Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Informatyki

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Inżynieria Systemów Informatycznych

Opracowanie i implementacja algorytmów przydziału zasobu porcjowanego

Adam Małkowski

numer albumu 265760

promotor
dr hab. inż. Krzysztof Pieńkosz

Warszawa 2017

Streszczenie

Tytuł: *Opracowanie i implementacja algorytmów przydziału zasobu porcjowanego*

W pracy rozpatrywany jest problem pakowania pojemników elementami częściowo podzielnymi, będący jednym z modeli problemu alokacji zasobów. Podzielność jest ograniczana przez parametr narzucający minimalny rozmiar fragmentu powstałego po podziale elementu. W pracy dokonany jest przegląd dotychczas znanych algorytmów rozwiązywania tego problemu. Zaproponowane zostają trzy algorytmy konstrukcyjne rozwiązujące problem pakowania elementów częściowo podzielnych oraz dwa algorytmy poprawy mające za zadanie poprawić upakowanie otrzymane w wyniku zastosowania algorytmu konstrukcyjnego.

Słowa kluczowe: *problem alokacji zasobu, problem pakowania pojemników, optymalizacja, algorytmy heurystyczne*

Abstract

Title: *Development and implementation of algorithms for parted resource allocation*

This paper concerns with the bin packing problem with partially fragmentable items, which is one of the models of resource allocation problem. In the case analyzed in this paper it is assumed that there is given a minimal allowed size of pieces for which items may be fragmented. The paper contains review of existing algorithms solving this version of the problem. There are proposed three construction algorithms solving bin packing problem with partially fragmentable items and two algorithms that improve the results found by the construction algorithms.

Keywords: *resource allocation, bin packing problem, optimization, heuristic*

Dziękuję Panu dr. hab. inż. Krzysztofowi Pieńkoszowi
za okazaną pomoc i wsparcie

załącznik do zarządzenia nr 28/2016 r.
Rektora PW

Politechnika Warszawska

„załącznik nr 3 do zarządzenia nr 24/2016 Rektora PW

....
miejscowość i data

....
imię i nazwisko studenta

....
numer albumu

....
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

....
czytelny podpis studenta"

Spis treści

Wprowadzenie

Problem alokacji zasobów powszechnie występujący w otaczającej rzeczywistości jest bardzo ogólny i może być dostosowany do wielu sytuacji - co za tym idzie trudno znaleźć ogólne metody alokacji działające z rozsądną wydajnością oraz dające rozsądne wyniki. W związku z tym występuje bardzo wiele wariantów problemu oraz związanych z nimi modeli. W przypadku problemu alokacji należy zdefiniować pojęcia zasobu i podmiotu. Zasób to byt, który posiadamy lub możemy posiadać i który dystrybuujemy. Podmiotem nazywamy zaś byt potrzebujący uzyskania konkretnej ilości określonego zasobu.

Niniejsza praca dotyczy algorytmów alokacji zasobu porcjowanego do podmiotów częściowo podzielnych. Zasób porcjowany to taki, który jest dostępny w porcjach. Zwiększenie dostępności takiego zasobu wiążę się z koniecznością zdobycia kolejnej porcji zasobu, nawet jeżeli nie zostanie ona w pełni wykorzystana. W ogólnym przypadku każda porcja zasobu może mieć różny rozmiar, jednak w niniejszej pracy analizowany jest przypadek, w którym porcje zasobu mają ustalony rozmiar.

Modele alokacji zasobu porcjowanego określa się jako problemy pakowania pojemników (ang. BPP - Bin Packing Problem). Problemy te interpretują przydział zasobu jako pakowanie pojemnika pewnymi elementami. Elementy są tutaj interpretowane jako podmioty. Pojemność pojemnika reprezentuje rozmiar porcji zasobu - w niniejszej pracy jednakową dla wszystkich pojemników. Rozmiar elementu reprezentuje wartość zapotrzebowania danego podmiotu. W klasycznym przypadku problemu pakowania pojemników elementy są niepodzielne - dany element należy w całości zapakować do jednego pojemnika. Wariant z częściowo podzielnymi elementami dopuszcza możliwość podziału elementu na fragmenty podczas procesu pakowania, jednakże tylko w pewnym ograniczonym stopniu. W literaturze [3, 4, 6, 11] opisano różne podejścia do dzielenia elementów na fragmenty. Niniejsza praca skupia się na podejściu narzucającym pewien minimalny rozmiar fragmentu.

Celem pracy jest dokonanie analizy obecnie istniejących algorytmów rozwiązywania problemu alokacji zasobu porcjowanego do podmiotów częściowo podzielnych, a następnie próba zmodyfikowania ich oraz zaproponowania nowych, tak aby osiągnąć pewną poprawę generowanych upakowań.

W ramach pracy rozpatrywane zostaną zarówno algorytmy konstrukcyjne, generujące pewne konkretne upakowanie, jak również algorytmy poprawy, których działanie sprowadza się do próby modyfikacji otrzymanego wcześniej upakowania, tak aby je poprawić. Rozpatrywane w pracy algorytmy, zarówno konstrukcyjne jak i poprawy, są algorytmami heurystycznymi.

Rozdział 1 opisuje szczegółowo modele alokacji zasobu porcjowanego. Rozdział 2 jest przeglądem obecnie istniejących algorytmów rozwiązywania problemu pakowania pojemników elementami częściowo podzielnymi. Rozdział 3 dotyczy proponowanych w niniejszej pracy metod rozwiązywania rzeczonego problemu. Rozdział 4 zawiera wyniki oraz wnioski z eksperymentów dotyczących działania opisanych algorytmów. Ostatni rozdział zawiera podsumowanie.

1. Modele alokacji zasobu porcjowanego

1.1 Klasyczny problem pakowania pojemników

W klasycznym modelu pakowania pojemników, zwanym również problemem BPP (ang. Bin Packing Problem), należy znaleźć upakowanie elementów j ze zbioru $N = \{1, \dots, n\}$ o rozmiarach w_j do jak najmniejszej liczby pojemników. Każdy pojemnik ma pojemność C.

Dla klasycznego modelu pakowania pojemników przyjmuje się założenie, że pojedynczy podmiot realizuje swoje zapotrzebowanie korzystając z dokładnie jednej porcji zasobu - co przekłada się na założenie, że pojedynczy element może być zapakowany jedynie w całości do jednego pojemnika. Założenie to ma uzasadnienie w wielu sytuacjach praktycznych. W [6] przytoczony został przykład opisujący problem zagospodarowania czasu pracowników w firmie serwisowej poprzez przypisanie im konkretnych zleceń. W opisanym problemie zasobem porcjowanym jest czas pracy pracownika, a konkretną porcją jest jeden dzień roboczy jednej osoby. Realizacja tego samego zlecenia na raty przez różnych pracowników jest zarówno dla klienta jak również właściciela firmy często nie do zaakceptowania - potencjalnie zwiększa koszty oraz zmniejsza jakość realizacji zlecenia.

Problem pakowania jest $NP-trudny$ [3], w związku z tym do rozwiązywania go wykorzystuje się głównie algorytmy heurystyczne.

Istnieje bardzo wiele algorytmów rozwiązywania problemu BPP. Najprostszą klasą tych algorytmów są podejścia listowe. Metody te, w trakcie swojego działania, pakują kolejne elementy z wstępnie ustalonej listy oraz wykorzystują odpowiednią regułę pakowania elementów.

Jednymi z najpopularniejszych metod listowych są algorytmy FF (First Fit) oraz BF (Best Fit) [3]. Dla obu metod porządek elementów na liście jest dowolny. Algorytm FF pakuje kolejne elementy z listy do pojemnika o najniższym numerze, do którego da się je zapakować. Algorytm BF różni się od FF metodą wyboru pojemnika - w tym przypadku element jest pakowany do takiego pojemnika, aby po zapakowaniu elementu zostało w nim najmniej wolnego miejsca. Algorytmy te dają lepsze rezultaty jeżeli zmodyfikuje się je, sortując wcześniej listę elementów zgodnie z porządkiem malejącym. Algorytmy stosujące takie sortowanie nazwane są BFD (Best Fit Decreasing) oraz FFD (First Fit Decreasing). Algorytmy te mimo swojej prostoty są dość skuteczne.

Algorytmy zorientowane na pojemniki (ang. BOH - Bin Oriented Heuristic), to takie, które w kolejnych krokach starają się jak najlepiej zapakować kolejne pojemniki. Oznacza to, że znajdują odpowiednie elementy do zapakowania do aktualnego pojemnika, po czym przechodzą do pakowania następnego pojemnika. Pakowanie pojedynczego pojemnika sprowadza się do wybrania pewnego podzbioru elementów i przypisania pojemnikowi wszystkich elementów z wybranego podzbioru.

Wiele algorytmów zorientowanych na pojemniki podczas pakowania pojedynczego pojemnika znajduje więcej niż jeden podzbiór elementów w kontekście zapakowania jednego pojemnika. Algorytmy działające w ten sposób przygotowują parę wariantów zapakowania pojedynczego pojemnika, a następnie wybierają najlepszy z wariantów. Przykładem tego typu algorytmów są B2F (Best Two Fit) oraz MBS (Minimum Bin Slack) [1]. Algorytm B2F bazuje na algorytmie FFD. Do zapakowania kolejnych pojemników wykorzystuje on dwa podzbiory elementów - pierwszy z nich jest to wynik zapakowania jednego pojemnika algorytmem FFD, a drugi powstaje przez modyfikacje pierwszego. Jeżeli w pierwszym pojemniku pozostaje wolne miejsce, zamieniany jest najmniejszy zapakowany do niego element na dwa mniejsze elementy, które zajmują więcej miejsca niż zamieniany element oraz mieszczą się do pojemnika. Algorytm MBS, w celu zapakowania kolejnego pojemnika, znajduje wszystkie warianty zapakowania pojemnika, takie, że nie da się zapakować żadnego dodatkowego elementu z pozostałych na liście. Następnie wybiera wariant, w którym zostaje najmniej wolnego miejsca w pojemniku.

Istnieją również próby dokładnego rozwiązywania problemu pakowania pojemników. Najczęściej stosowanymi technikami są metoda podziału i ograniczeń oraz technika generacji kolumn [3, 12].

1.2 Problem pakowania elementów częściowo podzielnych

Założenie o niepodzielności elementów w klasycznym problemie pakowania ma często silne uzasadnienie praktyczne, jednakże istnieją przykłady sytuacji życiowych, dla których natura elementów pozwala na ich podzielność. Gdy możliwa jest dowolna podzielność elementów, mówimy o problemie w wariancie ciągłym. Jest on spotykany w przypadkach alokacji zasobów, których struktura pozwala na dowolne ich dzielenie. Na placu budowy zapotrzebowanie na cement dla poszczególnych grup robotników może być zrealizowane z kilku worków - będących porcjami zasobu - nawet jeżeli w poszczególnych workach znajduje się bardzo mała część całego zapotrzebowania.

Rozwiązywanie problemu w wariancie ciągłym jest trywialne oraz wykonywalne z liniową złożonością względem liczby elementów wejściowych. Problem można rozwiązywać poprzez pakowanie kolejnych elementów do pojemnika o najniższym numerze, który nie jest w pełni zapakowany. W przypadku gdy nie da się zapakować całego elementu następuje jego fragmentacja - element jest dzielony na fragment o rozmiarze równym wolnej przestrzeni w wybranym pojemniku oraz fragment będący resztą. Pierwszy z fragmentów jest pakowany do wybranego pojemnika, drugi fragment jest pakowany do następnego pojemnika. Proces ten jest kontynuowany, aż zapakowane zostaną wszystkie elementy. Algorytm ten zawsze rozwiązuje problem uzyskując upakowanie optymalne złożone z liczby pojemników równej

$$\left\lceil \frac{\text{suma rozmiarów wszystkich elementów}}{\text{rozmiar pojemnika}} \right\rceil \tag{1}$$

Problem pakowania pojemników w wariancie ciągłym ma istotne znaczenie dla analizy problemu pakowania elementów częściowo podzielnych - liczność upakowania optymalnego dla problemu w wariancie ciągłym jest oszacowaniem dolnym liczności upakowań w wariancie częściowo podzielnym. W przypadku klasycznego problemu pakowania jest podobnie - liczność rozwiązania optymalnego pozwala na oszacowanie od góry rozwiązań problemu częściowo podzielnego dla tych samych danych. Jednak znalezienie rozwiązania optymalnego problemu bez podzielnych elementów jest zadaniem trudnym. Jako mniej dokładne oszacowanie górne, może być wykorzystana liczność rozwiązania problemu dyskretnego wygenerowanego przez jeden z prostych algorytmów listowych.

Istnieją przypadki praktyczne, dla których zarówno klasyczny problem pakowania jak i problem pakowania w wariancie ciągłym nie oddają w pełni specyfiki. W pracach [5, 6] opisany został problem minimalizacji odpadów powstających przy rozkroju kształtowników (podłużnych elementów metalowych). Kształtowniki dostępne są na rynku w pewnej standardowej długości, w praktyce natomiast nabywcy potrzebują kształtowników o zróżnicowanych długościach. W tej sytuacji, praktyką jest rozkrajanie zakupionych elementów na mniejsze, zgodne z zapotrzebowaniem, bądź spawanie, jeżeli potrzebny jest element o długości większej niż standardowa. W problemie tym dopuszczalne jest wycinanie zamawianych elementów niekoniecznie w całości z tej samej porcji zasobu. Dopuszczalne jest również wykorzystanie kształtowników zespawanych z kilku krótszych elementów, jednakże występuje ograniczenie na gęstość spawów wynikające z przyczyn technologicznych - wielokrotne spawanie zmniejsza jakość otrzymanego elementu.

Inny przykład został opisany w publikacjach [3,4]. Publikacje opisują problem harmonogramowania danych przesyłanych przez sieć telewizji kablowej. Transmisja danych jest tam modelowana jako strumień ponumerowanych szczelin, gdzie każda ze szczelin, pozwala na przesłanie pewnej porcji danych. Raz na jakiś czas, centralna stacja telewizji kablowej publikuje informacje na temat przydziału najbliższych szczelin konkretnym klientom lub ich grupom. Problem polega na przydzieleniu pakietów poszczególnych klientów do szczelin. Rozpatrywane są dwa rodzaje alokacji pakietów. Pierwszy dotyczy pakietów, których przesyłanie jest związane ze sztywnymi rygorami czasowymi (np. połączenia o stałej szybkości transmisji). Został on nazwany w publikacji [3] *fixed location*. Drugi, nazwany w publikacji [3] *free location*, dotyczy pozostałych pakietów. Mogą one być kojarzone z dowolnymi szczelinami. Problem rozwiązywany jest w dwóch fazach. W pierwszej alokowane są wszystkie pakiety *fixed location*. Szczeliny im przydzielane, są wybierane tak, aby spełnić wymagane rygory. Druga faza polega na przydzieleniu pakietów *free location* do pozostałych szczelin. Pomiędzy przydzielonymi pakietami *fixed location* występują grupy wolnych szczelin. Grupy szczelin można interpretować jako pojemniki, pakiety *free location* jako elementy. Dodatkowo, pakiety *free location* mogą zostać podzielone i wysłane oddzielnie, jednakże wiąże się to z dodaniem do poszczególnych powstałych fragmentów dodatkowych informacji. Druga faza rozwiązywania może być modelowana jako problem pakowania pojemników elementami częściowo podzielnymi.

Problemy pakowania elementów częściowo podzielnych pozwalają na podzielność elementu w pewnym ograniczonym stopniu. Różni autorzy różnie podchodzili do modelowania tego zjawiska. W publikacji [11] zaproponowano narzucenie ograniczenia na liczbę fragmentów, z których składa się element. W publikacjach [3, 11] zaproponowano podejście, w którym fragmentacja wiąże się z obecnością pewnego narzutu realizowanego poprzez zwiększenie rozmiaru fragmentów. Ma to modelować sytuacje, w których dzielenie lub łączenie elementów powoduje pewne straty materiału wynikające z procesu technologicznego. Kolejnym z podejść jest dodanie pewnej funkcji kosztu zwieszającej się wraz z kolejnymi podziałami, jednakże nie wpływającej na rozmiar fragmentów [3].

Niniejsza praca skupia się na ograniczeniu możliwości dzielenia elementów poprzez wprowadzenie wartości wyznaczającej minimalny rozmiar fragmentu powstałego w wyniku podziału elementu. Prowadzi to do definicji problemu pakowania pojemników w wariancie częściowo podzielnym [6].

Problem BPP_β

Dany zbiór elementów $j \in N$ o rozmiarach w_j należy zapakować do jak najmniejszej liczby pojemników o rozmiarach C. Podczas pakowania elementy można dzielić na mniejsze fragmenty pod warunkiem, że rozmiary tych fragmentów nie będą mniejsze niż β.

Parametr β jest nieujemny oraz określa dopuszczalny zakres podziału elementów, jednocześnie ograniczając liczbę części na jakie mogą być podzielone elementy. Model ten reprezentuje sytuacje, w których fragmenty materiału mniejsze niż β mogą być problematyczne w kontekście ich przetwarzania lub być zbyt małe ze względu na naturę problemu.

Problem BPP_β zgodnie z [6] jest problemem klasy złożoności $NP - trudne$.

1.3 Wykorzystana notacja

- C – rozmiar pojemnika
- M – zbiór pojemników
- m – liczność zbioru pojemników
- N – zbiór elementów

- n – liczność zbioru elementów
- w_j – rozmiar elementu o indeksie j
- c_i – wolne miejsce w pojemniku i
- β – parametr podzielności, minimalna wielkość fragmentu powstałego w wyniku podziału elementu
- Element podzielny – element o rozmiarze $w \geq 2\beta$
- Element niepodzielny – element o rozmiarze $w < 2\beta$

2. Algorytmy rozwiązywania BPP_β

2.1 Proste algorytmy listowe

Metody rozwiązywania BPP_β są w dużej mierze adaptacjami algorytmów rozwiązywania BPP. Niniejszy rozdział będzie dotyczył adaptacji najprostszych algorytmów listowych Best Fit, Bin First Bit i Bin Best Fit do problemu pakowania z elementami częściowo podzielnymi. Algorytmy Bin First Fit (BinFF) oraz Bin Best Fit (BinBF) są algorytmami zorientowanymi na pojemniki i w trakcie swojego działania próbują zapakować kolejne pojemniki – w przypadku BinFF elementami o najmniejszych numerach indeksu, które da się zapakować do aktualnie pakowanego pojemnika, a w przypadku BinBF elementami, które po dodaniu do aktualnego pojemnika pozostawią najmniej wolnego miejsca.

Pakowanie elementów w BPP_β różni się od pakowania elementów w BPP faktem, że analizując element j możliwe jest zapakowanie do pojemnika niekoniecznie całego elementu o rozmiarze w_j, ale również jego fragment o rozmiarze z przedziału $[\beta,\ w_j - \beta]$. Rozmiar fragmentu nie może być mniejszy niż β, więc minimalnym rozmiarem fragmentu jest β, a maksymalnym $w_j - \beta$, tak aby drugi z fragmentów również był co najmniej równy β. Zgodnie z tą zasadą, w pracy [6] została zaproponowana następująca reguła pakowania elementów częściowo podzielnych do pojemników.

Reguła pakowania 1

Dla analizowanego elementu j i pojemnika i:

- Jeżeli $w_j \leq c_i$ (rozmiar elementu j jest mniejszy bądź równy od wolnego miejsca w pojemniku i) to zapakuj elementu j do pojemnika i,
- Jeżeli $w_j > c_i$, $w_j \geq 2\beta$ i $c_i \geq \beta$ to podziel element j oraz zapakuj do pojemnika i element o rozmiarze równym $\min\{c_i, w_j - \beta\}$,
- W przeciwnym przypadku, nie pakuj elementu j do pojemnika i.

Zgodnie z tą regułą w pracy [6] został zdefiniowany algorytm $BinFF_\beta$.

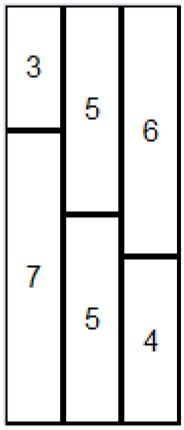
Algorytm $BinFF_\beta$ (Bin First Fit)

1. Utwórz listę elementów L.
2. Pobierz kolejny pusty pojemnik i.
3. Dla pojemnika i, znajdź element j z listy L o najmniejszym numerze indeksu, który da się zapakować do pojemnika i zgodnie z regułą pakowania 1. Jeżeli podczas pakowania dojdzie do podziału elementu, pozostały po podziale fragment dodaj do listy L.
4. Jeżeli pojemnik nie jest pusty oraz istnieją elementy, które da się do niego zapakować, wróć do punktu 3.
5. Jeżeli pojemnik jest pełen lub nie istnieją elementy, które da się do niego zapakować oraz lista L nie jest pusta, wróć do punktu 2.
6. Jeżeli lista elementów L jest pusta, zakończ działanie algorytmu.

