Sprawozdanie nr 5
03.06.2022

ALGORYTMY I STRUKTURY DANYCH semestr letni, rok akademicki 2021/2022 piątek 9:45-11:15

1. Andrei Kartavik, Indeks: 153925
2. Ivan Kaliadzich, Indeks: 153936

Wstęp

Celem tego sprawozdania jest zapoznanie się z metodą programowania dynamicznego oraz strategią heurystyczną na przykładzie problemu plecakowego.

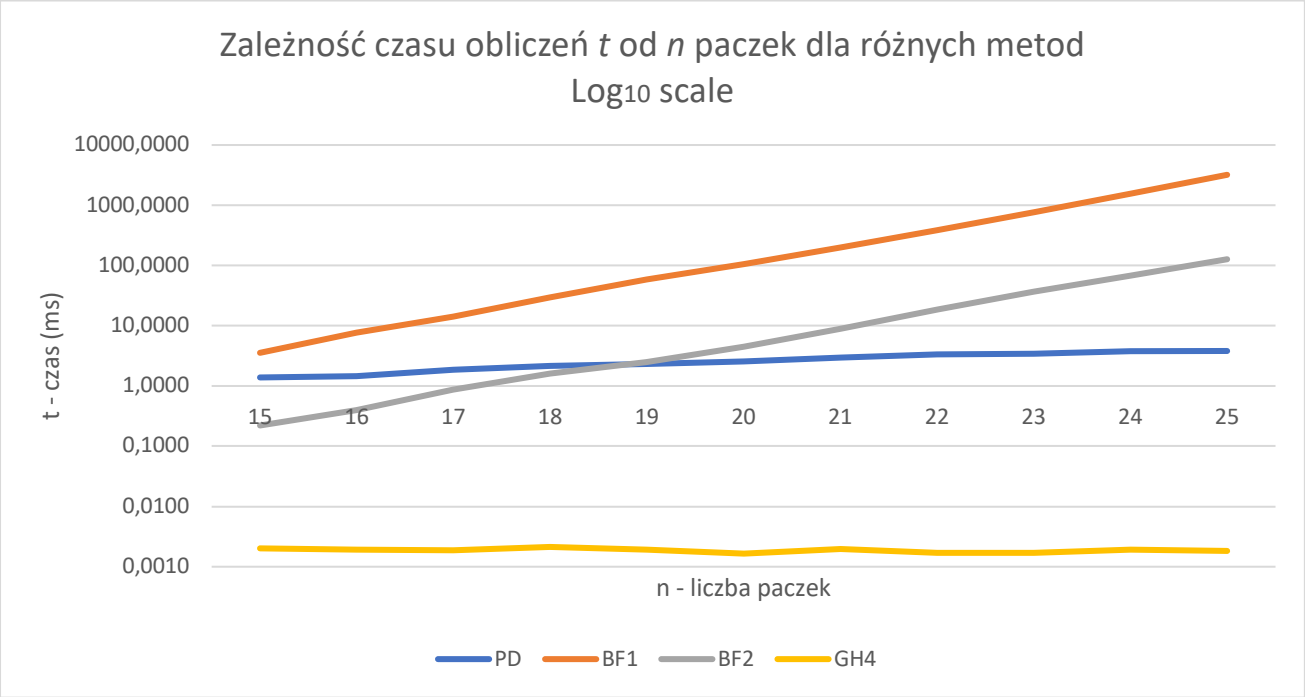
Wszystkie pomiary były wykonywane dla $s(a_i)$ z zakresu [10, 1 000] i $b(a_i)$ z zakresu [100, 10 000].

Przy badaniu zależności czasu obliczeń *t* od liczby paczek *n* dla metody programowania dynamicznego (PD), pełnego przeglądu (BF_1), pełnego przeglądu z eliminacją rozwiązań niedopuszczalnych (BF_2) i heurystycznej z 4 regułami wyboru paczki: losową (GH_1), $\min\{s(a_i)\}$ (GH_2), $\max\{w(a_i)\}$ (GH_3) i $\max\{w(a_i)/s(a_i)\}$ (GH_4); pomiary były wykonywane dla *n* z zakresu [15, 25], z krokiem co 1 (11 punktów pomiarowych) i $b = 50\% \Sigma s(a_i)$.

Przy badaniu zależności czasu obliczeń *t* od liczby paczek *n*, $b = 50\% \Sigma s(a_i)$ i $75\% \Sigma s(a_i)$, dla metody PD pomiary były wykonywane dla *n* z zakresu [100, 1000], z krokiem co 100; dla BF_1 i BF_2 pomiary były wykonywane dla *n* z zakresu [15, 24], z krokiem co 1; dla GF_4 pomiary były wykonywane dla *n* z zakresu [1000, 10 000], z krokiem co 1000.

