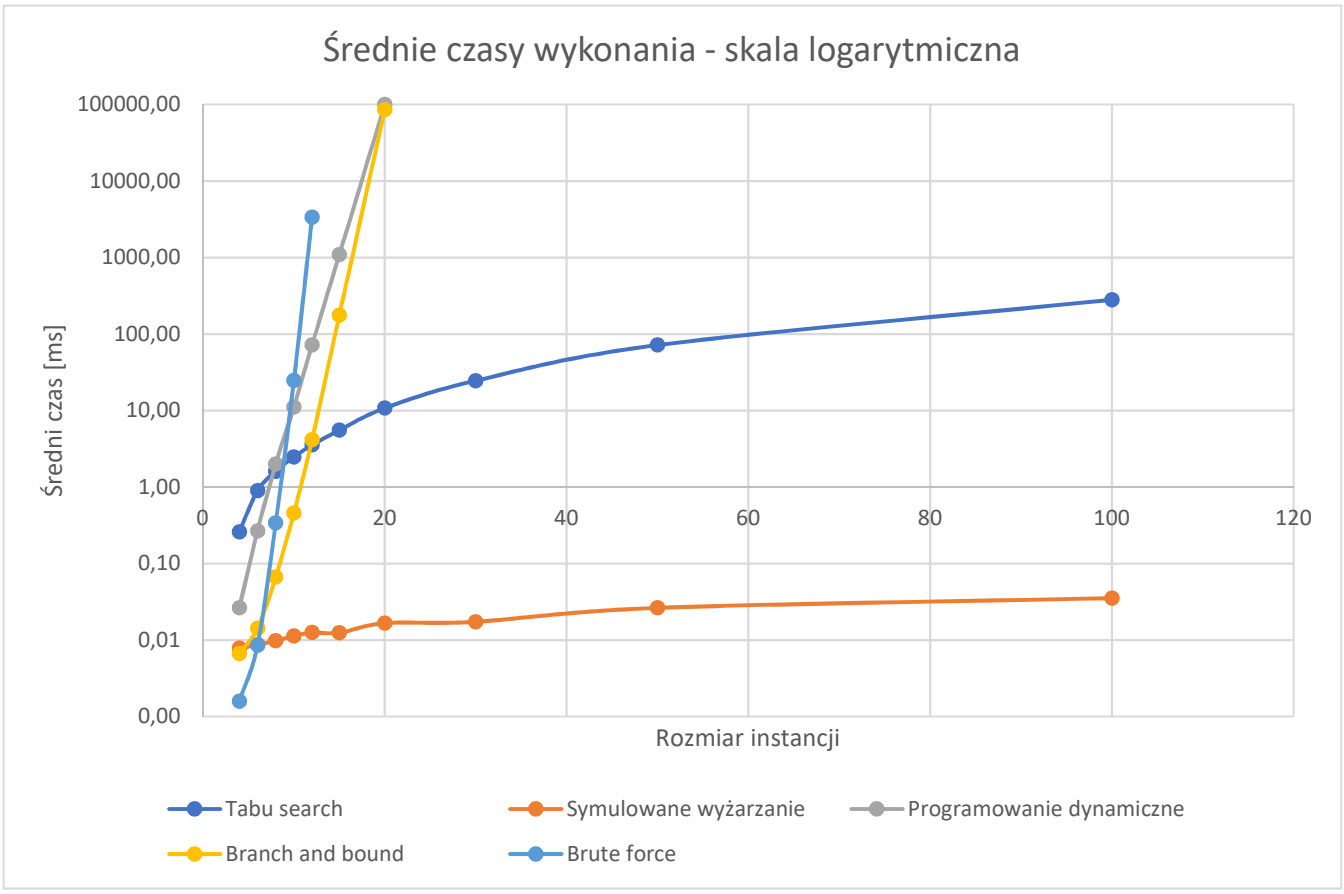
4. Uzyskane wyniki