Przykład działania algorytmu $BinFF_\beta$

Dane wejściowe: $C = 10, \beta = 3, L = \{7, 5, 4, 8, 6\}$

Pobierany jest pojemnik o indeksie 1. Zapakowany zostaje element o rozmiarze 7. Następnie analizowane są elementy o rozmiarach 5 i 4 ale nie udaje się ich zapakować do pojemnika. Element o rozmiarze 8 jest dzielony na dwa fragmenty o rozmiarach 5 i 3. Fragment o rozmiarze 3 jest zapakowany do pojemnika o indeksie 1, tym samym zapełniając go. Fragment o rozmiarze 5 jest zwracany do listy L. Pobierany jest drugi pojemnik oraz zapakowany do niego element o wielkości 5. W kolejnym kroku do pojemnika zostaje zapakowany kolejny element o rozmiarze 5, tym samym zapełniając pojemnik. Następnie pobierany jest trzeci pojemnik i zapakowane do niego kolejno elementy o wielkościach 4 i 6. W ten sposób zapełniony zostaje 3 pojemnik oraz zapakowane zostają wszystkie elementy z listy L. Rysunek 2.1 przedstawia upakowanie wygenerowane w przykładzie.



Rysunek 2.1. Rezultat pracy algorytmu BinFF$_\beta$ dla opisanego przykładu. Kolumny oznaczają pojemniki, białe prostokąty to zapakowane elementy.

W publikacji [6] został zdefiniowany również algorytm BinBF$_\beta$ różniący się od algorytmu BinFF$_\beta$ sposobem wyboru pakowanego elementu.

Algorytm BinBF$_\beta$ (Bin Best Fit)

1. Utwórz listę elementów L.
2. Pobierz kolejny pusty pojemnik i.
3. Dla pojemnika i, znajdź element j z listy L po dodaniu którego, zgodnie z regułą pakowania 1, w pojemniku i zostanie najmniej miejsca. Jeżeli podczas dodawania, dojdzie do podziału elementu, pozostały po podziale fragment dodaj do listy L.
4. Jeżeli pojemnik nie jest pusty oraz istnieją elementy, które da się do niego zapakować, wróć do punktu 3.
5. Jeżeli pojemnik jest pełen lub nie istnieją takie elementy, które da się do niego zapakować oraz lista L nie jest pusta, wróć do punktu 2.
6. Jeżeli lista elementów L jest pusta, zakończ działanie algorytmu.

Eksperymentalne badania [6] pokazują, że w większości przypadków algorytm BinBF$_\beta$ generuje upakowania lepsze niż algorytm BinFF$_\beta$.

Algorytmy BinFF$_\beta$ i BinBF$_\beta$ nie zakładają wcześniejszego uporządkowania listy elementów. Istnieją ich warianty wykorzystujące sortowanie – BinFFD$_\beta$ (Bin First Fit Decreasing), BinFFI$_\beta$ (Bin First Fit Increasing), BinBFD$_\beta$ (Bin Best Fit Decreasing), BinBFI$_\beta$ (Bin Best Fit Increasing). Wstępne sortowanie poprawia średnią jakość upakowań generowanych przez opisane algorytmy.

2.2 Algorytm BinFFSL$_\beta$

W pracy [6] został zdefiniowany algorytm BinFFSL$_\beta$ (ang. Bin First Fit Small Large). Jest on znacznie lepszy od algorytmów opisanych w rozdziale 2.1. Dokładniejsze wykorzystanie pojemników wynika z wykorzystywania dużych i podzielnych elementów, głównie do dopełniania wolnego miejsca w pojemnikach. W pierwszej kolejności zostają pakowane elementy niepodzielne, z uwagi na ich mniejszą elastyczność. Zgodnie z tym podejściem zaproponowana została reguła pakowania, która pakuje pojemniki do pełna albo do pojemności co najwyżej $C - \beta$.

Dzięki temu, tak zapakowane pojemniki można dopełnić elementami o rozmiarach $w_j > 2\beta$. Najlepsze wyniki algorytm osiąga w przypadku występowania elementów $\gg 2\beta$.

Reguła pakowania 2

Dla analizowanego elementu j i pojemnika i:

- Jeżeli $w_j = c_i$ lub $w_j \leq c_i - \beta$ zapakuj cały element j do pojemnika i,
- Jeżeli $w_j \in (c_i - \beta, c_i)$ oraz element j jest podzielny, to zapakuj tylko fragment elementu j o wielkości $w_j - \beta$,
- Jeżeli $w_j \in (c_i, c_i + \beta)$, $c_i \geq 2\beta$ oraz element j jest podzielny, to zapakuj fragment elementu j o wielkości $c_i - \beta$,
- Jeżeli $w_j \geq c_i + \beta$ oraz element j jest podzielny, to zapakuj fragment elementu j o wielkości c_i,
- Jeżeli nie zachodzi żaden z powyższych warunków, nie pakuj elementu j do pojemnika i.

Korzystanie z reguły pakowania 2 ma sens wtedy, gdy struktura elementów pozostałych do zapakowania daje nadzieję na dopełnianie pojemników – czyli gdy istnieją elementy podzielne i są one w stanie dopełnić pojemnik. W związku z tym, zaproponowane są następujące warunki rezygnowania z reguły pakowania 2 na rzecz reguły pakowania 1:

- W1 – pierwszy element na liście L jest niepodzielny i jego rozmiar jest większy niż $C - \beta$ (ale nie większy od c_i),
- W2 – wszystkie pozostałe elementy w liście L są niepodzielne,
- W3 – zachodzi $c_i < 2\beta$ i rozmiary w_j wszystkich pozostałych elementów na liście L są w przedziale $(c_i - \beta, c_i + \beta)$.

Po wystąpieniu któregoś z tych warunków algorytm dokańcza pakowanie danego pojemnika stosując algorytm BinBF$_\beta$ (wyjątkowo, gdy wystąpi warunek W1, pakowany (zgodnie z regułą pakowania 1) jest element podczas analizy którego wystąpił i dopiero po zapakowaniu jego, algorytm zmienia działanie na zgodne z BinBF$_\beta$).

Schemat algorytmu BinFFSL$_\beta$ jest następujący.

Algorytm BinFFSL$_\beta$ (Bin First Fit Small Large$_\beta$)

1. Utwórz listę elementów L zgodnie z uporządkowaniem:
   a. Na początku elementy niepodzielne (o rozmiarze $w_j < 2\beta$) w kolejności nierosnącej,
   b. Następnie elementy podzielne (o rozmiarze $w_j \geq 2\beta$) w kolejności niemalejącej.
2. Pobierz kolejny pusty pojemnik i.
3. Dopóki pojemnik nie jest pełny i nie jest spełniony warunek W1, W2 lub W3 zastosuj regułę pakowania 2 dla kolejnych elementów j z listy L. Elementy zapakowane usuń z listy L. Fragmenty, które pozostały po podziale elementów umieść w liście L zachowując sposób jej uporządkowania.
4. Jeżeli wystąpił warunek W1 zapakuj element j do pojemnika i zgodnie z regułą pakowania 1.
5. Jeżeli pojemnik i nie jest pusty zastosuj dla niego algorytm BinBF$_\beta$.

6. Jeżeli lista L jest pusta to zakończ działanie algorytmu, w przeciwnym przypadku idź do punktu 2.

W publikacji [6] opisano szereg pozytywnych własności algorytmu BinFFSL$_\beta$ w przypadku występowania dużych podzielnych elementów. Udowodnione zostało następujące twierdzenie.

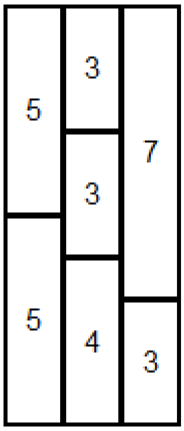
Twierdzenie. *Jeżeli dla instancji I problemu BPP$_\beta$ zachodzi $C \geq 3\beta$ oraz $w_j \geq 3\beta$, $\forall j \epsilon N$ to algorytm BinFFSL$_\beta$ wyznacza rozwiązanie optymalne używające $\lceil \sum_{j \epsilon N} w_j / C \rceil$ pojemników.*

Przykład działania algorytmu BinFFSL$_\beta$

Dane wejściowe: $C = 10, \beta = 3, Elementy = \{7, 5, 4, 8, 6\}$

Lista elementów po posortowaniu wygląda następująco: $L = \{5, 4, 6, 7, 8\}$.

Pobierany jest pojemnik o indeksie 1. Zapakowany do niego zostaje element o rozmiarze 5. Zgodnie z regułą pakowania 2 nie zostają pakowane elementy o rozmiarach 4, 6 i 7 gdyż ich zapakowanie pozostawiałoby wolne miejsce $c_i < \beta$. Element o rozmiarze 8 jest dzielony na fragmenty o rozmiarach 5 i 3. Powstały fragment o wielkości 5 jest pakowany do pojemnika 1, dopakowując do pełna pierwszy pojemnik. Fragment o rozmiarze 3 jest dodawany do listy L zachowując jej sortowanie. Po tym zabiegu lista jest postaci: $L = \{4, 3, 6, 7\}$. Pobierany jest pusty pojemnik o indeksie 2. Pakowany do niego jest element o rozmiarze 4 a następnie element o rozmiarze 3. Element o wielkości 6 jest dzielony na dwa fragmenty, oba o rozmiarze 3. Jeden z nowo powstałych fragmentów pakowany jest do pojemnika 2, drugi dodawany do listy L (w tym momencie $L = \{3,7\}$). Tym samym zapakowany zostaje pojemnik drugi oraz pobierany pojemnik trzeci. Zostają do niego zapakowane elementy kolejno o wielkościach 3 i 7. Lista L zostaje pusta i algorytm kończy działanie generując upakowanie złożone z trzech pojemników co jest rozwiązaniem optymalnym dla zadanych danych wejściowych. Rysunek 2.2 przedstawia upakowanie wygenerowane w przykładzie.



Rysunek 2.2. Rezultat pracy algorytmu BinFFSL$_\beta$ dla opisanego przykładu. Kolumny oznaczają pojemniki, białe prostokąty to zapakowane elementy.

3. Proponowane algorytmy

3.1 Algorytm BinFFAW$_\beta$

Algorytm BinFFSL$_\beta$ stara się pakować w pierwszej kolejności elementy najmniej elastyczne. W niniejszym rozdziale opisany zostanie listowy algorytm BinFFAW$_\beta$ (ang. Bin First Fit Average Weight), który opiera się na innym podejściu do oceny elastyczności elementów. Przyjmuje się w nim, że najmniej elastyczne są elementy niepodzielne, których rozmiar jest bliski 2β. Elastyczność elementów podzielnych wynika z liczby fragmentów, na które możemy je podzielić oraz z średniej wielkości fragmentów, zakładając najczęstsze możliwe dzielenie. Element o rozmiarze 2β można podzielić na dwa małe fragmenty o rozmiarze β, zaś element o rozmiarze 2.9β, po odkrojeniu fragmentu o rozmiarze β, pozostawia fragment o rozmiarze 1.9β - który wcześniej został określony jako jeden z najmniej elastycznych. Wynika z tego pomysł sortowania listy elementów zgodnie z malejącą wartością $\frac{w_j}{\left\lfloor \frac{w_j}{\beta} \right\rfloor}$ (elementy o rozmiarach równych wartości $\frac{w_j}{\left\lfloor \frac{w_j}{\beta} \right\rfloor}$ są ustawiane malejąco zgodnie z ich rozmiarem). Wzór ten jest jednak stosowany dla elementów j takich, że $w_j > \beta$, w przeciwnym razie $\left\lfloor \frac{w_j}{\beta} \right\rfloor$ wynosi 0. Elementy o $w_j < \beta$ ustawiane są na końcu listy, w porządku malejącym, w celu dopełniania luk w pojemnikach. W wyniku przeprowadzonych eksperymentów sortowanie zostało zmodyfikowane dla problemów, w których wartość parametru $\beta \epsilon \left[\frac{C}{5}, \frac{C}{3}\right]$. Elementy o rozmiarze $w_j > C - \beta$ umieszczane są na liście, przed elementami o rozmiarach $w_j < \beta$, w porządku malejącym. Z uwagi na ich rozmiar elementy te są znacznie bardziej elastycze niż pozostałe elementy podzielne.

Podczas opracowywania algorytmu przez autora niniejszej pracy analizowane były różne warianty sortowania – jedną z większych trudności było znalezienie najlepszego miejsca na elementy o rozmiarze mniejszym niż β. Ostateczna postać sortowania jest rezultatem analizy eksperymentów i próbą znalezienia najlepszego rozwiązania dla jak największej ilości danych testowych.

W trakcie działania algorytm próbuje pakować kolejne elementy z listy. Do zapakowania konkretnego elementu wykorzystana jest reguła pakowania 1. Metoda podejmuje dodatkowe działania w sytuacji, w której po zapakowaniu elementu j do pojemnika i, w pojemniku pozostaje mniej wolnego miejsca niż β. Jest to sytuacja, w której nie da się otrzymać fragmentu, który dało by się zapakować. Metoda próbuje znaleźć element lepiej wypełniający pojemnik i niż element j. W tym celu szuka elementu, przeglądając dalej listę, którego zapakowanie zamiast elementu j pozostawi w pojemniku najmniej wolnego miejsca. O ile taki element istnieje, algorytm pakuje go do pojemnika i zamiast elementu j. Heurystyka ta nie występuje w przypadku pakowania pierwszego elementu do pojemnika. Pakowanie pojemnika zaczyna się od zapakowania pierwszego elementu z listy L, który w ramach tej metody jest uważany za najmniej elastyczny. Elementem zamiennym byłby w tym przypadku bardzo duży element lub fragment o znacznie większej elastyczności. Następnie, jeżeli pojemnik nie został zapakowany do pełna, algorytm kontynuuje przeglądanie listy od miejsca występowania największego elementu niepodzielnego, mniejszego lub równego wolnemu miejscu w pojemniku.

Pełny schemat algorytmu BinFFAW$_\beta$ jest przedstawiony poniżej.

Algorytm BinFFAW$_\beta$:

1. Utwórz listę elementów L.
2. Posortuj listę L zgodnie z następującymi zasadami:

    a. Elementy o wadze w przedziale $w_j \in [\beta, C-\beta]$ dla $\beta \in \left[\frac{C}{5}, \frac{C}{3}\right]$ oraz $w_j \in (\beta, \infty)$ dla pozostałych wartości parametru β posortuj w porządku malejącym względem wartości $\frac{w_j}{\left\lceil \frac{w_j}{\beta} \right\rceil}$, a w przypadku równych wartości $\frac{w_j}{\left\lceil \frac{w_j}{\beta} \right\rceil}$ na podstawie rozmiaru elementów. Posortowane elementy dodaj do listy L.
    b. Dla $\beta \in \left[\frac{C}{5}, \frac{C}{3}\right]$ - ustaw elementy o wadze w przedziale $w_j \in (C-\beta,\ \infty)$ w porządku niemalejącym. Posortowane elementy dodaj na koniec listy L.
    c. Elementy o $w_j \le \beta$ posortuj w porządku nierosnącym. Posortowane elementy dodaj na koniec listy L.
3. Pobierz kolejny pusty pojemnik i.
4. Dopóki pojemnik nie jest pełny zastosuj regułę pakowania 1 dla kolejnych elementów j z listy L. Elementy zapakowane usuń z listy L. Fragmenty, które pozostały po podziale elementów umieść w liście L zachowując sposób jej uporządkowania.
    a. Jeżeli po zapakowaniu elementu j do pojemnika i w pojemniku zostaje mniej wolnego miejsca niż β ($c_i < \beta$), to przeglądaj kolejne elementy szukając elementu lepiej wypełniającego pojemnik i niż element j. Zakończ poszukiwania, jeżeli znajdziesz element, który dokładnie wypełnia pojemnik albo przejrzysz wszystkie elementy. Jeżeli znajdzie się taki element, zapakuj go zamiast elementu j.
    b. Jeżeli po kroku 4a. w pojemniku pozostanie wolne miejsce ($c_i \neq 0$) ustaw aktualny element na największy element niepodzielny o wartości $w_j \le c_i$.
5. Jeżeli lista L jest pusta to zakończ działanie algorytmu, w przeciwnym przypadku idź do punktu 3.

Analiza złożoności:

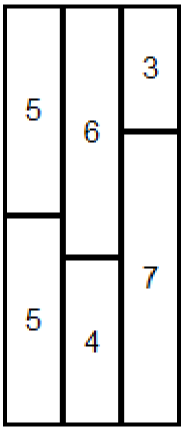
Algorytm ma złożoność wielomianową i jest ona rzędu $O(n^2)$. Na złożoność składa się sortowanie $O(\text{n} \log \text{n})$ oraz co najwyżej podwójne przejrzenie listy dla każdego pojemnika. Potencjalne drugie przejrzenie listy polega na cofnięciu się do elementów niepodzielnych jeżeli w pojemniku zostanie wolne miejsce $c_i < \beta$ po zakończeniu kroku 4a.

Przykład działania algorytmu BinFFAW$_\beta$

Dane wejściowe: $C = 10, \beta = 3, Elementy = \{7, 5, 4, 8, 6\}$

Lista elementów po posortowaniu wygląda następująco: $L = \{5, 4, 7, 6, 8\}$.

Pobierany zostaje pojemnik o indeksie 1. Pakowany zostaje do niego element o rozmiarze 5. Element o wielkości 4 może być zapakowany, ale jego zapakowanie pozostawiłoby w pojemniku wolną przestrzeń o rozmiarze 1. Sprawdzane zostają kolejne elementy (zapakowanie fragmentu o rozmiarze 4 z elementu o rozmiarze 7 pozostawi również wolne miejsce o wielkości 1, zapakowanie fragmentu o rozmiarze 3 z elementu o rozmiarze 6 pozostawi wolne miejsce o wielkości 2), aż do znalezienia elementu o rozmiarze 8. Jest on dzielony na fragmenty o rozmiarach 5 i 3. Fragment o wielkości 5 pakowany jest do pierwszego pojemnika tym samym zapełniając go. Fragment o wielkości 3 jest dodawany do listy L, która po jego dodaniu wygląda następująco $L = \{4, 7, 6, 3\}$. Pobierany jest drugi pojemnik i zapakowany zostaje do niego element o romiarze 4. Zapakowanie fragmentu o rozmiarze 4 z elementu 7 pozostawiłoby w pojemniku wolne miejsce o rozmiarze 2. Zapakowanie elementu o wielkości 6 dopełnia pojemnik. Pobierany zostaje trzeci pojemnik. Pakowany do niego jest element o rozmiarze 7, a następnie element o rozmiarze 3. Algorytm kończy swoje działanie gdyż lista elementów jest pusta. Zostały użyte 3 pojemniki. Jest to rozwiązanie optymalne dla tego przykładu. Rysunek 3.1 przedstawia upakowanie wygenerowane w przykładzie.



Rysunek 3.1. Rezultat pracy algorytmu BinFFAW$_\beta$ dla opisanego przykładu. Kolumny oznaczają pojemniki, białe prostokąty to zapakowane elementy.

3.2 Algorytm BinFFSL'$_\beta$

Autor niniejszej pracy, znaczą część czasu przeznaczył na próby modyfikacji algorytmu BinFFSL$_\beta$ jako, że w badaniach przeprowadzonych w pracy [6] okazał się najlepszym algorytmem do rozwiązywania problemu pakowania elementów częściowo podzielnych.

Efektem tej analizy jest powstanie nowego algorytmu nazwanego BinFFSL'$_\beta$ w trzech wariantach.

Pierwszy wariant algorytmu BinFFSL'$_\beta$ oznaczany jako BinFFSL'1$_\beta$, różni się od algorytmu BinFFSL$_\beta$ działaniem kroku 3, a dokładniej, sposobem przeglądania elementów. W BinFFSL$_\beta$, po zapakowaniu elementu j jest analizowany element, który przed zapakowaniem elementu j, miał indeks $j + 1$. Jest to konsekwencja heurystyki mówiącej, że każdy kolejny element pakowany do pojemnika, musi być coraz bardziej elastyczny (czyli zgodnie z logiką sortowania listy, najmniej elastyczne są duże elementy niepodzielne, a najbardziej elastyczne są duże elementy podzielne). Takie podejście przeglądania listy pozwala na osiągnięcie bardzo sensownej złożoności algorytmu $O(n^2)$ wynikającej z jednego przebiegu listy w kroku 3 i jednego przebiegu listy w kroku 4. W metodzie BinFFSL'1$_\beta$ po zapakowaniu niepodzielnego elementu j przeglądanie listy L rozpoczynane jest od początku. Heurystyka ta jest zainspirowana poniższą obserwacją.

Analizowany jest niepodzielny element rozmiaru $\beta + a$, wolne miejsce wynosi $2 * \beta + b$, gdzie $b < a, 0 < a < \beta$. Taki element nie jest dodawany do pojemnika, gdyż pozostałoby tam niecałe β miejsca. Załóżmy, że najbliższy element, który da się zapakować zgodnie z regułą pakowania 2 ma rozmiar $\beta + b - a$. Po zapakowaniu go do pojemnika, wolne miejsce wyniesie $\beta + a$. W standardowej wersji algorytmu, iterowane będą kolejne elementy i pozostała luka zostanie wypełniona albo bardzo małym elementem oraz częścią elementu długiego, albo jedynie częścią elementu długiego. W wersji zmodyfikowanej wykorzystany zostanie element $\beta + a$.