Przy badaniu średniego błędu popełnianego przez poszczególne heurystyki dla różnych *b* = 25;50;75 %, obliczenia były wykonywane dla *n* z zakresu [100, 1000] z krokiem co 100 (10 punktów pomiarowych), gdzie błąd był obliczany zgodnie z formułą: $(X_{PD}-X_{GH}) / X_{PD} * 100\%$.

Porównanie strategii rozwiązania problemu plecakowego



n	PD	BF_1	BF_2	GH_4
15	1,3854	3,5534	0,2210	0,0020
16	1,4398	7,6192	0,3958	0,0019
17	1,8733	14,3428	0,8637	0,0019
18	2,1673	29,3903	1,6092	0,0021
19	2,2910	58,3219	2,5045	0,0019
20	2,5307	106,3074	4,4412	0,0017
21	2,9469	197,7507	8,9811	0,0020
22	3,3089	383,5154	18,6188	0,0017
23	3,4551	771,8144	37,1996	0,0017
24	3,7774	1557,3164	68,5084	0,0019
25	3,8127	3203,7129	127,0605	0,0018

Tablica 1: Zależność czasu obliczeń t od liczby paczek n dla różnych metod i b = 50% $\Sigma s(a_i)$

Problem plecakowy – jeden z najczęściej poruszanych problemów optymalizacyjnych. Nazwa zagadnienia pochodzi od maksymalizacyjnego problemu wyboru przedmiotów, tak by ich wartość sumaryczna była jak największa i jednocześnie one mieściły się w plecaku. Czyli przy podanym zbiorze elementów o podanej wadze i wartości należy wybrać taki podzbiór, by suma wartości była możliwie jak największa, a suma wag była nie większa od danej pojemności plecaka.
Decyzyjna wersja przedstawionego zagadnienia brzmi następująco: „Czy wartość co najmniej X może być osiągnięta bez przekraczania wagi Y?". Należy ta wersja problemu plecakowego do klasy NP-zupełnych, co jest udowodnione, więc dotychczas nie istnieje algorytmu, który rozwiązałby ten problem w czasie wielomianowym. A z powodu tego, że problem optymalizacyjny jest nie łatwiejszy niż problem decyzyjny, wersja optymalizacyjna problemu plecakowego należy do klasy NP-trudnych. Klasa problemów trudnych nie jest jednorodna, ponieważ pseudowielomianowy algorytm problemu plecakowego klasyfikujemy jako problem NP-zupełny, a problem nierozwiązywalny w czasie pseudowielomianowym nazywamy silnie NP-zupełny.

Pierwsza strategia rozwiązania tego problemu jest strategią pełnego przeglądu, czyli brute-force. Polega ona na tym, że musimy sprawdzić wszystkie możliwe podzbiory przedmiotów i wyznaczyć ten, w którym suma wartości przedmiotów jest jak najbardziej maksymalna, a suma wag nie przekręci maksymalną pojemność plecaka. W celu wygenerowania wszystkich podzbiorów wybraliśmy metodę generowania wszystkich *n*-bitowych binarnych liczb (jak np. 0000, 0001, …. , 1111), w których 1 odpowiadała temu, czy musimy dodać a_i przedmiot do naszego rozwiązania, a 0 oznaczało to, że nie musimy dodawać tego przedmiotu do rozwiązania. Złożoność obliczeniowa tej operacji wynosi $O(2^n)$, ale jeszcze musimy przejść przez każdy podzbiór naszych przedmiotów w celu znalezienia potrzebnego nam zbioru elementów, co daje nam złożoność $O(n)$ i w wyniku, złożoność takiego podejścia wynosi $O(n*2^n)$. To jest bardzo wolna strategia ze względu na złożoność czasową, ale jesteśmy pewni, że rozwiązanie problemu plecakowego tą metodą jest naprawdę optymalne.

Druga strategia rozwiązania tego problemu jest strategią pełnego przeglądu z eliminacją rozwiązań niedopuszczalnych, czyli metoda oparta na algorytm z powracaniem. Działa ona na podobnej zasadzie do metody brute-force… To znaczy, że nadal musimy sprawdzić każdy możliwy podzbiór zbioru naszych elementów, ale różnica jest w tym, że przy tworzeniu podzbioru sprawdzamy czy suma wag tych elementów, które już zostały dodane do rozwiązania, nie

przekracza pojemności plecaka. I jeżeli przekracza, to skończymy generować ten podzbiór, bo nie ma sensu w tym, żeby dodawać elementy do plecaka, który już jest wypełniony. Ta metoda ma taką samą złożoność jak i metoda brute-force, czyli O($n*2^n$), ale w rzeczywistości, jak możemy zobaczyć na wykresie i w tablice 1, będzie ona szybsza od metody brute-force. Jest to nadal dosyć wolna strategia rozwiązania problemu plecakowego, ale zarówno jak i w przypadku metody brute-force, jesteśmy pewni, że w końcu otrzymamy optymalne rozwiązanie.

