Projekt 15: Sklep internetowy

Jan Cichowlas i Jan Pogłód

Listopad 2023

1 Wstęp

Celem projektu było przedstawienie optymalnego algorytmu dla problemu, w którym sklep internetowy w celu usprawnienia zamówień wymaga obsługi maksymalnej liczby klientów przy pewnych ograniczeniach. Sklep internetowy ustala następujące dane wejściowe:

- Liczba pracowników wynosi m, gdzie $1 \leq m \leq 10$.
- Każdy pracownik pracuje przez T czasu dla $1 \leq T \leq 8$.
- Sklep odwiedza n klientów, którzy składają zamówienia. Klienci nie są priorytetowani przez sklep oraz mogą złożyć co najwyżej jedno zamówienie. $1 \leq n \leq 100$.
- Każdy klient składa zamówienie przez t_i czasu i wiemy, że $1 \leq t_i \leq 5$.

Jako wyjście należało w najoptymalniejszy sposób przedstawić grafik obsłużonych klientów przez każdego z m pracowników, przy założeniu, że żaden z nich nie pracuje ponad T czasu pracy. Stworzony w ten sposób algorytm jako wyjście miał prezentować m ciągów postaci $(z_1,...,z_j)$ dla każdego z pracowników, przy czym z_i jest numerem zamówienia dla $i=1,...,j$ oraz $\sum_{i=1}^{j} t_i \leq T$.

2 Rozpoznanie problemu

Podstawowym warunkiem zapewnienia sklepowi obsługi największej liczby klientów jest odpowiedni przydział zamówień do każdego z pracowników. Rozwiązując ten problem należy obsłużyć jak najwięcej spośród $z_1,...,z_n$ zamówień przydzielając je w optymalny sposób do każdego z pracowników. Przy założeniu, że w sklepie pracowałby tylko jeden pracownik można zauważyć, że jest to szczególny przypadek *"problemu plecakowego"*, który również opiera się na odpowiedniej selekcji przedmiotów (zamówień) do plecaka (pracownika).

2.1 Definicja 1 - Problem Plecakowy

Dany jest zbiór n elementów wśród których każdy ma wartość p_i oraz wagę w_i i plecak o pojemności C. Przy zmiennej decyzyjnej x_i należy znaleźć podzbiór elementów, dla którego:

$$\max \sum_{i=1}^{n} p_i x_i, \tag{1}$$

przy ograniczeniu:

$$\sum_{i=1}^{n} w_i p_i \leq C. \tag{2}$$

Zmienna decyzyjna x_i określa, czy *i-ty* przedmiot powinien znaleźć się w plecaku. Dyskretny problem plecakowy nazwiemy:

1. **Binarnym** jeśli każdy z elementów występuje w plecaku dokładnie raz. Wtedy x_i jest zmienną dyskretną postaci $x_i \in \{0, 1\}$.

2. **Ograniczonym.** Wtedy x_i jest zmienną całkowitą, ograniczoną $x_i \in [0, b_i]$, gdzie b_i jest predefiniowaną liczbą elementów pewnego typu.

3. **Nieograniczonym.** Wtedy x_i jest zmienną całkowitą, nieograniczaną $x_i \geq 0$, wtedy gdy nie ma ograniczenia na liczbę elementów pewnego typu.

4. **Wielowymiarowym**, jeśli plecak lub przedmioty mają co najmniej dwa wymiary. Wówczas $x_i \in \{0, 1\}$ oraz c_i jest wartością wagi w *m-tym* plecaku *i=1,..,m.*

2.2 Przydział zamówień dla pracowników

Zauważmy, że zadanie przydziału odpowiednich zamówień dla pracowników możemy powiązać z wielowymiarowym problemem plecakowym. Wówczas zmienne decyzyjne określające, czy klient *i-ty* powinien zostać obsłużony przez *m-tego* pracownika są binarne, ponieważ każdy z klientów może być przydzielony do któregoś pracownika dokładnie raz. Ponadto $c_1 = c_2 = ... = c_m = T$, ponieważ każdy z pracowników pracuje dokładnie taką samą liczbę godzin wynoszącą T. W przypadku postawionego problemu wagi są czasami realizacji zamówienia i wynoszą kolejno $t_1, ..., t_n$ oraz żaden z klientów nie jest priorytetowany, a więc możemy pominąć wartość p_i we wzorze (1), gdyż dla każdego klienta są sobie równe (możemy założyć, że $p_1 = p_2 = ... = p_n = 1$).

