

Laboratorium Optymalizacji Kombinatorycznej

Algorytm heurystyczny dla problemu Job-Shop Scheduling

Prowadzący: prof. dr hab. inż. Maciej Drozdowski

Krzysztof Marciniak - 106574,

Dominik Galewski - 106575

Poznań, dnia 30 listopada 2012 r.

1. Wstęp

1.1. Wprowadzenie

Tematem niniejszego sprawozdania jest analiza przykładowego, stworzonego przez nas na potrzeby tego ćwiczenia, algorytmu heurystycznego rozwiązującego problem JSP. W dalszej części zawarte zostaną zarówno informacje dotyczące analizowanego problemu jak i stworzonego do rozwiązania go algorytmu a także zestawienie wygenerowanych wyników oraz czasów wykonania.

1.2. Opis problemu

Idea JSP (problem szeregowania Job-Shop) może zostać ukazana następująco:

Mając dane na wejściu sekwencje wykonania operacji (ang. *task*) [numerowane od 0 do $m-1$ i opisane przez dwie wielkości: *czas trwania* oraz *numer maszyny*] zwane zadaniami (ang. *job*) [numerowane od 0 do $n-1$] oraz m maszyn, do których przypisane są poszczególne operacje, znaleźć długość **najkrótszego** uszeregowania, które wykonuje wszystkie operacje w zadanej kolejności tak, aby żadna z operacji na danej maszynie nie nakładała się (zakładamy, iż maszyny nie mogą wykonywać zadań równolegle)

Można wykazać, zakładając iż znany problem komiwojażera jest NP-trudny, że opisywany tu problem także jest NP-trudny - zakładając $m=1$ uzyskujemy problem komiwojażera (pojedyncza maszyna staje się komiwojażerem, czasy trwania operacji - odległościami między miastami, zaś poszczególne zadania [złożone z jednej operacji, gdyż ilość maszyn jest równa ilości operacji] to nic innego jak miasta). Problemu takiego nie można więc rozwiązać znajdując rozwiązanie optymalne w czasie wielomianowym - wymaga to albo czasu wykładniczego, albo znalezienia dowolnego rozwiązania poprawnego w czasie wielomianowym przy pomocy algorytmu heurystycznego - ten ostatni przypadek został opisany w tym sprawozdaniu.

1.3. Środowisko testowe

Testy algorytmu przeprowadzone zostały na następującej maszynie:

- cpu: AMD Phenom II X4 965 3.9 GHz
- pamięć: Kingston HyperX DDR3 1600MHz 8GB
- system: Ubuntu Linux 12.10 32-bit

Powyższa konfiguracja pozwoliła uzyskać bardzo dobre wyniki czasowe, jednak zdajemy sobie sprawę z tego, iż na innej maszynie testowej mogłyby one być gorsze. Nie mieliśmy jednak (niestety) możliwości sprawdzenia działania programu testowego na innej maszynie.

1.4. Uwagi

Program testowy wykorzystuje funkcję `clock_gettime()` z nagłówka `ctime` lub `time.h`, która jest dostępna tylko pod systemami linuxowymi; próba kompilacji kodu załączonego do sprawozdania pod systemem Windows lub innym nie bazującym na jądrze *NIX może zakończyć się błędem.

2. Algorytm

2.1. Opis algorytmu

Ideą działania algorytmu stworzonego przez nas do rozwiązania JSP jest gromadzenie operacji wykonywanych na danym etapie" (dla tego samego identyfikatora operacji) i przypisywanie ich do określonych maszyn w tabeli przydziałów. Takie ustawienie pozwala na rozwiązywanie konfliktów metodą FIFO - operacje zadań o niższych identyfikatorach zostają wykonane szybciej niż pozostałe. Oczywiście samo wrzucanie zadań konfliktów nie rozwiązuje, właściwe rozwiązanie nakładania się operacji na maszynie polega na obliczeniu która z wartości jest większa: czas zakończenia ostatniej operacji analizowanego zadania czy też czas zakończenia ostatniej operacji na maszynie którą teraz sprawdzamy. Wynika to stąd, iż określona operacja nie może zacząć się wykonywać wcześniej niż zakończy się operacja poprzedzająca ją (pozwala to m.in. zachować porządek przewidziany w zadaniu), jednak może wykonać się później - jest to opisany przypadek konfliktu operacji i wymaga on od operacji oczekiwania na zwolnienie danej maszyny (a to nastąpi wówczas, gdy skończy się wykonywać przetwarzana operacja co zostanie odnotowane w odpowiedniej tablicy).

