Algorytm zachłanny dla szeregowania zadań równoległych z przyległymi przydziałami procesorów

Paweł Kierkosz 155995
Kamil Bryła 155988
E-mail do kontaktu – pawel.kierkosz@student.put.poznan.pl

1. Opis problemu

Ogólnie mówiąc, rozważany w sprawozdaniu problem polega na przydziale pewnej liczby zadań do wykonania na wielu procesorach równoległych, w taki sposób, aby suma czasów zakończenia wszystkich zadań była możliwie najmniejsza. Przyjmijmy oznaczenia oraz terminologię, jakie stosowane są w wybranych publikacjach [2, 5]. Niech n oznacza liczbę zadań, które mają zostać wykonane, a $T = \{T_1, T_2, \ldots, T_n\}$ niech będzie zbiorem tych zadań. Z kolei przez m oznaczmy liczbę procesorów, a przez $P = \{P_1, P_2, \ldots, P_m\}$ zbiór równoległych procesorów. Dalej wprowadźmy następujące oznaczenia:

r_j – moment gotowości zadania T_j, tj. najszybszy możliwy czas jego rozpoczęcia,

p_j – czas trwania zadania T_j (długość zadania),

$size_j$ lub s_j – liczba procesorów przydzielonych do zadania T_j (rozmiar lub szerokość zadania).

Zakładamy ponadto, że procesory są identyczne (jednakowo szybkie), a zadania przydzielane są do procesorów w sposób przyległy (indeksy procesorów przypisanych do każdego z zadań muszą być kolejnymi liczbami naturalnymi) oraz ciągły (zadanie już rozpoczęte musi być realizowane w sposób nieprzerwany przez p_j jednostek czasu na takim samym zestawie procesorów).

Każdy sposób przydziału zadań do procesorów spełniający wymagane warunki będziemy nazywać harmonogramem lub uszeregowaniem zadań. Jako funkcję celu, czyli kryterium optymalizacji, przyjmiemy sumę czasów zakończenia wszystkich zadań, którą będziemy oznaczać przez $SumC_j$. Często stosowanym kryterium jest czas realizacji harmonogramu (czyli jego długość) oznaczany przez C_{max}. Oba te kryteria podlegają minimalizacji.

2. Opis algorytmu

Przy spełnieniu pewnych warunków problem szeregowania zadań równoległych może być utożsamiany z problemem dwuwymiarowego pakowania lub rozkroju pasa (SPP - Strip packing problem) polegającym na upakowaniu zbioru prostokątnych elementów na prostokątnym pasie o skończonej szerokości, ale nieskończonej długości, w taki sposób, aby zużycie materiału było jak najmniejsze [3, 6]. Z literatury znane są różne metody optymalizacji tego zagadnienia. Wiele z nich bazuje na heurystycznych metodach rozmieszczania kolejnych elementów na pasie. Z bardziej znanych można wymienić algorytmy: bottom-left (BL), bottom-left-fill (BLF), best-fit (BF) [1, 3, 4]. Wspólną cechą tych

strategii jest zasada rozmieszczania kolejnych elementów w najniżej położonym punkcie spośród najbardziej na lewo wysuniętych punktów wolnego obszaru na pasie, w którym nie rozmieszczono jeszcze innych elementów. Różnice natomiast związane są z tym, czy najpierw wybierany jest element, a następnie punkt w którym jest on umieszczany (BL, BLF), czy też kolejność jest odwrotna (BF) oraz w sposobie wypełniania „luk" powstających podczas rozmieszczania elementów na pasie.

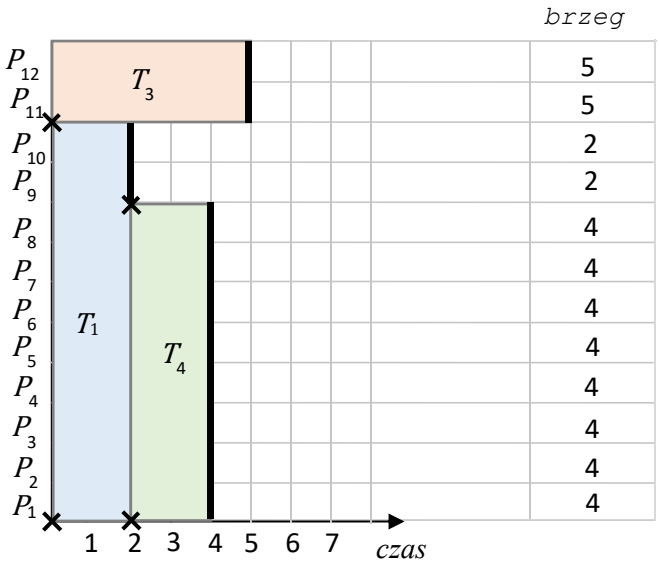
My w swoim algorytmie zdecydowaliśmy się użyć zmodyfikowanej przez nas heurystyki best-fit opisanej w pracy [3]. Działanie algorytmu zostanie zilustrowane na następującym przykładzie:

Przykład 1. Mamy 12 identycznych procesorów i na nich ma zostać wykonanych 5 zadań. Parametry tych zadań podane są w poniższej tabelce.

T	r_j	p_j	$size_j$
T_1	0	2	10
T_2	0	4	4
T_3	0	5	2
T_4	0	2	8
T_5	1	2	6

W naszym algorytmie zachłannym, dla przyjętego kryterium optymalizacji ($SumC_j$.), o kolejności przydzielania zadań do procesorów będzie decydował czas ich trwania. Im krótszy czas (mniejsza długość), tym wcześniej dane zadanie może zostać wykonane.

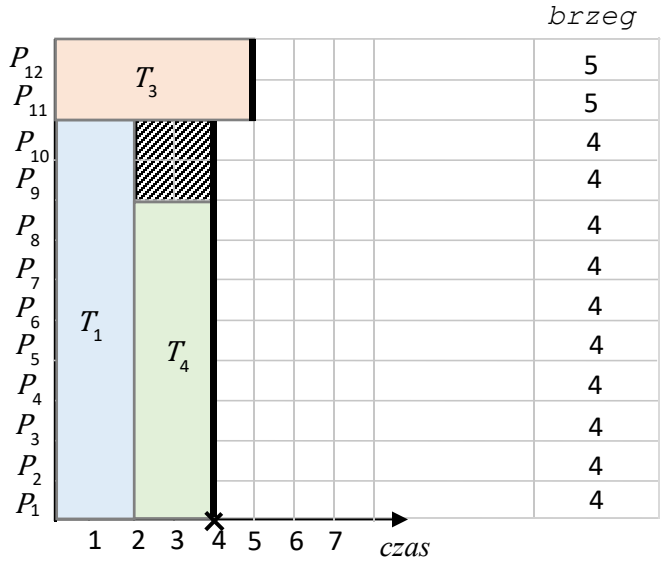
W każdej iteracji algorytmu, w której do procesorów przypisywane jest zadanie (na pasie ustawiany jest prostokątny element) ewentualnie przestój, aktualizowane są wartości wektora zawierającego *m* liczb, który dalej nazywać będziemy *brzegiem*. Zawarte są w nim informacje dotyczące brzegu aktualnego harmonogramu częściowego (jeszcze nie ukończonego), czyli czasu pracy każdego z *m* procesorów. W źródłowej pracy [3] brzeg ten nazywany jest „skyline".



Rys. 1. Ilustracja pojęcia brzegu oraz punktu zaczepienia

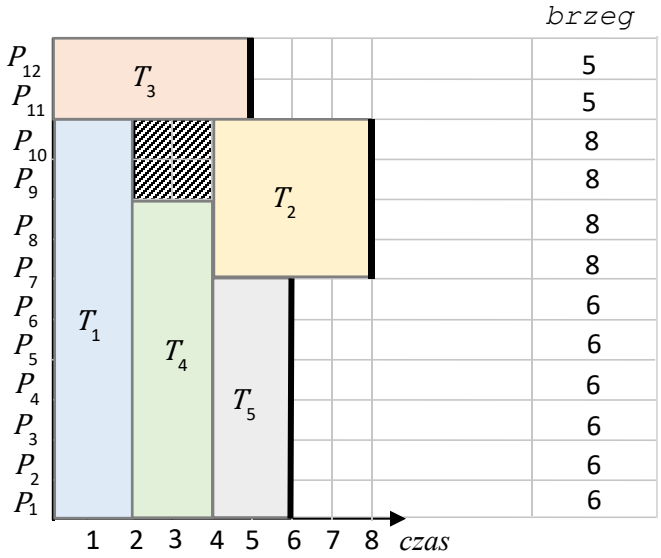
Rysunek 1 przedstawia przykładowy harmonogram częściowy dla przykładu 1, odpowiadający mu brzeg (pogrubione krawędzie) oraz jego reprezentacja w postaci wektora. Na początku algorytmu, gdy do procesorów nie są przydzielone jeszcze żadne zadania, brzeg jest wektorem składającym się z samych zer.