Trzecia strategia jest strategią programowania dynamicznego. Programowanie dynamiczne to technika lub strategia projektowania algorytmów, stosowana przeważnie do rozwiązywania zagadnień optymalizacyjnych. Ta metoda opiera się na podziale rozwiązywanego problemu na podproblemy zależne od siebie, którą cechuje własność optymalnej podstruktury. W przypadku problemu plecakowego to oznacza, że wybór k elementów jest optymalny, jeśli wybór k-1 był optymalny. Więc, żeby znaleźć optymalne rozwiązanie dla *n* elementów o rozmiarze plecaka *b*, musimy znaleźć optymalne rozwiązanie dla *n-1* elementów i rozmiarze plecaka b. Wiedząc to, możemy stworzyć funkcję, która pomoże nam znaleźć rozwiązanie tego problemu. Wygląda ona następująco:

Funkcja f(*i, l*), gdzie i – to numer elementu a_i ze zbioru wszystkich przedmiotów, czyli z zakresu [0, …, n], a l – to dostępne miejsce w plecaku, czyli z zakresu [0, …, b], wynikiem której jest maksymalna wartość plecaka, którą uda się uzyskać dla *n* przedmiotów nie przekraczając rozmiaru plecaka *b* w wyniku wykonania algorytmu. W przypadku gdy mamy jakiś i, ale l = 0, co oznacza, że mamy element, ale nie mamy miejsca w plecaku, optymalne rozwiązanie f(i, 0) wynosi 0, bo nie mając miejsca w plecaku, nie jesteśmy w stanie nic do niego włożyć, więc wartość jest równa 0. W analogicznym przypadku, gdy mamy jakiś l, ale i = 0, czyli mamy dostępne miejsce, ale nie mamy elementu do włożenia, to wartość f(0, l) znów wynosi 0, bo nie mając przedmiotów wartość plecaka jest równa 0. Ale, gdy mamy jakiś i, l nie równe 0, musimy sprawdzić, czy opłaca się włożenie elementu a_i do plecaka, czy nie. Co prowadzi nas do tego, że jeżeli waga elementu s(a_i) przekracza dostępne miejsce w plecaku, to nie możemy go włożyć, więc zawartość plecaka musi zostać taka sama, czyli: jeżeli s(a_i) > l, to f(i, l) = f(i-1, l). A gdy waga elementu nie przekracza dostępnego miejscu w plecaku, to musimy sprawdzić, czy jego dołożenie prowadzi nas do większej wartości, czyli: jeżeli s(a_i) >= l, f(i, l) = max { f(i - 1, l), f(i - 1, l - s(a_i)) + w(a_i)}, gdzie w(a_i) – wartość elementu a_i. I w wyniku wykonania algorytmu, otrzymamy największą wartość, którą możemy uzyskać nie przekraczając rozmiaru plecaka b. Ale jeszcze na pewno

chcemy zrozumieć, jakie przedmioty zostały włożone do plecaka. I w tym przypadku, wystarczy nam sprawdzić, czy dokładaliśmy pewny przedmiot do plecaka podczas wykonania algorytmu. Musimy zacząć od wartości funkcji f(n, b) i porównać ją z wartością f(n-1, b), jeżeli są one równe, to znaczy że nie dołożyliśmy przedmiotu n do plecaka, a zatem musimy sprawdzić to samo dla elementu f(n-1, b). Ale gdybyś f(n, b) != f(n-1, b), to oznaczałoby, że dołożyliśmy przedmiot n do plecaka i teraz musimy sprawdzić to samo dla elementu f(n-1, b – s(a_n)).
Złożoność obliczenia wartości f(n, b) wynosi O(n*b), bo dla każdego elementu z zakresu [0, …, n] musimy otrzymać wartość dla dostępnego miejsca w plecaku z zakresu [0, …, b]. Po obliczeniu wartości w celu otrzymania zbioru włożonych przedmiotów, musielibyśmy sprawdzić *n* przedmiotów – O(n), więc w wyniku ten algorytm ma złożoność czasową O(*n*b*). Mogłoby się wydawać, że ten algorytm działa w czasie wielomianowym, ale to nie jest tak. Bo zmieniając parametr *b*, rozmiar naszego problemu nie zmienia się w żaden sposób, bo zmieniamy tylko rozmiar plecaka, a nie liczbę dostępnych elementów, więc ten algorytm jest pseudowielomianowy, przecież możemy udowodnić, że b zmienia się wykładniczo.
Jak możemy zobaczyć na wykresie i w tablice, jest on dość szybki w porównaniu z powyższymi metodami, i znów jesteśmy pewni, że otrzymamy optymalne rozwiązanie! Ale wadą może być to, że implementacja takiego algorytmu jest trochę cięższa, niż w powyższych przypadkach.

