

Sprawozdanie dot. problemu szeregowania zadań $P3||L_{max}$

3 kwietnia 2006 roku

1 Sformułowanie problemu

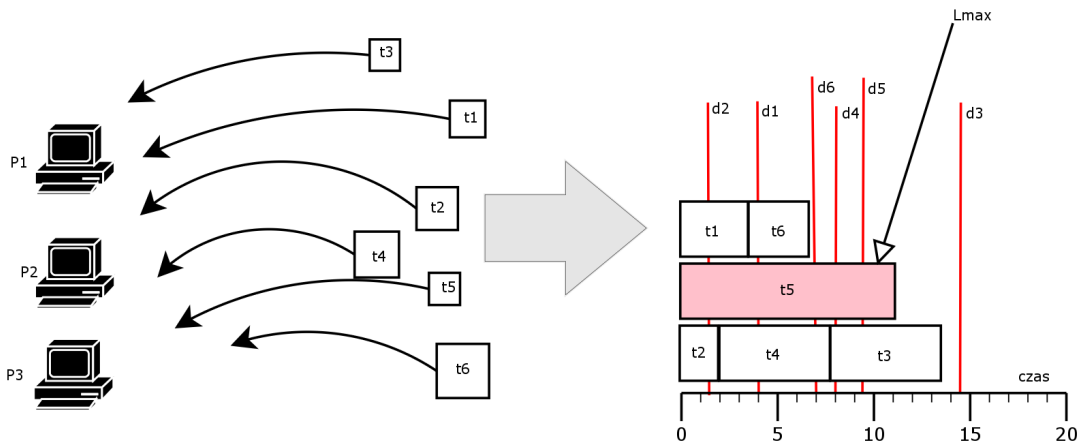
Problem uszeregowania zadań, którego dotyczy sprawozdanie precyzyjnie opisywany, jako

$$P3||L_{max} \quad (1)$$

polega na przydzieleniu zadań t trzem jednakowym procesorom P i uporządkowaniu ich w taki sposób, by maksymalne opóźnienie (L_{max}) dowolnego zadania było jak najmniejsze.

By móc określić opóźnienie l , każde zadanie charakteryzuje się czasem c , jaki jest potrzebny do jego wykonania i terminem d , na kiedy ma zostać wykonane.

Procesory są jednakowe i pracują równolegle, bez przerwy. Nie ma więc różnicy, na którym procesorze będzie wykonane dane zadanie. Również zadania nie są powiązane ze sobą zależnościami, a więc mogą być wykonane w dowolnej kolejności. Po rozpoczęciu zadanie nie może być przerwane.



2 Opis algorytmu

2.1 Wersja 1. Szybkie sortowanie

Przed napisaniem algorytmu, zespół postawił przed sobą trudne zadanie odpowiedzi na pytanie, co jest przyczyną największych opóźnień. Doświadczenie podpowiada, że rozwiązywanie zadań na ostatnią chwilę powoduje, że planowany czas przekracza maksymalny termin i powstaje opóźnienie. Również proste przykłady, po przełożeniu sytuacji rzeczywistych na model oparty na symbolach t (zadanie), d (deadline) i l (opóźnienie), wykazały prawdziwość doświadczeń. Dlatego pierwsza wersja algorytmu w najprostszy sposób sortuje zadania wg. terminu zakończenia d , w taki sposób, że najpilniejsze zadania są wykonywane na początku, a najmniej pilne, na końcu. W implementacji posłużono się algorytmem *quicksort*. W drugiej fazie, algorytm przydzielał kolejno zadania do najwcześniej wolnych procesorów.

Podsumowując, algorytm bierze pod uwagę d i łatwo się skaluje dla różnej liczby procesorów.

2.2 Wersja 2. Najpilniejsze, najpierw dłuższe

Mimo, że bardzo prosty, algorytm w pierwszej wersji dosyć dobrze sobie radził z średnimi losowymi danymi, ujawniła się kolejna zależność. Otóż uporządkowując zadania, wg informacji, na kiedy będą potrzebne, warto również wziąć

pod uwagę fakt, ile zajmą czasu. To znaczy, jeśli zadanie zajmuje dużo czasu, warto je zacząć jeszcze wcześniej, niż zadanie które kończy się podobnie, a zajmuje mniej czasu.

By móc w łatwy sposób sortować wg. takiego kryterium, do definicji zadania dodano pole *d_bezc*, którego warość to po prostu $d_bez c = d - c$.

Po przeprowadzeniu wielu testów (szczegóły w Wynikach) poznane zostały kolejne właściwości algorytmu. Algorytm dawał optymalne, lub bardzo dobre wyniki dla zadań o zbliżonej długości, natomiast tracił skuteczność, gdy zadania miały bardzo różne długości. Zachowanie to było podobne do zachowania pierwszej wersji, tj. wprowadzona zmiana nie przyniosła znaczącej poprawy algorytmu.