Drugą bardzo ważną zmienną wektorową używaną w programie jest *pkt_zaczep.* Przechowywane są w niej współrzędne tzw. punktu zaczepienia (oznaczone krzyżykami na rysunku 1), tj. punktu, w którym znajdzie się dolny lewy wierzchołek rozmieszczanego elementu, ewentualnie przestoju. Jest on wyznaczany w każdej iteracji algorytmu dla aktualnego brzegu. Jego pierwsza współrzędna *x* jest równa najmniejszej wartości zapisanej w wektorze *brzeg*, a druga (*y*) jest równa indeksowi (indeksujemy od 0) pierwszej komórki, w której to *x* występuje. Dla układu z rysunku 1 i odpowiadającemu mu brzegowi, punkt zaczepienia będzie miał współrzędne (2, 8). Wcześniejsze punkty zaczepienia miały współrzędne: (0, 0), (0, 10), (2,0). Patrząc na układ z rysunku 1 łatwo można stwierdzić, że w punkcie zaczepienia (2, 8) nie można „zaczepić" już żadnego z pozostałych dwóch elementów: T_2 i T_5, ponieważ otrzymalibyśmy rozwiązanie niedopuszczalne – elementy nachodziłyby na siebie bądź wychodziły poza krawędź pasa. W takim przypadku w powstałą „lukę" (też nazywaną „niszą") wstawiamy tzw. *przestój.* W graficznej interpretacji harmonogramu przestój taki będzie prostokątem o szerokości równej szerokości luki oraz takiej długości, aby krawędź przestoju zrównała się niższą krawędzią sąsiadujących z luką elementów. Jeżeli luka przylega do górnej lub dolnej krawędzi pasa, to przestój wyrównujemy do krawędzi sąsiadującego z luką elementu. Na rysunku 2 przedstawiono układ z rysunku 1 uzupełniony o przestój oraz nowo wyznaczony brzeg i kolejny punkt zaczepienia.



Rys. 2. Wstawianie przestoju do harmonogramu

Po wstawieniu do układu z rysunku 2 dwóch nierozmieszczonych jeszcze elementów otrzymalibyśmy końcowy już harmonogram zamieszczony na rysunku 3 oraz odpowiadający mu brzeg.



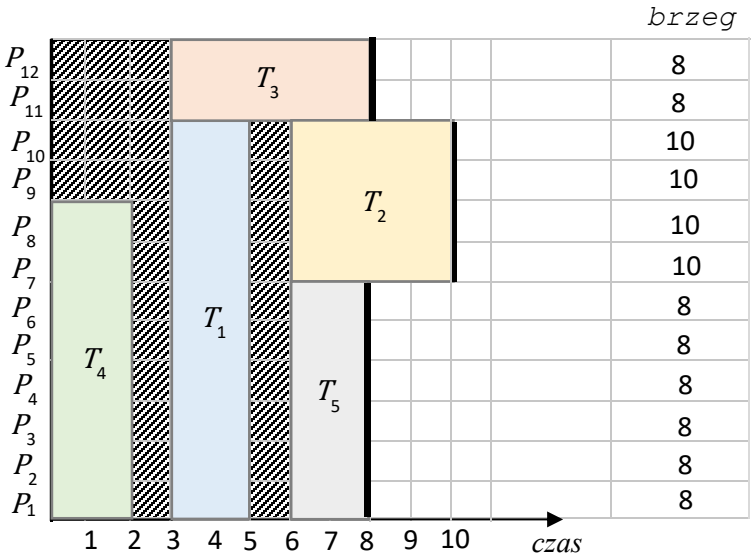
Rys. 3. Harmonogram otrzymany do przykładu 1

Sumując teraz czasy zakończenia kolejno przydzielanych zadań: 2 + 5 + 4 + 6 + 8 = 25 otrzymujemy wartości funkcji celu, czyli w tym przypadku $SumC_j = 25$. Warto zauważyć, że korzystając z danych zapisanych w ostatnim wektorze *brzeg* łatwo można odczytać wartość C_{max} dla danego harmonogramu. Jest to po prostu największa występująca w nim wartość, tj. w naszym przykładzie $C_{max} = 8$. Analizując układ przedstawiony na rysunku 3 można również zauważyć, że wyznaczony harmonogram nie przedstawia rozwiązania optymalnego. Lepsze rozwiązanie, z wartością $SumC_j = 23$ oraz $C_{max} = 7$, otrzymalibyśmy rozmieszczając zadania w kolejności: T_4, T_2, T_3, T_5, T_1. Wynika to z faktu, że algorytm zachłanny jest algorytmem przybliżonym i nie zawsze daje rozwiązania optymalne.

Konieczność wstawienia przestoju może być związana nie tylko z niedopuszczalnym nachodzeniem na siebie elementów bądź wystawaniem elementu poza krawędzie pasa, ale może wynikać również z ograniczenia w postaci momentów gotowości poszczególnych zadań. Może mianowicie zdarzyć się sytuacja, że żadnego elementu nie można wstawić w danym puncie zaczepienia (pomimo, że spełnione są wspomniane wcześniej warunki geometrycznej dopuszczalności), ponieważ nie pozwalają na to momenty gotowości pozostałych do rozmieszczenia zadań. Sytuację taką ilustruje harmonogram z rysunku 4, otrzymany dla zmodyfikowanej wersji poprzedniego przykładu:

Przykład 2. Liczba procesorów – 12, liczba zadań – 5 oraz:

T	r_j	p_j	$size_j$
T_1	3	2	10
T_2	6	4	4
T_3	3	5	2
T_4	0	2	8
T_5	6	2	6



Rys. 4. Harmonogram otrzymany do przykładu 2

Układy tego typu jak ten przedstawiony na rysunku 4 nie występują w przypadku problemów pakowania i rozkroju. Dlatego też musieliśmy zmodyfikować podejście stosowane w tych zagadnieniach, tak aby nasz algorytm uwzględniał momenty gotowości poszczególnych zadań.

W każdej iteracji opracowanego przez nas algorytmu zachłannego wykonywane będą następujące operacje:

1. Dla danego rozmieszczenia elementów (informacja, które elementy zostały już na pasie rozmieszczone przechowywana jest w zmiennej wektorowej *zadania_przydzielone*) wyznaczamy brzeg, a następnie punkt zaczepienia.
2. Spośród elementów, które jeszcze nie zostały rozmieszczone na pasie (wektor: *zadania_do_przydzielenia)* wyznaczamy zbiór tych wszystkich elementów, które mogą być zaczepione w danym punkcie (wektor: *zadania_w_punkcie_zaczepienia*). W tym celu sprawdzamy, czy dany element nie zachodzi na inne elementy, nie wychodzi poza krawędź pasa oraz czy wartość r_j danego elementu pozwala na jego rozmieszczenie w danym punkcie zaczepienia.
3. Sprawdzamy, czy zbiór *zadania_w_punkcie_zaczepienia* jest pusty i w zależności od odpowiedzi wykonujemy jeden z następujących kroków:
   a) jeżeli nie jest pusty, to z tego zbioru wybieramy zadanie o najkrótszym czasie trwania i umieszczamy je w harmonogramie (rozmieszczamy na pasie) oraz aktualizujemy zmienne: *zadania_przydzielone, zadania_do_przydzielenia*,
   b) jeżeli jest pusty, to w danym punkcie zaczepienia umieszczamy przestój.

Operacje te będą wykonywane dopóki są jeszcze jakieś elementy do rozmieszczenia (wektor *zadania_do_przydzielenia* nie jest pusty).