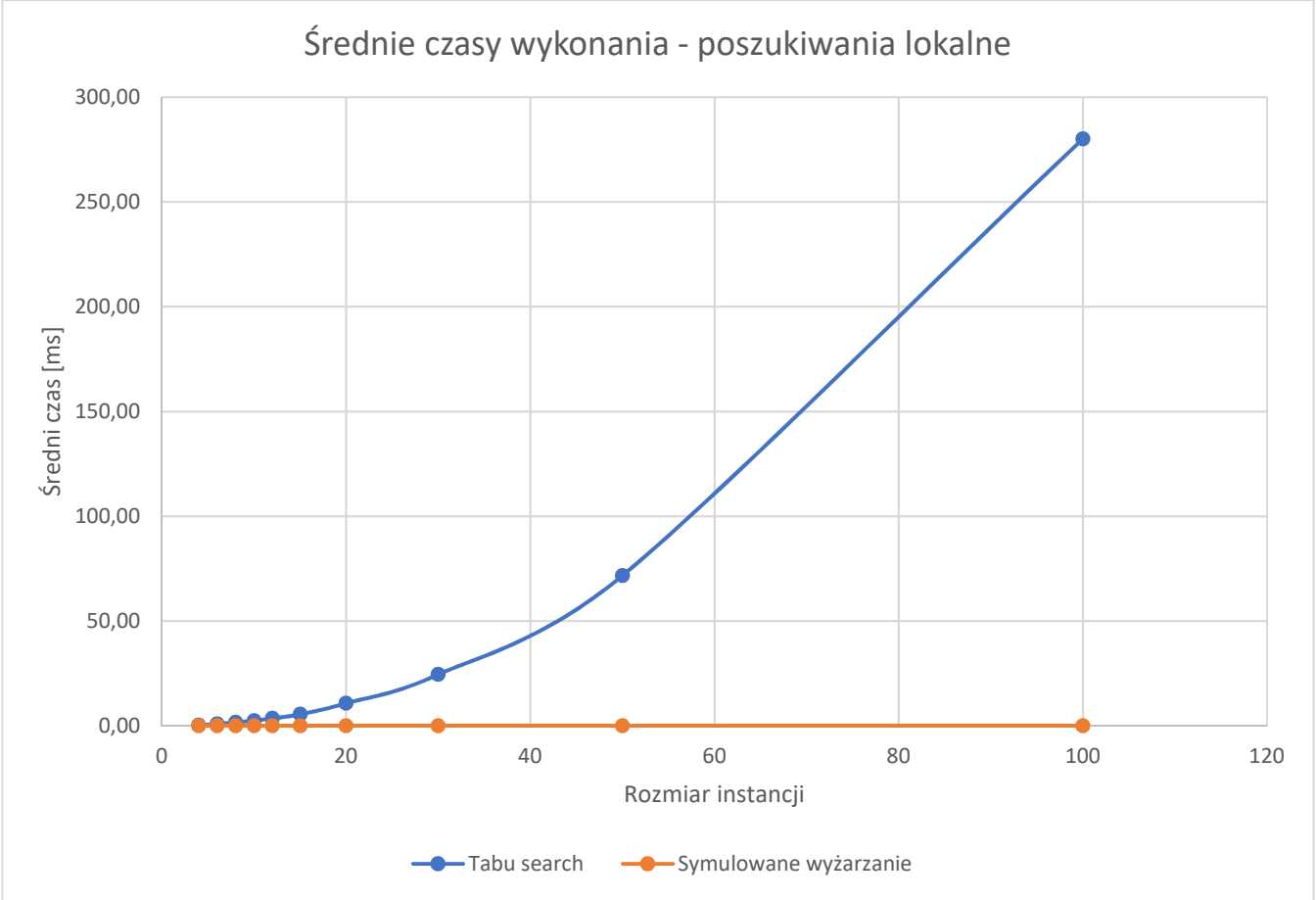
Uzyskane wynik przedstawia tabela:

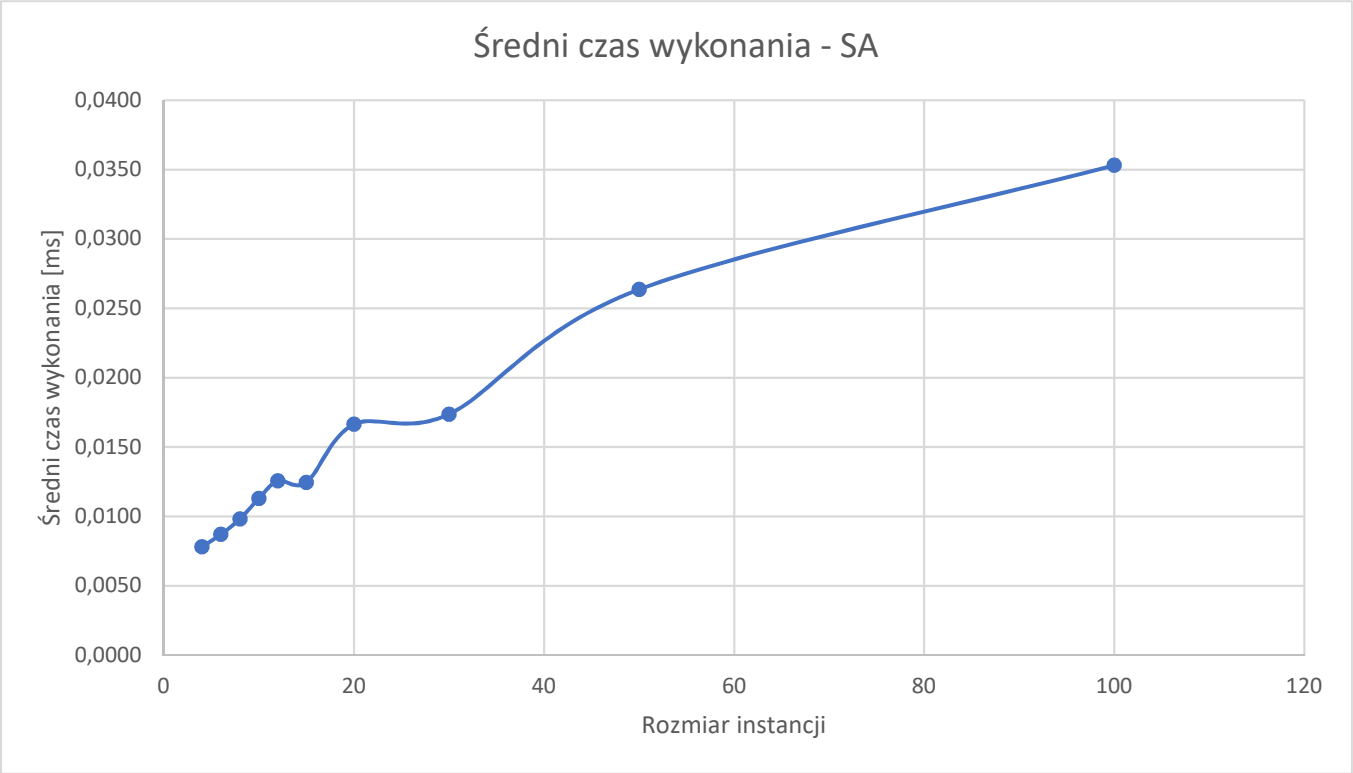
Rozmiar	Średni czas				
	Tabu Search	SA	Held-Karp	BnB	Brute Force
[1]	[ms]	[ms]	[ms]	[ms]	[ms]
4	0,26	0,0078	0,027	0,0066	0,0016
6	0,90	0,0087	0,27	0,0143	0,0086
8	1,60	0,0098	1,99	0,0665	0,3391
10	2,47	0,0113	11,1	0,4596	24,7
12	3,55	0,0126	72,0	4,1562	3353,3
15	5,52	0,0125	1090,1	176,0	
20	10,7	0,0166	99347,0	85171,74	
30	24,5	0,0174			
50	71,7	0,0264			
100	280,1	0,0353			