2.3 Postawienie zadania

Na podstawie powyższych wniosków możemy zbudować problem optymalnego przydziału zamówień dla pracowników.
Dany jest zbiór n elementowy wśród których każdy ma wagę t_i i m pracowników o maksymalnym czasie obsługi T. Przy binarnej zmiennej decyzyjnej x_i należy znaleźć podzbiór elementów dla każdego pracownika, dla którego:

$$\max \sum_{i=1}^{n} x_i, \tag{3}$$

przy ograniczeniu:

$$\sum_{i=1}^{n} t_i \leq T. \tag{4}$$

3 Implementacja algorytmu

3.1 Definicja 2 - Programowanie dynamiczne

Programowanie dynamiczne to technika projektowania algorytmów rozwiązujących m.in. problemy optymalizacji. W podejściu dynamicznym, problem dzielony jest na mniejsze podproblemy, a następnie optymalne rozwiązanie problemu wyznaczane jest przez znalezienie optymalnych rozwiązań tych podproblemów.

3.2 Tworzenie algorytmu z wykorzystaniem programowania dynamicznego

Przykładem wykorzystania programowania dynamicznego jest stworzenie dwuwymiarowej tablicy dynamicznej, zwanej później *dp*, która w przypadku danego problemu wybierać będzie najbardziej optymalne czasy obsługi dla *i-tego* pracownika. Dla każdego z pracowników tworzona jest nowa tablica *dp* zawierająca dwa wymiary reprezentujące liczbę klientów n oraz maksymalny czas pracy pracownika T. Tak jak i dla klasy algorytmów programowania dynamicznego na początku rozważane są mniejsze podproblemy, co oznacza, że badana jest sytuacja, gdy pracownik pracował kolejno czas 1,2,...,T. Tą informację przechowują kolumny tablicy. Natomiast w wierszach tablicy rozpatrywana jest liczba klientów do obsługi, ponownie zaczynając od sytuacji, gdy należy obsłużyć tylko jednego klienta, następnie tylko dwóch, aż ostatecznie 1,...,n klientów. W rezultacie tablica *dp* na kolejnych indeksach przechowuje pewną zmienną $y_{i,w}$, która odpowiada na pytanie ile pracownik mógłby obsłużyć klientów przy czasie w, gdzie $w=1,...,T$ oraz wiedząc, że do sklepu przyszło i, gdzie $i=1,...,n$ klientów. Następnie należy skonstruować tą zmienną w taki sposób, żeby rozpoznawała ona, czy obecnie postawiona liczba klientów i czas są lepsze od poprzednich, które była rozpatrywane. Oznacza to, że dla każdego $i=1,...,n$ oraz $w=1,...,T$ musi ona spełniać poniższy warunek, przy zadanym wcześniej ciągu $times_i$, który na *i-tym* miejscu przechowuję czas zamówienia *i-tego* klienta:

1. **Jeżeli** $times_{i-1} >$ w **to** $y_{i,w} = y_{i-1,w}$
2. **W przeciwnym razie** $y_{i,w} = \max(y_{i-1,w},\ y_{i-1,w-times_{i-1}} + 1)$

(1) Otrzymaliśmy warunek na wartości $y_{i,w}$ w tablicy dp o indeksie *i, w*. Dla każdego kolejnego klienta sprawdzamy, czy czas obsługi jego zamówienia zmieści się w *w-tym* czasie pracy pracownika. Jeżeli nie, to wartość zmiennej $y_{i,w}$ przepisujemy z poprzedniej względem ilości pracowników - innymi słowy kolejnego klienta ten pracownik nie może już obsłużyć, więc wraca do obsługiwania poprzedniej ilości klientów. W przeciwnym przypadku ten kolejny klient może zostać obsłużony, bo czas jego zamówienia zmieści się w *w-tym* czasie pracy pracownika, a więc zmiennej $y_{i,w}$ musimy przypisać decyzję o obsłużeniu kolejnego klienta. W tym celu sprawdzamy, czy pracownik mógłby obsłużyć poprzedniego klienta, pod warunkiem, że miałby do dyspozycji $w - times_{i-1}$ czasu pracy, a więc dokładnie tyle ile miałby gdyby najpierw obsłużył obecnego pracownika, a dopiero później tego, którego obsłużył poprzednio. Do tej wartości dodajemy liczbę 1 reprezentującą jego wartość, ponieważ poprzednio założyliśmy, że każdy z pracowników jest jednakowo priorytetowany. W tym momencie możemy zauważyć, że nasz algorytm liczy się z konsekwencjami swoich wyborów i sprawdza, czy w obecnym kroku zamiana kolejności obsługi pracowników ma jakiekolwiek znaczenie. W przypadku, gdyby się okazało, że ta obsługa jest mniej optymalna, niż w przypadku kroku *(i-1,w)*, to wstawiamy w miejsce zmiennej $y_{i,w}$, tą bardziej optymalną korzystając z funkcji max.
(2) W ten sposób dla każdego pracownika tworzona jest tablica z decyzją wyboru najbardziej optymalnych klientów do obsługi. Następnym krokiem jest dodanie optymalnych wyborów do grafiku danego pracownika. Wybór nazwiemy optymalnym, jeśli w (1) okazało się, że zamiana klientów w *(i,w)*-tym kroku nie daje konsekwencji wyjścia po za rozpatrywany czas w. W tym celu należy sprawdzić warunek $y_{i-1,w}$!= $y_{i,w}$. Wówczas możemy wstawić czas zamówienia $times_{i-1}$ do tablicy zawierającej obsłużone czasy zamówień oraz klienta *i-1* do grafiku rozpatrywanego pracownika.
(3) Ostatnią czynnością do dokończenia algorytmu jest wstawić 0 w miejsce klientów, którzy zostali już obsłużeni przez wcześniejszych pracowników. Powtarzając czynność dla każdego pracownika możemy skończyć algorytm zwracając finalne grafiki pracowników.