W naszym algorytmie zachłannym w punkcie 3. a) do umieszczenia zadania w harmonogramie wybierane jest zadanie o najkrótszym czasie trwania (najmniejszej długości). W przypadku przyjętego kryterium takie podejście wydaje się intuicyjnie uzasadnione, ponieważ mamy wówczas większą szansę uzyskania harmonogramu, w którym suma czasów zakończenia wszystkich zadań będzie możliwie najkrótsza. Można przypuszczać, że dla innych kryteriów optymalizacji korzystniejszy mógłby się okazać inny sposób wyboru zadań do wykonania. Aby potwierdzić nasze przypuszczenia przeprowadziliśmy eksperymenty obliczeniowe, w których zmienialiśmy kryterium wyboru i do rozmieszczenia wybieraliśmy elementy o: największej szerokości, największej długości, najmniejszym polu, najmniejszej szerokości, najmniejszej długości. Testy przeprowadziliśmy dla dwóch kryteriów optymalizacji: $SumC_j$ i C_{max}. W tabelce 1 zamieściliśmy wyniki tych testów dla problemu „LANL-CM5-1994-4.1-cln.swf" z biblioteki Parallel Workloads Archive (https://www.cs.huji.ac.il/labs/parallel/workload/) oraz wybranych wartości *n.* Najlepsze uzyskane wyniki ze względu na oba kryteria zostały wyróżnione: dla $SumC_j$ pogrubioną czcionką, dla C_{max} podkreśleniem.

Tab. 1. Wartości $SumC_j$ i C_{max} uzyskane dla instancji „LANL-CM5-1994-4.1-cln.swf" w zależności od sposobu wyboru zadania oraz liczby n

Kryterium wyboru zadania	Kryterium optymalizacji	Liczba wczytywanych zadań (*n*)				
		100	1000	5000	10000	15000
Największe pole	$SumC_j$	1509571	141425648	4324553440	18114900819	42012874959
	C_{max}	28959	282484	1816050	3679919	6034005
Największa długość	$SumC_j$	1511571	133469692	4318688489	17916088987	42233194337
	C_{max}	28959	296208	1806578	3838866	6087404
Największa szerokość	$SumC_j$	1325446	143701707	4320112703	17968128296	41592994624
	C_{max}	29252	299653	1808401	3702224	6040561
Najmniejsze pole	$SumC_j$	**1288466**	128544008	4288533720	17643770647	41378342384
	C_{max}	34246	323129	1897166	4243353	6832699
Najmniejsza długość	$SumC_j$	1306631	**128200484**	**4287721775**	**17640612571**	**41346579011**
	C_{max}	35292	321918	1885229	4216704	6739278
Najmniejsza szerokość	$SumC_j$	1297651	129790384	4297755064	17872489991	42312290210
	C_{max}	34231	327358	1874634	4214409	6776884

Dla kryterium $SumC_j$ w większości przeprowadzonych testów najlepsze wyniki otrzymywaliśmy wybierając w kolejnych iteracjach algorytmu element (zadanie) o najmniejszej długości, a dla C_{max} o możliwie największym polu powierzchni. W celu przeprowadzenia dalszych testów efektywnościowych przygotowaliśmy też wersję naszego algorytmu, w którym w punkcie 3. a) element do rozmieszczenia wybierany był w sposób losowy.

W porównaniu do zwykłego algorytmu BL oraz niektórych innych algorytmów, dużą zaletą przedstawionej tu heurystycznej metody rozmieszczania elementów na pasie jest, to, że w otrzymanych układach (harmonogramach) nie powstają „dziury", w których zmieściłby się jakikolwiek element i to niezależnie od sposobu wybory kolejno rozmieszczanych elementów. Otrzymywane układy są już stosunkowo dobrze „upakowane".

3. Wprowadzenie do przeprowadzonych badań

W celu sprawdzenia efektywności naszego algorytmu przeprowadziliśmy badania na następujących plikach obowiązkowych: *DAS2-fs0-2003-1.swf*, *LANL-CM5-1994-3.1-cln.swf* oraz *SDSC-SP2-1998-3.1-cln.swf*. Postanowiliśmy wykonać również testy obliczeniowe na dwóch dodatkowych, wybranych przez nas, plikach: *CTC-SP2-1996-3.1-cln.swf* oraz *LANL-CM5-1994-4.1-cln.swf*. Motywacją do przeprowadzenia obszerniejszych badań było lepsze zobrazowanie efektywności algorytmu w zależności od wybranej funkcji celu. Wyniki otrzymane dla algorytmu zachłannego porównywaliśmy z wynikami uzyskanymi przy wykorzystaniu algorytmu, w którym w kolejnych iteracjach przydział zadań do procesorów odbywał się w sposób losowy (algorytm losowy).

Wykonaliśmy następujące testy, w których badaliśmy zależność:

- czasu działania algorytmu od liczby wczytanych elementów,
- wartości funkcji celu $SumC_j$ od liczby wczytanych elementów,
- czasu realizacji harmonogramu (C_{max}) od liczby wczytanych elementów.

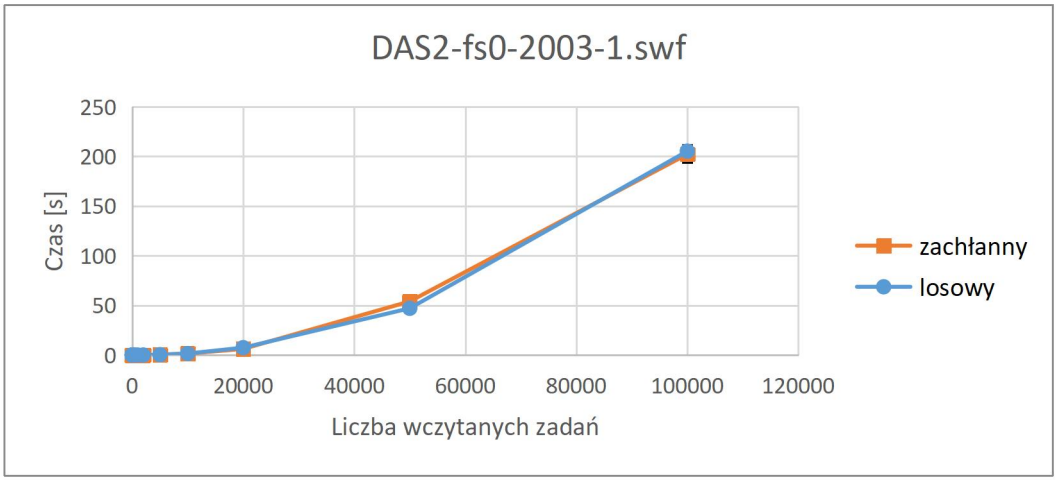
Pierwszy test wykonaliśmy w celu sprawdzenia, jak bardzo czas działania naszego algorytmu zależy od rozmiaru zadania. Został on wykonany, ponieważ w badanym problemie czas, w jakim wykonuje się algorytm, jest ważnym czynnikiem i podobnie, jak w innych zagadnieniach optymalizacji kombinatorycznej często decyduje o możliwości wykorzystania danego algorytmu w praktycznych zastosowaniach. W przeprowadzonych testach wyznaczaliśmy średnią arytmetyczną czasu działania algorytmu oraz odchylenie standardowe. Jak wiadomo, wyznaczenie średniej arytmetycznej wymaga wykonania więcej niż jednego pomiaru. W przypadku algorytmu losowego dla każdej instancji wyniki zostały uśrednione z dziesięciu uruchomień

Następne testy zostały wykonane w celu zbadania efektywności algorytmu zachłannego w porównaniu do algorytmu losowego. Jako kryterium optymalizacji przyjęliśmy sumę czasów zakończenia wszystkich zadań – $SumC_j$. W przypadku tego kryterium, w opracowanym przez nas algorytmie zachłannym, o kolejności wyboru zadań decydowała ich długość – zadania o krótszym czasie wykonania wybierane były jako pierwsze. Dodatkowo przeprowadziliśmy również eksperymenty obliczeniowe, w których porównaliśmy wyniki uzyskiwane dla algorytmu zachłannego i losowego w odniesieniu do kryterium C_{max}, czyli czasu realizacji harmonogramu. Tutaj algorytm zachłanny w kolejnych iteracjach wybierał zadania o możliwie największym polu powierzchni, obliczanego jako iloczyn długości trwania zadania przez jego szerokość tj. liczby procesorów przydzielonych do zadania.