Poniższe wykresy obrazują dane zawarte w tabelach:



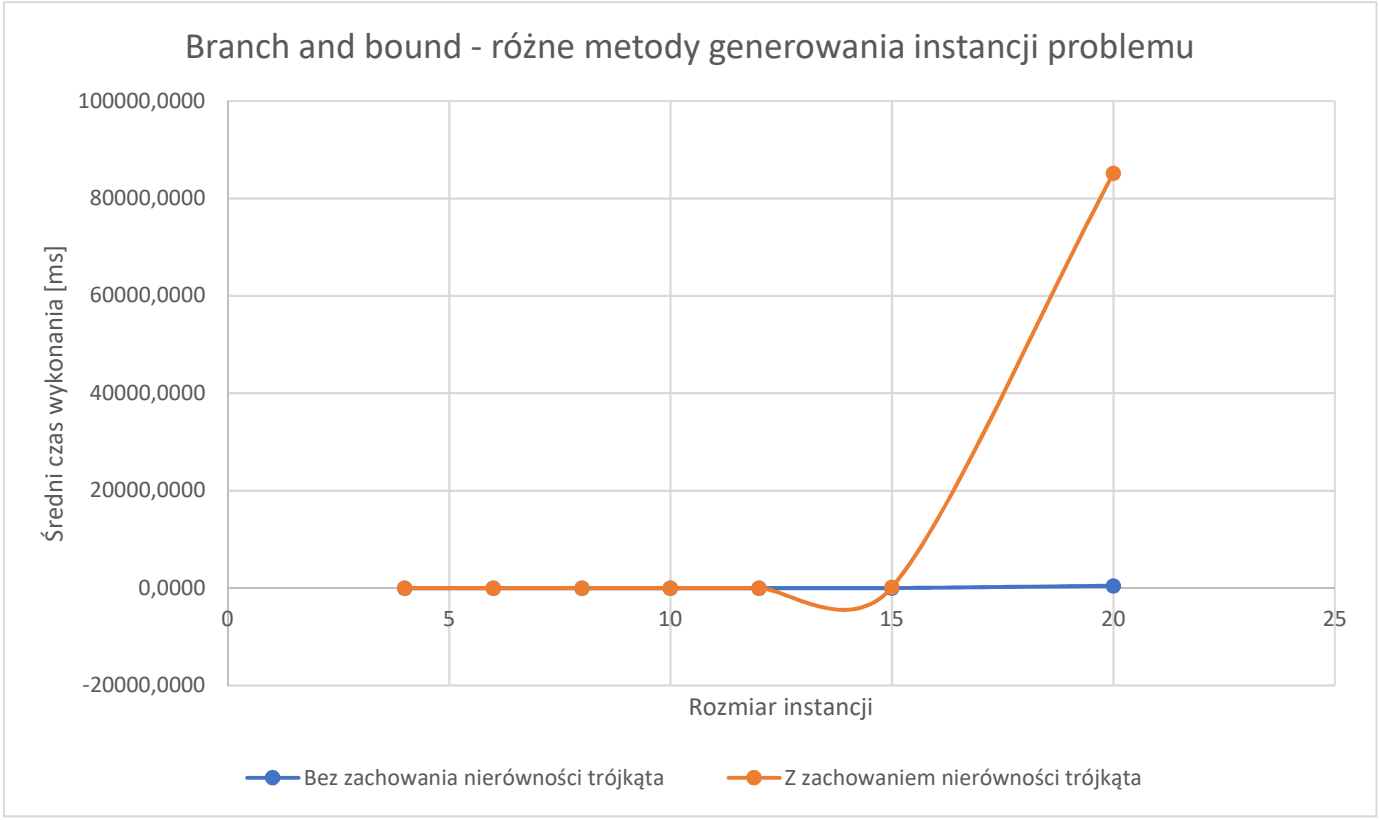

Średnie czasy wykonania - skala logarytmiczna
100000,00
10000,00
1000,00
100,00
10,00
1,00
0,10
0,01
0,00
Średni czas [ms]
0
20
40
60
80
100
120
Rozmiar instancji
Tabu search
Symulowane wyżarzanie
Programowanie dynamiczne
Branch and bound
Brute force