2.3 Wersja 3. Tabu Search

Kolejne testy, na coraz bardziej różnorodnych danych, wykazały że algorytm może znajdować L_{max} różniący się od optymalnego nieraz nawet o średnią długość jednego zadania (dla pakietów 7-9 zadań do uszeregowania na 2-3 procesory).

Zainspirowani algorytmem Tabu Search, poznanym na laboratoriach, uznaliśmy że właśnie tego nam brakuje. Przeglądając optymalne uszeregowania zauważyliśmy, że dla jednego zbioru zadań jest wiele rozwiązań o tym samym najlepszym L_{max} . Co więcej, na L_{max} składa się ciąg pewnych zadań, występujących po sobie zawsze w tej samej kolejności. Więc zawsze są co najmniej trzy najlepsze rozwiązania, wynikające z ułożenia takiej specyficznej sekwencji zadań na każdym z procesorów.

Następnie, porównanie z wynikami generowanymi przez nasz prosty algorytm wykazało, że uszeregowania mają zbliżone sekwencje (po których występuje zadanie z L_{max}), nieraz różniące się o jedno lub dwa zadania.

Jest to bardzo dobra sytuacja do wykorzystania algorytmu Tabu Search. Algorytm ten, rozpoczynając od bieżącego uporządkowania poszukuje w najbliższym otoczeniu uporządkowań lepszych, więc coraz bardziej zbliża wynik do optymalnego.

2.4 Opis działania wersji ostatecznej

2.4.1 Struktury danych

Algorytm jako argumenty przyjmuje tablicę zadań, ich liczbę i liczbę dostępnych procesorów. Każde zadanie ma określone *c* i *d*, porę rozpoczęcia (*start* - opisane niżej) oraz wyliczane na ich podstawie *d_bezc* i *l*.

Charakter operacji, jakie są wykonywane na zadaniach, stwarza z jednej strony potrzebę przetwarzania ich, jako tablicy jednowymiarowej - podczas sortowania. Z drugiej natomiast, chcielibyśmy móc łatwo obliczać *l* każdego zadania, po zamianie jego kolejności przez **tabu search**, a więc gdy zmieni się jego czas rozpoczęcia i zaktualizować opóźnienia wszystkich zadań następujących po nim na danym procesorze. Tak więc potrzebny jest dostęp do zadań jednocześnie w jednym wymiarze i z podziałem na procesory.

Dlatego każde zadanie posiada jeszcze pole *next*, ze wskaźnikiem do następnego zadania na tym samym procesorze. Zadanie ostatnie na danym procesorze ma wskaźnik do NULL. Zadanie początkowe rozpoznajemy po tym, że czas rozpoczęcia (*start*) ma wartość 0.

Poza zadaniami, swoje struktury mają także procesory. Są one wykorzystywane podczas działania algorytmu. Dla każdego procesora zapamiętywane jest, od kiedy jest wolny - pole *wolny_od*, oraz wskaźnik do ostatnio przydzielonego mu zadania - pole *last* - pole jest wykorzystywane przy tworzeniu wskaźników *next* pomiędzy zadaniami na tych samych procesorach.

2.4.2 Warunki początkowe

Tworzone jest *lproc* procesorów, wolnych od chwili 0. W tablicy zadań, określoną wartość mają tylko pola *num*, *c*, *d*, *d_bezc*.

2.4.3 Opis działania

1. Pierwszym etapem jest posortowanie zadań wg. pola *d_bezc* algorytmem **quicksort**. Po sortowaniu tablica zawiera zadania uporządkowane od najmniejszego do największego *d_bezc*.
2. Zadania są po kolei przydzielane procesorom wg. zasady, który procesor najwcześniej jest wolny, ten dostaje zadanie. Uzupełniane są więc pozostałe pola *start*, *l* i *next* w strukturach zadań.
3. Uruchamiany jest algorytm **Tabu search** z informacjami takimi, jak zbiór zadań, oraz wielkość tablicy Tabu. W ramach algorytmu **Tabu search**:

- (a) obliczane jest bieżące L_{max} dla danego uporządkowania.
- (b) dla każdej pary zadań $t1$ i $t2$ należących do zbioru zadań, jeśli dane podstawienie nie jest na liście tabu, obliczane jest L_{max} , jakie byłoby po zamianie tych zadań miejscami. Jednocześnie zapamiętywane jest, która zamiana dwóch zadań dałaby najmniejszy L_{max} .
- (c) Zamiana, przynosząca najlepszą poprawę L_{max} jest wprowadzana w życie, oraz dodawana do listy tabu. Zamiany są dokonywane tak długo, jak przynoszą jakąkolwiek poprawę.

Algorytm zwraca znaleziony L_{max} , natomiast tablica z zadaniami zawiera ostateczne uporządkowanie.