Przyjmując następujące oznaczenia:

- $operacja_{j,t}$ - operacja t zadania j ; opisana jest przez następujące wartości: moment startu t_{st} , czas działania dur , identyfikator maszyny $machine_id$ oraz identyfikator zadania job_id do którego należy.

W praktyce nie jest wykorzystywana dwuwymiarowa tablica *operacja* jak mogłoby sugerować oznaczenie, lecz tablica zadań *Jobs* złożona z tablic [a właściwie obiektów klasy Job zawierających te tablice oraz kilka innych istotnych wartości] operacji, przy czym operacja jest obiektem klasy Task

- *MachineUsage* - tablica m list operacji
- *StopTimes* - tablica m nieujemnych liczb całkowitych; na pozycji o indeksie i przechowywany jest moment zakończenia wykonywania ostatniej operacji na maszynie i

algorytm może zostać zdefiniowany następująco:

1. MachineUsage zainicjalizuj listami pustymi zaś StopTimes wypełnij wartościami 0
2. Przyjmij $t = 0$
3. Przyjmij $j = 0$
4. Przyjmując za $machine_id$ identyfikator maszyny, do której przypisana jest operacja t zadania j , dodaj operację $operacja_{j,t}$ na koniec listy $MachineUsage[machine_id]$
5. Zwiększ j o 1

6. Jeśli j jest równe liczbie zadań, przejdź do kolejnego punktu; w przeciwnym razie wróć do punktu 4.
7. Przyjmij $i = 0$; jest to numer obecnie analizowanej maszyny
8. Przyjmij $cur_task = 0$; jest to numer obecnie przetwarzanej operacji z listy *MachineUsage*[i]
9. Przyjmij za job_dur czas trwania zadania job_id , zaś za $task_start_time$ większą z dwóch wartości: *StopTimes*[$machine_id$] lub job_dur
10. Ustaw czas trwania analizowanej operacji cur_task na $task_start_time$, czas zakończenia operacji $task_end_time$ na $task_start_time + dur$, czas zakończenia przetwarzania na maszynie *StopTimes*[$machine_id$] na $task_end_time$ zaś czas trwania zadania job_id na $task_end_time$
11. Zwiększ cur_task o 1; jeśli jest równe długości listy *MachineUsage*[i], przejdź do kolejnego punktu; jeśli nie, wróć do punktu 8.
12. Zwiększ i o 1; jeśli jest równe ilości maszyn, przejdź do kolejnego punktu; w przeciwnym wypadku wróć do punktu 7.
13. Wyczyść wszystkie listy w *MachineUsage*
14. Zwiększ t o 1; jeśli jest równe ilości operacji (a zatem ilości maszyn), zakończ działanie - wszystkie operacje wszystkich zadań zostały przeanalizowane. W przeciwnym wypadku wróć do punktu 2.

5	2		
0	1	1	1
0	1	1	1
0	1	1	1
0	1	1	1
1	1	0	1

Tablica 1: Instancja testowa

2.2. Przykład

Przyjmijmy iż na wejściu podana została instancja Beasley’a (orlib) przedstawiona w tabeli 1. Istnieje 5 zadań złożonych odpowiednio z 4 operacji o długości 1 wykonywanych na maszynie 0 oraz jednej operacji o długości 1 na maszynie 1 na etapie pierwszym oraz odwrotnie (4 operacje na 1, jedna na 0) na etapie drugim; podział na etapy ma w tym wypadku wprowadzić sposób myślenia taki jak działanie algorytmu i nie jest on w żaden sposób oficjalną częścią problemu. Dla ułatwienia przyjmijmy iż operację o długości d na maszynie d oznaczать będziemy przez (m, d) . Ponieważ wszystko zostało już wyjaśnione, czas uszeregować owe operacje.

Na początku inicjalizujemy $MachineUsage[]$ - otrzymujemy $\{\emptyset, \emptyset\}$ a następnie ustawiamy $StopTimes[]$ na $\{0, 0\}$.