W efekcie, metoda BinFFSL'1$_\beta$ unika fragmentacji elementów podzielnych, gdy istnieje możliwość dopełnienia pojemnika elementami niepodzielnymi. Modyfikacja ta pogarsza złożoność algorytmu, gdyż po dodaniu każdego elementu niepodzielnego, cześć listy L jest przeglądana kolejny raz.

Algorytm BinFFSL'1$_\beta$

1. Utwórz listę elementów L zgodnie z uporządkowaniem:

   a. Na początku elementy niepodzielne (o rozmiarze $w_j < 2\beta$) w kolejności nierosnącej,
   b. Następnie elementy podzielne (o rozmiarze $w_j \geq 2\beta$) w kolejności niemalejącej.
2. Pobierz kolejny pusty pojemnik i.
3. Dopóki pojemnik nie jest pełny i nie jest spełniony warunek W1, W2 lub W3 zastosuj regułę pakowania 2 dla kolejnych elementów j z listy L. Elementy zapakowane usuń z listy L. Fragmenty, które pozostały po podziale elementów umieść w liście L zachowując sposób jej uporządkowania. W przypadku zapakowania elementu niepodzielnego, stosowanie reguły pakowania 2 kontynuuj od pierwszego elementu na liście L.
4. Jeżeli wystąpił warunek W1 zapakuj element j do pojemnika i zgodnie z regułą pakowania 1.
5. Jeżeli pojemnik i nie jest pusty zastosuj dla niego algorytm BinBF$_\beta$.
6. Jeżeli lista L jest pusta to zakończ działanie algorytmu, w przeciwnym przypadku idź do punktu 2.

Drugi wariant algorytmu BinFFSL'$_\beta$ oznaczany poprzez BinFFSL'2$_\beta$ różni się od algorytmu BinFFSL'$_\beta$ sortowaniem dla przypadków, w których parametr β ma wartość większą niż $(\max_{j\epsilon N} w_j)/3$. W BinFFSL$_\beta$ elementy podzielne są sortowane zgodnie z porządkiem niemalejącym, gdyż im element jest dłuższy tym można podzielić go na więcej części – co za tym idzie, jest bardziej elastyczny. Jednak dla przypadku gdy $\beta > (\max_{j\epsilon N} w_j)/3$ elementy podzielne mogą być dzielone na dokładnie dwa fragmenty. Wtedy bardziej elastyczne są elementy mniejsze, gdyż średni rozmiar elementów powstałych w wyniku podziału jest mniejszy. Sortowanie tych elementów w algorytmie BinFFSL'2$_\beta$ odbywa się w porządku nierosnącym. Modyfikacja ta nie wpływa na zmianę złożoności algorytmu.

Algorytm BinFFSL'2$_\beta$

1. Utwórz listę elementów L zgodnie z uporządkowaniem:
   a. Na początku elementy niepodzielne (o rozmiarze $w_j < 2\beta$) w kolejności nierosnącej,
   b. Następnie elementy podzielne (o rozmiarze $w_j \geq 2\beta$) ustaw: dla $\beta \leq (\max_{j\epsilon N} w_j)/3$ w kolejności niemalejącej; w przeciwnym przypadku, w kolejności nierosnącej.
2. Pobierz kolejny pusty pojemnik i.
3. Dopóki pojemnik nie jest pełny i nie jest spełniony warunek W1, W2 lub W3 zastosuj regułę pakowania 2 dla kolejnych elementów j z listy L. Elementy zapakowane usuń z listy L. Fragmenty, które pozostały po podziale elementów umieść w liście L zachowując sposób jej uporządkowania.
4. Jeżeli wystąpił warunek W1 zapakuj element j do pojemnika i zgodnie z regułą pakowania 1.
5. Jeżeli pojemnik i nie jest pusty zastosuj dla niego algorytm BinBF$_\beta$.
6. Jeżeli lista L jest pusta to zakończ działanie algorytmu, w przeciwnym przypadku idź do punktu 2.

Trzeci wariant algorytmu BinFFSL'$_\beta$ oznaczony jest poprzez BinFFSL'1+2$_\beta$. Jest to algorytm powstały jako połączenie pierwszego i drugiego wariantu BinFFSL'$_\beta$.

Algorytm BinFFSL'1+2$_\beta$

1. Utwórz listę elementów L zgodnie z uporządkowaniem:
   a. Na początku elementy niepodzielne (o rozmiarze $w_j < 2\beta$) w kolejności nierosnącej,
   b. Następnie elementy podzielne (o rozmiarze $w_j \geq 2\beta$) ustaw: dla $\beta \leq (\max_{j\epsilon N} w_j)/3$ w kolejności niemalejącej; w przeciwnym przypadku, w kolejności nierosnącej.

2. Pobierz kolejny pusty pojemnik i.
3. Dopóki pojemnik nie jest pełny i nie jest spełniony warunek W1, W2 lub W3 zastosuj regułę pakowania 2 dla kolejnych elementów j z listy L. Elementy zapakowane usuń z listy L. Fragmenty, które pozostały po podziale elementów umieść w liście L zachowując sposób jej uporządkowania. W przypadku zapakowania elementu niepodzielnego, stosowanie reguły pakowania 2 kontynuuj od pierwszego elementu na liście L.
4. Jeżeli wystąpił warunek W1 zapakuj element j do pojemnika i zgodnie z regułą pakowania 1.
5. Jeżeli pojemnik i nie jest pusty zastosuj dla niego algorytm BinBF_β.
6. Jeżeli lista L jest pusta to zakończ działanie algorytmu, w przeciwnym przypadku idź do punktu 2.

3.3 Połączenie BinFFAW_β oraz BinFFSL_β

Algorytmy BinFFAW_β oraz BinFFSL_β zostały zaprojektowane tak, aby unikać takiego pakowania pojemników, że pozostaje w nich wolne miejsce w przedziale $c_i < \beta$. Obie te metody mają podobną strukturę działania – sortują elementy zgodnie z ich elastycznością oraz pakują elementy zgodnie z pewną heurystyką, mającą na celu uniknąć pozostawiania w pojemnikach mniej niż β wolnego miejsca. W związku z tym, istnieje możliwość połączenia ich i stworzenia dwóch nowych algorytmów opartych o sortowanie z jednego oraz heurystykę z drugiego.

Algorytm BinFFSL'3_β wykorzystuje sortowanie z algorytmu BinFFSL_β, a schemat pakowania algorytmu BinFFAW_β. Schemat ten nie wymaga zmian w celu dostosowania do innego sortowania.

Algorytm BinFFSL'3_β

1. Utwórz listę elementów L zgodnie z uporządkowaniem:
   a. Na początku elementy niepodzielne (o rozmiarze $w_j < 2\beta$) w kolejności nierosnącej,
   b. Następnie elementy podzielne (o rozmiarze $w_j \geq 2\beta$) w kolejności niemalejącej.
2. Pobierz kolejny pusty pojemnik $\boldsymbol{i}$.
3. Dopóki pojemnik nie jest pełny zastosuj regułę pakowania 1 dla kolejnych elementów j z listy L. Elementy zapakowane usuń z listy L. Fragmenty, które pozostały po podziale elementów umieść w liście L zachowując sposób jej uporządkowania.
   a. Jeżeli po zapakowaniu elementu j do pojemnika i w pojemniku zostaje mniej wolnego miejsca niż β ($c_i < \beta$), to przeglądaj kolejne elementy szukając elementu lepiej wypełniającego pojemnik i niż element j. Zakończ poszukiwania, jeżeli znajdziesz element, który dokładnie wypełnia pojemnik albo przejrzysz wszystkie elementy. Jeżeli znajdzie się taki element, zapakuj go zamiast elementu j.
   b. Jeżeli po kroku 4a. w pojemniku pozostanie wolne miejsce ($c_i \neq 0$) ustaw aktualny element na największy element niepodzielny o wartości $w_j \leq c_i$.
4. Jeżeli lista L jest pusta to zakończ działanie algorytmu, w przeciwnym przypadku idź do punktu 3.

Algorytm BinFFAWP2_β wykorzystuje sortowanie występujące w algorytmie BinFFAW_β oraz schemat pakowania występujący w algorytmie BinFFSL_β. Reguła pakowania 2 jest niezależna od sortowania. Inaczej jest w przypadku warunków zaniechania reguły pakowania 2 na rzecz reguły pakowania 1. Dla algorytmu BinFFAWP2_β warunek W1 jest postaci:

- W1 – aktualny element jest niepodzielny i jego rozmiar jest większy niż $C - \beta$ (ale nie większy od c_i), Warunki W2 i W3 brzmią identycznie jak w przypadku BinFFSL$_\beta$. Warunek W3 jest kłopotliwy implementacyjnie - w przypadku sortowania z BinFFSL$_\beta$ można go bardzo łatwo sprawdzić, w przypadku sortowania z BinFFAW$_\beta$ wyszukanie najmniejszego i największego elementu w liście jest bardziej kłopotliwe - nie jest wspierane bezpośrednio sortowaniem.

Algorytm BinFFAWP2$_\beta$

1. Utwórz listę elementów $\boldsymbol{L}$.
2. Posortuj listę L zgodnie z następującymi zasadami:
   a. Elementy o wadze w przedziale $w_j \in [\beta, C-\beta]$ dla $\beta \in \left[\frac{C}{5}, \frac{C}{3}\right]$ oraz $w_j \in (\beta, \infty)$ dla pozostałych wartości parametru β posortuj w porządku malejącym względem wartości $\frac{w_j}{\left\lceil \frac{w_j}{\beta} \right\rceil}$, a w przypadku równych wartości $\frac{w_j}{\left\lceil \frac{w_j}{\beta} \right\rceil}$ na podstawie rozmiaru elementów. Posortowane elementy dodaj do listy L.
   b. Dla $\beta \in \left[\frac{C}{5}, \frac{C}{3}\right]$ - ustaw elementy o wadze w przedziale $w_j \in (C-\beta,\ \infty)$ w porządku niemalejącym. Posortowane elementy dodaj na koniec listy L.
   c. Elementy o $w_j \leq \beta$ posortuj w porządku nierosnącym. Posortowane elementy dodaj na koniec listy L.
3. Pobierz kolejny pusty pojemnik i.
4. Dopóki pojemnik nie jest pełny i nie jest spełniony warunek W1, W2 lub W3 zastosuj regułę pakowania 2 dla kolejnych elementów j z listy L. Elementy zapakowane usuń z listy L. Fragmenty, które pozostały po podziale elementów umieść w liście L zachowując sposób jej uporządkowania. W przypadku zapakowania elementu niepodzielnego, stosowanie reguły pakowania 2 kontynuuj od pierwszego elementu na liście L.
5. Jeżeli wystąpił warunek W1 zapakuj element j do pojemnika i zgodnie z regułą pakowania 1.
6. Jeżeli pojemnik i nie jest pusty zastosuj dla niego algorytm BinBF$_\beta$.
7. Jeżeli lista L jest pusta to zakończ działanie algorytmu, w przeciwnym przypadku idź do punktu 2.

3.4 Algorytmy poprawy

Dotychczas, analizowane były algorytmy konstrukcyjne – deterministycznie generujące rozwiązanie jedynie na podstawie zbioru elementów oraz parametrów wejściowych. Inną klasą algorytmów są algorytmy poprawy. W przypadku problemu BPP$_\beta$ przyjmują one na wejściu, poza parametrami, pewne upakowanie startowe i modyfikują je w określony sposób w celu poprawienia upakowania.

Podstawowy algorytm poprawy (PAP) opiera się na założeniu, że w otrzymanym upakowaniu pojemniki pełne są zapakowane zadowalająco, zaś pojemniki posiadające wolną przestrzeń są do poprawienia. Metoda usuwa z upakowania pojemniki niepełne, a elementy do nich zapakowane, dodaje do nowej listy L_P. Następnie, powstała lista jest przekazywana do pewnego algorytmu rozwiązywania BPP$_\beta$, w szczególności tego, który wygenerował początkowe upakowanie (jednakże nie jest to konieczne, dopuszczalne jest wykorzystanie innego algorytmu, prostego gdy najważniejsze jest szybkie działanie lub bardziej złożonego, gdy najistotniejsza jest minimalizacja liczby pojemników upakowania).

Podstawowy Algorytm Poprawy

1. Pobierz startowe upakowanie U wygenerowane dla problemu BPP_{β}.
2. Z upakowania U wyjmij pojemniki niepełne zaś elementy zapakowane do tych pojemników dodaj do listy L_P.
3. Uruchom algorytm rozwiązywania problemu BPP_{β} (np. ten sam, który wygenerował to upakowanie) na liście L_P.
4. Połącz upakowanie otrzymane w kroku 3 z upakowaniem U. Jeżeli otrzymane upakowanie jest lepsze niż niezmodyfikowane U, zakończ algorytm i zwróć zmodyfikowane upakowanie. W przeciwnym przypadku zakończ algorytm i zwróć niezmodyfikowane upakowanie U.

Podstawowy algorytm poprawy jest algorytmem deterministycznym, w związku z tym, przyniesie poprawę jedynie dla specyficznych upakowań, dla których elementy znajdujące się w pojemnikach niepełnych zostaną zapakowane lepiej zadanym algorytmem.

Podstawowy algorytm poprawy można zmodyfikować, poprzez uruchomienie go na liczniejszej liście L_P. Należy zatem do tej listy dodać elementy również z części pełnych pojemników startowego upakowania. W związku z różną naturą upakowań generowanych przez różne algorytmy ciężko znaleźć odpowiednią metodę wyboru konkretnych pojemników. Jednym z możliwych rozwiązań jest wprowadzenie czynnika losowego i wyboru pojemnika z zadanym prawdopodobieństwem. Dla zwiększenia prawdopodobieństwa rozsądnego wyboru pełnych pojemników do uwzględnienia, procedura jest powtarzana kilkukrotnie. Podczas każdej próby poprawy zwiększane jest prawdopodobieństwo dodania pełnego pojemnika – ma to spowodować oddalenie się od rozwiązania poprzedniego. Dodatkowo w pierwszej iteracji algorytmu sugerowane jest, aby prawdopodobieństwo wyboru pełnego pojemnika wynosiło 0, aby zapewnić, że wyniki będą zawsze co najmniej tak dobre jak wyniki PAP.

Randomizowany Algorytm Poprawy

1. Pobierz startowe upakowanie U wygenerowane dla problemu BPP_{β}.
2. Ustaw $i = 0$, gdzie i jest numerem aktualnej iteracji.
3. Dopóki $i < k$, gdzie k jest zadaną liczbą iteracji:
    a. Wyczyść L_P.
    b. Z upakowania U wyjmij pojemniki niepełne zaś elementy zapakowane do tych pojemników dodaj do listy L_P. Upakowanie z wyjętymi pojemnikami niepełnymi oznacz jako U^-.
    c. Z prawdopodobieństwem $p + i * q$, gdzie p i q są zadanymi parametrami, wyjmuj kolejne pojemniki z upakowania U^-, a elementy z nich dodawaj do listy L_P. W przypadku pierwszego wykonania tego kroku, p i q mają wartość 0.
    d. Uruchom algorytm rozwiązywania problemu BPP_{β} (np. ten sam, który wygenerował to upakowanie) na liście L_P. Otrzymane upakowanie oznaczmy U^+.
    e. Jeżeli upakowanie $U_{nowe} = U^+ \cup U^-$ jest złożone z mniejszej liczby pojemników niż upakowanie U, przypisz $U = U_{nowe}$, ustaw i = 0. W przeciwnym przypadku zwiększ i o jeden.
    f. Jeżeli upakowanie uzyskane w kroku e korzysta z takiej samej liczby pojemników co upakowanie optymalne dla problemu pakowania pojemników w wariancie ciągłym dla tych samych danych (wzór (1)), zakończ algorytm i zwróć U.
4. Jeżeli $i = k$, zakończ algorytm i zwróć upakowanie U.

Złożoność algorytmu jest rzędu $O(k * m) * złożoność\ algorytmu\ alokacji$.

W kroku 4f sprawdzanie rozmiaru optymalnego rozwiązania problemu dyskretnego odbywa się poprzez porównanie liczby pojemników w upakowaniu U z wartością $\left\lceil \frac{\sum_{j \in N} w_j}{\beta} \right\rceil$.

W kroku 3c występują parametry p i q. Są to odpowiednio prawdopodobieństwo bazowe wyboru pojemnika pełnego (na podstawie eksperymentów, sensowna jest wartość 10%) oraz parametr, zwiększający prawdopodobieństwo dodania pełnego pojemnika wraz z liczbą iteracji (na podstawie wyników przeprowadzanych eksperymentów, sensowna jest wartość w przedziale $\left[\frac{10\%}{k}, \frac{20\%}{k}\right]$). Algorytm ten wykonuje się znacznie dłużej niż Podstawowy Algorytm Poprawiający co wynika z k-krotnego wyliczania rozwiązania, jednakże daję znacznie większą nadzieję na poprawienie rozwiązania i zawsze zwraca upakowania nie gorsze od upakowania zwracanego przez PAP.

4. Eksperymenty i porównanie algorytmów

4.1 Źródła danych testowych i notacja

Eksperymenty opisane w niniejszej pracy były przeprowadzane na zbiorach danych wygenerowanych przez prof. dr. Armina Scholl z Uniwersytetu Friedricha Schillera w Jenie oraz dr. Roberta Kleina z Politechniki w Darmstadt w 2003 roku pobranych ze strony Uniwersytetu Friedricha Schillera w Jenie [8, 9, 10]. Zostały przygotowane przez nich 3 zbiory danych generowane losowo oraz różniące się charakterystyką i strukturą. Zbiory te wygenerowane zostały dla problemu BPP. Autorzy podali również optymalną liczbę pojemników dla podanych instancji problemu BPP.

Pierwszy ze zbiorów *"Data set 1 for BPP-1"* zawiera 720 instancji problemu. Instancje są w grupach po 20 sztuk o identycznych cechach. Cechy rozróżniające zbiory to:

- n – liczba elementów
  - potencjalne wartości: 50, 100, 200, 500
- w_j – przedział z którego elementy są generowane
  - potencjalne wartości: $[1, 100]$, [20, 100], [30, 100]
- c – rozmiar pojemników
  - potencjalne wartości: 100, 120, 150

Każda instancja jest określona nazwą "NxCyWz_v", gdzie symbole x, y, z i v są zastępowane przez konkretne liczby.

- $n = 50\ dla\ x = 1;\ n = 100\ dla\ x = 2;\ n = 200\ dla\ x = 3;\ n = 500\ dla\ x = 4$
- $c = 100\ dla\ y = 1;\ c = 120\ dla\ y = 2;\ c = 150\ dla\ \ y = 3$
- $\forall_{j \epsilon N}\ w_j\ \epsilon\ [1, 100]\ dla\ \ z = 1;\ w_j\ \epsilon\ [20, 100]\ dla\ z = 2;\ w_j\ \epsilon\ [30, 100]\ dla\ z = 4$
- $v\ \epsilon\ A \dots T$ – konkretne litery odpowiadają jednej z 20 instancji cechowanej określonymi parametrami N, C, W

Drugi ze zbiorów *"Data set 2 for BPP-1"* zawiera 480 instancji problemu. Instancje te są w grupach po 10 sztuk o identycznych cechach. Cechy rozróżniające zbiory są identyczne jak w przypadku zbioru *"Data set 1 for BPP-1"*, jednakże inna jest charakterystyka rozmiaru pojedynczego elementu. Występują tutaj parametry średniej wielkości elementu *avgWeight* oraz maksymalnego odchylenia od średniej oznaczonego jako *delta*. *avgWeight* może przyjąć wartości ze zbioru $\left\{\frac{c}{3}, \frac{c}{5}, \frac{c}{9}\right\}$. *delta* może przyjąć wartości $\{20\%, 50\%, 90\%\}$.

Każda instancja jest określona nazwą struktury "NxWyBzRv", gdzie symbole x, y, z i v są zastępowane przez konkretne liczby.

- $n = 50\ dla\ x = 1;\ n = 100\ dla\ x = 2;\ n = 200\ dla\ x = 3;\ n = 500\ dla\ x = 4$
- $avgWeight = \frac{c}{3}\ dla\ y = 1;\ avgWeight = \frac{c}{5}\ dla\ y = 2;\ avgWeight = \frac{c}{3}\ dla\ y = 3$
- $delta = 20\%\ dla\ z = 1;\ delta = 50\%\ dla\ z = 2;\ \ delta = 90\%\ dla\ z = 3$
- $v\ \epsilon\ \{0, 1, \dots, 9\}$ – konkretne liczby odpowiadają jednej z 10 instancji cechowanej określonymi parametrami N, W, B

Trzeci ze zbiorów *"Data set 3 for BPP-1"* zawiera 10 instancji problemu o takiej samej charakterystyce.

$$n = 200, c = 100000, \forall_{j \epsilon N} \; w_j \; \epsilon \; [20000, 35000]$$

Są to najtrudniejsze przypadki testowe dla problemu BPP. Każda instancja jest określona nazwą "HARDv", gdzie $v \; \epsilon \; \{0, 1, \dots, 9\}$ i odpowiada za identyfikacje konkretnej instancji.