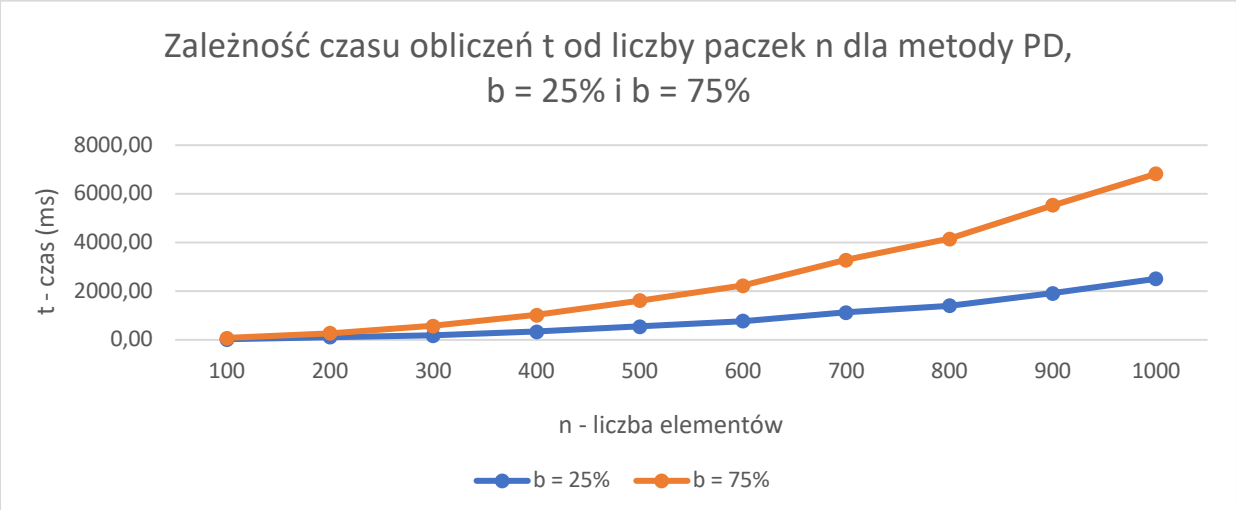
I ostatnia strategia rozwiązywania tego problemu jest strategią algorytmu heurystycznego, opierający się na regule wyboru max{w(a_i)/s(a_i)}. Algorytm heurystyczny – to algorytm niedający gwarancji znalezienia rozwiązania optymalnego, umożliwiający jednak znalezienie rozwiązania dość dobrego w rozsądnym czasie. W naszym przypadku działa on następująco:
Najpierw dla każdego przedmiotu musimy obliczyć parametr *ws*, który jest równy stosunku wartości przedmiotu do jego wagi, czyli *ws* = w(a_i)/s(a_i). Zatem musimy posortować nasze elementy według tego parametru *ws* i zaczynając od elementu z największą wartością *ws* (w przypadku, gdy sortujemy elementy w tablice, to jest ostatni element)*,* wkładamy go do plecaka i przechodzimy do następnego elementu z największą wartością *ws,* i ponownie wkładamy go do plecaka. Kontynuujemy ten proces aż do momentu, gdy wkładanie kolejnego elementu nie spowoduje przekroczenie rozmiaru plecaka. Złożoność tej metody zależy od wybranej metody sortowania, w naszym przypadku to metoda sortowania szybkiego, więc ten algorytm średnio ma złożoność O(*n*log$_2$n*). Jest to najszybsza strategia ze względu na złożoność czasową w porównaniu ze wszystkimi przedstawionymi metodami, ale jej główną wadą

jest to, że ta strategia nie gwarantuje optymalnego rozwiązania, a zazwyczaj tylko bardzo przybliżonego do niego.