2.4.4 Szczegóły implementacji: Lista Tabu

Funkcje listy Tabu spełnia tablica o wielkości *tabusize* podawanej podczas wywoływania algorytmu. Do jej obsługi służą funkcje: *in_tabu* oraz *add_to_tabu*. Z listą tabu jest także związany indeks, wskazujący numer komórki w tablicy, do którego wpisano najdawniejszą zamianę zadań. Początkowo jest to wartość 0, czyli początek tablicy. Po dokonaniu zamiany, nowa para zadań jest dodawana do tablicy w miejscu wskazywanym przez indeks, a indeks jest zwiększany o 1, modulo *tabusize*, by nie wyszedł poza zakres tablicy. W ten sposób, gdy, przy *tabusize* = 5, indeks wskazuje wartość 1, nowa zmiana zostanie dodana w komórce 1, a indeks zwiększony do 2. Po czasie, gdy indeks wskazuje wartość 4, nowa zamiana zostanie zapisana w komórce 4, a indeks przejdzie do 0. Tym samym, najstarsza podmiana (w komórce 0) zostanie wkrótce nadpisana. Początkowo lista tabu wypełniona jest zerami, gdyż podmiany elementów 0 na 0 i tak nie mogą się zdarzyć, więc jest tak, jakby lista nie zawierała żadnej podmiany.

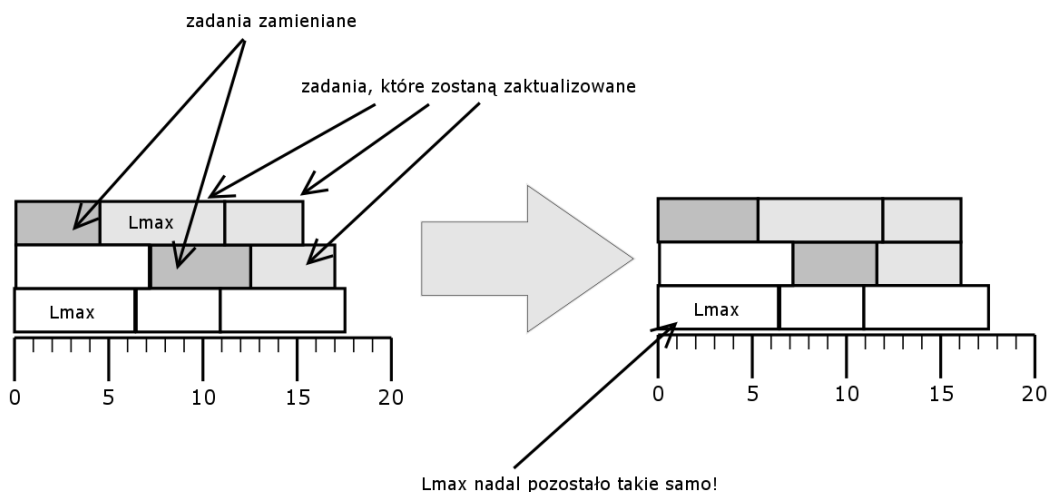
2.4.5 Szczegóły implementacji: Zamiana zadań

Wielokrotnie wspomniana wyżej zamiana dowolnych, różnych od siebie, zadań miejscami to:

- zamiana miejscami pól *c*, *d*, *d.bezc* i *num*. Natomiast pole *l* musi być na nowo policzone, z uwzględnieniem nowych wartości zamienionych miejscami pól, oraz pola *start*.
- Zmiana zadania powoduje także, że trzeba zaktualizować czasy wszystkich późniejszych zadań na danym procesorze. Dotyczy to obu zadań, które zamieniamy miejscami.

2.4.6 Uzasadnienie wielokrotnego obliczania L_{max}

W każdej iteracji Tabu search przed i po zamianie zadań L_{max} obliczane jest od nowa poprzez przeszukanie całego zbioru zadań. Wydaje się, że możnaby przejść tylko pola *l* tych zadań, które się zmieniły, jednak istnieje możliwość, że w zbiorze zadań występuje kilka zadań o takim samym *l*, będącym jednocześnie L_{max} . Wówczas, gdy po dokonaniu zmian, tylko jedno z tych zadań będzie miało nowe, mniejsze *l*, to drugie zadanie nadal będzie miało największe *l* dla całego zbioru.



2.4.7 Nadmiarowość danych

Zadania nie muszą mieć pola *num* ze swoim numerem, jednak pozostawiono je ze względu na późniejszą łatwość wygenerowania graficznej reprezentacji szeregowania, łatwo czytelnej dla ludzkiego oka. Prostą ilustrację bieżącego uporządkowania wyświetla funkcja *stan*.