Tu rozpoczynamy pierwszy etap. Na odpowiednie listy wrzucamy operacje przypisane konkretnym zadaniom; po zakończeniu uzupełniania tablica $MachineUsage[]$ wygląda tak:

$\{[(0, 1), (0, 1), (0, 1), (0, 1)], [(1, 1)]\}$

Najpierw przetwarzamy listę $MachineUsage[0]$: jako że $StopTimes[0]$ wynosi 0, a zadanie do którego należy operacja jeszcze nie zaczęło się wykonywać, za $task_start_time$ podstawiamy $\max\{0, 0\} = 0$. Dodajemy do tego czas trwania operacji (1) i otrzymujemy długość trwania zadania 1 oraz wartość $StopTimes[0]$ równą 1; analogicznie obliczamy wykonania dla pozostałych operacji na liście otrzymując czasy startu i zakończenia $\{[0, 1], [1, 2], [2, 3], [3, 4]\}$. Zadania są w tym wypadku wywoływane bezpośrednio po sobie co nie powinno dziwić - przydzielamy maszyny w kolejności takiej, w jakiej ubiegają się o nie konkretne operacje. Po zakończeniu przetwarzania tej listy wartość $StopTimes[0]$ wynosi 4, natomiast $StopTimes[1]$ pozostaje bez zmian.

Ostatnie do przetworzenia pozostaje zadanie (1, 1) które zostanie wykonane w taki sam sposób jak 1. operacja tego etapu - zostanie przydzielone na starcie (w czasie $t = 0$) i po 1 jednostce czasu zakończy się zmieniając wartość $StopTimes[1]$ z 0 na 1 oraz zmieniając czas trwania zadania do którego należy z 0 na 1.

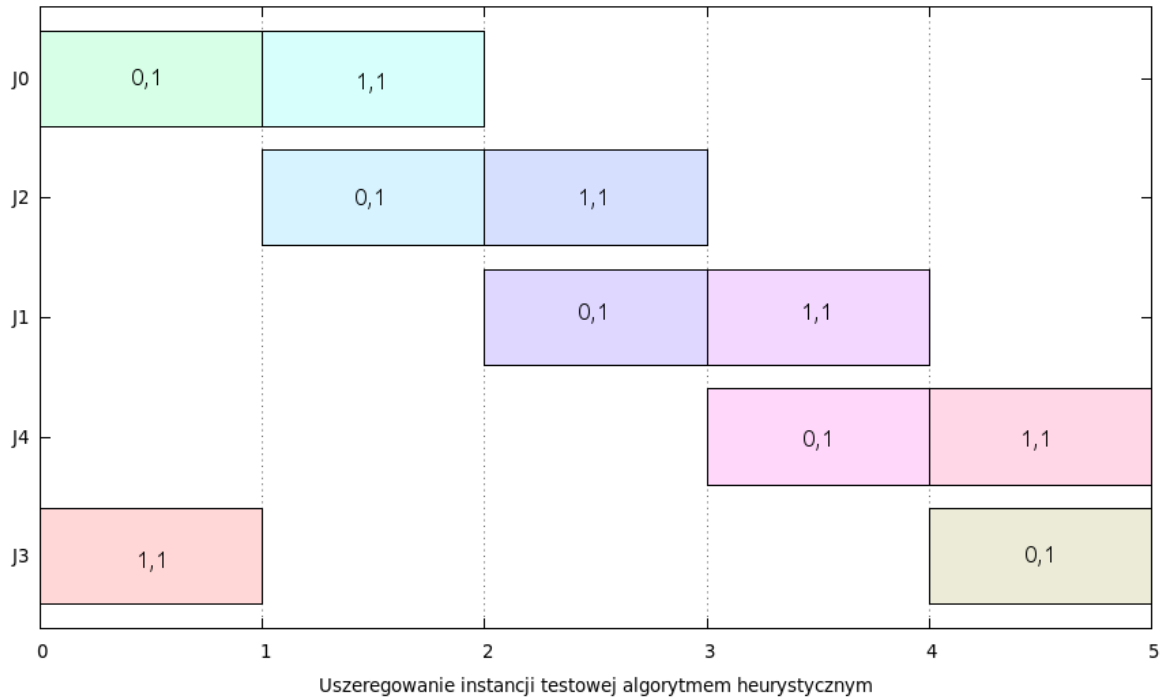
Podsumowując 1. etap: czasy trwania zadań wynoszą odpowiednio 1, 2, 3, 4 oraz 1, $MachineUsage$ zostaje wyczyszczone zaś $StopTimes$ wynosi $[4, 1]$

Czas zacząć więc etap 2. Jest on analogiczny do etapu 1., więc ograniczymy się do opisu jednej operacji: zadania 4. Zaczynamy więc po raz kolejny od wypełnienia list $MachineUsage$, w tym wypadku będzie ona wyglądać tak:

$\{[(0, 1)], [(1, 1), (1, 1), (1, 1), (1, 1)]\}$

Obliczmy więc czas rozpoczęcia wykonywania operacji (0, 1) na maszynie 0. Wartość $StopTimes[0]$ wynosi 4, zaś czas trwania zadania - 1. $\max\{4, 1\} = 4$ które staje się momentem rozpoczęcia przetwarzania operacji na maszynie 0. Dodajemy do tego 1, które jest czasem przetwarzania operacji, i - uzyskując 5 - wrzucamy do $StopTimes[0]$ (nie będzie ono już istotne w tym wypadku, jednak generalnie jest to ważne).