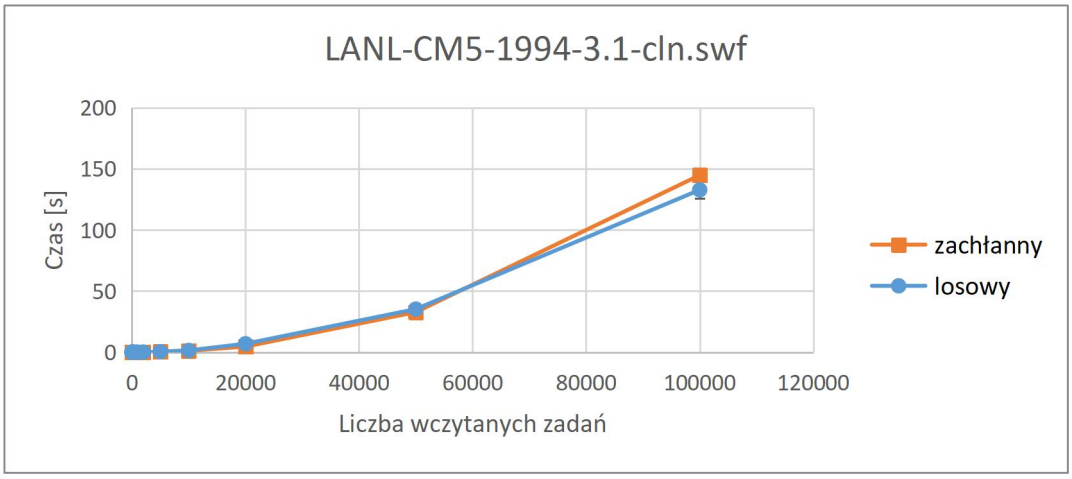
W celu poprawnego porównania efektywności wyboru zadania w sposób losowy (algorytm losowy) i systematyczny (algorytm zachłanny), w testach badających zależność wartości funkcji celu oraz czasu realizacji harmonogramu od liczby wczytanych elementów, wyniki dla algorytmu losowego zostały uśrednione z kilku uruchomień. Wyniki przeprowadzonych testów dla danej instancji, tak jak w poprzednim przypadku zostały uśrednione z dziesięciu uruchomień. Aby lepiej móc zobrazować wyniki postanowiliśmy wykonać wykresy punktowe z prostymi liniami.

4. Wyniki testów obliczeniowych

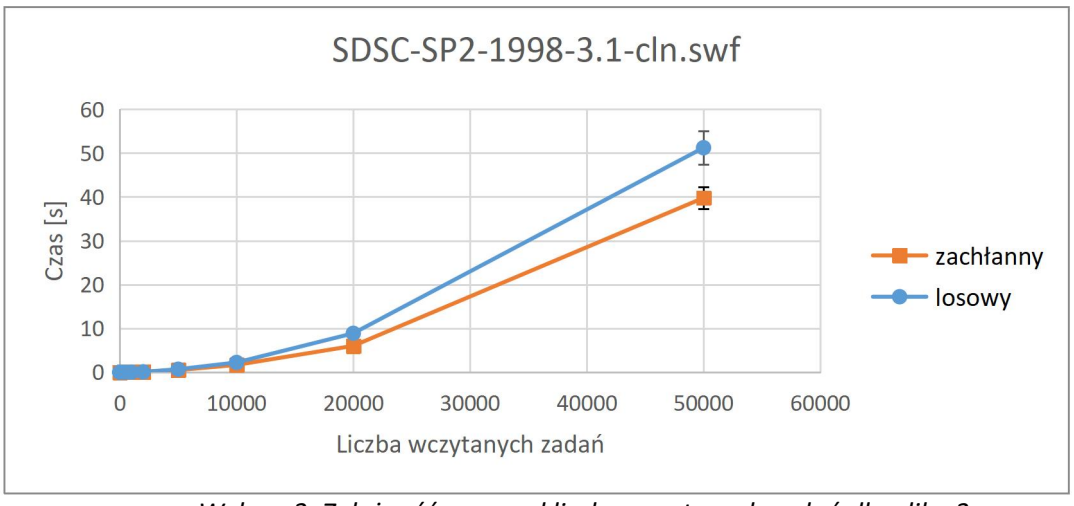
Jak wspomnieliśmy powyżej, pierwsze testy obrazują zależność czasu obliczeń od liczby wczytanych elementów.



Wykres 1. Zależność czasu od liczby wczytanych zadań dla pliku 1.



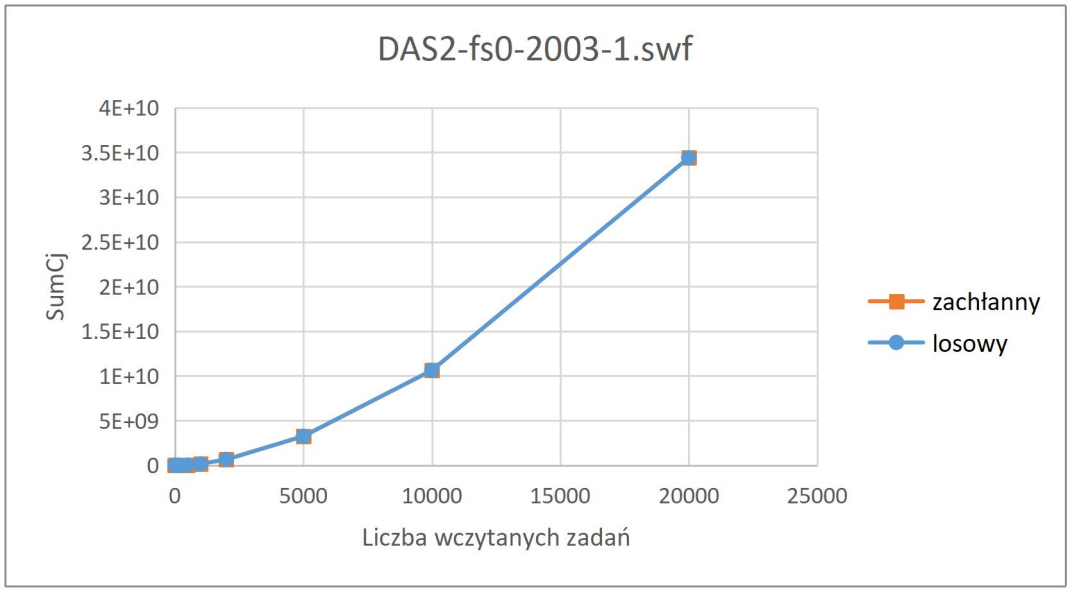
Wykres 2. Zależność czasu od liczby wczytanych zadań dla pliku 2.



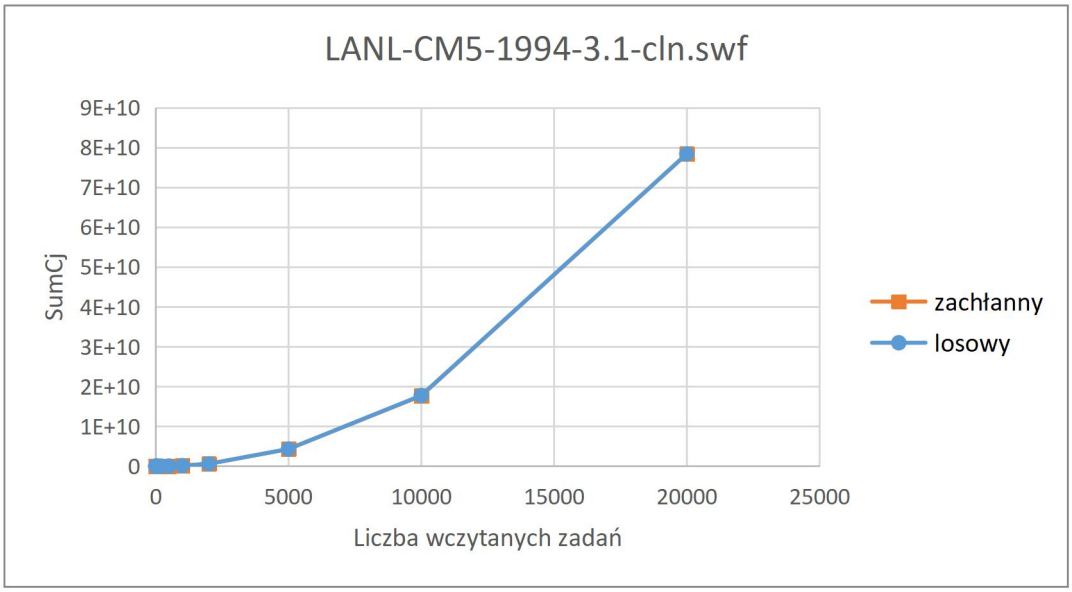
Wykres 3. Zależność czasu od liczby wczytanych zadań dla pliku 3.

Jak można było podejrzewać, niezależnie czy używamy algorytmu zachłannego, czy losowego zależność czasu obliczeń od liczby wczytanych elementów jest zbliżona dla obu algorytmów. Wynika to z faktu, że jedyna różnica między nimi polega na odmiennym sposobie przydziału kolejnych zadań do wykonania przez procesory. Widoczne powyżej wykresy sugerują, że czas pracy algorytmów rośnie wielomianowo wraz ze wzrostem rozmiaru zagadnienia. Dla algorytmu zachłannego można było uzyskać jeszcze krótsze czasy obliczeń przeprowadzając najpierw sortowanie zadań według przyjętego kryterium i przydzielaniu ich kolejno w tej posortowanej kolejności.