Wszystkie eksperymenty przeprowadzone w tej pracy korzystają z opisanych zbiorów danych. Na potrzeby eksperymentów elementy należące do opisanych zbiorów są podzielne, a o stopniu ich podzielności decyduje parametr β określający minimalny rozmiar fragmentu oraz będący parametrem wejściowym wykorzystanych metod.

4.2 Implementacja algorytmów

Biblioteka implementująca algorytmy rozwiązywania problemu BPP_{β} została napisana w języku C++ 11. Wybór języka był podyktowany głównie szybkością działania - wszystkie opisane algorytmy są dość proste w implementacji, co za tym idzie, nie ma potrzeby wykorzystywania języków bardzo wysokiego poziomu, które za swą złożoność płacą wolniejszym działaniem. C++ 11 poza szybkością oferuje szeroką bibliotekę standardowych algorytmów dzięki której można uniknąć implementacji kontenerów, algorytmów sortowania oraz części kodu związanego z pobraniem danych z plików. Kod został napisany w paradygmacie obiektowym i jest biblioteką udostępniającą funkcje uruchamiające algorytmy. Implementacja odbyła się w środowisku Microsoft Visual Studio 2013 na licencji studenckiej dla studentów Politechniki Warszawskiej. Wybór narzędzia Microsoftu był podyktowany językiem, znajomością tego narzędzia oraz mnogością funkcjonalności ułatwiających pisanie oraz utrzymywanie kodu.

Do przechowywania upakowań skorzystano ze struktury typu **`deque<deque<Element>>`** - zewnętrzny kontener odpowiada za przechowywanie pojemników a wewnętrzny reprezentuje pojemnik. Klasa **`Element`** zawiera informacje o poszczególnym elemencie

```
class Element
{
public:
  //metody pobierające i ustawiające
private:
  int index;
  int value;
};
```

Wartość **`index`** jednoznacznie identyfikuje, którym elementem albo fragmentem, którego elementu zadanego na wstępie jest dany obiekt. Wartość ta propaguje się przy podziale elementów. Wartość **`value`** odpowiada wadze danego elementu lub fragmentu.

Kontener **`deque`** ze standardowej biblioteki C++ został wybrany ze względu na szybki dostęp do dowolnego elementu jak również możliwość pracy jak na kontenerze typu listowego.

W większości przypadków do sortowania wykorzystana jest funkcja `sort` ze standardowej biblioteki C++ wraz z funkcjami porównującymi elementy napisanymi odpowiednio do różnych porządków występujących w implementacji.

Eksperymenty z użyciem opisanej biblioteki odbywają się na maszynie klasy PC z procesorem Intel Core i5-4570 taktowanym częstotliwością 3.20 GHz działającym na systemie operacyjnym Windows 7.

Obsługa biblioteki

Za funkcjonalność biblioteki odpowiada klasa `AlgorthmsBBP`, której metody pozwalają na wywołanie konkretnych algorytmów. Biblioteka korzysta z pól pomocniczych jej obiektów wspólnych dla wielu algorytmów - efektem tego jest absolutny zakaz wykonywania równolegle więcej niż jednej instancji jednego algorytmu na jednym obiekcie tejże klasy. Każdy wątek powinien posiadać swój własny obiekt `AlgorthmsBBP`.

Aby rozpocząć pracę z biblioteką należy utworzyć nowy obiekt klasy `AlgorthmsBBP` oraz skonfigurować go poprzez ustawienie parametrów. Parametry są ustawianie przy pomocy metod `setPARAM(value)` gdzie "PARAM" to nazwa parametru. Dostępne parametry to:

- `B` – parametr β (domyślnie `0`)
- `C` – rozmiar pojemnika (domyślnie `0`)
- `CorrectionIterationCount` – liczba powtórzeń w przypadku RAP (domyślnie `0`)
- `P` – prawdopodobieństwo dodania pełnego pojemnika dla RAP (domyślnie `0`)
- `Q` – wzrost prawdopodobieństwa dodania pełnego pojemnika dla RAP (domyślnie `0`)
- `Debug` – parametr w przypadku ustawienia na `true` powoduje uaktywnienie szeregu zabezpieczeń w trakcie wykonywania programu, potencjalnie przydatne w przypadku wprowadzania zmian w kodzie biblioteki (domyślnie `false`)
- `PrintLvl` – parametr określa stopień dokładności wypisywania informacji o przebiegu działania algorytmów, przyjmuje wartości `0, 1, 2` (domyślnie `0`)

Aktualną wartość parametru można sprawdzić za pomocą metody `getPARAM()`.
Aby uruchomić wybraną metodę pakowania należy skorzystać z jednej z metod:

- `BFD (deque<Element> input)` - Best Fit Decreasing
- `BFDB (deque<Element> input)` - Best Fit Decreasing$_\beta$
- `BinFFDB (deque<Element> input)` - Bin First Fit Decreasing$_\beta$
- `BinBFIB (deque<Element> input)` - Bin Best Fit Increasing$_\beta$
- `BinFFSLB(deque<Element> input, bool versionPrim1, bool versionPrim2, Correction withCorrection)` - Bin First Fit Small Large$_\beta$
  - `bool versionPrim1` - ustawienie parametru powoduje uruchomienie algorytmu BinFFSL'1$_\beta$
  - `bool versionPrim2` - ustawienie parametru powoduje uruchomienie algorytmu BinFFSL'2$_\beta$
  - W przypadku ustawienia obu powyższych parametrów metoda uruchamia algorytm BinFFSL'1+2$_\beta$
  - `Correction withCorrection`- parametr pozwala na automatyczne uruchomienie algorytmu poprawy po zakończeniu pakowania, możliwe wartości: `Correction::NO`, `Correction::BASE`, `Correction::RANDOM`
- `BinFFAWB(deque<Element> input, bool withBinFFSLsort, Correction withCorrection)` - Bin First Fit Average Weight$_\beta$
  - `bool withFFSLsort` - ustawienie parametru powoduje uruchomienie algorytmu BinFFSL'3$_\beta$
- `BinFFAWP2B(deque<Element> input, Correction withCorrection)` - Bin First Fit Average Weight Pack 2$_\beta$
- `correct (Correction type, deque<deque<Element>> input, Algorithm alg)` - funkcja uruchamiająca algorytm poprawy

- `Correction type` - parametr definiujący typ algorytmu poprawy
- `Algorithm alg` - parametr pozwala wybrać funkcje pomocniczą algorytmu poprawy

Wszystkie opisane metody zwracają obiekt typu `deque<deque<Element>>` przechowujący wygenerowane upakowanie. W przypadku algorytmów konstrukcyjnych parametr `deque<Element> input` przyjmuje zbiór elementów do zapakowania. W przypadku algorytmu poprawy `deque<deque<Element>> input` przyjmuje upakowanie startowe, parametr `alg` jest związany z następującym typem wyliczeniowym:

```
enum Algorithm {
  BFD, BFDB, BinFFD, BinBFIB, BinFFAWB, BinFFAWP2B, BinFFSLB
};
```

4.3 Porównanie prostych algorytmów

W niniejszym podrozdziale omówione są eksperymenty z rozwiązywaniem problemu BPP$_\beta$ korzystając z prostych algorytmów listowych. Przytoczone wyniki będą służyć za kontekst do analizy algorytmów zaproponowanych i opisanych w pracy.

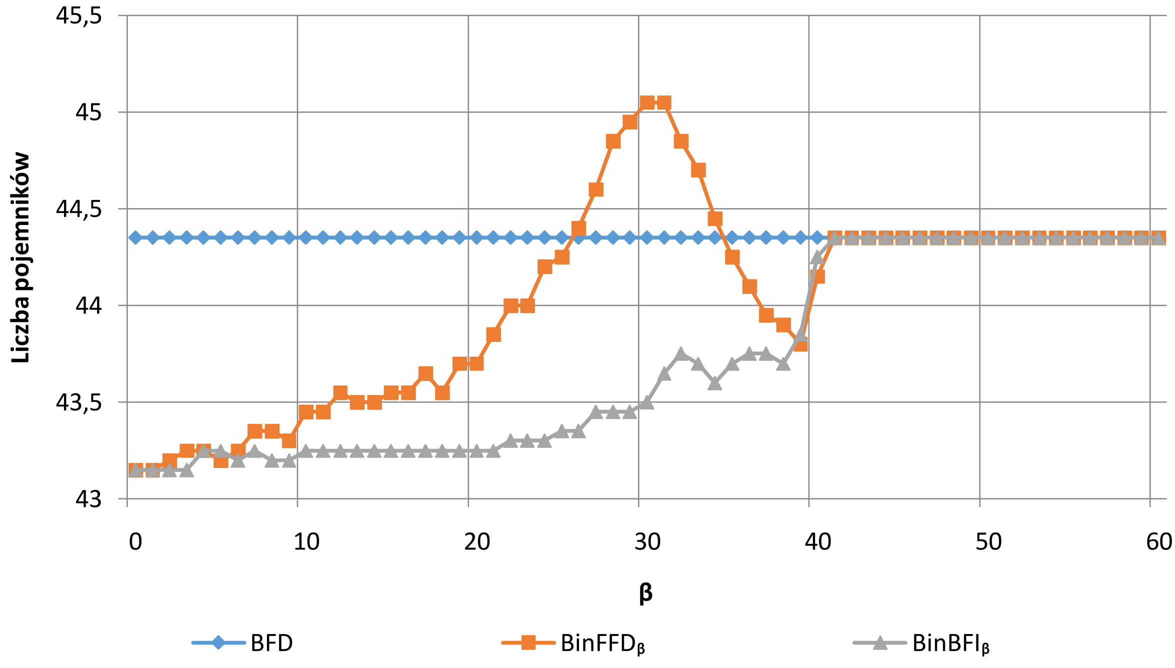
Porównanie algorytmów będzie prezentowane na 4 instancjach danych wejściowych ze zbiorów [8, 9, 10].

Do prostych algorytmów na potrzeby tej pracy zaliczają się algorytmy Bin First Fit$_{\beta,}$ Bin Best Fit$_\beta$ wraz z ich wersjami poprzedzonymi sortowaniem. Reprezentantami tej klasy algorytmów do porównań zostały wybrane BinFFD$_\beta$ oraz BinBFI$_\beta$. Przytoczone zostaną również wyniki algorytmu BFD dla klasycznego problemu pakowania (bez stosowania podziału elementów).

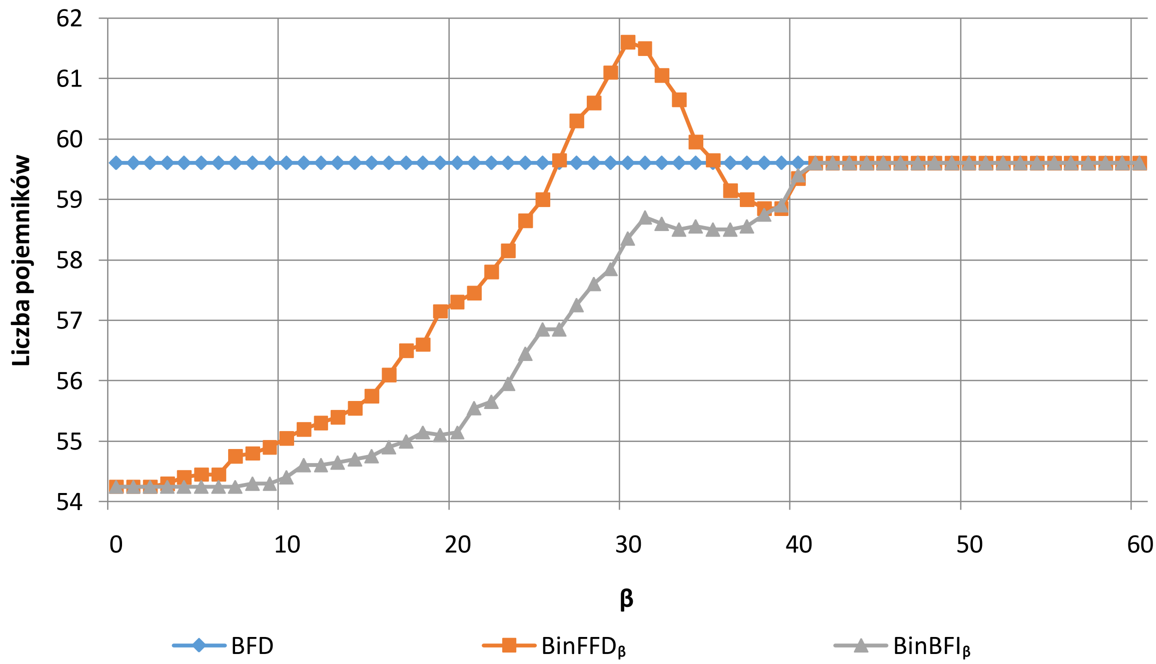
Liczba pojemników na wykresie oznacza średnią liczbę pojemników w upakowaniach wygenerowanych przez algorytm na zbiorze danych testowych N2C2W1 dla wszystkich 20 instancji problemu dla danego algorytmu. Na osi OX zaznaczona jest wartości parametru β.

Na potrzeby eksperymentów, parametr β jest całkowitoliczbowy.

Dane N2C2W1 charakteryzują się występowaniem elementów zarówno bardzo małych jak również bliskich rozmiarem wartości C. Wykres 4.1 pokazuje, że ewidentnie najlepiej radzi sobie algorytm BinBFI$_\beta$. Algorytm BinFFD$_\beta$ w pewnym, dość dużym zakresie argumentów, daje wyniki gorsze od wyników osiągniętych dla klasycznego problemu pakownia. Wraz z instancjami danych opublikowane zostały [8, 9, 10] liczności upakowań optymalnych dla problemu BPP. Algorytm BFD dla danych N2C2W1 generuje upakowania korzystające z tej samej liczby pojemników co upakowania optymalne. W związku z tym wyniki algorytmu BFD są oszacowaniem górnym sensownych wyników problemu pakowania elementów częściowo podzielnych. Oszacowaniem dolnym jest wartość optymalna rozwiązania problemu pakowania pojemników w wariancie ciągłym wyrażonego wzorem (1). Jest to wartość osiągana przez wszystkie opisywane metody rozwiązywania problemu BPP$_\beta$ dla $\beta = 0$.

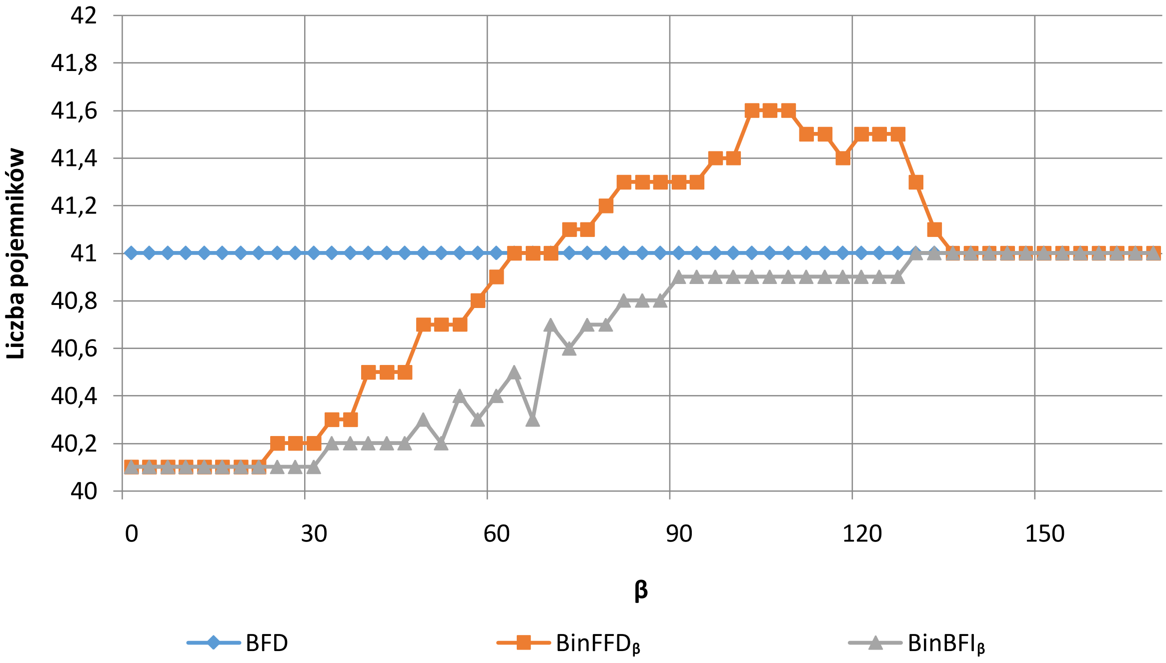


Wykres 4.1 Porównanie działania prostych algorytm dla danych N2C2W1



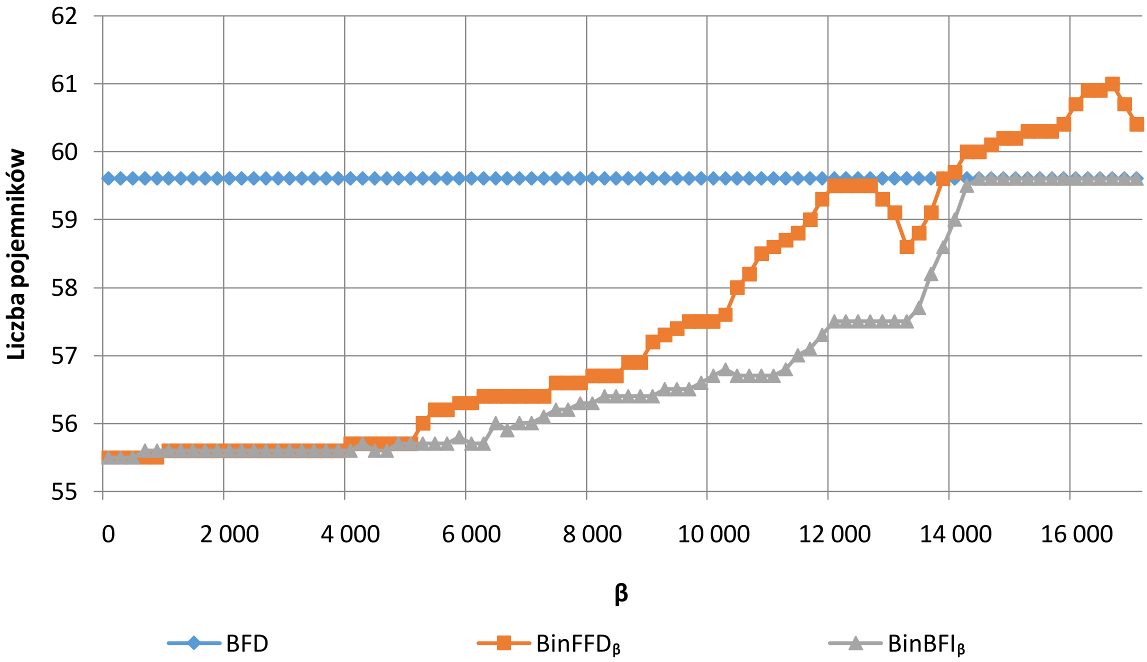
Wykres 4.2 Porównanie działania prostych algorytm dla danych N2C2W4

Analizując wykres 4.2 pokazujący rezultaty dla danych N2C2W4 można wyciągnąć podobne wnioski. Tutaj także algorytm BFD rozwiązuje klasyczny problem pakowania optymalnie. Główną zauważalną różnicą jest wyżej położona charakterystyka działania algorytmów w wariancie częściowo podzielnym względem zarówno rozwiązania klasycznego problemu pakowania jak i rozwiązania problemu ciągłego. Wynika to z braku elementów bardzo małych, które potrafią uzupełnić małe wolne przestrzenie w pojemnikach.



Wykres 4.3 Porównanie działania prostych algorytm dla danych N2W2B2

Kolejny zbiór, N2W2B2 (Wykres 4.3) zawiera dane jeszcze trudniejsze dla wariantu częściowo podzielnego. Zbiór ten charakteryzuje się średnim rodzajem elementu równym $c/_5 = 200$. Oznacza to, że już dla $\beta \geq 90$ przeciętny element jest bardzo słabo podzielny. Również dla tych danych algorytm BFD wyznacza rozwiązanie optymalne. Średnia liczba pojemników na wykresach dotyczących danych N2W2B2 jest wyliczana dla $\beta = 3 * k,\ k = 0, 1, \ldots, 56$. Linie łączące punkty są dodane dla zwiększenia czytelności wykresu.