Pozostałe operacje z $MachineUsage[1]$ przetwarzamy w taki sam sposób i ostatecznie uzyskujemy uszeregowanie przedstawione na wykresie Gantta na następnej stronie.



2.3. Analiza złożoności

Upraszczając skrajnie powyższy opis algorytmu (Sprowadzając go jedynie do głównych pętli) i zapisując go w składni języka C++ możemy uzyskać następującą postać:

Listing 1: uproszczony zapis kodu na potrzeby analizy złożoności

```

1  for(int t = 0; t < m; t++) {
2      for(int j = 0; j < n; j++) {
3          // uzupełnianie MachineUsage
4      }
5      for(int i = 0; i < m; i++) { // obliczanie czasow
6          /*
7              niezależnie od wielkości poszczególnych list
8              w tym miejscu wykonanych zostanie n operacji
9              (ponieważ tyle operacji zostało wrzuconych
10             jedna linie wyżej)
11          */
12      }
13  }
```

Łatwo zauważyć, iż w ten sposób wykonane zostanie $O(m \cdot (n + mn)) = O(mn \cdot (m + 1))$ operacji, co daje złożoność kwadratową od ilości maszyn/operacji. Warto pamiętać o tym, iż jest to także kwestia implementacji - uzyskanie takiej złożoności możliwe jest m.in. dzięki zastosowaniu struktur danych typowych dla języka C++; w wypadku innych języków złożoność ta może być większa.

2.4. Kilka słów o implementacji

Jak już wspomnieliśmy wcześniej, wykorzystane zostało tu programowanie zorientowane obiektowo, co pozwala na uzyskanie dostępu do wartości charakteryzujących operacje/zadania w czasie liniowym bez zwiększenia narzutu pamięciowego (oraz zwiększa wygodę pisania i czytelność

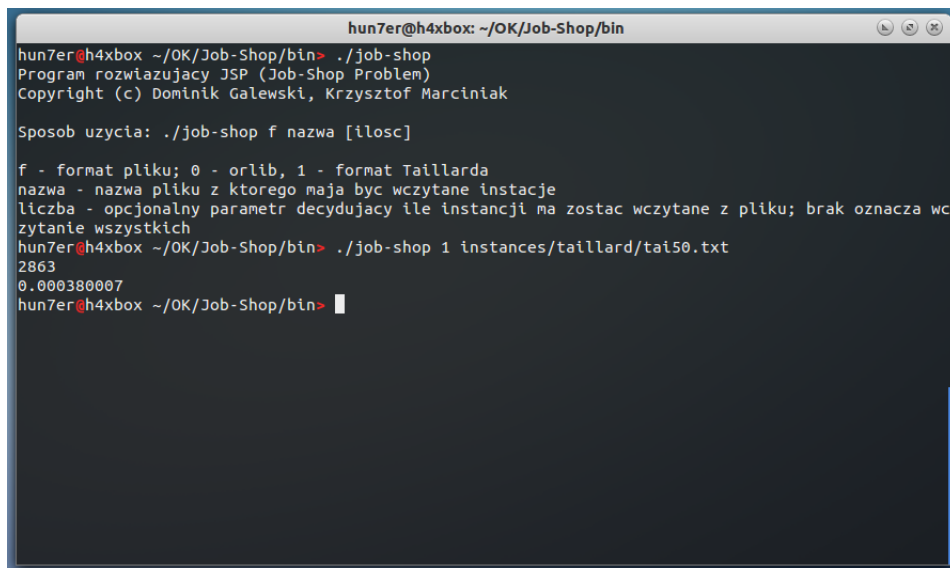
kodu). Zamiast standardowych dynamicznych tablic oraz tworzonych specjalnie na potrzeby projektu list wykorzystane zostały kontenery *std::vector* oraz listy *std::list*. Operacje wejścia/wyjścia odbywają się poprzez strumienie (nagłówek *iostream*), operacje na plikach także przez strumienie (*fstream*) zaś czytanie wartości z plików oraz ich konwersja na liczby przez *sstream*.