Badanie wpływu *n* i *b* na efektywność czasową poszczególnych metod

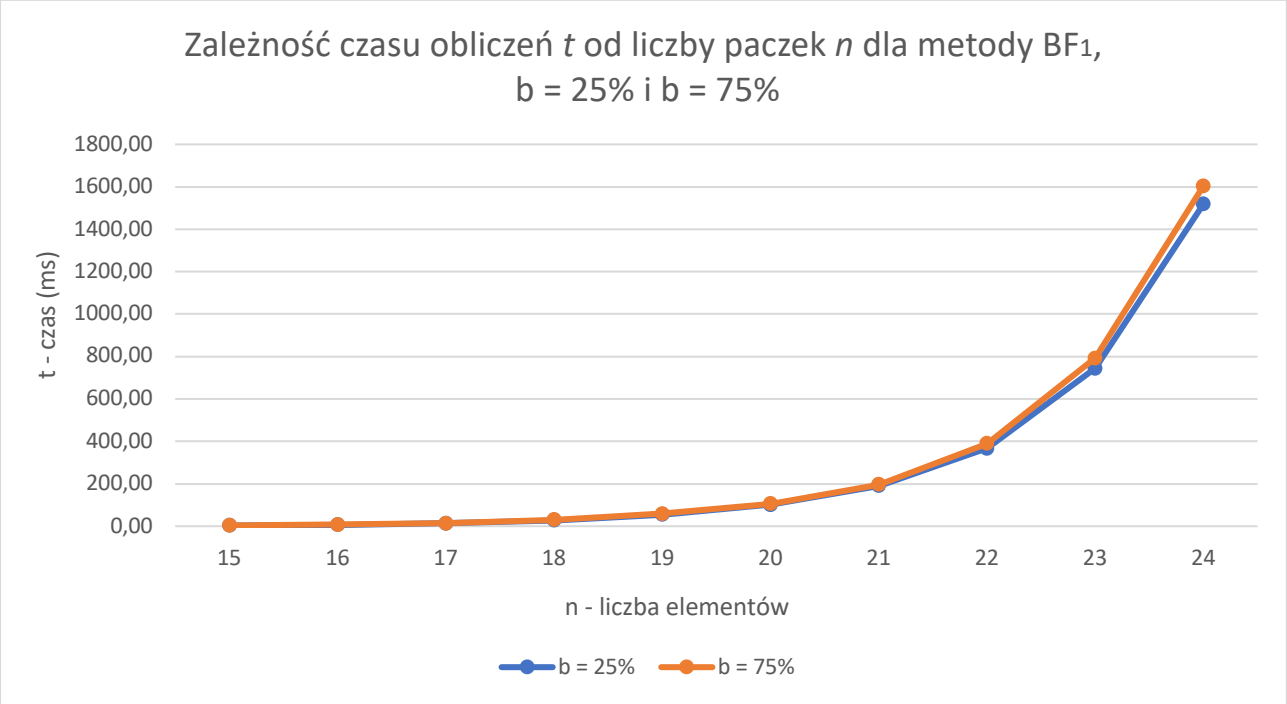
PD		
n	**b**	**t = f(n)**
100	25%	27,04
	75%	81,21
200	25%	109,21
	75%	276,95
300	25%	184,54
	75%	576,02
400	25%	333,20
	75%	1017,76
500	25%	545,34
	75%	1611,13
600	25%	778,67
	75%	2227,37
700	25%	1123,10
	75%	3284,01
800	25%	1404,74
	75%	4151,40
900	25%	1913,56
	75%	5539,48
1000	25%	2508,49
	75%	6820,49

Tablica 2: Zależność czasu obliczeń t od liczby paczek n dla metody PD i b = 25;75% $\Sigma s(a_i)$



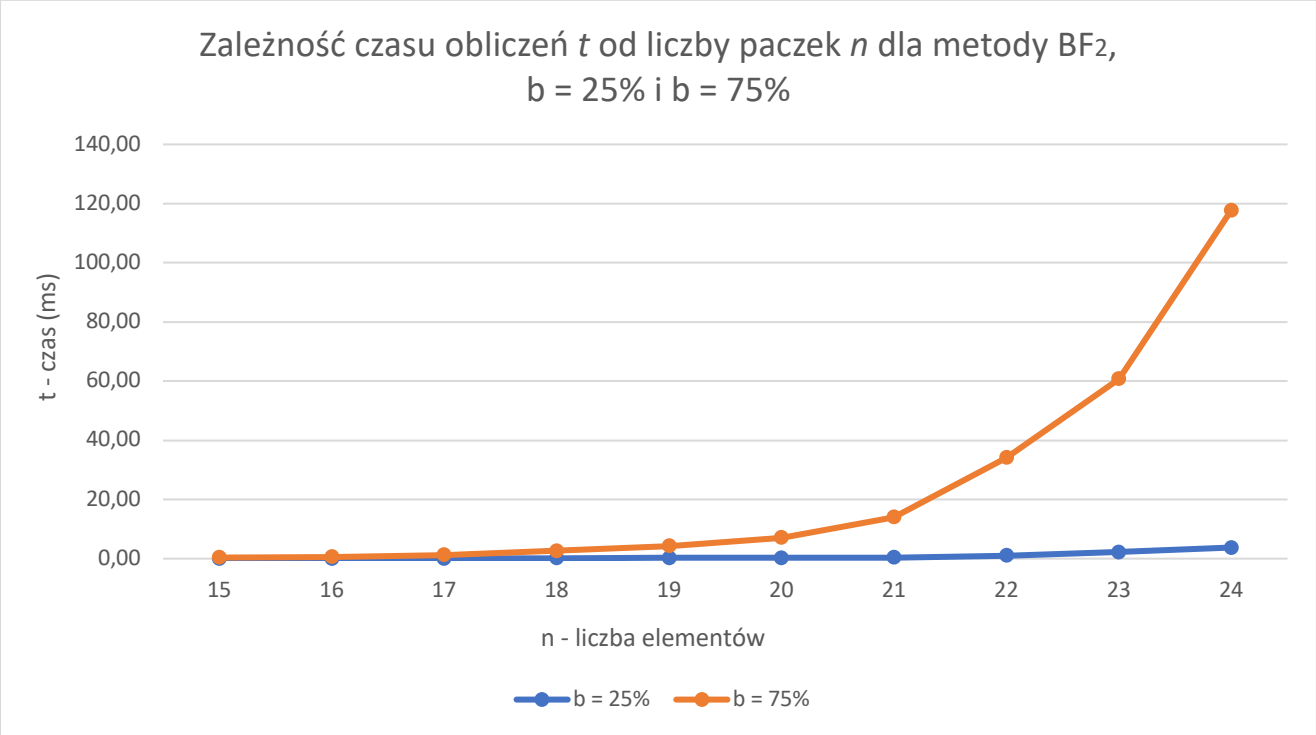
BF1		
n	**b**	**t = f(n)**
15	25%	3,44
	75%	3,64
16	25%	7,02
	75%	7,38
17	25%	13,84
	75%	14,45
18	25%	27,31
	75%	30,02
19	25%	54,60
	75%	59,31
20	25%	100,89
	75%	106,02
21	25%	190,13
	75%	196,33
22	25%	367,16
	75%	389,25
23	25%	743,90
	75%	791,82
24	25%	1518,31
	75%	1603,69