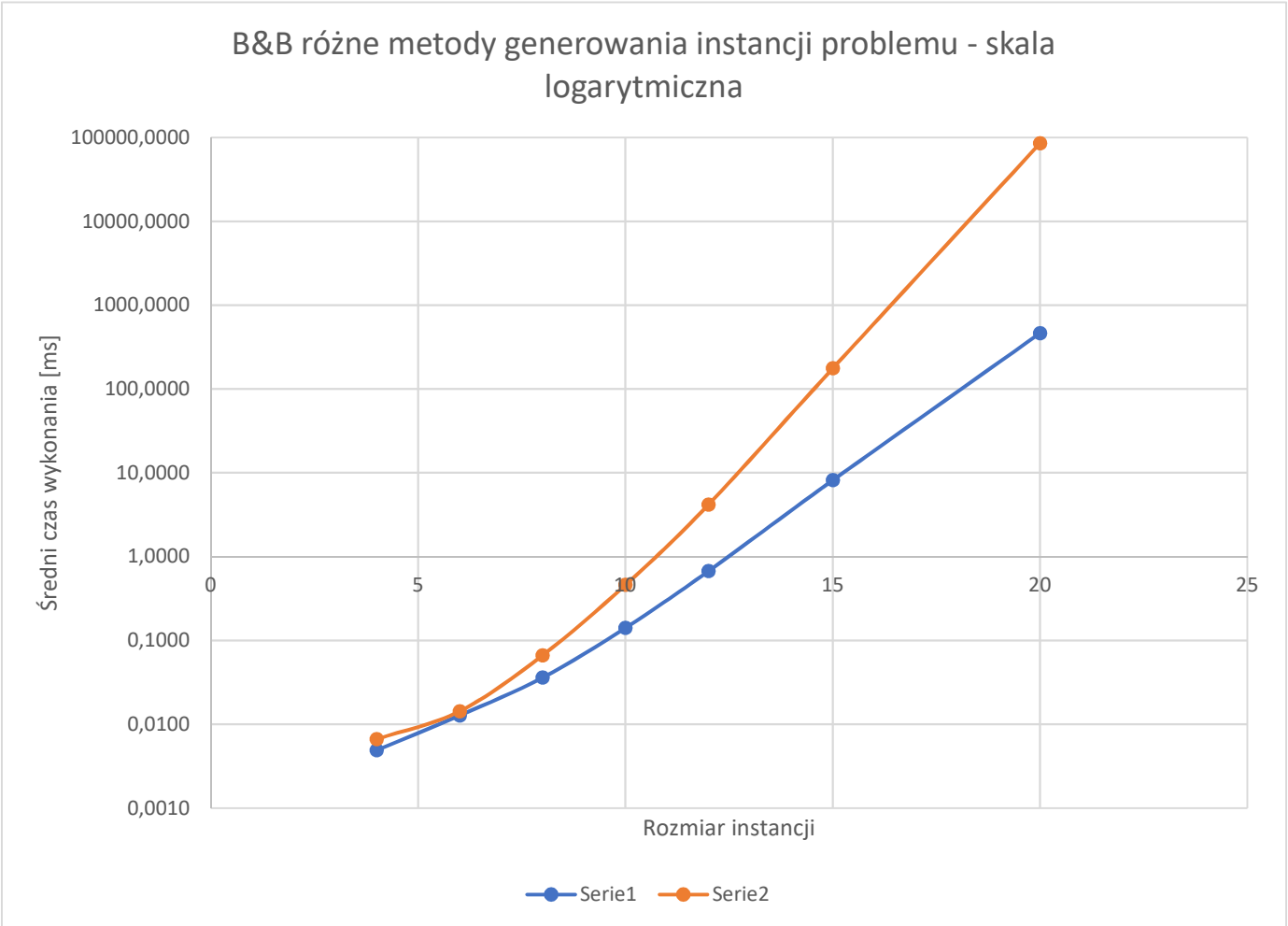

Średnie czasy wykonania - poszukiwania lokalne
300,00
250,00
200,00
150,00
100,00
50,00
0,00
Średni czas [ms]
0
20
40
60
80
100
120
Rozmiar instancji
Tabu search
Symulowane wyżarzanie



W sprawozdaniu do pierwszego etapu projektu zaznaczono potrzebę zmiany sposobu generowania instancji problemu. Największy wpływ zmiana miała na czas działania algorytmu podziału i ograniczeń (B&B). Czasy wykonania tego algorytmu dla starego i nowego sposobu generowania instancji problemu przedstawia poniższa tabela i wykresy:

Rozmiar	Średni czas	
	Stare pomiary	Nowe pomiary
[1]	[ms]	[ms]
4	0,0049	0,0066
6	0,0128	0,0143
8	0,0360	0,0665
10	0,141	0,460
12	0,672	4,16
15	8,1	176,0
20	462,6	85171,7


Branch and bound - różne metody generowania instancji problemu
100000,0000
80000,0000
60000,0000
40000,0000
20000,0000
0,0000
-20000,0000
Średni czas wykonania [ms]
0
5
10
15
20
25
Rozmiar instancji
Bez zachowania nierówności trójkąta
Z zachowaniem nierówności trójkąta


B&B różne metody generowania instancji problemu - skala logarytmiczna
100000,0000
10000,0000
1000,0000
100,0000
10,0000
1,0000
0,1000
0,0100
0,0010
Średni czas wykonania [ms]
0
5
10
15
20
25
Rozmiar instancji
Serie1
Serie2

5. Wnioski i uwagi końcowe