3. Testy

3.1. Wprowadzenie

Kod źródłowy programu wykorzystywanego do testów jak i skrypty powłoki BASH można pobrać tutaj: <https://github.com/hun7err/Job-Shop>

Przykładowe sposoby użycia skryptów i programu przedstawione zostały na odpowiednich zrzutach ekranu na rysunkach 1-3.

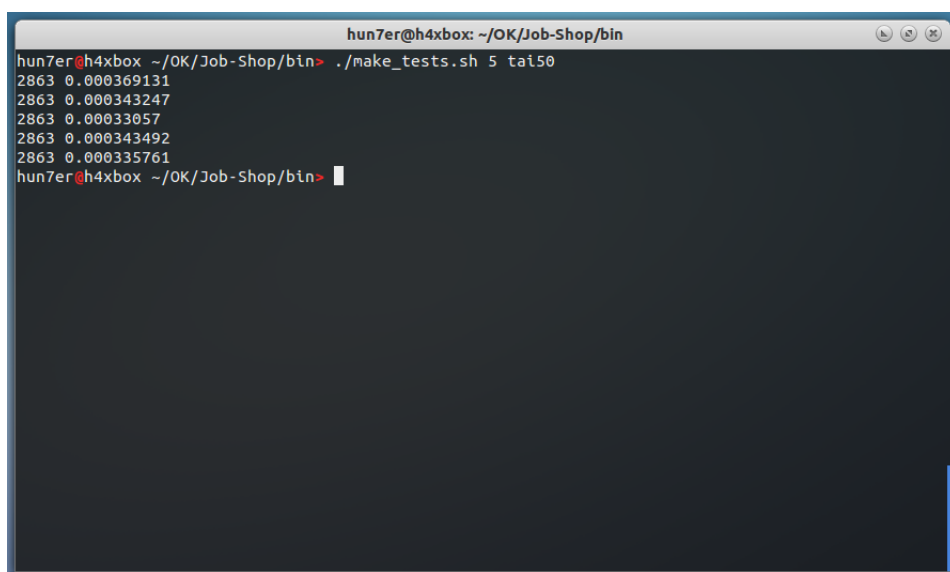


```
hun7er@h4xbox: ~/OK/Job-Shop/bin
hun7er@h4xbox ~/OK/Job-Shop/bin> ./job-shop
Program rozwiązujący JSP (Job-Shop Problem)
Copyright (c) Dominik Galewski, Krzysztof Marciniak

Sposob użycia: ./job-shop f nazwa [ilosc]

f - format pliku; 0 - orlib, 1 - format Taillarda
nazwa - nazwa pliku z ktorego maja byc wczytane instancje
liczba - opcjonalny parametr decydujący ile instancji ma zostac wczytane z pliku; brak oznacza wczytanie wszystkich
hun7er@h4xbox ~/OK/Job-Shop/bin> ./job-shop 1 instances/taillard/ta150.txt
2863
0.000380007
hun7er@h4xbox ~/OK/Job-Shop/bin> 
```

Rysunek 1: Przykładowe użycie programu testowego



```
hun7er@h4xbox: ~/OK/Job-Shop/bin
hun7er@h4xbox ~/OK/Job-Shop/bin> ./make_tests.sh 5 ta150
2863 0.000369131
2863 0.000343247
2863 0.00033057
2863 0.000343492
2863 0.000335761
hun7er@h4xbox ~/OK/Job-Shop/bin> 
```

Rysunek 2: Przykładowe użycie skryptu automatyzującego wykonanie programu testowego

```

hun7er@h4xbox: ~/OK/Job-Shop/bin
hun7er@h4xbox: ~/OK/Job-Shop/bin> ./jobcount_tests.sh 5 tai50
1148 0.000139662
1148 0.000109753
1310 0.000119431
1374 0.000128051
1401 0.000146885
hun7er@h4xbox: ~/OK/Job-Shop/bin> ./jobcount_tests.sh 10 tai50
1148 0.000141443
1148 0.000110007
1310 0.000118373
1374 0.000128054
1401 0.000138991
1442 0.000142734
1550 0.000148761
1611 0.000164097
1656 0.000165862
1672 0.00017484
hun7er@h4xbox: ~/OK/Job-Shop/bin>

```

Rysunek 3: Przykładowe użycie skryptu wykonującego n pierwszych zadań instancji

n	w_heur	w_dol	w_gor
20	1937	1213	1362
21	2312	1217	1663
22	2283	1314	1626
23	2196	1248	1574
24	2233	1284	1660