Tablica 3: Zależność czasu obliczeń t od liczby paczek n dla metody BF1 i b = 25;75% $\Sigma s(a_i)$



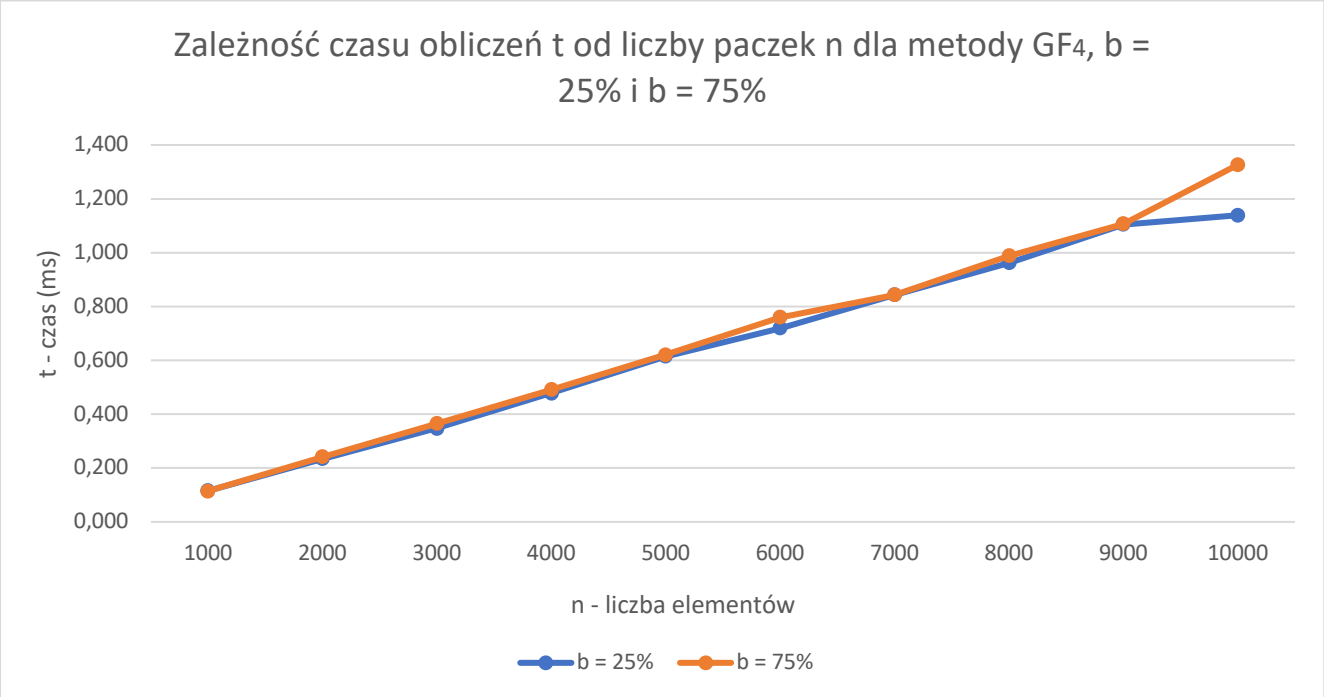
BF2		
n	**b**	**t = f(n)**
15	25%	0,03
	75%	0,36
16	25%	0,07
	75%	0,63
17	25%	0,09
	75%	1,26
18	25%	0,18
	75%	2,68
19	25%	0,25
	75%	4,24
20	25%	0,26
	75%	7,08
21	25%	0,38
	75%	13,98
22	25%	1,00
	75%	34,08
23	25%	2,16
	75%	60,78
24	25%	3,75
	75%	117,74

Tablica 4: Zależność czasu obliczeń t od liczby paczek n dla metody BF2 i b = 25;75% $\Sigma s(a_i)$