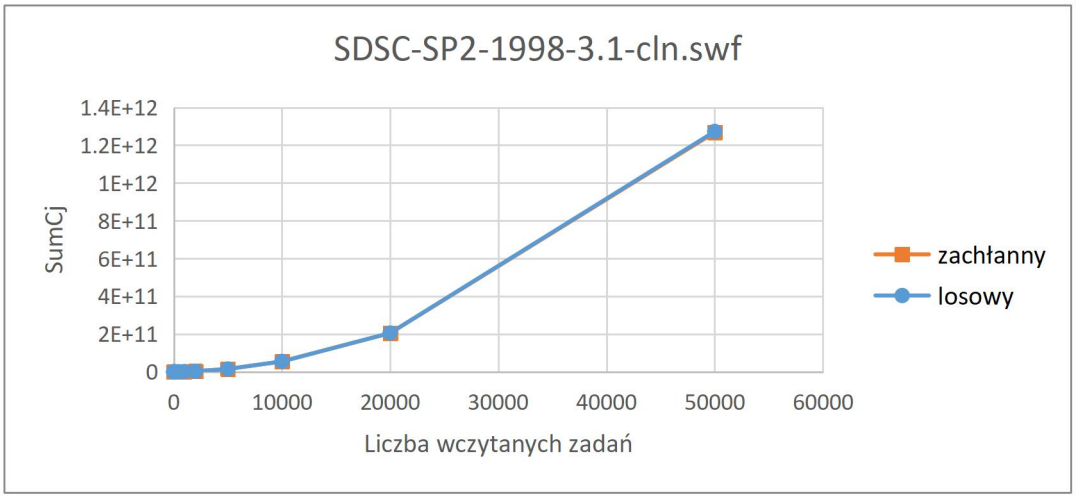
Kolejne wykresy, ilustrują związek między uzyskiwaną wartością funkcji celu, a liczbą wczytanych zadań.



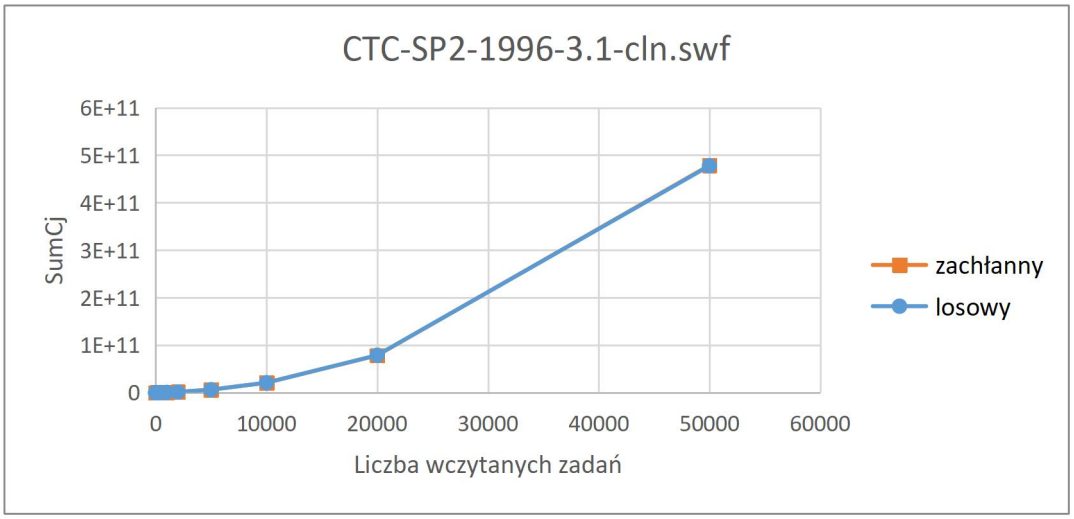
Wykres 4. Zależność wartości funkcji celu od liczby wczytanych zadań dla pliku 1.



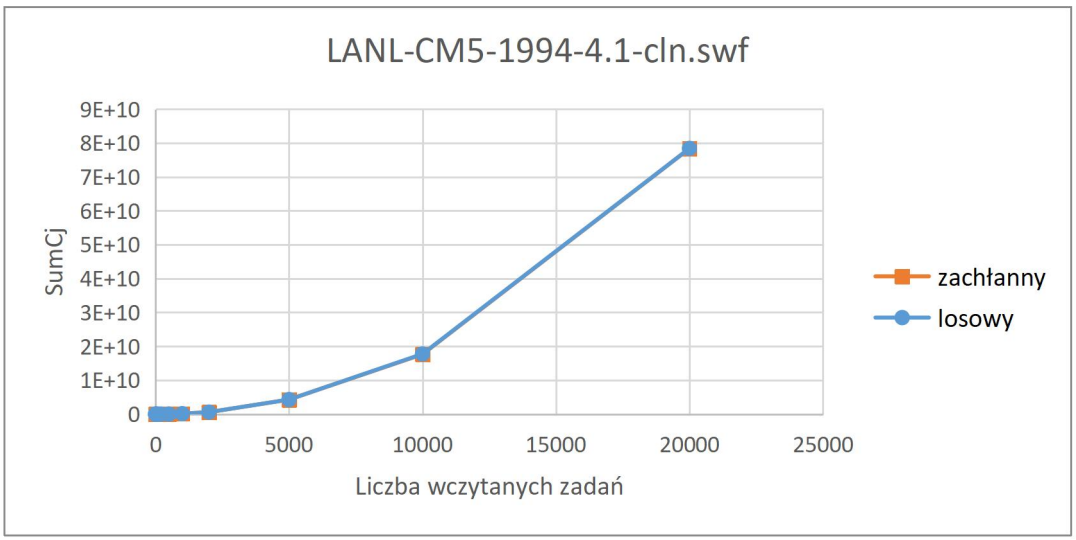
Wykres 5. Zależność wartości funkcji celu od liczby wczytanych zadań dla pliku 2.



Wykres 6. Zależność wartości funkcji celu od liczby wczytanych zadań dla pliku 3.

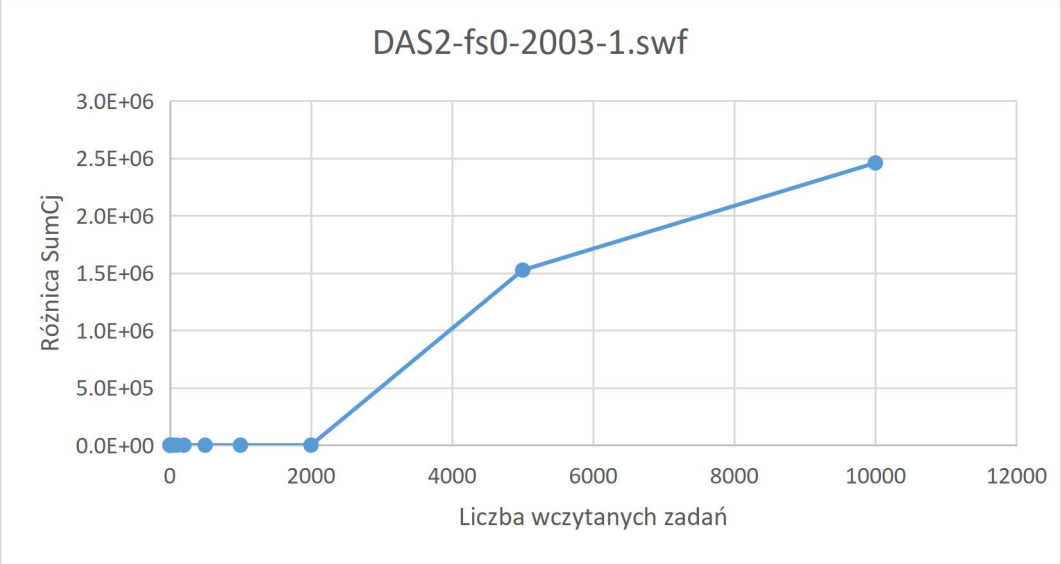


Wykres 7. Zależność wartości funkcji celu od liczby wczytanych zadań dla pliku 4.