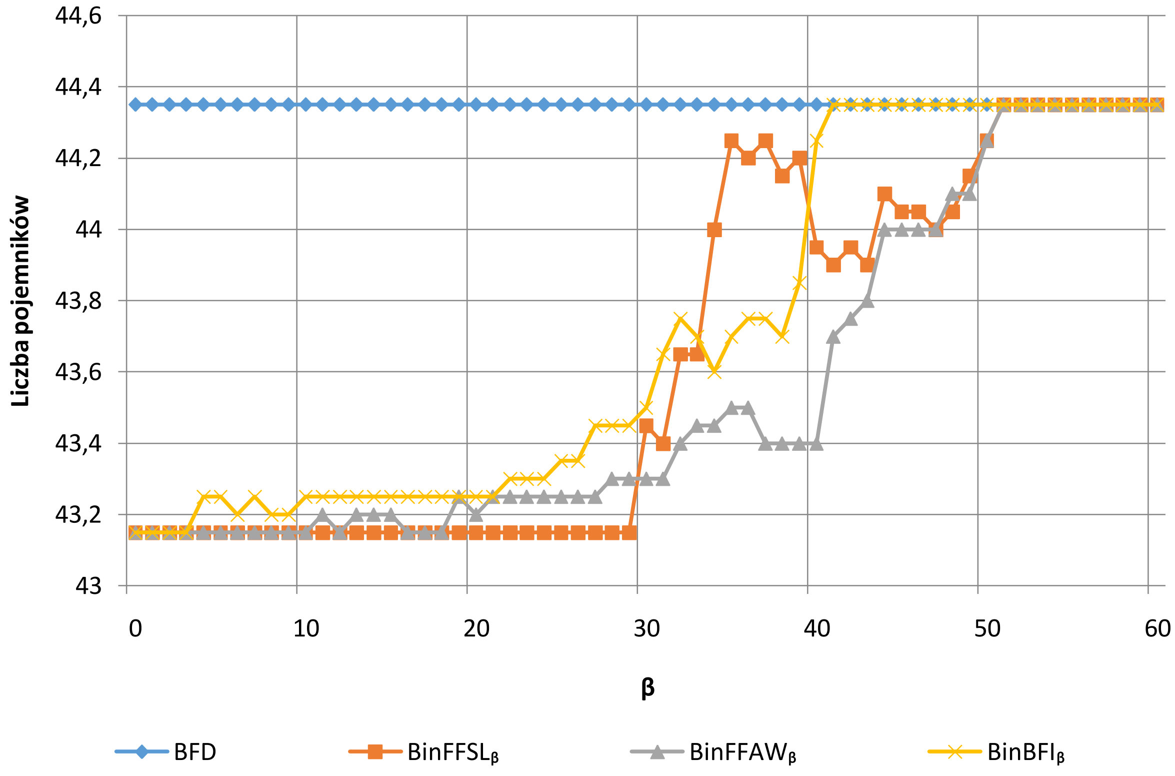
Wykres 4.4 Porównanie działania prostych algorytm dla danych HARD

Wykres 4.4 przedstawia wyniki działania prostych algorytmów dla danych HARD. Mimo nazwy wskazującej na trudność, są to jedne z prostszych danych dla problemu pakowania elementów częściowo podzielnych, gdyż największe elementy są ciągle około 3 razy mniejsze od rozmiaru pojemnika. W związku z tym, różnice między wynikami osiąganymi przez różne algorytmy dla tych danych są mniejsze. Widać tutaj dość podobne wyniki osiągane przez wszystkie metody, mimo, że najczęściej algorytmy BinFFD_β prezentują się znacznie gorzej niż BinBFI_β.

Na wykresach dla zbiorów danych HARD średnia liczba pojemników jest wyliczana dla $\beta = 200 * k,\ k = 0, 1, \dots, 100$. Wynika to z bardzo dużego rozmiaru pojedynczych elementów i małych zmianach upakowań dla bliskich sobie wartości parametru β.

Wyniki otrzymane w tym rozdziale będą służyć za wyniki referencyjne, w szczególności rezultaty osiągane przez algorytm BFD jako pewna granica jakości upakowania oraz punkt odniesienia.

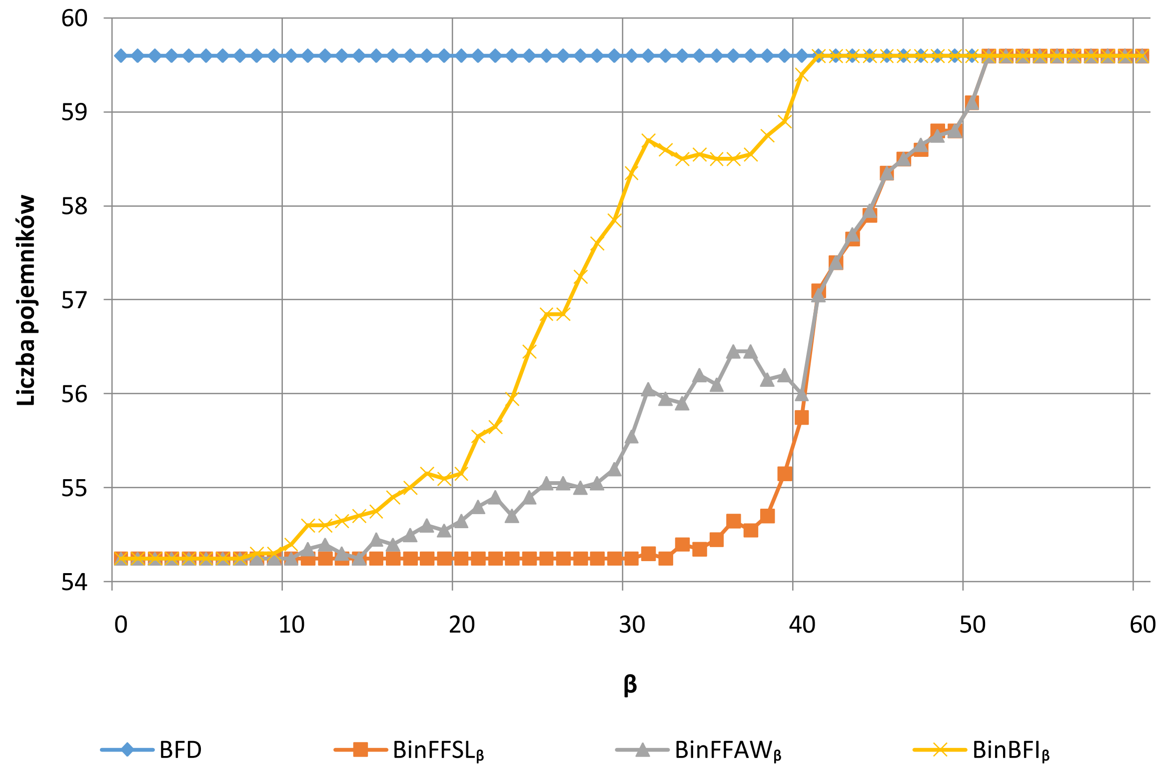
4.4 Eksperymenty z algorytmem BinFFAW_β



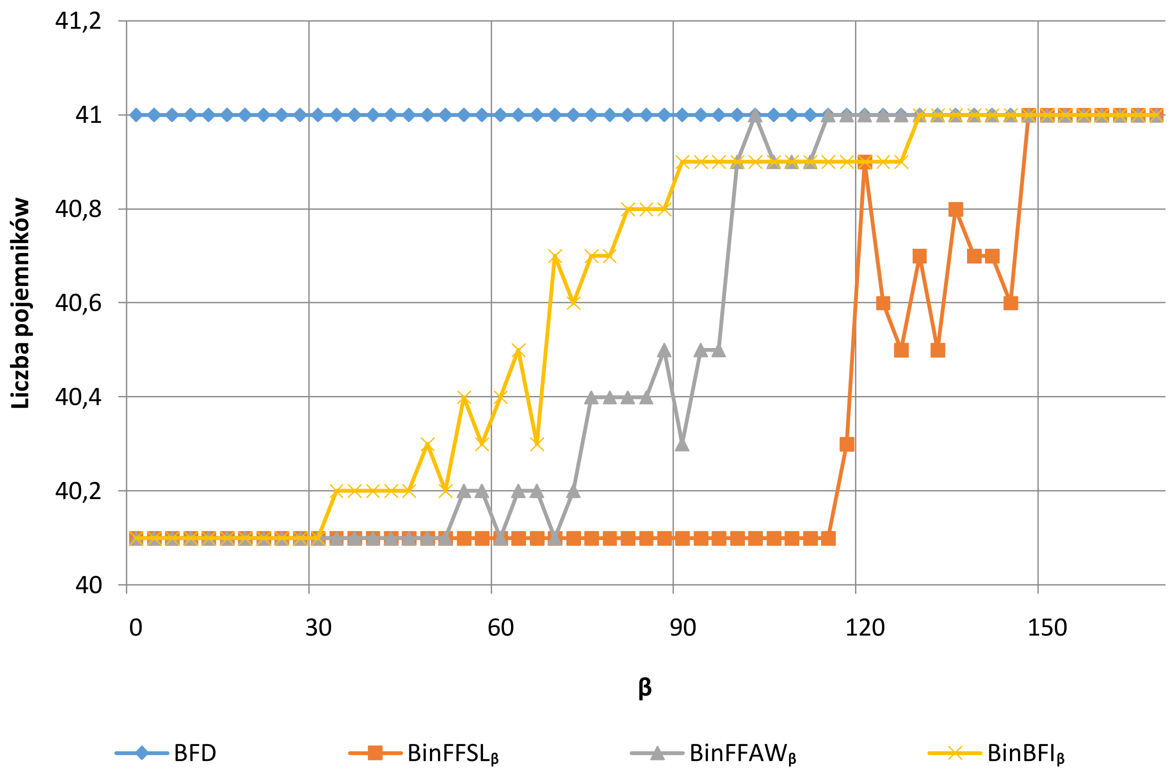
Wykres 4.5 Wyniki działania BinFFAW_β dla danych N2C2W1

Działanie i efekty osiągane przez BinFFAW_β zostaną przedstawione w porównaniu do standardowej wersji algorytmu BinFFSL_β, algorytmu BinBFI_β oraz algorytmu BFD. Na przykładzie danych N2C2W1 (Wykres 4.5) widać, że BinFFAW_β daje wyniki znacznie lepsze niż proste algorytmy oraz algorytm BinFFSL_β dla tych danych w zakresie $\beta > 30$. Różnica pomiędzy wynikami działania algorytmu BinFFSL_β a algorytmu BinFFAW_β jest bardzo duża. Jest to charakterystyczne dla danych NxCxW1 charakteryzowanych występowaniem bardzo małych elementów.

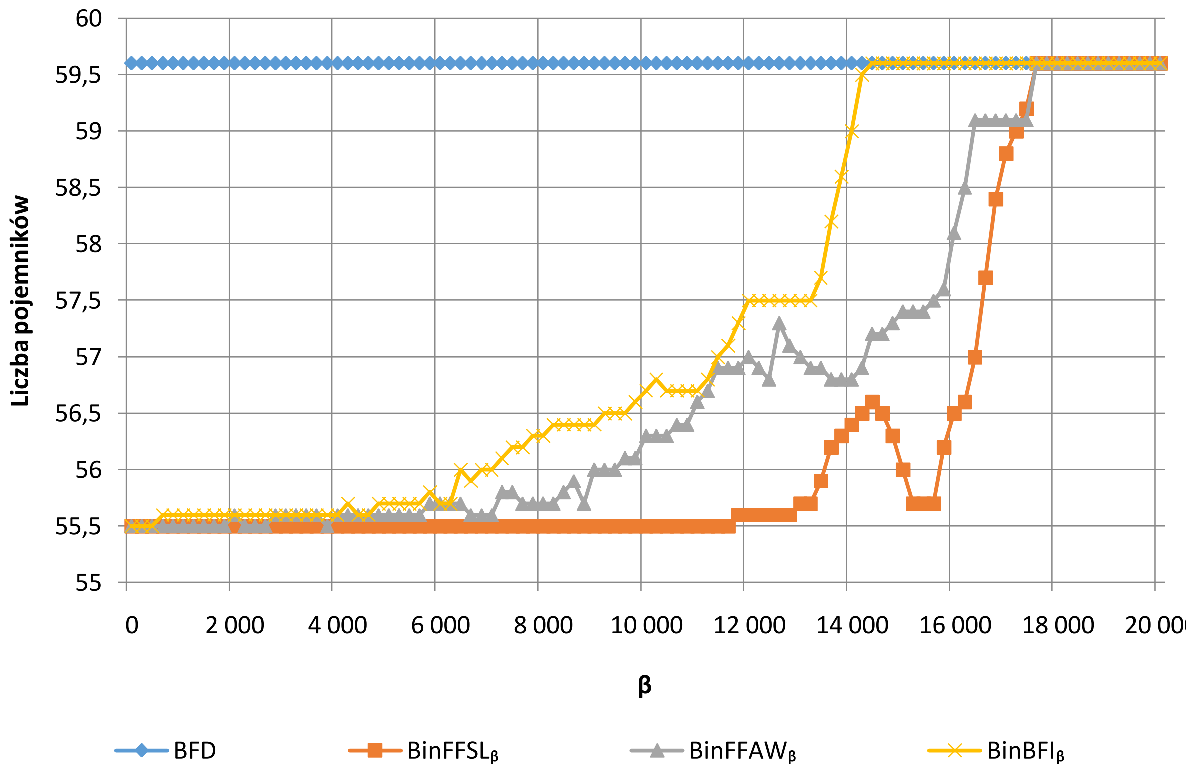
Algorytm BinFFAW_β, zgodnie z oczekiwaniami działa dość szybko, generacja upakowań dla całkowitoliczbowych wartości $\beta \in [0, 60]$ zajęła 10,73 sekundy, gdzie generacja upakowań w analogicznym przypadku dla algorytmu BinFFSL_β trwała 8,583 sekund.



Wykres 4.6 Wyniki działania algorytmu BinFFAW$_\beta$ dla danych N2C2W4



Wykres 4.7 Wyniki działania algorytmu BinFFAW$_\beta$ dla danych N2W2B2



Wykres 4.8 Wyniki działania algorytmu BinFFAW$_\beta$ dla danych HARD

Na wykresie 4.6 przedstawiającym wyniki dla danych N2C2W4 linie związane z poszczególnymi metodami są od siebie bardziej oddalone. Algorytm BinFFSL$_\beta$ daje znacznie lepsze rezultaty niż dla N2C2W1. Algorytm BinFFAW$_\beta$ osiąga wyniki znacznie lepsze niż BinBFI$_\beta$ na całym przedziale oraz bardzo podobne do wyników BinFFSL$_\beta$ dla $\beta < 10$ oraz $\beta > 40$. Na środku przedziału prezentuje się gorzej.

Wyniki dla kolejnych dwóch zbiorów danych (Wykresy 4.7 i 4.8) potwierdzają dotychczasowe spostrzeżenia. Algorytm BinFFAW$_\beta$ jest szybki i daje wyniki pomiędzy wynikami BinFFSL$_\beta$, a wynikami algorytmów prostych. Dla danych N2W2B2 dla dość niskiego β algorytm zaczął osiągać wyniki identyczne jak dla problemu dyskretnego. Te dane mają też pewną specyficzną cechę - różnica pomiędzy średnią wartością rozwiązań klasycznego problemu pakowania a problemu pakowania w wariancie ciągłym wynosi niecały 1 przy liczbach rzędu 40. Oznacza to, że wystarczy jeden pojemnik więcej wygenerowany dla jednej instancji problemu, i średnia wartość rośnie o 0,05 co na tym wykresie jest już zauważalne. Przy tak małych różnicach, ciężko o wyciąganie daleko idących wniosków.

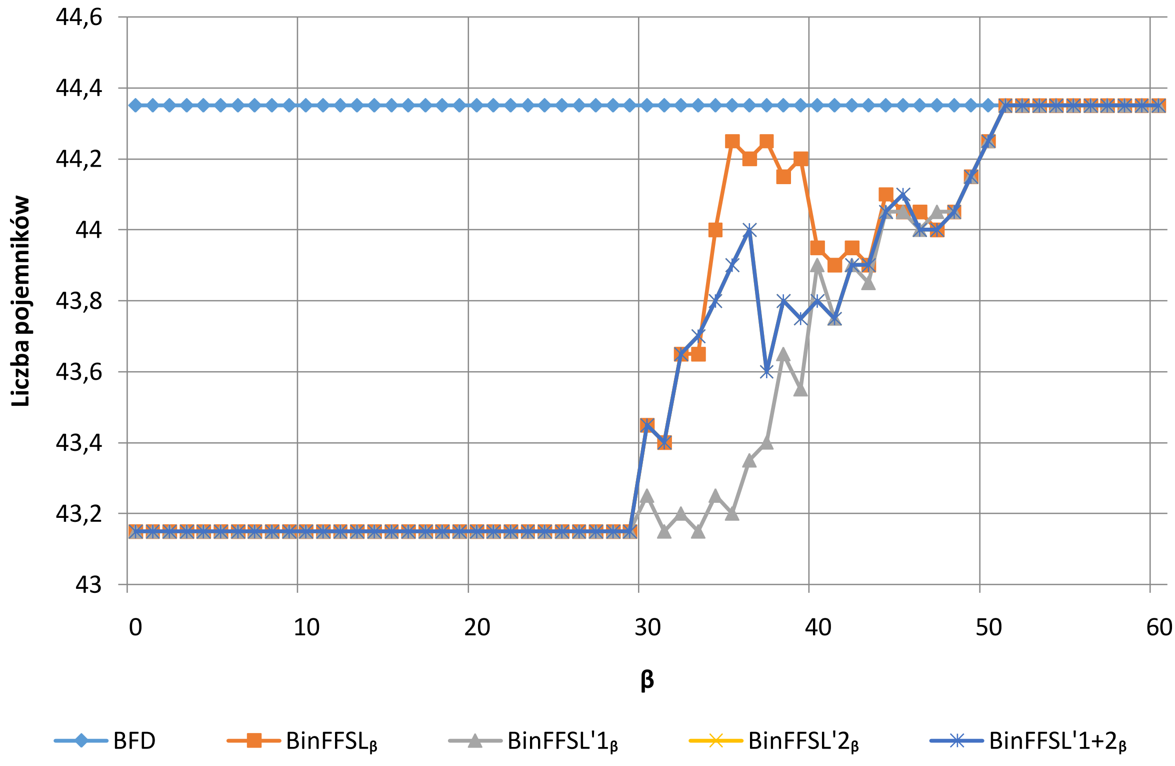
Warte do odnotowania, przy okazji analizy danych HARD jest, że BinFFAW$_\beta$ daje wyniki lepsze niż wyniki dla problemu dyskretnego dla $\beta < 17\,500$ gdzie algorytmy proste sprawdzały się jedynie do około $\beta < 14\,000$.

Podsumowując, istnieją zbiory danych dla których algorytm BinFFAW$_\beta$ sprawdza się znacznie lepiej niż BinFFSL$_\beta$. Są to zbiory o znacznej liczbie elementów bardzo małych ($w_j < \beta$). Dla pozostałych zbiorów danych, BinFFAW$_\beta$ generuje upakowania wysokiej jakości, porównywalne do upakowań generowanych przez BinFFSL$_\beta$, słabsze od nich dla średnich wartości β.

4.5 Eksperymenty z algorytmem BinFFSL'$_\beta$ w różnych wariantach

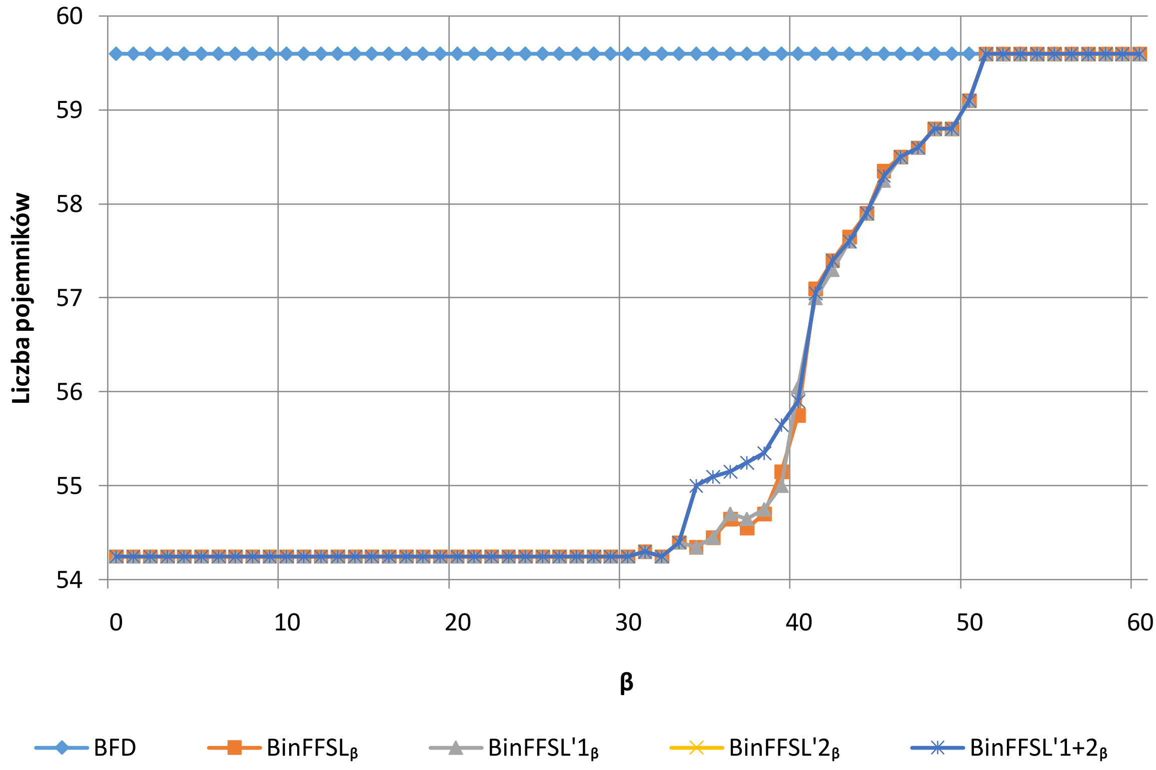
W niniejszym podrozdziale przedstawione zostaną rezultaty eksperymentów działania algorytmu BinFFSL'$_\beta$ w różnych wariantach. Algorytm oznaczony jako BinFFSL'1+2$_\beta$ jest to wariant BinFFSL'$_\beta$ zawierający zmiany zarówno z BinFFSL'1$_\beta$ jak i BinFFSL'2$_\beta$. W wariancie BinFFSL'1$_\beta$ modyfikacja polega na powtórnym przetwarzaniu listy po dodaniu elementu niepodzielnego. W BinFFSL'2$_\beta$ - zmianie uległo sortowanie.