GF4		
n	**b**	**t = f(n)**
1000	25%	0,115
	75%	0,115
2000	25%	0,234
	75%	0,242
3000	25%	0,348
	75%	0,366
4000	25%	0,478
	75%	0,491
5000	25%	0,615
	75%	0,621
6000	25%	0,719
	75%	0,760
7000	25%	0,843
	75%	0,843
8000	25%	0,963
	75%	0,988
9000	25%	1,105
	75%	1,108
10000	25%	1,139
	75%	1,326

Tablica 5: Zależność czasu obliczeń t od liczby paczek n dla metody GF4 i b = 25;75% $\Sigma s(a_i)$



Czas obliczeń we wszystkich strategiach rośnie wraz z ilością paczek. Zarówno możemy zobaczyć, że rozmiar plecaka nie ma znacznego wpływu na metodę BF_1 i wynika to z tego, że ta strategia polega na generowaniu wszystkich możliwych podzbiorów paczek - $O(2^n)$, i dalszym sprawdzaniu każdego z tych podzbiorów na spełnienie warunków naszego problemu - O(n), co razem wynosi $O(n*2^n)$, więc rozmiar plecaka nie ma wpływu na złożoność czasową tej metody.

Dla metody heurystycznej GF_4 rozmiar plecaka *b* też nie ma zauważalnego wpływu na czas obliczeń, ponieważ musimy te elementy najpierw posortować według jakiegoś parametru zgodnie z wybraną metodą sortowania (w naszym przypadku quicksort) – $O(n*log_2 n)$, a zatem musimy przejść przez ten zbiór dodając elementy do plecaka, aż do momentu, gdy dołożenie następnego elementu nie przekroczy rozmiar plecaka – O(n). Razem to daje nam złożoność $O(n*log_2 n)$, więc rozmiar plecaka też nie ma znacznego wpływu na złożoność czasową tej strategii.

W strategii BF_2 rozmiar plecaka ma wpływ na czas działania algorytmu, ponieważ im mniejszy jest rozmiar plecaka, tym wcześniej suma wag wybranych przedmiotów przekracza jego pojemność i tym wcześniej cała gałąź rozwiązań zostaje odrzucona.

Dla metody programowania dynamicznego rozmiar plecaka ma największy wpływ, ponieważ podczas działania tego algorytmu musimy dla każdego przedmiotu z zakresu [0, …, n] obliczyć wartość plecaka dla dostępnego miejsca z zakresu [0, …, b], więc ten algorytm ma złożoność $O(n*b)$, stąd i wynikają te różnice czasowe w tablice 2 i na wykresie.

Badanie średniego błędu popełnianego przez poszczególne heurystyki dla różnych b

b = 25%				
n	**GH_1**	**GH_2**	**GH_3**	**GH_4**
100	47,11%	17,51%	22,46%	0,81%
200	52,03%	18,36%	20,36%	0,19%
300	57,20%	13,34%	22,95%	0,01%
400	53,58%	13,75%	22,51%	0,14%
500	60,66%	13,27%	20,91%	0,12%
600	53,92%	12,30%	22,65%	0,13%
700	56,32%	12,36%	20,66%	0,10%
800	56,10%	12,17%	23,44%	0,11%
900	53,93%	13,25%	23,51%	0,12%
1000	57,13%	14,74%	23,00%	0,04%
Średnia	54,80%	14,11%	22,25%	0,18%

Tablica 6: Średni błąd popełniany przez poszczególne heurystyki dla b = 25%

b = 50%				
n	**GH_1**	**GH_2**	**GH_3**	**GH_4**
100	30,43%	9,35%	4,69%	0,45%
200	41,92%	12,56%	6,27%	0,13%
300	38,32%	11,94%	6,96%	0,14%
400	37,85%	13,23%	6,20%	0,03%
500	35,69%	13,15%	5,57%	0,04%
600	35,66%	12,33%	6,56%	0,09%
700	38,49%	12,84%	6,71%	0,05%
800	41,05%	12,31%	7,44%	0,04%
900	39,92%	12,67%	6,33%	0,11%
1000	37,36%	15,19%	6,56%	0,08%
Średnia	37,67%	12,56%	6,33%	0,12%

Tablica 7: Średni błąd popełniany przez poszczególne heurystyki dla b = 50%

b = 75%				
n	**GH_1**	**GH_2**	**GH_3**	**GH_4**
100	26,87%	8,77%	0,79%	0,00%
200	18,91%	7,84%	2,18%	0,06%
300	20,36%	12,33%	1,22%	0,06%
400	18,19%	8,09%	1,90%	0,13%
500	23,31%	8,13%	1,54%	0,03%
600	20,88%	8,64%	1,76%	0,02%
700	19,30%	8,60%	1,86%	0,04%
800	18,86%	9,06%	1,12%	0,04%
900	20,83%	9,14%	1,56%	0,07%
1000	21,95%	8,98%	1,65%	0,05%
Średnia	20,95%	8,96%	1,56%	0,05%