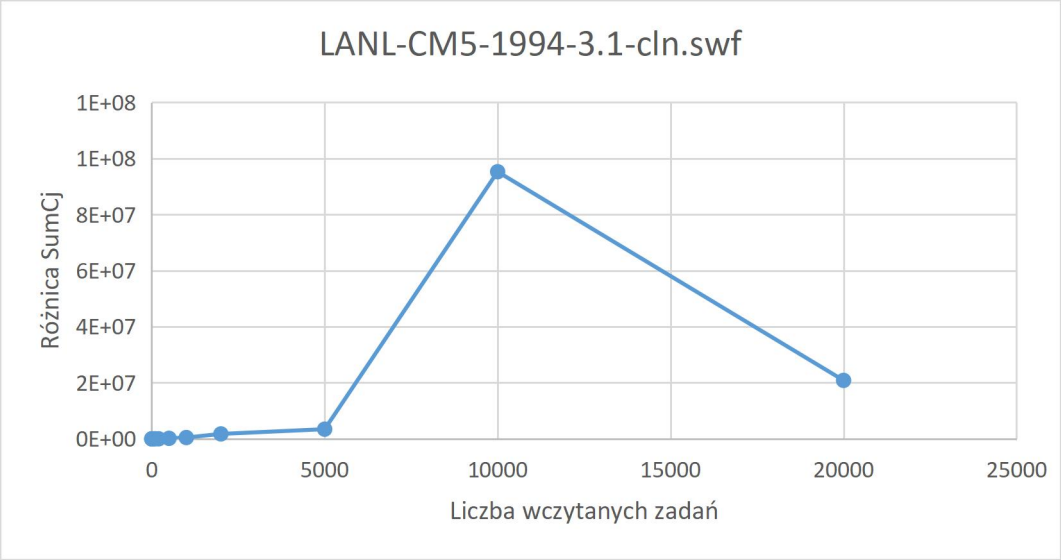


Wykres 8. Zależność wartości funkcji celu od liczby wczytanych zadań dla pliku 5.

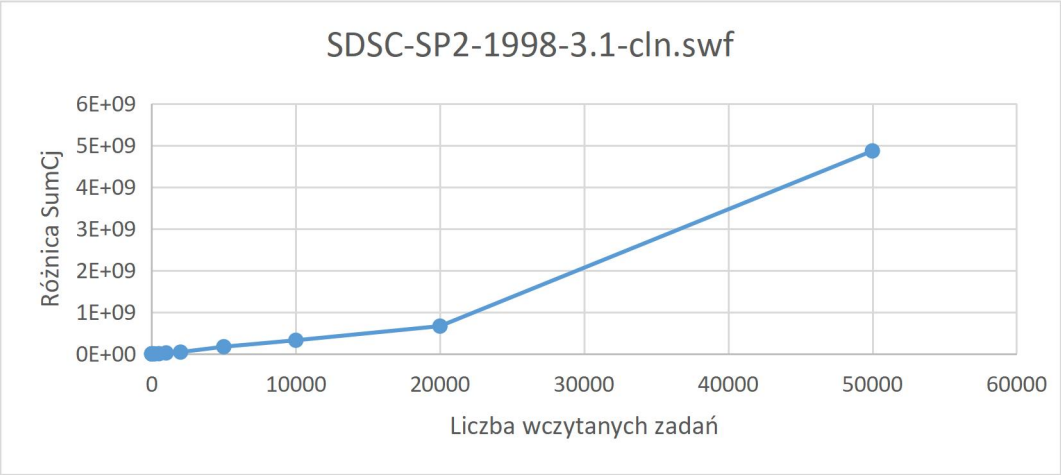
Jak można zauważyć wykorzystaliśmy tutaj również wcześniej wspomniane, dodatkowe pliki w celu lepszego zobrazowania wyników. Analiza zamieszczonych wykresów może w pierwszej chwili sugerować, że nie ma prawie żadnej różnicy w otrzymywanych wynikach w zależności od tego, czy użyjemy algorytmu zachłannego, czy losowego. Brak widocznej różnicy między tymi dwoma algorytmami wynika jednak z bardzo dużych wartości funkcji celu, jakie otrzymujemy dla problemów z dużą liczbą wczytywanych zadań (efekt skali). Aby lepiej móc porównać oba algorytmy wykonaliśmy wykresy zależności różnicy uzyskiwanych wyników od liczby wczytanych elementów. Różnica ta polega na odjęciu od siebie wartości funkcji celu dla algorytmu z wyborem losowym i wartości funkcji cel algorytmu zachłannego. Oczekujemy wyników powyżej osi *Ox*, gdyż wtedy algorytm zachłanny będzie lepszy.



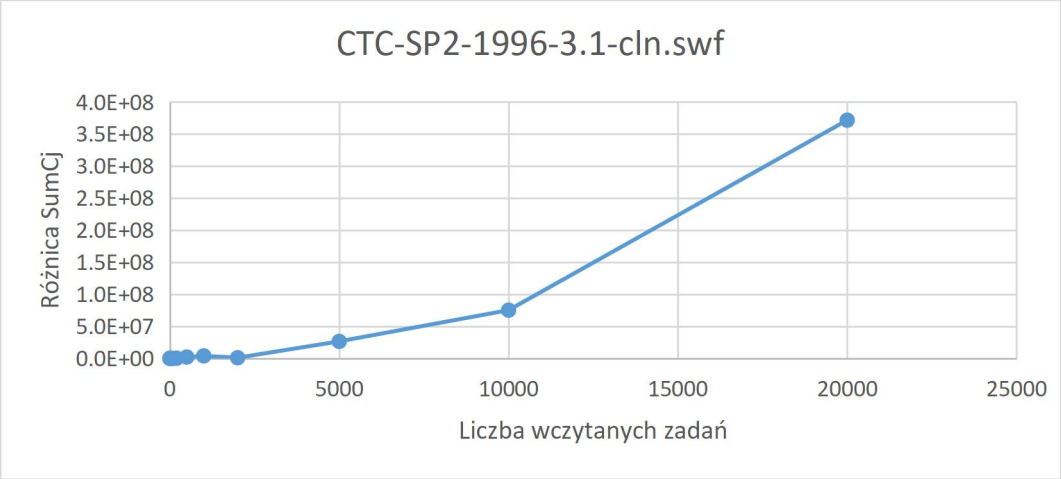
Wykres 9. Zależność różnicy wartości funkcji celu (losowy-zachłanny) od liczby wczytanych zadań dla pliku 1.



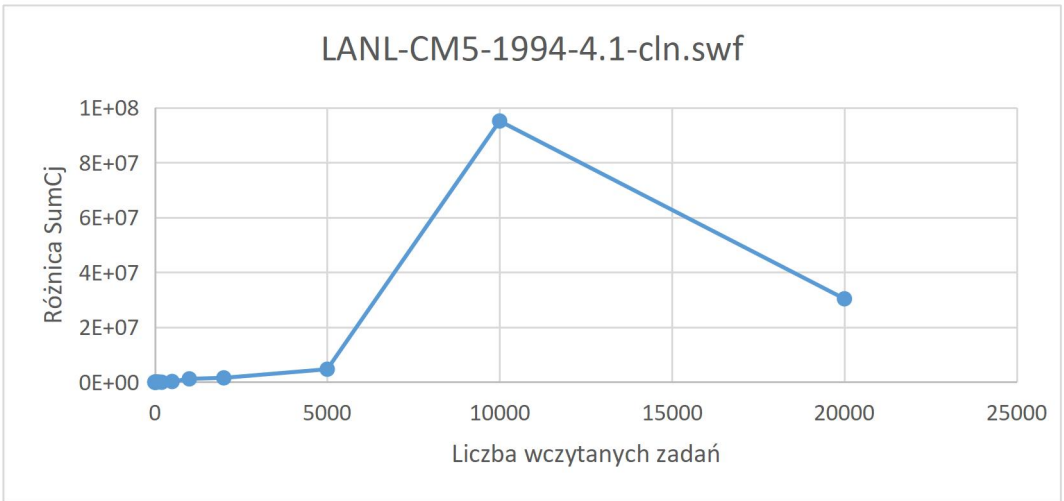
Wykres 10. Zależność różnicy wartości funkcji celu (losowy-zachłanny) od liczby wczytanych zadań dla pliku 2.



Wykres 11. Zależność różnicy wartości funkcji celu (losowy-zachłanny) od liczby wczytanych zadań dla pliku 3.