2.4.8 Optymalizacje

1. Wiele uwagi poświęcono temu, w jaki sposób dla jednowymiarowej tablicy oddać charakter dwuwymiarowej tablicy, gdzie jednym wymiarem jest czas, a drugim liczba procesorów. Rozważano m.in. wersje:
 - (a) zadania dla 1 procesora na początku tablicy, zadania dla drugiego procesora na końcu tablicy. Powstało jednak pytanie, gdzie wstawić zadania z 3 procesora, lub jak zrobić algorytm dla większej ilości procesorów.
 - (b) stworzyć od razu tablicę dwuwymiarową. Jednak, stworzenie tablicy o wielkości *liczba_procesorów*liczba_zadań* byłoby bardzo nadmiarowe, bo potrzebna jest tylko tablica o wielkości *liczba_zadań*.
 - (c) każde zadanie ma numer procesora, na jakim jest wykonywane. Wówczas i tak trzeba było jednak przechodzić wiele elementów tablicy, które były przypisane do nie interesujących nas procesorów i niepotrzebnie je sprawdzać.
2. W trakcie tabu search, przy podmianach, pole *d_bezc* nie jest kopiowane, gdyż było ono potrzebne tylko na etapie sortowania.

2.5 Przykłady wykonania algorytmu

2.5.1 Badanie rozwiązań przy użyciu różnych metod

Zadanie: Do 3 równoległych procesorów należy przydzielić 5 zadań tak, aby największe opóźnienie (L_{max}) było jak najmniejsze. Zadania mają następujące dane:

```
t0: c0=50 d0=50
t1: c1=25 d1=26
t2: c2=1 d2=7
t3: c3=3 d3=25
t4: c4=5 d4=20
```

Najprostszym rozwiązaniem problemu jest posortowanie zadań wg. ustalonego kryterium, a następnie przydzielenie do procesorów. Sortowanie może się odbywać wg. oczekiwanych terminów zakończenia (*d*) lub wg. czasów trwania zadań (*c*) albo też wg. różnic między czasami trwania zadań a ich oczekiwanymi czasami zakończenia (*d_bezc*):

1. Zadania są sortowane wg. parametru *d*. Otrzymany ciąg zadań jest następujący: *t2, t4, t3, t1, t0*. Wynik przydzielenia zadań do procesorów przedstawiono poniżej:

```
0 (t2 c=1 d=7 l=-6 s=0) (t1 c=25 d=26 l=0 s=1)
1 (t4 c=5 d=20 l=-15 s=0)
2 (t3 c=3 d=25 l=-22 s=0) (t0 c=50 d=50 l=3 s=3)
```

Nie jest to jednak rozwiązanie optymalne. $L_{max} = 3$.

2. Sortowanie odbywa się wg. parametru *c*. Zadania są porządkowane od najszybszego do najwolniejszego. Po posortowaniu kolejność zadań jest następująca: *t2, t3, t4, t1, t0*. Po przydzieleniu zadań do procesorów otrzymujemy następujące wyniki:

```
0 (t2 c=1 d=7 l=-6 s=0) (t1 c=25 d=26 l=0 s=1)
1 (t3 c=3 d=25 l=-22 s=0) (t0 c=50 d=50 l=3 s=3)
2 (t4 c=5 d=20 l=-15 s=0)
```

Największe maksymalne opóźnienie jest szukane poprzez sprawdzanie opóźnień wszystkich zadań. Jeśli algorytm trafi na rozwiązanie lepsze od dotychczasowego, zapamiętuje je jako najlepsze L_{max} . Dla takiego rozwiązania $L_{max}=3$. Widzimy, że ustawienie zadań dla tego przykładu jest takie samo jak przy sortowaniu wg. kryterium oczekiwanego czasu zakończenia zadania (*d*), inne są tylko procesory na których zadania są ustawione.

3. Zadania można także uporządkować wg. różnicy między czasem trwania a oczekiwanym czasem zakończenia (*d.bezc*). Po wykonaniu algorytmu **quicksort** otrzymano ciąg zadań: t_0, t_1, t_2, t_4, t_3 . Każde z zadań jest umieszczane na pierwszym najwcześniej dostępnym procesorze. W wyniku takiego algorytmu otrzymujemy rozwiązanie:

```
0 (t0 c=50 d=50 l=0 s=0)
1 (t1 c=25 d=26 l=-1 s=0)
2 (t2 c=1 d=7 l=-6 s=0) (t4 c=5 d=20 l=-14 s=1) (t3 c=3 d=25 l=-16 s=6)
```

Sprawdziwszy opóźnienia wszystkich zadań okazuje się, że większe opóźnienie wynosi 0 i jest lepsze od poprzednich rozwiązań.