Porównując dane z wykresów 4.9 i 4.1widać, że algorytmy z rodziny BinFFSL$_\beta$ osiągają znacznie lepsze wyniki niż algorytmy proste. W przypadku algorytmów prostych, wyniki identyczne jak dla klasycznego BPP osiągały dla $\beta > 40$, gdzie algorytmy z rodziny BinFFSL$_\beta$ dopiero dla $\beta > 51$. Dla wartości $\beta < 30$ omawiane algorytmy regularnie znajdują upakowania optymalne, na podstawie wzoru (1). Różnice pomiędzy poszczególnymi wariantami BinFFSL'$_\beta$, są widoczne dla wartości β pomiędzy 30 a 51. Modyfikacja zmieniając sortowanie dominuje nad modyfikacje z powtórnym przetwarzaniem listy elementów - w efekcie na wykresie nie widać różnic na liniach je opisujących (pomiędzy algorytmami BinFFSL2'$_\beta$ i BinFFSL'1+2$_\beta$). Bardzo dużą poprawę przynosi algorytm BinFFSL1'$_\beta$ - niestety, jest to cecha rozwiązań z *"Data set 1 for BPP-1"* z $W = 1$ - dla pozostałych zbiorów danych testowych wyniki BinFFSL1'$_\beta$ są znacznie mniej obiecujące.

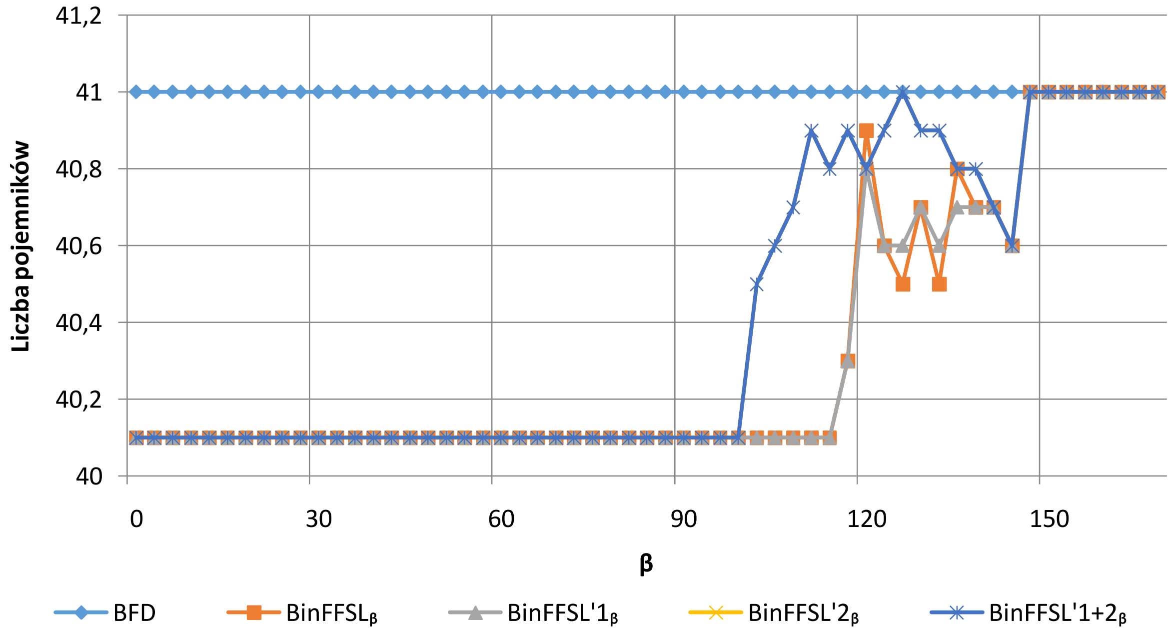


Wykres 4.9 Porównanie działania wariantów algorytmu BinFFSL'$_\beta$ dla danych N2C2W1

Również dla zbioru N2C2W4 (Wykres 4.10) algorytm BinFFSL$_\beta$ osiąga upakowania bardzo wysokiej jakości - rozwiązania zaczynają odbiegać od poziomu rozwiązania optymalnego problemu w wariancie ciągłym dopiero dla $\beta > 33$ a jest to spowodowane kompletnym brakiem elementów, które można podzielić bardziej niż na dwa fragmenty (elementy w tych zbiorach mogą mieć maksymalnie rozmiar 100 - czyli $\beta > 33$ mogą być dzielone jedynie na dwie części) oraz bardzo małą liczbą elementów podzielnych. Wpływ modyfikacji na wyniki w tym przypadku jest znacznie mniej zauważalny. BinFFSL'1_β na przedziale $30 < \beta < 45$ daje raz lepsze, raz gorsze wyniki od wersji standardowej - przy czym jest minimalnie wolniejsza (generowanie upakowań dla danych N2C2W4 dla 60 różnych wartości parametru β - czas wykonania algorytmu BinFFSL$_\beta$ - 12,191, a BinFFSL'1_β - 12,529). BinFFSL'2_β oraz BinFFSL'$1+2_\beta$ dla żadnej z wartości parametru β nie przyniosły poprawy upakowań.



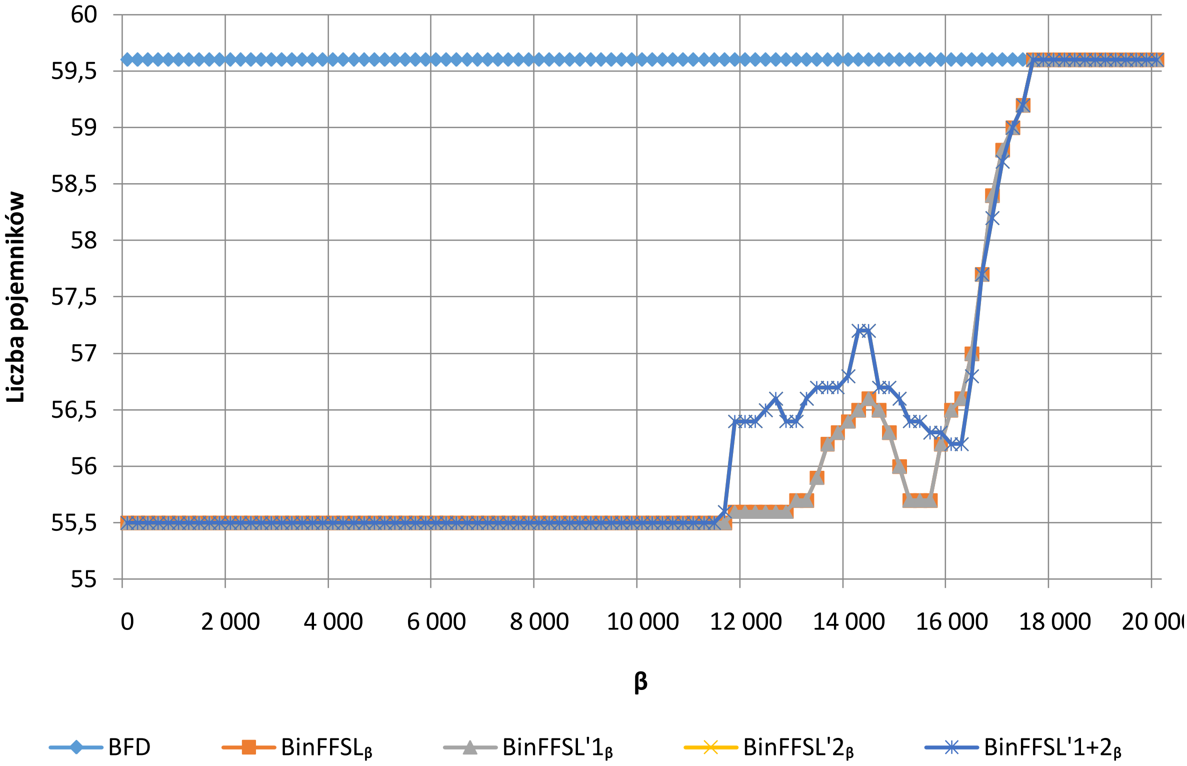
Wykres 4.10 Porównanie działania wariantów algorytmu BinFFSL'$_\beta$ dla danych N2C2W4



Wykres 4.11 Porównanie działania wariantów algorytmu BinFFSL'$_\beta$ dla danych N2W2B2

Kolejne dwa wykresy (4.11 i 4.12) potwierdzają, że algorytmy BinFFSL'2$_\beta$ oraz BinFFSL'1+2$_\beta$ (mimo poprawy rozwiązania dla danych HARD dla $\beta \approx 16\,000$) są gorsze niż standardowy BinFFSL$_\beta$. Algorytm BinFFSL'1$_\beta$ w przypadku danych N2W2B2 daje mniej więcej takie same wyniki jak standardowa wersja algorytmu, lecz w zależności od punktu, raz jest od niego gorszy a raz nie.

Podsumowując algorytm BinFFSL'2$_\beta$ okazał się korzystny tylko dla bardzo specyficznych danych i najprawdopodobniej nie warto go stosować. Stosowanie BinFFSL'1$_\beta$ daje zaś nadzieję znalezienia rozsądniejszego rozwiązania i potencjalnie warto z niego korzystać, ewentualnie próbować rozwiązywać problem oboma algorytmami i wybierać korzystniejszą opcje. Algorytm BinFFSL'1$_\beta$ generuje upakowania znacznie lepsze od BinFFSL$_\beta$ na danych NxCxW1, jednocześnie generującą upakowania na podobnym poziomie dla innych zbiorów danych. Mimo zmiany złożoności algorytm ciągle zachowuje rozsądny czas działania (np. dla wygenerowania upakowań na potrzeby wykresu dla wszystkich instancji danych HARD czas działania algorytmu BinFFSL$_\beta$ to 25,092 przy czasie BinFFSL'1$_\beta$ wynoszącym 26,176 sekundy).

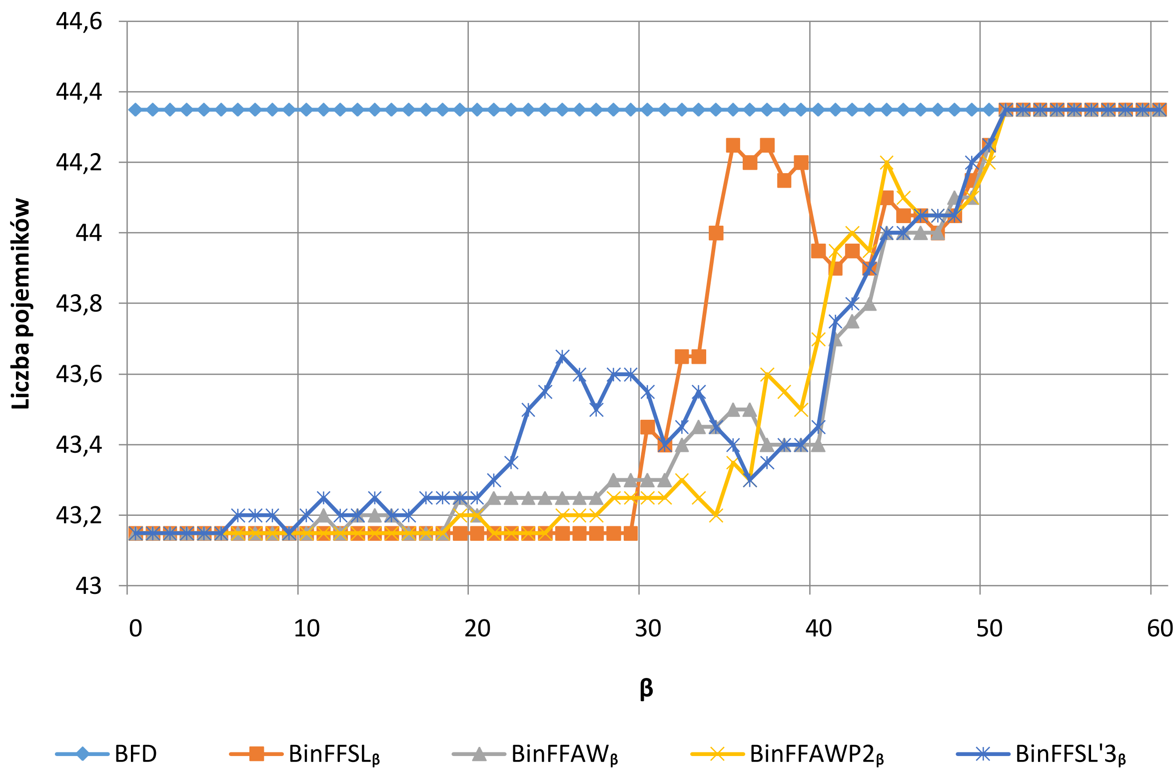


Wykres 4.12 Porównanie działania wariantów algorytmu BinFFSL'$_\beta$ dla danych HARD

4.6 Eksperymenty z algorytmami BinFFAWP2$_\beta$ i BinFFSL'3$_\beta$

Algorytmy BinFFAWP2$_\beta$ oraz BinFFSL'3$_\beta$, opisane w rozdziale 3.3, powstały jako wynik połączenia algorytmów BinFFAW$_\beta$ oraz BinFFSL$_\beta$. BinFFAWP2$_\beta$ korzysta z sortowania zdefiniowanego dla algorytmu BinFFAW$_\beta$ oraz schematu pakowania algorytmu BinFFSL$_\beta$. BinFFSL'3$_\beta$, zgodnie z nazwą, wykorzystuje sortowanie z algorytmu BinFFSL$_\beta$. W BinFFSL'3$_\beta$ schemat pakowania elementów pochodzi z BinFFAW$_\beta$.

Dla danych N2C2W1 (Wykres 4.13) wyniki BinFFSL'3$_\beta$ okazały się być gorsze niż wyniki BinFFAW$_\beta$. Zmiana sortowania w algorytmie BinFFSL$_\beta$ przynosi dość sensowne wyniki. Średnia liczba pojemników w upakowaniach utworzonych przez BinFFAWP2$_\beta$ jest niższa niż dla BinFFSL$_\beta$ i porównywalna do otrzymanej przy użyciu BinFFAW$_\beta$. Kosztem jednak jest czas wykonania - o ile algorytm BinFFSL'3$_\beta$ działa w czasie porównywalnym do BinFFAW$_\beta$, o tyle BinFFAWP2$_\beta$ działa kilkukrotnie wolniej niż BinFFSL$_\beta$ - wynika to z innej struktury listy elementów, w której na końcu nie zawsze występują elementy podzielne i algorytm BinFFAWP2$_\beta$ znacznie częściej dochodzi do sytuacji, w której dopełnia elementy algorytmem BinBF$_\beta$ który ma znacznie gorszą złożoność (dodanie każdego elementu wymaga przejrzenia całej listy elementów w przypadku sortowania proponowanego algorytmu). Poza tym, sprawdzanie warunku W3 ze zmienionym sortowaniem w zastosowanej implementacji jest bardziej czasochłonne.

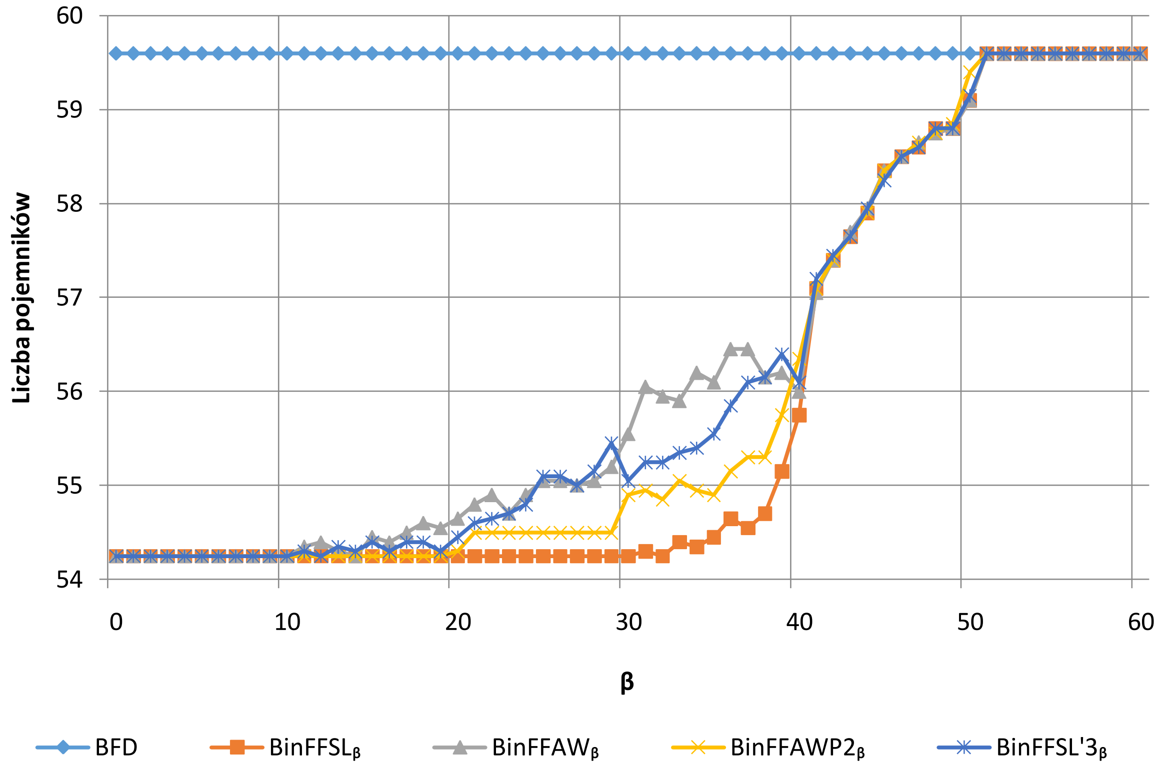


Wykres 4.13 Wyniki działania algorytmów BinFFAWP2$_\beta$ i BinFFAW'3$_\beta$ dla danych N2C2W1

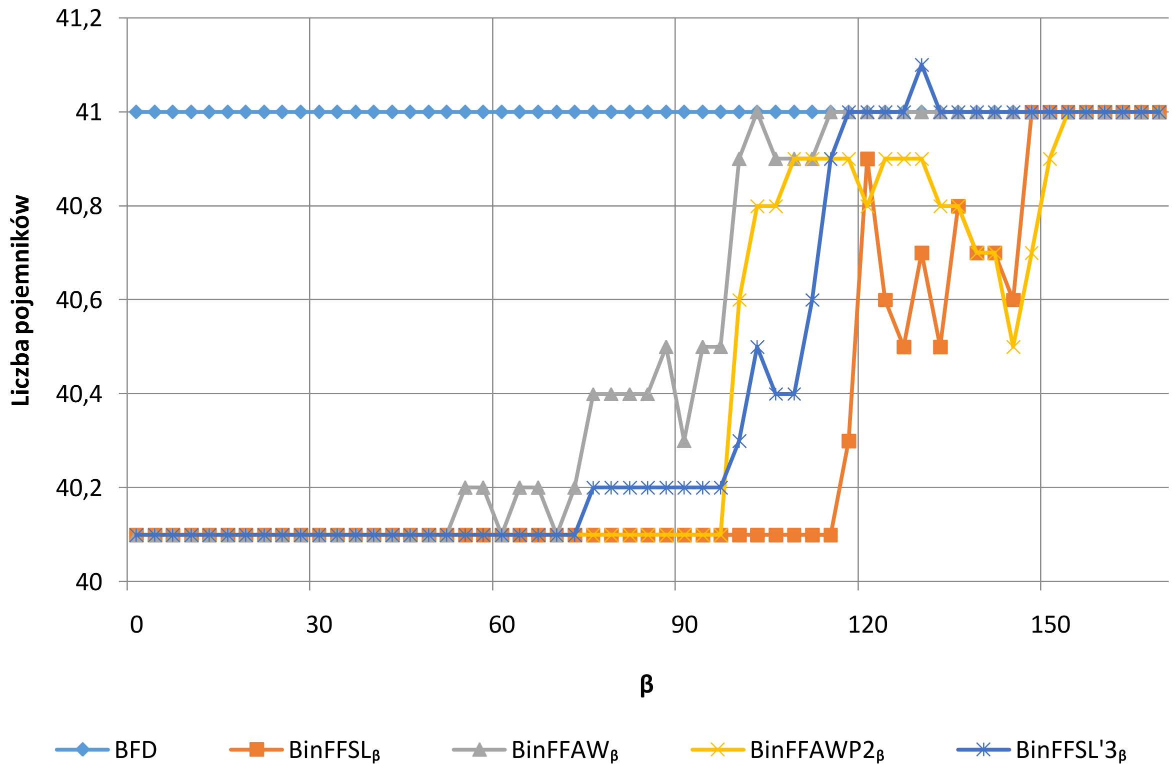
W przypadku danych N2C2W4 zgodnie z wykresem 4.14 algorytm BinFFSL'3$_\beta$ znajduję minimalnie lepsze rozwiązania od BinFFAW$_\beta$ dla $30 < \beta < 40$, zaś algorytm BinFFAWP2$_\beta$ działa gorzej niż BinFFSL$_\beta$. Również tutaj można zauważyć znaczny wzrost czasu działania BinFFAWP2$_\beta$ względem BinFFSL$_\beta$.