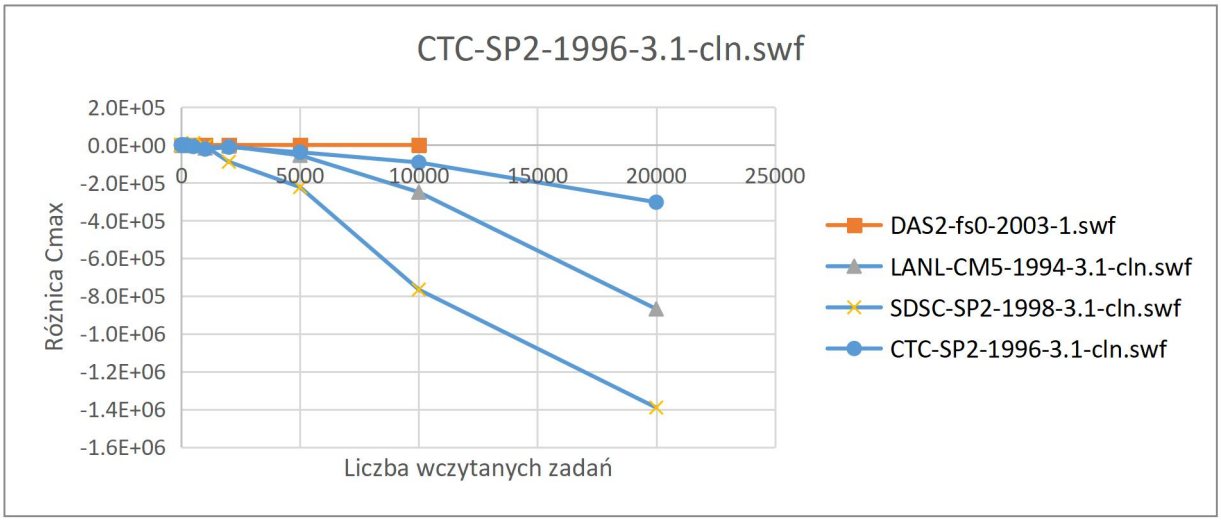
Wykres 12. Zależność różnicy wartości funkcji celu (losowy-zachłanny) od liczby wczytanych zadań dla pliku 4.



Wykres 13. Zależność różnicy wartości funkcji celu (losowy-zachłanny) od liczby wczytanych zadań dla pliku 5.

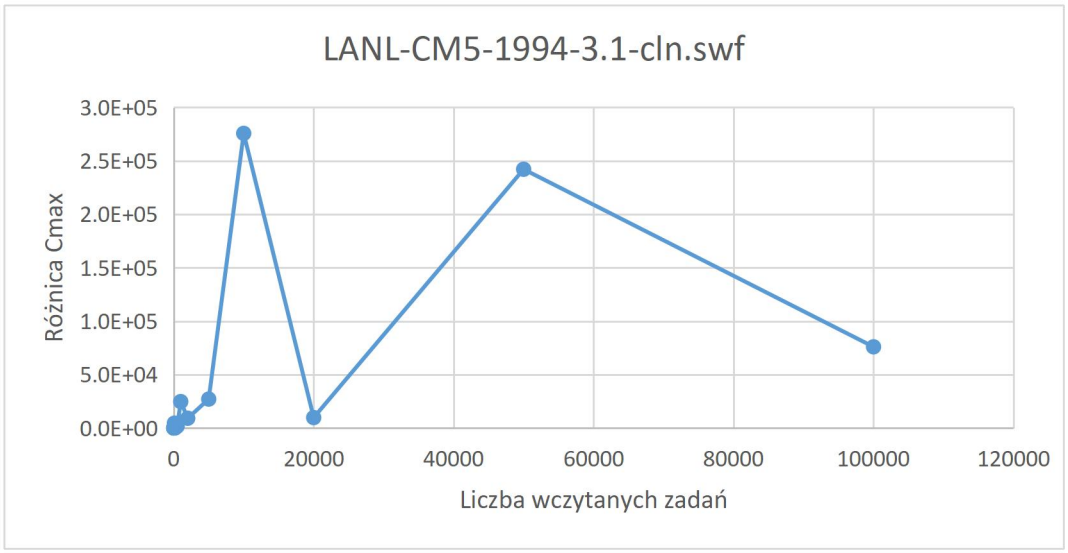
Analizując zamieszczone powyżej wykresy można wnioskować, że algorytm zachłanny daje lepsze wyniki niż algorytm losowy, przynajmniej w odniesieniu do użytych instancji testowych. Ciężko jednak stwierdzić, czy istnieje tutaj jakikolwiek trend. Wykorzystaliśmy tutaj również dodatkowy plik (*CTC-SP2-1996-3.1-cln.swf*), ze względu na bardzo satysfakcjonujące wyniki. Dla prawie każdego z użytych plików testowych widzimy, że badana różnica rośnie wraz z liczbą wczytanych elementów. Wynik dla pliku LANL-CM5-1994-3.1-cln.swf pokazuje nam bardzo dużą różnicę dla 10000 wczytanych zadań. Odbiega to od reszty wyników w znaczny sposób. Może to być spowodowane tym, że porównujemy się do algorytmu losowego, albo wynikać ze specyfiki (trudnej do wykrycia bez dokładniejszych testów) samej instancji testowej.

Jak już wcześniej wspomniano, przeprowadziliśmy również dodatkowe badania dla minimalizacji kryterium C_{max}, czyli długości harmonogramu. Poniższy zbiorczy wykres przedstawia nam różnicę miedzy wartościami funkcji celu otrzymywanymi dla algorytmu wybierającego zadanie w sposób losowy i systematyczny.

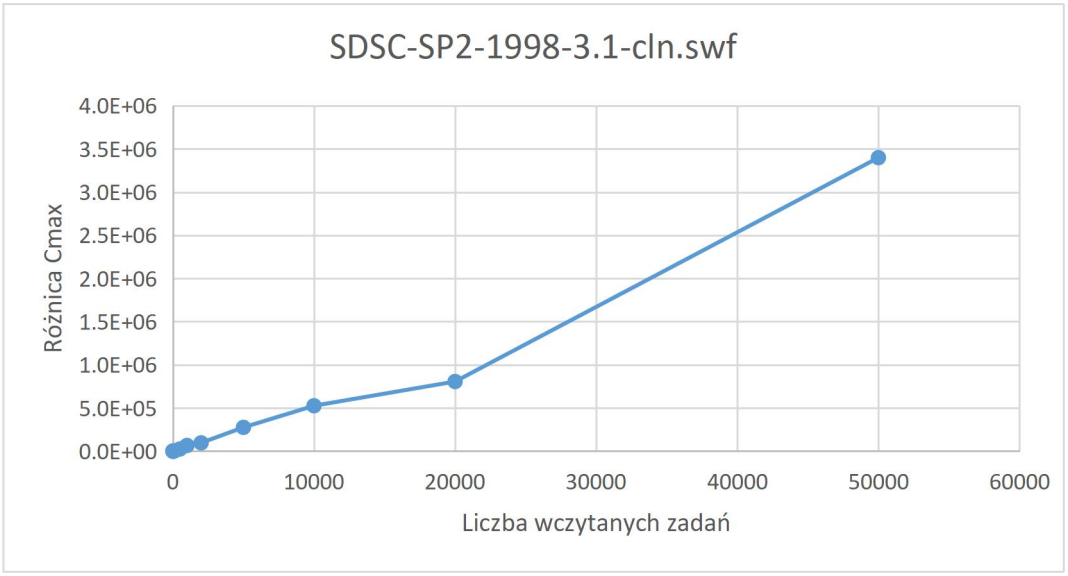


Wykres 14. Zależność różnicy wartości funkcji C_{max} (losowy-zachłanny) od liczby wczytanych zadań dla plików 1, 2, 3, 4