Tablica 2: Jakość rozwiązań. Długość uszeregowania dla instancji tai20-tai24: w_heur - długość wygenerowanego uszeregowania, w_dol - długość najkrótszego uszeregowania, w_gor - długość najdłuższego uszeregowania

We wszystkich przypadkach testowych, aby zminimalizować błąd pomiaru, dokonane zostały serie pomiarów po 5 uruchomieniach programu testowego każda. Wyniki zamieszczone w sekcji "Wyniki"

3.2. Wyniki

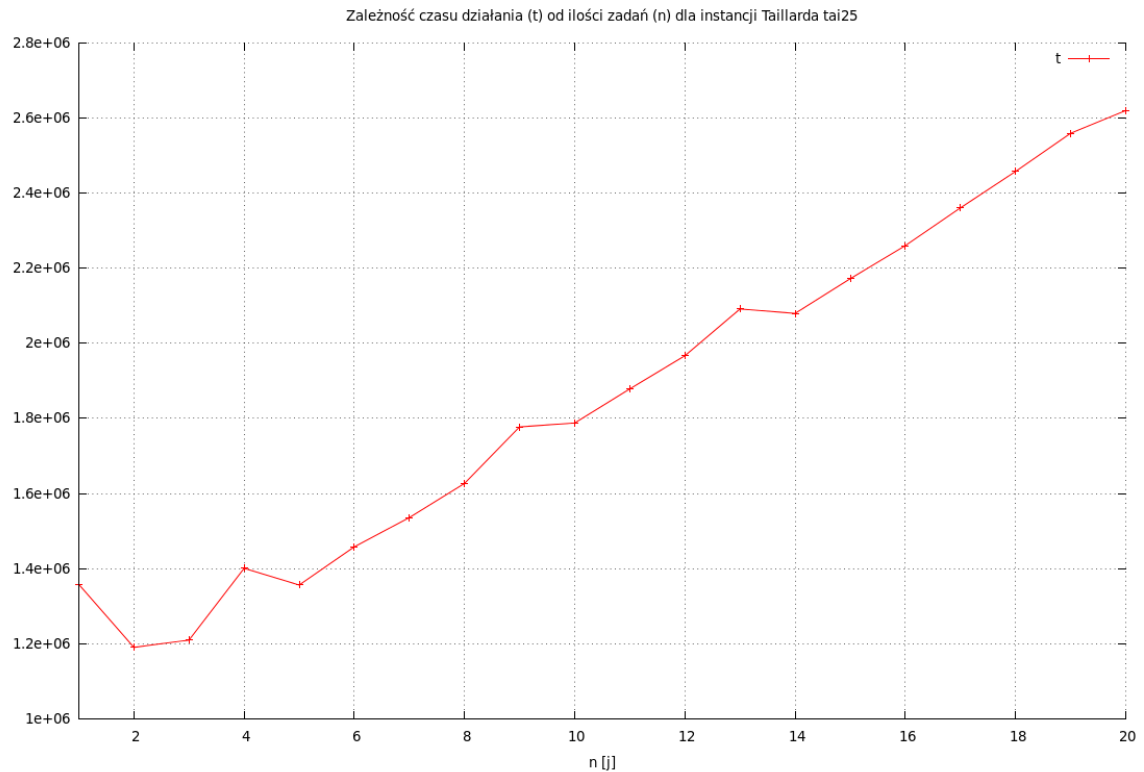
Wyniki testów (odpowiednio pomiaru czasu dla instancji Taillarda tai20-24 oraz dodatkowo jakości rozwiązań dla tai25 i czasu dla liczby zadań rosnącej od 1 do 20 przedstawione zostały w tabelach oraz zobrazowane na odpowiednich wykresach zamieszczonych poniżej.