Tablica 8: Średni błąd popełniany przez poszczególne heurystyki dla b = 75%

śr. błąd	**GH_1**	**GH_2**	**GH_3**	**GH_4**
b = 25%	54,80%	14,11%	22,25%	0,18%
b = 50 %	37,67%	12,56%	6,33%	0,12%
b = 75%	20,95%	8,96%	1,56%	0,05%
średnia	37,81%	11,88%	10,05%	0,12%

Tablica 9: Średni błąd popełniany przez poszczególne heurystyki dla różnych b

Już ustaliliśmy, że algorytm heurystyczny – to algorytm niedający gwarancji znalezienia rozwiązania optymalnego, umożliwiający jednak znalezienie rozwiązania dość dobrego w rozsądnym czasie. Algorytm zachłanny to taki, który w celu wyznaczenia rozwiązania w każdym kroku dokonuje zachłannego (najlepiej rokującego w danym momencie) wyboru rozwiązania częściowego. Innymi słowy, algorytm zachłanny dokonuje wyboru wydającego się w danej chwili najlepszym, kontynuując rozwiązanie podproblemu wynikającego z podjętej decyzji. Algorytm listowy najpierw porządkuje dane według pewnego parametru, a następnie rozpatruje je według ustalonej kolejności. Metody zachłanne i listowe można wykorzystać tylko wtedy, kiedy potrafimy wyznaczyć kryterium oceny jakości poszczególnych rozwiązań. Wszystkie strategie GH są algorytmami heurystycznymi, a wszystkie oprócz GH_1 są jeszcze listowymi i zachłannymi.

Jak możemy zauważyć w powyższych tablicach, dla wszystkich metod GH z wzrostem ładowności *b* maleje niedokładność rozwiązania. Największe zmiany zachodzą dla GH_1 i GH_3, ponieważ pierwszy przedmiot o stosunkowo dużym rozmiarze, lecz małej wartości może zająć dużą część ładowności samochodu. Strategie GH_2 i GH_4 w swoim kryterium zawierają rozmiar przedmiotów, więc są one mniej podatne na nieoptymalne rozwiązania. Oraz możemy zobaczyć, że z wzrostem liczby paczek *n* niedokładność rośnie dla metod GH_1 i GH_3, a dla metody GH_4 na odwrót – maleje. Na metodę GH_2 liczba paczek *n* nie ma dużego wpływu.

GH_4 średnio jest najbardziej dokładną strategią, ponieważ uwzględnia ona i wartość, i wagę przedmiotu. GH_3 jest trochę gorsza niż GH_4, ponieważ uwzględnia ona tylko wartość przedmiotu, ale nadal jest dobrą strategią, bo do plecaka dokłada się przedmiotów o największych wartościach, więc szansa na to, że będziemy mieli rozwiązanie bliskie do optymalnego jest dość optymistyczna. Metoda GH_2 uwzględnia tylko rozmiar przedmiotu i dlatego niedokładność tej metody jest dość duża. A skuteczność metody GH_1 jest w dużej mierze przypadkowa, ponieważ kolejne elementy są wybierane losowo, więc niedokładność tej metody jest największa w porównaniu z innymi metodami.

Algorytmy zachłanne nie muszą być heurystykami (jak na przykład algorytm Dijkstry służący do znajdowania najkrótszej ścieżki w grafie).

Wnioski

Metoda rozwiązania brute-force problemu plecakowego jest dość wolna, ale niezawodna. Możemy być pewni, że ona doprowadzi nas do optymalnego rozwiązania. Dlatego warto ją stosować, gdy liczba przedmiotów jest mała i dążymy do rozwiązania optymalnego, oraz gdy chcemy prostego ze względu na implementację algorytmu.

Metoda rozwiązania oparta na algorytm z powracaniem jest lepsza niż metoda brute-force, ponieważ pomaga ona eliminować rozwiązania niedopuszczalne. Warto ją stosować, gdy dążymy do optymalnego rozwiązania i liczba przedmiotów jest nadal nie duża.

Metoda programowania dynamicznego jest bardzo dobrą strategią rozwiązania tego problemu, bo ona również jak i powyższe metody prowadzi nas do rozwiązania optymalnego, oraz działa szybciej niż one. Ale wadą może być to, że jest ona wrażliwa na rozmiar plecaka, bo im większy jest jego rozmiar, tym więcej czasu zajmie działanie tego algorytmu. Oraz implementacja takiego algorytmu jest trochę trudniejsza w porównaniu z innymi algorytmami.

Strategia heurystyczna jest najszybszą metodą do rozwiązania problemu plecakowego, ale jej główną wadą jest to, że ona nie gwarantuje nam rozwiązania optymalnego. Dlatego warto ją stosować, gdy potrzebujemy szybkiego rozwiązania i jesteśmy gotowi stracić na dokładności otrzymanego wyniku.