2.5.2 Użycie Tabu Search

Zadanie:

```
t0: c0=2 d0=2
t1: c1=20 d1=20
t2: c2=40 d2=40
t3: c3=5 d3=7
t4: c4=2 d4=5
t5: c5=1 d5=10
```

- Algorytm sortuje zadania wg. *d.bezc*. Po posortowaniu zadania uporządkowane są w kolejności: $t_5, t_4, t_1, t_0, t_2, t_3$.
- Zadania są przydzielane do procesorów stosując zasadę, że przydzielamy zadanie do pierwszego najwcześniej dostępnego procesora.

Na początku każdy procesor jest dostępny, więc pierwsze trzy zadania są przydzielane do kolejnych procesorów:

```
0 (t5 c=40 d=40 l=0 s=0)
1 (t4 c=20 d=20 l=0 s=0)
2 (t1 c=2 d=2 l=0 s=0)
```

Następnie szukamy pierwszego najwcześniej dostępnego procesora. W programie została w tym celu wprowadzona zmienna `procesory[i].wolny_od`.

```
1 (t2 c=29 d=129 l=-100 s=0) (t1 c=17 d=100 l=-54 s=29) (t4 c=161 d=188 l=19 s=46)+
2 (t3 c=124 d=132 l=-8 s=0)+
```

Oznacza to, że być może i kolejna iteracja petli w algorytmie **Tabu Search** poprawi uszeregowanie. Jednak po jej wykonaniu okazuje się, że nie znaleziono lepszego uporządkowania. Powyższe rozwiązanie jest najlepszym znalezionym uporządkowaniem.

Program **wynik**, zwrócił maksymalne opóźnienie $L_{max} = 9$. Lepszy, od znalezionego przez algorytm.

3 Wyniki

Program był testowany w każdej z wersji. Do testowania zostały dodatkowo napisane programy **wynik** oraz **generator**.

Generator produkuje pliki wejściowe ze zbiorami zadań do uszeregowania. Podając odpowiednie parametry przy wywoływaniu go, można określić liczbę zadań, które zostaną wygenerowane (domyślnie 10), średnią długość zadań (domyślnie 10), maksymalną różnicę długości poszczególnych zadań od długości średniej. Można także określić minimalny i maksymalny termin zakończenia zadań.

Wynik to program rozwiązujący dane zadanie szeregowania poprzez sprawdzenie wszystkich permutacji zadań wejściowych. Porównując wyniki tego programu oraz programu szeregującego, ocenialiśmy jakość algorytmu. Ze względu na dużą złożoność obliczeniową ($O(n!)$) sprawdzano zbiory o 7-9 zadaniach. Dla większej ilości zadań, program działał zbyt długo.

Do kompleksowego przetestowania algorytmu użyto prostego skryptu powłoki Linuksa:

```
#!/bin/sh

tests_count=100
timeout_min=1
timeout_max=500

for ltasks in 7 8 9; do
  for clen in 10 20 50 100 200; do
    for crdiv in 10 5 3 1; do
      crandomness='echo $(( $clen / $crdiv ))'
      echo ./test.sh $tests_count -l $ltasks -c $clen -r $crandomness -t $timeout_min -u $timeout_max
      ./test.sh $tests_count -l $ltasks -c $clen -r $crandomness -t $timeout_min -u $timeout_max
    done;
  done ;
done
$
```

Czyli łącznie 60 zestawów po 100 testów, kolejno dla zbiorów po 7, 8 i 9 zadań, o średnich długościach 10, 20, 50, 100, 200, takich, że długości różnią się o 10%, 20%, 33%, 100% względem średniej długości zadania. Skrypt `test.sh` wywołuje po prostu programy `generator`, `wynik` i `szereguj` \$tests_count razy i porównuje wynik szeregowania z wynikiem optymalnym.

3.1 Algorytm a rozwiązanie optymalne

Pierwszy test, to sprawdzenie, dla jakich zestawień parametrów, uporządkowania nie są optymalne. Na 60 zestawów, w 22 były błędne uporządkowania.

lp.	<i>lproc</i>	<i>c</i>	$\pm c$	l.błędów(%)
1	7	100	20	1
2	7	100	33	2
3	7	100	100	2
4	7	200	20	7
5	7	200	40	7
6	7	200	66	12
7	7	200	200	19
8	8	50	50	1
9	8	100	33	2
10	8	100	100	6
11	8	200	20	17
12	8	200	40	13
13	8	200	66	18
14	8	200	200	26
15	9	50	50	2
16	9	100	20	2
17	9	100	33	1
18	9	100	100	8
19	9	200	20	14
20	9	200	40	14
21	9	200	66	21
22	9	200	200	36

Dodatkowo przeprowadzono testy:

lp.	l_{proc}	c	$\pm c$	l.błędów(%)
1	8	500	50	12
2	8	500	100	30
3	8	500	166	40
4	8	500	500	42
5	8	1000	100	27
6	8	1000	200	34
7	8	1000	333	37
8	8	1000	1000	50

Gdzie c w tabeli to średnia długość zadania, a $\pm c$ to maksymalna wartość, o jaką długość zadania może się różnić od długości średniej. A więc dla $c = 1000$ i $\pm c = 1000$, zadania mogą mieć długość w przedziale $[0, 2000]$.