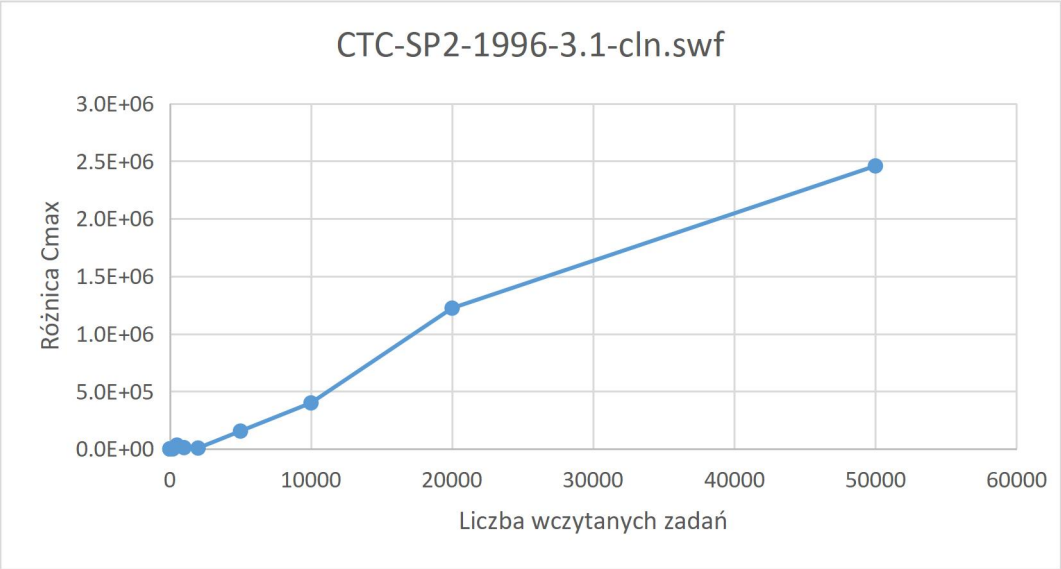
Jak możemy zauważyć algorytm losowy prezentuje się tutaj lepiej, ponieważ wszystkie wartości są na lub poniżej osi *Ox*. Pokazuje to, że dotychczasowe kryterium (najmniejsza długość zadania) doboru następnego zadania jest nieefektywne. Zdecydowaliśmy się zatem sprawdzić, czy inny sposób wyboru kolejnych zadań nie dałby lepszych wyników w odniesieniu do kryterium w postaci czasu realizacji harmonogramu. Wyniki jednego z takich testów zaprezentowaliśmy wcześniej w tabeli 1. Po sprawdzeniu kilku różnych kryteriów zauważyliśmy, że najlepsze rezultaty daje w tym przypadku wybór zadań o największym polu powierzchni (długość zadania * liczba przydzielonych procesorów). Poniżej przedstawiliśmy wykresy różnic (obliczanych w taki sam sposób jak poprzednio) miedzy wartościami funkcji celu C_{max} dla algorytmu losowego i zachłannego.



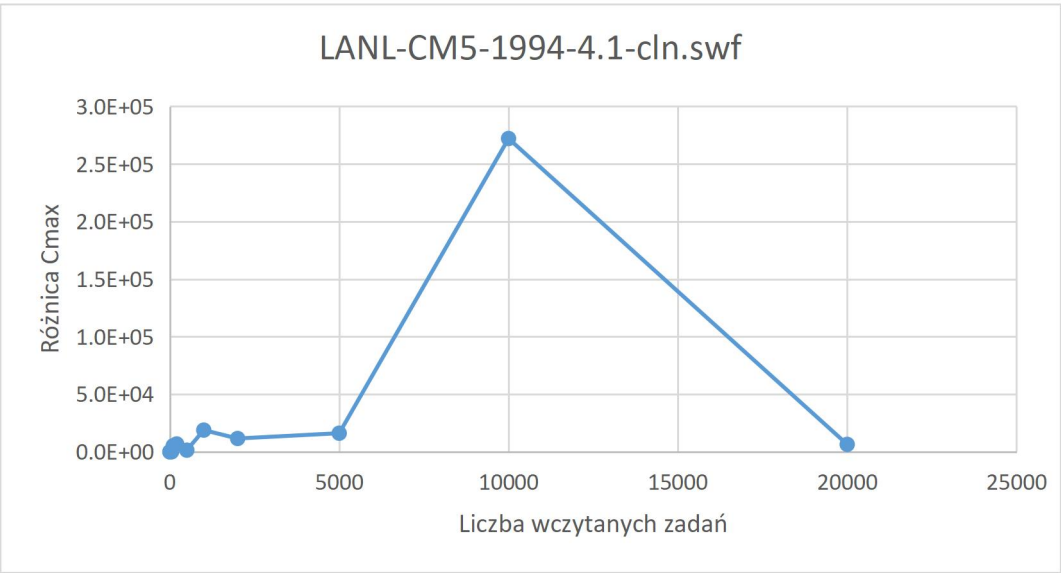
Wykres 15. Zależność różnicy wartości funkcji C_{max} (losowy-zachłanny) od liczby wczytanych zadań dla pliku 2 z nowym kryterium wyboru



Wykres 16. Zależność różnicy wartości funkcji C_{max} (losowy-zachłanny) od liczby wczytanych zadań dla pliku 3 z nowym kryterium wyboru



Wykres 17. Zależność różnicy wartości funkcji C_{max} (losowy-zachłanny) od liczby wczytanych zadań dla pliku 4 z nowym kryterium wyboru



Wykres 18. Zależność różnicy wartości funkcji C_{max} (losowy-zachłanny) od liczby wczytanych zadań dla pliku 5 z nowym kryterium wyboru

Jak możemy zauważyć, wybór nowego kryterium w postaci pola powierzchni był trafny. Dla każdego pliku różnica ta była zawsze większa bądź równa zero. Ponownie dla plików *SDSC-SP2-1998-3.1-cln.swf* oraz *CTC-SP2-1996-3.1-cln.swf* możemy zauważyć trend wskazujący, że różnica ta rośnie wraz ze wzrostem liczby wczytanych zadań.

5. Wnioski z przeprowadzonych badań

Analizując wyniki przeprowadzonego badania efektywności opracowanego przez nas algorytmu zachłannego, można wyciągnąć kilka istotnych wniosków.

Pierwsze testy, dotyczące czasu działania algorytmu, wykazały, że niezależnie od zastosowanego podejścia (algorytm zachłanny czy losowy) czas trwania rośnie wielomianowo wraz z liczbą wczytanych elementów. Nawet dla instancji o dużym rozmiarze algorytm był w stanie wygenerować rozwiązanie w stosunkowo krótkim czasie. Ogólnie rzecz biorąc algorytmy zachłanne mają na ogół niską złożoność obliczeniową, co sprawia, że są atrakcyjne z punktu widzenia efektywności czasowej.

Jeżeli natomiast chodzi o „jakość" uzyskanych wyników w odniesieniu do kryterium optymalizacji, to pobieżna analiza wykresów zależności między otrzymywaną wartością funkcji celu, a liczbą wczytanych zadań mogłaby sugerować, że nie ma istotnej różnicy pomiędzy algorytmem zachłannym a losowym. Dodatkowe badania, uwzględniające różnicę między rezultatami obu algorytmów, pokazały jednak, że algorytm zachłanny w zdecydowanej większości przeprowadzonych testów osiąga lepsze wyniki niż algorytm losowy.

Dalsze testy pokazały, że w sytuacji zmiany kryterium optymalizacji istotne staje się przeprowadzenie weryfikacji przyjętej w algorytmie zachłannym metody wyboru zadań do wykonania. Po zmianie kryterium z $SumC_j$ na C_{max} algorytm zachłanny, w którym przydział zadań odbywał się na podstawie czasu ich trwania, generował gorsze rozwiązania niż algorytm losowy. Sytuacja uległa odwróceniu, gdy dokonaliśmy modyfikacji algorytmu polegającej na tym, że w każdej iteracji do umieszczenia w harmonogramie wybierane było zadanie o największym polu powierzchni, rozumianym jako iloczyn wymiarów zadania.

Podsumowując, przeprowadzone badania sugerują, że algorytm zachłanny, w którym wybór kolejnych zadań do wykonania odbywa się w sposób deterministyczny i zgodny z jakimś ustalonym porządkiem daje na ogół lepsze wyniki niż algorytm losowy. Dodatkową zaletą algorytmu zachłannego jest to, że (w przeciwieństwie do algorytmu losowego) generowane przez niego rozwiązania są całkowicie powtarzalne. W przypadku tego typu algorytmów należy jednak starannie dobrać kolejność, w jakiej wybierane są zadania, ponieważ od tego zależy ich efektywność.

Literatura:

[1] Baker, B. S., E. G. Coffman, Jr., R. L. Rivest. (1980). Orthogonal packings in two dimensions. SIAM Journal on Computing, 9(4), 808–826.

[2] Błądek, I., Drozdowski, M., Guinand, F., Schepler, X. (2015). On contiguous and non-contiguous parallel task scheduling. Journal of Scheduling, 18, 487–495.

[3] Burke, E.K., Kendall, G., Whitwell, G. (2004). A new placement heuristic for the orthogonal stock-cutting problem. Operations Research, 52 (4), 655–671.

[4] Chazelle, B., (1983). The bottom-left bin packing heuristic: an efficient implementation. IEEE Transaction on Computers, 32 (8), 697–707.

[5] Drozdowski, M. (2009). Scheduling for parallel processing. London: Springer.

[6] Leung, S. C.H., Zhang, D., Sim, K. M. (2011). A two-stage intelligent search algorithm for the two-dimensional strip packing problem. European Journal of Operational Research, 215(1), 57-69.