Czasy wykonania algorytmów poszukiwania lokalnego dla stałych parametrów pokrywają się z przewidywaniami: na wykresie dla Tabu Search widać zależność kwadratową (proporcjonalną do rozmiaru sąsiedztwa), z kolei wykres dla symulowanego wyżarzania można uznać za wykres funkcji liniowej z dodaną stałą.

Patrząc na wykresy można dojść do wniosku, że algorytm symulowanego wyżarzania jest zawsze szybszy niż Tabu Search, jednak warto pamiętać, że dużo zależy od wybranych parametrów.

Wyraźnie widać też na wykresach, że czasy wykonania algorytmów przeszukiwania lokalnego są już dla całkiem niedużych instancji znacznie mniejsze niż algorytmów dokładnych – ceną za to jest oczywiście brak gwarancji rozwiązania optymalnego. Jednak najczęściej nie jest konieczne rozwiązanie optymalne, zadowalamy się rozwiązaniem wystarczająco dobrym.

Wielką siłą ale i słabością prezentowanych algorytmów jest możliwość dość dokładnego dostrojenia ich działania przez dobór parametrów: pozwala to zdecydować, jak długo jesteśmy gotowi poszukiwać lepszego rozwiązania, a jednocześnie przez to konieczna staje się jakaś wiedza *a priori* na temat instancji problemu, aby możliwe było dobranie odpowiednich parametrów.

Warto zauważyć też, jak bardzo zmiana sposobu generowania instancji problemu wpłynęła na czas wykonania algorytmu B&B. Przedstawione dane potwierdzają, że poprzedni sposób generowania instancji bardzo mocno promował zastosowaną heurystykę, podczas gdy przy danych bardziej „realnych” ta heurystyka nie radzi sobie już tak dobrze.

6. Spis treści