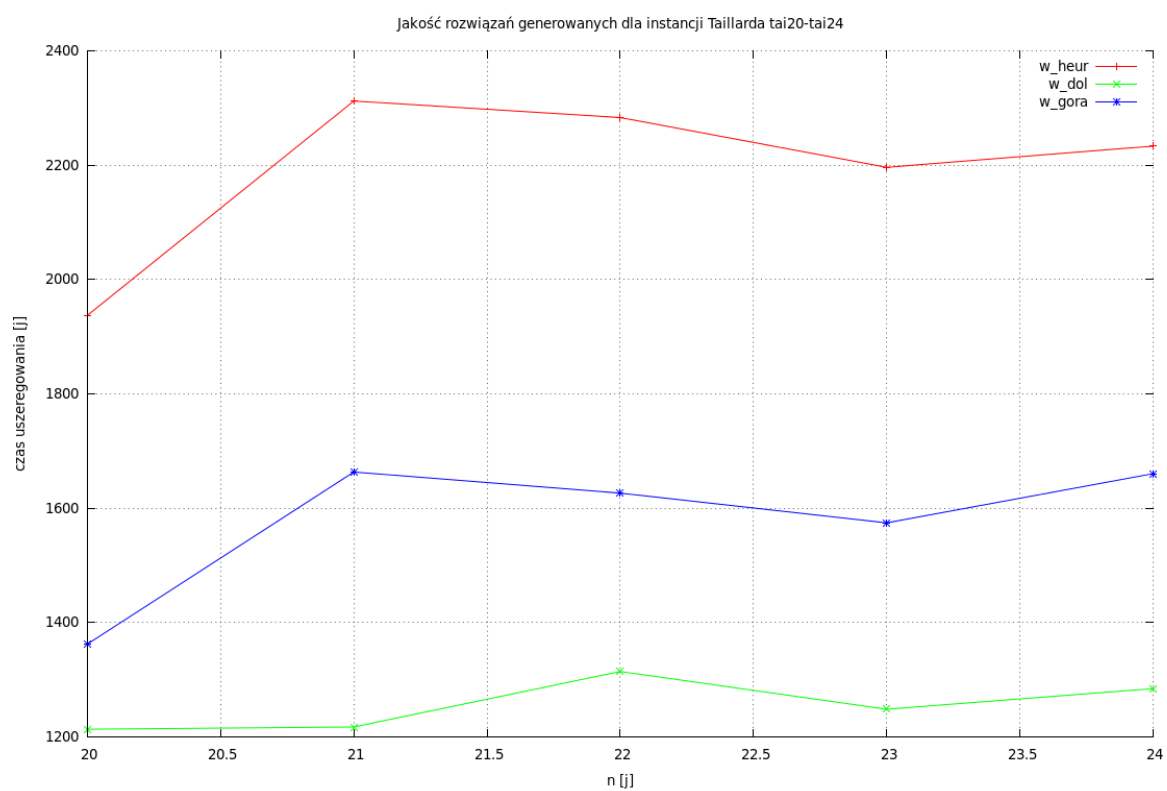
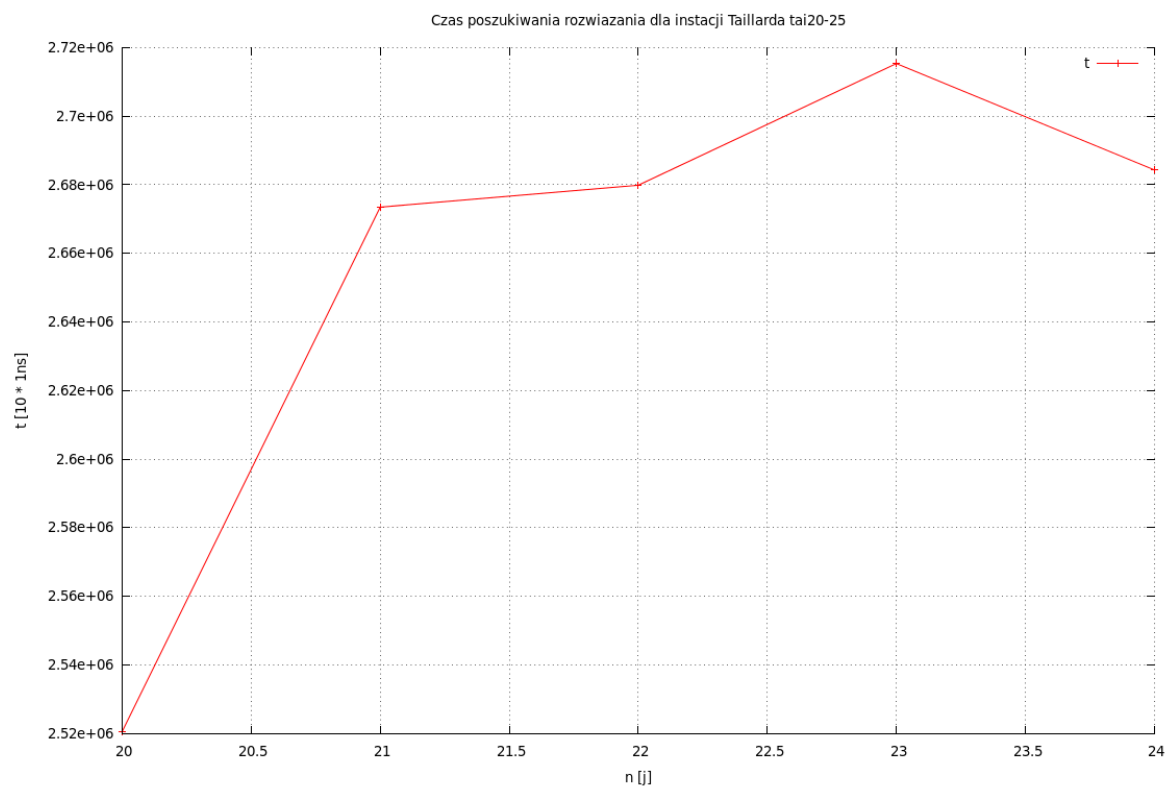
n	pomiar 1	pomiar 2	pomiar 3	pomiar 4	pomiar 5	średnia
20	0.000249262	0.000217277	0.000210625	0.000372336	0.000210778	0.0002520556
21	0.000289934	0.000259922	0.000276765	0.000255635	0.000254478	0.0002673468
22	0.000290523	0.000259013	0.000277318	0.000254619	0.000258459	0.0002679864
23	0.000293287	0.000266415	0.000279266	0.000260178	0.000258515	0.0002715322
24	0.000303617	0.000259151	0.000265288	0.000257259	0.00025684	0.000268431