Algorytm znajduje więc rozwiązanie optymalne dla małych zbiorów zadań, lub dla zbiorów z zadaniami o zbliżonej długości. Im więcej zadań, oraz im większa różnorodność czasu trwania, tym mniej jest wyników optymalnych.

3.2 Różnica między rozwiązaniami, a OPT

Wiedząc już, kiedy wynik algorytmu nie jest optymalny, powstaje pytanie, na ile nie jest on różny od poprawnego. Wyniki testu odpowiadającego na to pytanie przedstawia poniższa tabela.

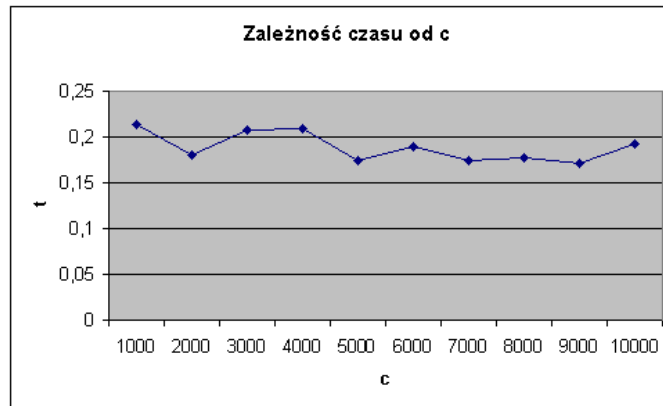
l_{proc}	c	$\pm c$	$avg\Delta$	$\frac{avg\Delta}{c}$
7	100	20	-2,00	0,02
7	100	33	-8,50	0,09
7	100	100	-4,50	0,05
7	200	20	-9,14	0,05
7	200	40	-8,43	0,04
7	200	66	-14,17	0,07
7	200	200	-44,05	0,22
8	50	50	-3,00	0,06
8	100	33	-2,50	0,03
8	100	100	-11,67	0,12
8	200	20	-5,00	0,03
8	200	40	-10,15	0,05
8	200	66	-13,83	0,07
8	200	200	-22,31	0,11
9	50	50	-9,50	0,19
9	100	20	-3,50	0,04
9	100	33	-4,00	0,04
9	100	100	-13,88	0,14
9	200	20	-7,00	0,04
9	200	40	-7,50	0,04
9	200	66	-16,10	0,08
9	200	200	-31,64	0,16
8	500	50	-11,75	0,02
8	500	100	-23,60	0,05
8	500	166	-29,60	0,06
8	500	500	-45,33	0,09
8	1000	100	-21,59	0,02
8	1000	200	-33,03	0,03
8	1000	333	-68,78	0,07
8	1000	1000	-79,68	0,08

Kolumna $avg\Delta$ podaje średnią (cały czas trzeba pamiętać, że jeden wiersz to jeden zestaw 100 zbiorów zadań) różnicę L_{max} między rozwiązaniami, a OPT. Bardziej interesująca jest ostatnia kolumna $\frac{avg\Delta}{c}$, czyli stosunek $avg\Delta$ do średniej długości zadań. Wniosek z analizy tej kolumny jest taki, że błąd algorytmu wynosi ok. $\frac{1}{10}c$.

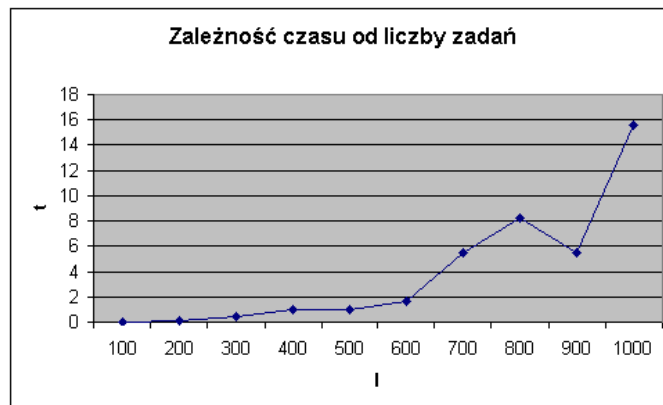
3.3 Testy wydajności algorytmu

Aby określić, które czynniki mają wpływ na szybkość działania algorytmu, wykonano pomiary szybkości działania w zależności od zmiany jednej zmiennej. Poniżej zostały umieszczone wykresy zależności czasu od zmiany długości zadania (c), od liczby zadań (l) oraz od $\pm c$. Argumenty pozostałe, tj. minimalny i maksymalny czas skończenia zostały pominięte, gdyż mają podobny wpływ, jak $\pm c$. Jest tak dlatego, że zmienna d jest wykorzystywana zawsze wtedy, gdy c (podczas obliczania $d_{bez c}$, obliczania l , podczas **exchange**).