Wykresy 4.15 i 4.16 pokazują podobne tendencje. Wartym zauważenia jest, że zarówno BinFFSL'3$_\beta$ jak i BinFFAWP2$_\beta$ dla pewnych wartości parametru β wygenerowały upakowanie gorsze od rozwiązania klasycznego problemu pakowania dla tych samych danych.

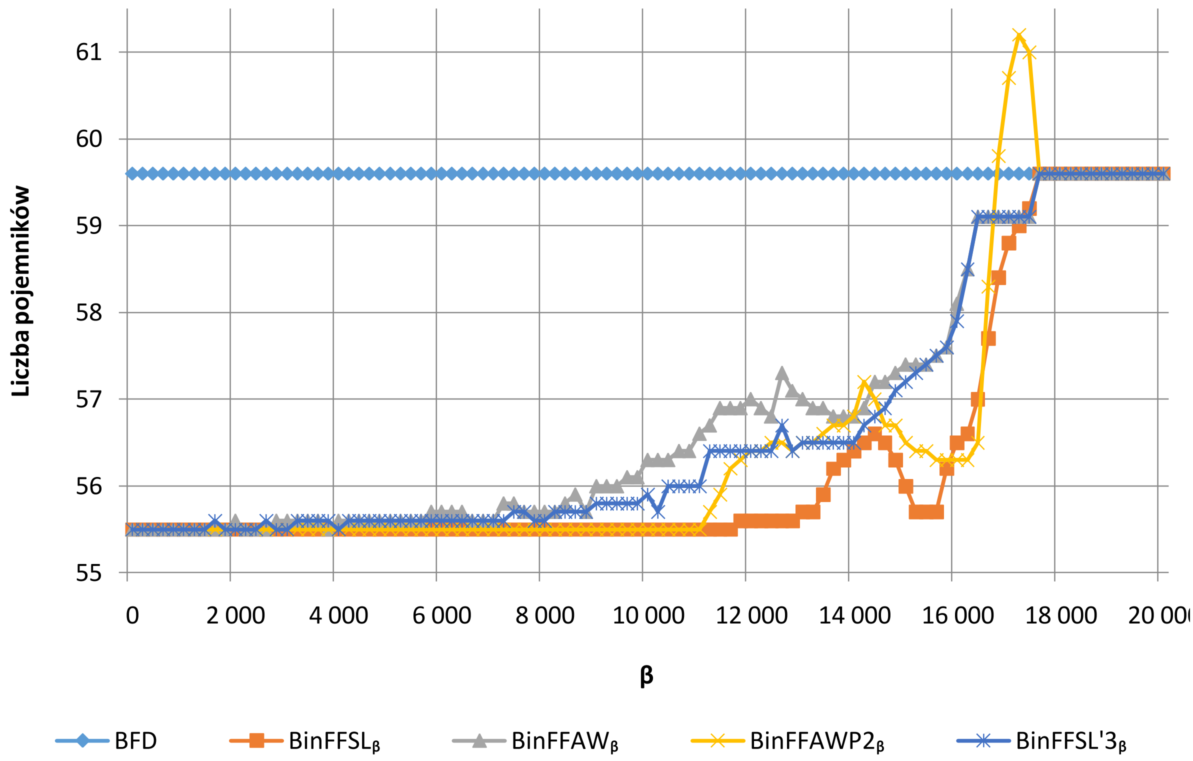
Podsumowując, zmiana sortowania w BinFFSL'3$_\beta$ dla części przypadków daje rozwiązania lepsze niż BinFFAW$_\beta$, nie wpływa również znacząco na wydajność. Przeciwnie wygląda sytuacja dla BinFFSL$_\beta$ - tutaj zmiana sortowania i stworzenie algorytmu BinFFAWP2$_\beta$ nie dość, że pogarsza osiągane rozwiązania, to jeszcze kilkukrotnie spowalnia czas działania algorytmu.



Wykres 4.14 Wyniki działania algorytmów BinFFAWP2$_{\beta}$ i BinFFAW'3$_{\beta}$ dla danych N2C2W4



Wykres 4.15 Wyniki działania algorytmów BinFFAWP2$_{\beta}$ i BinFFAW'3$_{\beta}$ dla danych N2W2B2



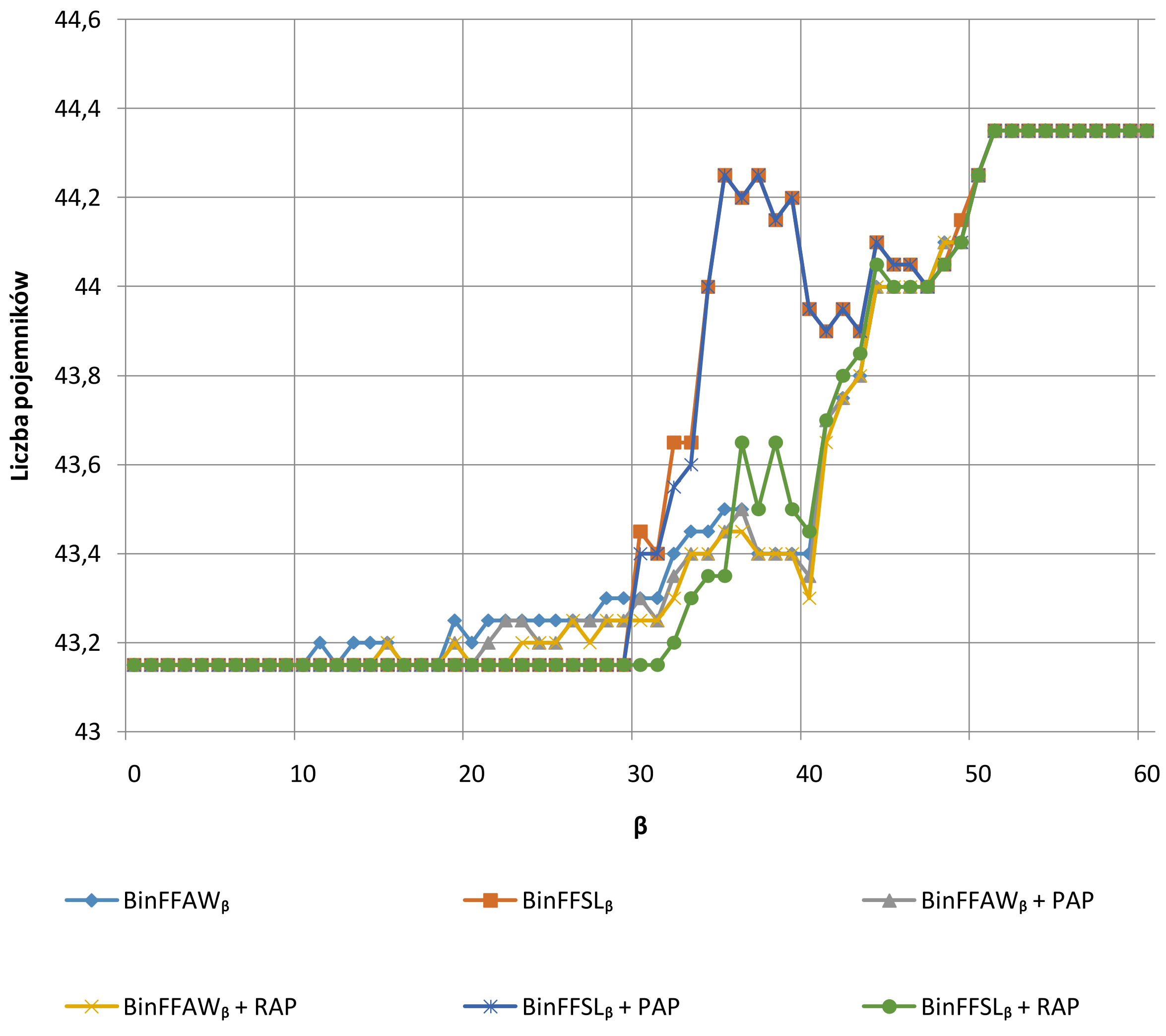
Wykres 4.16 Wyniki działania algorytmów BinFFAWP2$_\beta$ i BinFFAW'3$_\beta$ dla danych HARD

4.7 Eksperymenty z algorytmami poprawy

W niniejszym rozdziale zostaną opisane eksperymenty wykorzystania algorytmów poprawy wraz z algorytmami BinFFSL$_\beta$ oraz BinFFAW$_\beta$. W eksperymentach algorytm RAP pracuje z parametrami $q = 10\%$, $p = 1\%$, które zostały dobrane podczas tworzenia rzeczonego algorytmu. W eksperymentach w ramach algorytmu pomocniczego działa ten sam algorytm, który wygenerował upakowanie startowe.

W przypadku danych N2C2W1 (Wykres 4.17) zarówno PAP jak i RAP poprawiają jakość upakowań zarówno wygenerowanych przez BinFFSL$_\beta$ jak i BinFFAW$_\beta$. Efekty działania PAP są dość małe i zauważalne dla paru wartości parametru β. Działanie RAP dla upakowań wygenerowanych przez BinFFAW$_\beta$ jest równomierne dla $20 < \beta < 40$, jednakże największy sukces algorytmu RAP to poprawa upakowań generowanych przez BinFFSL$_\beta$. Udało się tutaj poprawić liczność upakowań o średnio około 0,5-1 pojemnika. Jest to o tyle imponujący wynik, że różnica pomiędzy rozwiązaniem optymalnym dla problemu pakowania w wariancie ciągłym a dla klasycznego problemu pakowania wynosi 1,2.

Bardzo pozytywną cechą tych algorytmów poprawy jest pewność, że rozwiązanie będzie nie gorsze od rozwiązania startowego. Relacja taka występuje również między algorytmem podstawowym i randomizowanym - jako, że dla pierwszej iteracji algorytmu randomizowanego parametru p i q wynoszą 0, to w praktyce wykonuje się Podstawowy Algorytm Poprawy, i jeżeli daje on poprawę dla danych wejściowych, to poprawione już rozwiązanie jest rozwiązaniem bazowym dla dalszych iteracji.

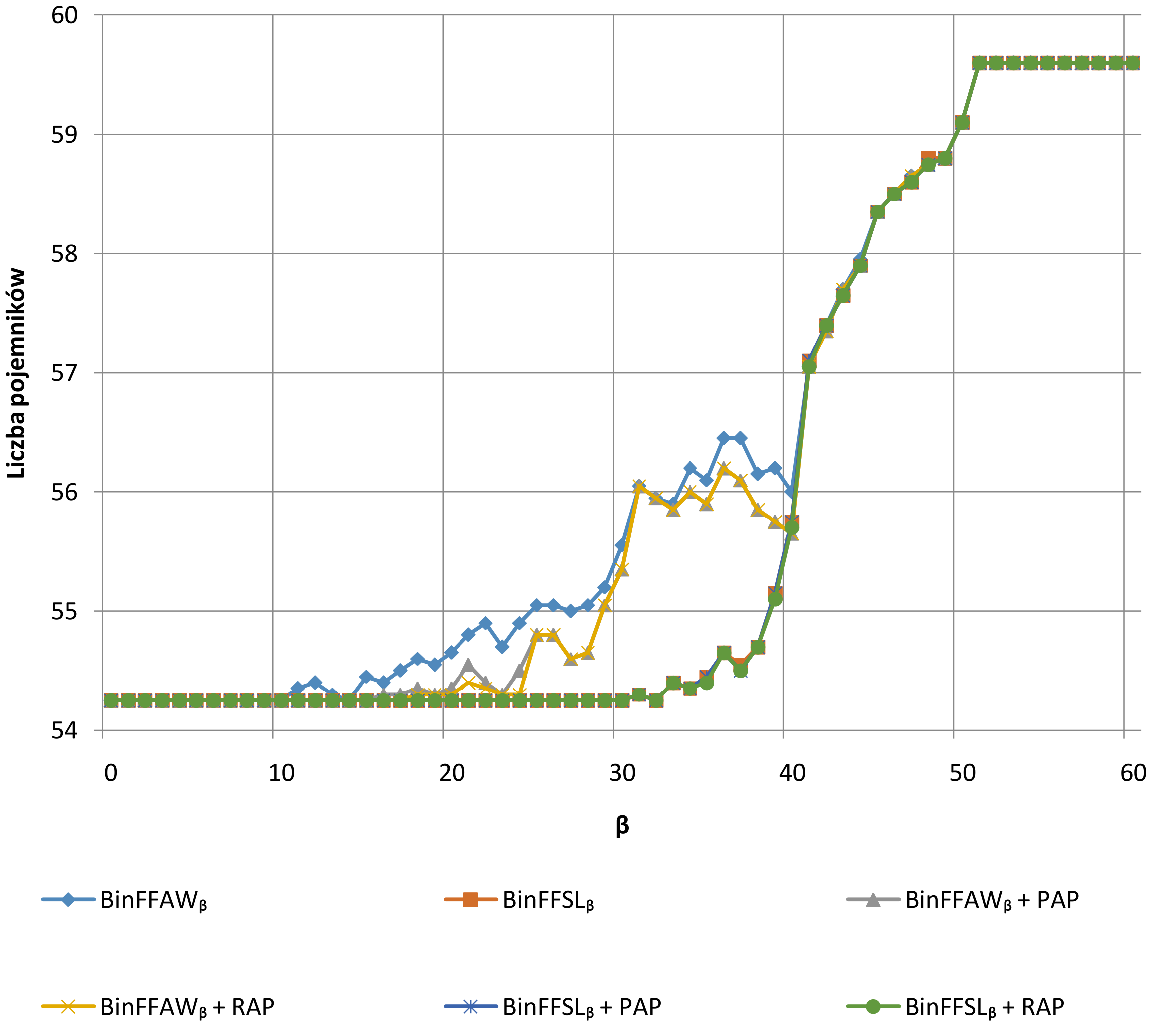


Wykres 4.17 Wyniki działania algorytmów poprawy dla danych N2C2W1

Zgodnie z wykresem 4.18 algorytmy poprawy sprawdziły się bardzo dobrze jeżeli chodzi o $BinFFAW_{\beta}$. Dla wartości $10 < \beta < 41$ w znacznej większości przypadków uruchomienie algorytmów poprawy znajduje lepsze upakowanie. Dla $\beta < 25$ często nowe rozwiązania są optymalnymi. Średnia liczba pojemników zmniejsza się, w zależności od β, od 0.05 pojemnika do około 0.5 pojemnika.

Poprawa w przypadku $BinFFSL_{\beta}$ widoczna jest dla 7 wartości parametru β. W każdym z tych przypadków średnia liczba pojemników poprawiła się o 0.05 pojemnika.

Algorytmy poprawiające są dość czasochłonne, bo mimo, że konstruują w praktyce tylko rozwiązania cząstkowe, to robią to wielokrotnie a wraz ze wzrostem wartości parametru β lista elementów L_p wykorzystywana w obu algorytmach wydłuża się.



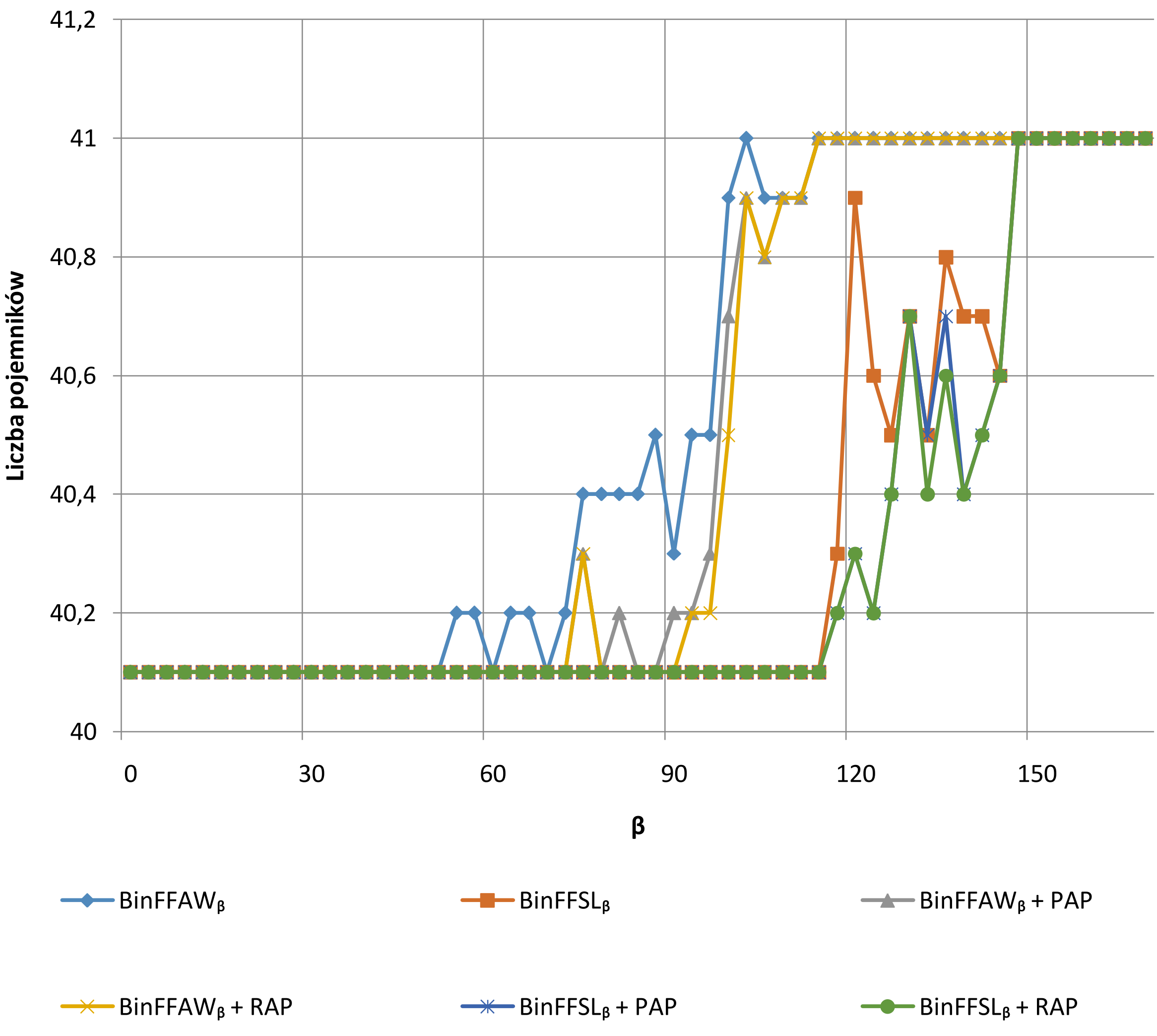
Wykres 4.18 Wyniki działania algorytmów poprawy dla danych N2C2W4

W tabeli 4.1 widać, że narzut czasowy Podstawowego Algorytmu Poprawy jest w większości przypadków akceptowalny (mimo, że dla danych HARD i BinFFSL_β trwa aż 24 sekundy - czyli więcej niż sam podstawowy algorytm). Bardziej czasochłonny jest Randomizowany Algorytm Poprawy - wynika to z wielokrotnego próbowania poprawy, dość kosztownych operacji generowania liczb losowych oraz wielokrotnego przeglądania listy pojemników oraz ich wnętrz w ciągu każdego przebiegu. Wysokie wartości czasu trwania algorytmów są związane z faktem, że dotyczą generacji kilkudziesięciu upakowań, nie pojedynczego.

Średnia poprawa jakości w tabeli 4.1 jest liczona jako średnia wartość różnic pomiędzy licznością upakowań generowanych przez algorytm podstawowy, a licznością upakowań generowanych przez algorytm poprawy na tym upakowaniu. Wartości trzeba brać z pewną dozą ostrożności - uwzględniają one zerową poprawę dla upakowań, które na starcie są optymalne oraz upakowań z wysoką wartością β, które są bardzo bliskie rozwiązań problemu klasycznego. Mimo wszystko widać, zarówno z wykresów jak i z tabeli, że algorytmy dają sensowne rezultaty. Ich kosztowność jest o tyle mniej bolesna, że w przypadku praktycznym rzadko kiedy generowana jest charakterystyka dla tak wielu wartości parametru β.

Tabela 4.1 Ocena wydajności i jakości działania algorytmów poprawy. Czas trwania liczony jest wraz z czasem trwania podstawowego algorytmu.