Tablica 3: Czas wykonywania dla instancji tai20-tai24

n	pomiar 1	pomiar 2	pomiar 3	pomiar 4	pomiar 5	średnia
1	0.000136891	0.000138124	0.000133193	0.000137867	0.000132761	0.0001357672
2	0.000106611	0.000130659	0.000112512	0.000134897	0.000110248	0.0001189854
3	0.000118974	0.000120523	0.000127592	0.000119854	0.000117705	0.0001209296
4	0.000200902	0.000125864	0.000120298	0.000128804	0.000124368	0.0001400472
5	0.00013748	0.000134921	0.000131103	0.000137612	0.000136895	0.0001356022
6	0.000144395	0.000145004	0.000147446	0.000148803	0.000143128	0.0001457552
7	0.000155117	0.000148213	0.000157253	0.000154215	0.000153007	0.000153561
8	0.000161741	0.000162315	0.00016641	0.000159693	0.000163067	0.0001626452
9	0.000169827	0.000167233	0.00018334	0.000183315	0.00018489	0.000177721
10	0.000182562	0.000179737	0.000175149	0.000178495	0.000177803	0.0001787492
11	0.00020098	0.000184428	0.000183737	0.000183721	0.000186297	0.0001878326
12	0.000195668	0.000200248	0.000198276	0.000193531	0.000195608	0.0001966662
13	0.000209934	0.000202208	0.000217379	0.000206239	0.000209872	0.0002091264
14	0.000205723	0.00020741	0.000206064	0.000211212	0.000209396	0.000207961
15	0.000218323	0.000222152	0.000212904	0.000215359	0.000217206	0.0002171888
16	0.000227722	0.000222366	0.000229543	0.000227292	0.000223028	0.0002259902
17	0.000231938	0.00023416	0.000237762	0.00024043	0.000235943	0.0002360466
18	0.000246603	0.000250798	0.000253854	0.000238947	0.000238493	0.000245739
19	0.000254034	0.000255494	0.000262385	0.000252613	0.000255069	0.000255919
20	0.000256652	0.000258174	0.000263599	0.000273241	0.000258198	0.0002619728

Tablica 4: Pomiary czasu wykonania dla instancji tai25 z liczbą zadań rosnącą od 1 do 20





4. Wnioski

Jak widać po tabelach oraz wykresach generowane wyniki nie są w zadanym przedziale $\langle w_dol, w_gora \rangle$ (przynajmniej dla tai25), jednak dla wspomnianej instancji czas uszeregowania tworzy podobną krzywą co w_gora , co pozwala nam sądzić iż wygenerowane rozwiązanie jest dopuszczalne. Algorytm (oraz jego implementacja) zostały sprawdzone dla instancji testowych oraz instancji stworzonych tylko i wyłącznie na potrzeby wyszukiwania potencjalnych błędów, co pozwala sądzić, iż generowane rozwiązania są poprawne.