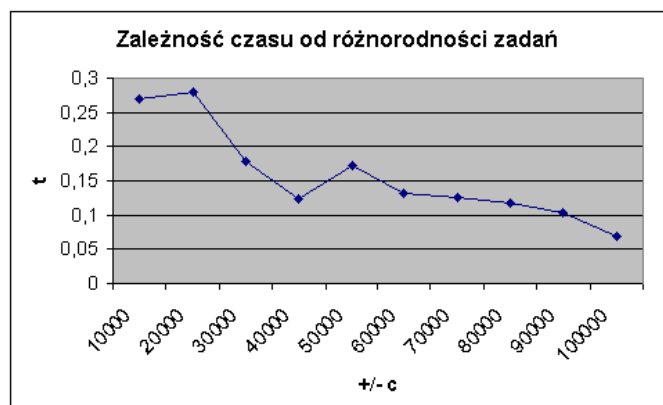
Przedziały, dla których wykonano pomiary, zostały dobrane doświadczalnie, by możliwie wyraźnie przedstawić zależność między zmiennymi, a czasem. Czas na wykresach jest podawany w sekundach. Każdy pomiar przeprowadzono kilka razy, wybrano wartości największe.



Zależność jest stała, a więc algorytm może szeregować zadania o dowolnej długości w zbliżonym czasie.



Zależność czasu od liczby zadań jest z kolei wielomianowa. Ponieważ zależność ta jest wyraźnie widoczna już dla „niewielkich” zbiorów zadań, można przypuszczać, że nie wynika ze złożoności sortowania (różnice czasu quicksort dla zbiorów o 100, a o 1000 elementów są minimalne), a z metody przydzielania procesora, lub Tabu search.

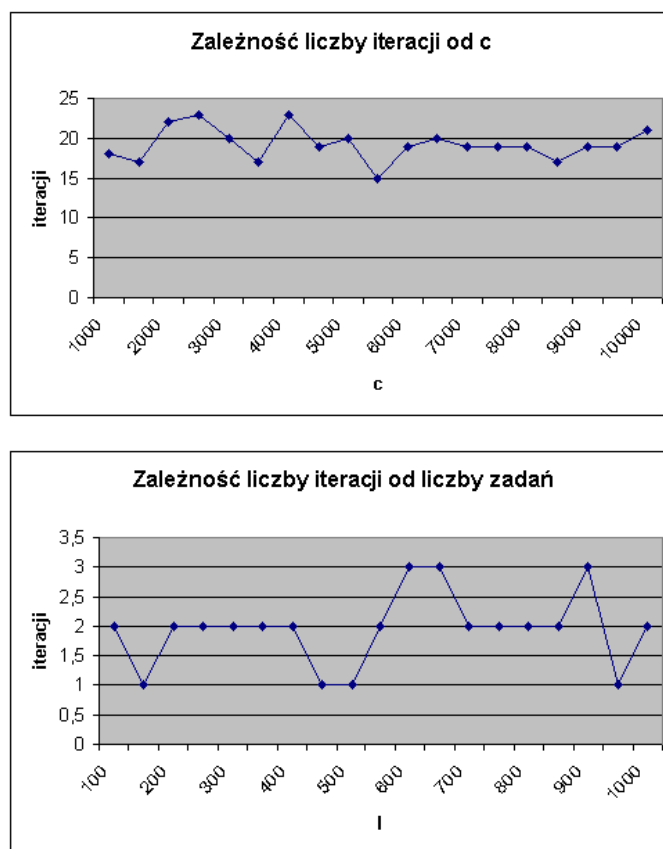


Testy przeprowadzono dla zadań o $c = 100000$. Zależność czasu od $\pm c$ jest również wielomianowa, lub nawet liniowa. Jednak ma dużo mniejszy współczynnik, niż w poprzednim przypadku, gdyż staje się widoczna dopiero dla 100krotnie większych problemów. Co ciekawe, algorytm ma większe problemy ze znalezieniem rozwiązania dla zadań o zbliżonych długościach ($c = [90000, 110000]$), niż dla zadań bardzo różnorodnych ($c = [0, 200000]$).