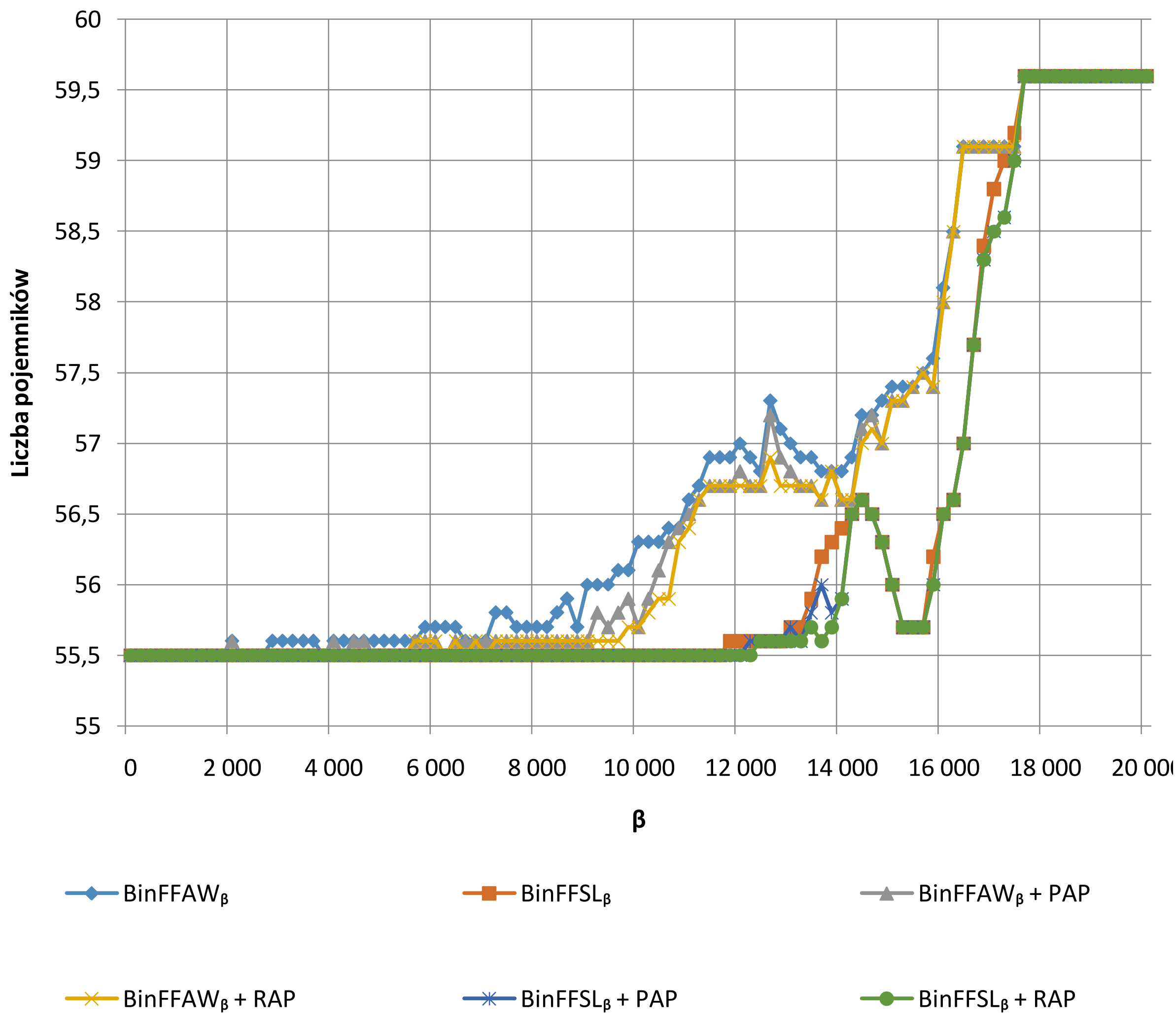
Zbiór danych		$BinFFAW_{\beta}$		$BinFFSL_{\beta}$	
		Podstawowy Algorytm Poprawy	Randomizowany Algorytm Poprawy	Podstawowy Algorytm Poprawy	Randomizowany Algorytm Poprawy
N2C2W1	Czas trwania[s] bez alg. popraw.	10,73		8,583	
	Czas trwania[s]	11,743	25,135	14,268	38,105
	Średnia poprawa jakości	0,013	0,021	0,004	0,107
N2C2W4	Czas trwania[s] bez alg. popraw.	12,191		9,992	
	Czas trwania[s]	16,288	53,068	17,593	54,375
	Średnia poprawa jakości	0,116	0,125	0,002	0,005
N2W2B2	Czas trwania[s] bez alg. popraw.	10,681		7,363	
	Czas trwania[s]	11,556	29,934	14,438	36,994
	Średnia poprawa jakości	0,049	0,058	0,032	0,035
HARD	Czas trwania[s] bez alg. popraw.	25,097		18,77	
	Czas trwania[s]	34,264	95,153	42,262	119,552
	Średnia poprawa jakości	0,093	0,124	0,028	0,036



Wykres 4.19 Wyniki działania algorytmów poprawy dla danych N2W2B2

Dla danych HARD (Wykres 4.20) algorytm poprawy zwiększa jakość rozwiązania praktycznie dla połowy dziedziny parametru β w szczególności osiągając rozwiązania optymalne albo bardzo ich bliskie.

Podsumowując, zaproponowane algorytmy poprawy często poprawiają otrzymane upakowania. Najlepsze efekty uzyskują z upakowaniami znajdowanymi przez algorytm $BinFFAW_{\beta}$, ale również dla $BinFFSL_{\beta}$ czasami udaje im się znaleźć lepsze upakowanie. Wymagają jednak znacznej ilości czasu (zwłaszcza RAP). W rozdziale 1.3 przytoczone zostały dwa przykłady. Pierwszy z nich opisuje problem zakupu kształtowników. Rozwiązywanie go warto wspierać algorytmami poprawy, gdyż czas nie jest tutaj kluczowy. Drugi przykład dotyczył harmonogramowania pakietów w czasie rzeczywistym. W tym przypadku, należy się dokładnie zastanowić nad wykorzystaniem algorytmów poprawy oraz dokonać wyboru będącego kompromisem pomiędzy jakością znajdowanych upakowań a czasem działania algorytmu.



Wykres 4.20 Wyniki działania algorytmów poprawy dla danych HARD

4.8 Poprawianie jakości rozwiązań wykorzystujące charakterystykę

Problem BPP_β charakteryzuje, że każde dopuszczalne upakowanie problemu BPP_β dla konkretnych danych i zadanej wartości parametru β jest poprawne również dla tych samych danych i wyższej wartości parametru β. Dzięki temu, mając wygenerowaną charakterystykę działania algorytmów dla zadanych danych wejściowych i wielu wartościach β możemy ją poprawić poprzez określenie wartości rozwiązania dla danej wartości $\beta = x$ jako najniższej wartości rozwiązania ze zbioru wartości rozwiązań dla $\beta \in (x, \infty)$ - w przypadku wielu charakterystyk przytoczonych w rozdziale 4 można zauważyć, że zdarza się, że algorytmy dla wyższych wartości β osiągają lepsze wartości. W sytuacjach, w których bardzo istotne jest znalezienie jak najlepszego rozwiązania, można spróbować, mimo poszukiwania rozwiązania dla określonej wartości β, wygenerować większą liczbę rozwiązania również dla wyższych wartości β oraz wybierać najlepsze rozwiązanie. W doborze liczby dodatkowych rozwiązań można kierować się na podstawie relacji między rozmiarem elementów, a aktualną wartością β. Algorytmy konstrukcyjne przedstawione w niniejszej pracy w trakcie działania rozbudowują dotychczas wygenerowane rozwiązanie cząstkowe, co za tym idzie, zrównoleglenie obliczeń w ich przypadku jest zadaniem trudnym, może niewykonalnym, jednak w sytuacji rozwiązywania problemów z rożnymi wartościami β, w bardzo prosty sposób można uruchomić parę instancji algorytmu na różnych rdzeniach procesora.

Przykładem, w którym takie podejście było by uzasadnione jest szukanie rozwiązania przy pomocy algorytmu BinFFSL$_\beta$ dla danych HARD dla $\beta \approx 14\,000$. Przy $\beta \approx 15\,500$ następuje gwałtowna poprawa jakości otrzymywanych rozwiązań i odkrycie tego może pozwolić na oszczędność około 1 porcji zasobu.

4.9 Wnioski

Eksperymenty wykonane w ramach pracy pokazują, że działanie poszczególnych algorytmów można analizować oddzielnie dla trzech przedziałów wartości parametru β - przedziale początkowym, środkowym oraz końcowym. Dokładna długość przedziałów jest różna w zależności od danych, dla których zostaje rozwiązywany problem. Początkowy przedział jest najdłuższy, zaczyna się od $\beta = 0$ i dotyczy realizacji, dla których istnieje bardzo dużo elementów podzielnych w tym elementów o wadze znacznie przekraczającej 2β. Przedział środkowy charakteryzuje się mniejszym udziałem elementów podzielnych na liście L oraz małą, ale niezerową liczbą elementów znacznie przekraczających 2β. Przedział końcowy zaczyna się wraz z końcem przedziału środkowego a kończy w miejscu, dla którego nie występują już elementy podzielne na liście L. Jest to najkrótszy przedział. Na przedziale końcowym, wraz z wzrostem wartości β zauważalne jest znaczne pogarszanie się upakowań. Realizacje dla ostatnich wartości parametrów β na tym przedziale znajdują upakowania o liczności identycznej z licznością upakowań znajdowanych przez algorytm BFD. W realizacjach mających wartości parametru β z przedziału końcowego na liście występuje mała liczba elementów podzielnych. Elementy podzielne są charakteryzowane tam podzielnością co najwyżej na trzy części.

Na przedziale początkowym najlepiej radzą sobie algorytmy BinFFSL$_\beta$ oraz BinFFSL'$_\beta$ - działają one szybko i zazwyczaj znajdują upakowania optymalne. Również wyniki BinFFAW$_\beta$ są satysfakcjonujące, w części przypadków udaje mu się znaleźć upakowania optymalne albo im bardzo bliskie. Pod względem czasu działania metoda BinFFAW$_\beta$ jest porównywalna do BinFFSL$_\beta$. Metoda BinBFI$_\beta$, będąca najlepszą wśród metod prostych, tworzy upakowania praktycznie zawsze gorsze od upakowań generowanych przez BinFFAW$_\beta$.

Algorytmy poprawy działające dla wartości β z przedziału początkowego przynoszą dość małe korzyści. Jest to spowodowane generowaniem upakowań optymalnych lub im biskich w tym przedziale wartości β - w przypadku tych drugich, algorytmy poprawy często sprowadzają je do rozwiązań optymalnych.

Wśród algorytmów konstrukcyjnych działających dla wartości β z przedziału środkowego wyniki są bardziej zróżnicowane. Średnio najlepsze wyniki również w tym przypadku daje algorytm BinFFSL$_\beta$, ale dla konkretnych instancji problemu zdarza się, że BinFFAW$_\beta$ lub BinFFSL'1$_\beta$ generują lepsze upakowania. Liczność pojemników w upakowaniach zaczyna zwiększać się i oddalać od liczności upakowań dla problemu pakowania w wariancie ciągłym. Algorytmy proste, pomijając BinBFI$_\beta$, przestają się sprawdzać a BinBFI$_\beta$ rozwiązuje problem wolniej oraz gorzej niż BinFFAW$_\beta$ oraz BinFFSL$_\beta$.

Algorytmy poprawy działające dla β z przedziału środkowego dają istotną poprawę jakości, zwłaszcza, gdy upakowanie startowe jest generowane przez BinFFAW$_\beta$. Również dla upakowań otrzymanych metodą BinFFSL$_\beta$ zdarza się, że algorytmy poprawy poprawiają upakowanie, jednak rzadziej i w mniejszym stopniu niż w połączeniu z upakowaniem z BinFFAW$_\beta$ – jest to kolejnym potwierdzeniem bardzo wysokiej jakości upakowań generowanych przez BinFFSL$_\beta$.

W przypadku rozwiązywania problemów z β leżącą na przedziale końcowym najlepiej spisują się również $BinFFSL_{\beta}$, $BinFFSL'1_{\beta}$ oraz $BinFFAW_{\beta}$. Jedną z najlepszych cech $BinFFAW_{\beta}$ jest generowanie sensownych wyników dla tego typu wartości β - może tutaj konkurować z $BinFFSL_{\beta}$. Jest to też motywacja do dalszej próby rozwijania metody. Są to najtrudniejsze realizacje problemu i algorytmy proste nie wykorzystują ich potencjału generując upakowania zbliżone do rozwiązań generowanych w problemie BPP.

W pracy zidentyfikowana została klasa zbiorów danych, reprezentowana przez zbiory NxCxW1, dla której algorytm $BinFFSL_{\beta}$, na przedziale środkowym, znajduje znacznie słabsze upakowania niż algorytm $BinFFAW_{\beta}$. Zbiory NxCxW1 charakteryzowane są obecnością elementów z przedziału $w_j \in [1{,}100]$ przy rozmiarze $C \in \{100, 120, 150\}$. Na przedziale środkowym, znaczna część elementów tych zbiorów ma rozmiar $w_j < \beta$. Dla takich danych, algorytm $BinFFSL_{\beta}$ radzi sobie bardzo słabo. Problem znajdowania upakowań dla tego typu danych, poza algorytmem $BinFFAW_{\beta}$, dobrze rozwiązuje algorytm $BinFFSL'1_{\beta}$ oraz połączenie $BinFFSL_{\beta}$ i jednego z algorytmów poprawy.

Eksperymenty pokazują, że wykorzystanie algorytmów poprawy dla takich wartości parametru β daje nadzieję na poprawę jakości upakowań – co nawet w połączeniu z dużym czasem działania jest podstawą do korzystania z nich. Złożoność tych algorytmów jest wielomianowa a czasy bezwzględne akceptowalne i w przypadku wykorzystania do wsparcia w planowaniu zakupów w wielkich przedsięwzięciach mogą okazać się przydatne. Najlepsze pod względem jakości uzyskanych upakowań wyniki można uzyskać używając RAP wraz z heurystyką opisaną w podrozdziale 4.7 oraz algorytmami $BinFFAW_{\beta}$ oraz $BinFFSL_{\beta}$.

Podsumowanie

W pracy dokonany został przegląd istniejących metod rozwiązywania problemu pakowania pojemników elementami częściowo podzielnymi. Zostały zaproponowane trzy nowe algorytmy konstrukcyjne oraz dwa algorytmy poprawy.

Opracowane algorytmy konstrukcyjne generują upakowania znacznie lepiej niż algorytmy proste. Cechuje je sensowna, wielomianowa złożoność obliczeniowa. Algorytm $BinFFAW_{\beta}$ dla wielu instancji danych oraz ograniczeń podzielności działa porównywalnie do $BinFFSL_{\beta}$. Zwłaszcza ciekawe jest jego działanie dla sytuacji, w których bardzo mała część elementów jest podzielna oraz ich podzielność jest charakteryzowana małą elastycznością - wtedy jakość generowanych przez $BinFFAW_{\beta}$ upakowań jest bardzo wysoka i nieosiągalna dla algorytmów prostszych. Algorytm $BinFFSL'1_{\beta}$ działa w podobnym czasie do $BinFFSL_{\beta}$ oraz generuje rozwiązania w większości przypadków bardzo podobne ale dla szczególnych przypadków zdarza się mu osiągnąć znacznie lepszy wynik. Algorytmy $BinFFSL'_{\beta}$ oraz $BinFFAW_{\beta}$ mogą stanowić bazę do dalszych modyfikacji - zawierają one wiele parametrów oraz własności, które można zmieniać bez porzucania ogólnej koncepcji działania.

Oba zaproponowane algorytmy poprawy, mimo swojej kosztowności, dają nadzieję na poprawę upakowań nawet bardzo wysokiej jakości. Algorytm PAP jest dość szybki. Algorytm RAP działa wolniej ale jest znacznie bardziej perspektywiczny. Korzystają one z algorytmu pomocniczego, którym może być dowolny algorytm konstrukcyjny rozwiązywania BPP_{β}. Daje to dużą swobodę wyboru oraz możliwość dostosowania do potrzeb jeżeli chodzi o czas działania, jakość generowanych upakowań oraz o dopasowanie się do specyfiki otrzymanego upakowania startowego.

Z uwagi na złożoność problemu należącą do klasy $NP - trudne$, wszystkie omawiane algorytmy opierają się na heurystykach – skutkuje to występowaniem pewnych anomalii. W trakcie eksperymentów pokazano, że istnieje klasa danych dla której algorytm $BinFFSL_{\beta}$ osiąga znacznie gorsze wyniki niż w większości przypadków. Dla klasy tej najlepiej radzi sobie algorytm $BinFFAW_{\beta}$. Algorytm $BinFFSL'1_{\beta}$ i połączenie $BinFFSL_{\beta}$ z algorytmem poprawy dają również lepsze wyniki niż algorytm $BinFFSL_{\beta}$ w przypadku tej klasy danych. Mimo to, na podstawie przeprowadzonych eksperymentów, w większości przypadków najlepszym algorytmem rozwiązywania problemu BPP_{β} pozostaje $BinFFSL_{\beta}$ i niniejsza praca jest tego potwierdzeniem.

Bibliografia

[1] FLESZAR, K.; CHARALAMBOUS, C., Average-weight-controlled bin-oriented heuristics for the one-dimensional bin-packing problem. *European Journal of Operational Research*, 2011, 210.2: 176-184.

[2] MARTELLO, S.; TOTH, P., *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990, Bin-packing problem, s. 221-240.

[3] MENAKERMAN, N.; ROM, R., Bin packing with item fragmentation. W: *Workshop on Algorithms and Data Structures*. Springer Berlin Heidelberg, 2001. s. 313-324.

[4] NAAMAN, N.; ROM, R., Packet scheduling with fragmentation. W: *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. IEEE, 2002. s. 427-436.

[5] PIEŃKOSZ, K.; TRATKIEWICZ, K., Problem optymalizacji rozkroju i spawania kształtowników. Zeszyty Naukowe Wydziału Mechanicznego, Politechnika Koszalińska, 35, 2004, s. 193-200.

[6] PIEŃKOSZ, K., Wybrane modele i metody optymalizacji alokacji zasobów. *Prace Naukowe Politechniki Warszawskiej. Elektronika*, 2010, 174

[7] PIEŃKOSZ, K., Bin packing with restricted item fragmentation. *Control & Cybernetics*, 2014, 43.4, s. 547-559.

[8] SCHOLL, A.; KLEIN, R., Data set 1 for BPP-1, 2003, [online], [dostęp 15 maja 2017]. Dostępny w Internecie: https://www2.wiwi.uni-jena.de/Entscheidung/binpp/bin1dat.htm

[9] SCHOLL, A.; KLEIN, R., Data set 2 for BPP-1, 2003, [online], [dostęp 15 maja 2017]. Dostępny w Internecie: https://www2.wiwi.uni-jena.de/Entscheidung/binpp/bin2dat.htm

[10] SCHOLL, A.; KLEIN, R., Data set 3 for BPP-1, 2003, [online], [dostęp 15 maja 2017]. Dostępny w Internecie: https://www2.wiwi.uni-jena.de/Entscheidung/binpp/bin3dat.htm

[11] SHACHNAI, H.; TAMIR, T.; YEHEZKELY, O., Approximation schemes for packing with item fragmentation. *Theory of Computing Systems*, 2008, 43.1, s. 81-98.

[12] VANDERBECK, F, Computational study of a column generation algorithm for bin packing and cutting stock problems. *Mathematical Programming*, 1999, 86.3, s. 565-594.

Wykaz symboli i skrótów

Wykaz symboli	
Symbol	Wyjaśnienie
C	Rozmiar zasobu
M	Zbiór pojemników
m	Liczność zbioru M
N	Zbiór elementów
n	Liczność zbioru N
w_j	Rozmiar elementu o indeksie j
c_j	Wolne miejsce w pojemniku o indeksie j
x_{ij}	Decyzja alokacyjna pomiędzy elementem j a pojemnikiem i
β	Minimalny rozmiar fragmentu w przypadku dzielenia elementów

Wykaz skrótów	
Skrót	**Rozwinięcie**
BPP	Bin Packing Problem
FF	First Fit
FFD	First Fit Decreasing
BF	Best Fit
BFD	Best Fit Decreasing
BinFF	Bin First Fit
BinBF	Bin Best Fit
BinBFD	Bin Best Fit Decreasing
BinBFI	Bin Best Fit Increasing
B2F	Best Two Fit
MBS	Minimum bin slack
BOH	Bin Oriented Heuristic
BinFFSL	Bin First Fit Small Large

Skrót	**Rozwinięcie**
W1	Warunek 1
W2	Warunek 2
W3	Warunek 3
BinFFAW	Bin First Fit Average Weight
BinFFAWP2	Bin First Fit Average Weight Pack 2
PAP	Podstawowy Algorytm Poprawy
RAP	Randomizowany Algorytm Poprawy

Spis rysunków, wykresów i tabel

Spis tabel

Opis zawartości płyty CD

Wraz z pracą inżynierską załączona zostaje płyta CD zawierająca:

- Plik w formacie pdf zawierający tekst pracy
- Folder „*src*" ze źródłami biblioteki użytej do eksperymentów