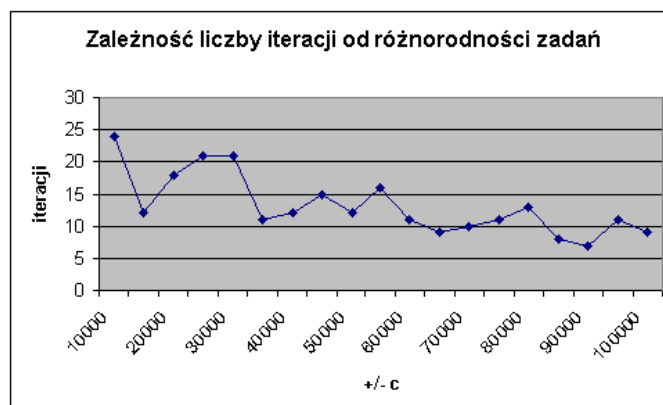
3.4 Analiza działania Tabu Search

Działanie **Tabu Search** analizowane było poprzez zliczanie liczby iteracji, potrzebnych do rozwiązania określonych problemów. Liczba ta mówi nam, jak wiele razy trzeba było poprawiać wynik po sortowaniu, a więc jak dobry jest sposób sortowania. Podobieństwa w poniższych zależnościach i poprzednich zależnościach od czasu, pozwolą stwierdzić, czy to **Tabu search** ma kluczowy wpływ na złożoność obliczeniową algorytmu.

Przedziały, dla których wykonano pomiary, ponownie zostały dobrane doświadczalnie, by możliwie wyraźnie przedstawić zależność między zmiennymi, a liczbą iteracji. Każdy pomiar przeprowadzono kilka razy, wybrano wartości największe. Aby móc liczyć liczbę iteracji, skompilowano program sortuj ze zdefiniowaną zmienną **VERBOSE**.



Na podstawie dwóch powyższych wykresów, działanie **Tabu search** nie jest zależne od ani od liczby zadań.



Zależność na powyższym wykresie jest podobna do zależności czasu od różnorodności zadań. A więc to **Tabu search** zajmuje więcej czasu dla zadań o podobnej długości, niż dla zupełnie różnych.

3.5 Złożoność obliczeniowa algorytmu

Przeprowadzono analizę średniej i pesymistycznej złożoności czasowej. Poprzez n oznaczono liczbę zadań, natomiast l - liczbę procesorów. Średnia złożoność, wynikająca z utworzenia l procesorów (l), średniej złożoności sortowania ($n \log n$), przydzielenia n zadań do l procesorów (nl) i złożoności **Tabu search** ($k(n^2 * exchange + exchange)$).

$$O(l + n \log n + nl + k(n^2 * exchange + exchange)) \quad (2)$$

Określenie złożoności **Tabu search**, samo w sobie jest wyzwaniem, dlatego posłużono się zmiennymi k (liczba iteracji **Tabu search**) i $exchange$ (złożoność funkcji **exchange**). Niestety zespół nie był w stanie w sposób analityczny określić średniej liczby iteracji, dlatego korzystając z analizy doświadczalnej (poprzednie wykresy), przyjęto że $k = \pm c$.

Natomiast złożoność **exchange** wynika z tego, ile jest zadań do zaktualizowania ($update_l$), po tym, które akurat zmieniamy, na tym samym procesorze. Dotyczy 2 zamienianych miejscami zadań ($O(2update_l)$). Wiedząc, że przydział zadań do procesorów, w typowych przypadkach prowadzi do równomiernego podziału liczby zadań na trzy procesory, $update_l$, należy do przedziału $[0 - \frac{1}{3}n]$, więc średnio $update_l = \frac{1}{6}n$, stąd $O(exchange) = O(2update_l) = O(2 * \frac{1}{6}n) = O(\frac{1}{3}n)$. Ostateczna postać wzoru:

$$O(l + n \log n + nl + (\pm c)(n^2 * \frac{1}{3}n + \frac{1}{3}n)) = O(l + n \log n + nl + (\pm c)\frac{1}{3}(n^2 + 1)n) = O(n^4) \quad (3)$$

W najgorszym przypadku, złożoność sortowania wynosi $O(n^2)$, oraz może zajść przypadek, że **Tabu search** dokona zamian dla wszystkich elementów ($k = n$). Natomiast najgorszą sytuacją podczas wykonywania **exchange** jest poprawienie $n - 2$ elementów, w sytuacji gdy wszystko jest na jednym procesorze, poza 2 zadaniami blokującymi dwa pozostałe procesory. Nie może się zdarzyć, że jeden procesor będzie miał wszystkie (więcej niż 1 zadanie), gdy inny procesor nie ma żadnych zadań. Złożoność dla przypadku pesymistycznego:

$$O(l + n^2 + nl + n(n^2 * 2(n - 2) + 2(n - 2))) = O(n^4) \quad (4)$$

Nawet wówczas algorytm pozostaje wielomianowy, a więc warunek zadania pozostaje spełniony.

4 Wnioski

1. Dekomponując trudny problem na szereg mniejszych, można go rozwiązać stosując algorytmy do tych mniejszych problemów.
2. Algorytmy aproksymacyjne, mimo że w większości przypadków dają dobre wyniki, mogą nieraz się potknąć i dla pewnych specyficznych danych znaleźć bardzo słaby wynik. Jednak tak, czy inaczej zrobią to w rozsądnym czasie.