4 Pseudokod

Na podstawie powyższych kroków podejścia do zadania od mniejszych podproblemów do głównego problemu stworzyliśmy poniższy algorytm. Dla każdego pracownika tworzymy tablicę dynamiczną jak w (1). W dalszej, środkowej części kodu decydujemy, których klientów dodać do grafiku pracownika bazując na krokach z (2). W ostatniej części kodu przypisujemy zero dla tych z klientów, którzy zostali już obsłużeni przez poprzednich pracowników. Tą procedurę powtarzamy dla każdego z m pracowników.

Algorithm 1 Pseudokod dla rozwiązania zadania

```
1: procedure MultiKnapsack(m,n,times,T)
2:     employerShedule ← makeBlankList(m)
3:     employerTimes ← makeBlankList(m)

4:     for employer ← 0 to m − 1 do
5:         dp ← makedpList(n, T)
6:         for i ← 1 to n + 1 do
7:             for w ← 0 to T do
8:                 if times[i − 1] = 0 or times[i − 1] > w then
9:                     dp[i][w] ← dp[i − 1][w]
10:                else
11:                    dp[i][w] ← max(dp[i − 1][w], dp[i − 1][w − times[i − 1]] + 1)
12:                end if
13:            end for
14:        end for

15:        w ← T
16:        j ← m − 1 − employer
17:        for i ← n to 1 step −1 do
18:            if dp[i − 1][w]! = dp[i][w] then
19:                w ← w − times[i − 1]
20:                if sum(employerTimes[j]) + times[i − 1] ≤ T then
21:                    if times[i − 1]! = 0 then
22:                        employerTimes[j].addElement(times[i − 1])
23:                        employerShedule[j].addElement(i − 1)
24:                    end if
25:                end if
26:            end if
27:        end for

28:        toRemove ← employerTimes[j]
29:        for el ∈ toRemove do
30:            index ← myIndex(times, el)
31:            times[index] ← 0
32:        end for
33:    end for
34:    return employerShedule
35: end procedure
```

Dodatkowe funkcje:
makeBlankList - funkcja tworzy pustą tablicę długości m
makedpList - tworzy tablicę dwuwymiarową n x T wypełnioną zerami
myIndex - wyszukuje pierwszy index elementu el w liście times
addElement - dodaje element do listy

5 Przykład działania algorytmu

Aby lepiej zrozumieć w jaki sposób działa nasza tablica dynamiczna rozpatrzmy prosty przykład. Załóżmy, że sklep internetowy zatrudnia dwóch pracowników oraz że każdy z nich ma maksymalny czas pracy równy 4. Rozpatrzmy sytuację, w której do sklepu przyszło 4 klientów i wiemy, że czas obsługi ich zamówień to kolejno [2,2,4,1]. Przeanalizujmy działanie wyżej postawionego algorytmu, aby zobaczyć jakie decyzje będzie on podejmował. Algorytm analizuje jednego pracownika naraz, a więc na początku skupia się na najszybszym rozdysponowaniu czasów pracy dla pierwszego pracownika. W pierwszym kroku algorytm tworzy dla niego tablicę dynamiczną dp, której zasady działania są takie, jak w Sekcji 3. Po włączeniu algorytmu wynik pracy tablicy dynamicznej został poniżej przedstawiony jako macierz (tablica dwuwymiarowa) w której rozpatrywane są poszczególne podproblemy.

	Czas = 0	Czas = 1	Czas = 2	Czas = 3	Czas = 4
0 klientów	0	0	0	0	0
1 klient	0	0	1	1	1
2 klientów	0	0	1	1	2
3 klientów	0	0	1	1	2
4 klientów	0	1	1	2	2

Tabela 1: Tablica dynamiczna dla pierwszego pracownika

Przeanalizujmy decyzje podjęte przez powyższą tablicę. W kolejnych kolumnach rozpatrywane są podproblemy, w których pracownik zamiast danego czasu pracy, wynoszącego 4, pracowałby mniej, kolejno: 0 czasu, 1 czasu, 2 czasu, 3 czasu aż w końcu 4 jednostki czasu pracy. Natomiast w wierszach tablica bierze pod uwagę mniejszą liczbę klientów do obsługi tzn. na początku sprawdza, co by się stało, gdyby zamiast 4 do sklepu przyszło 0, 1, 2, 3 klientów, aż w ostatniej kolumnie 4. Tablica bierze klientów do rozpatrzenia w kolejności, w jakiej zostali oni podani w danej tablicy. W ten sposób w *i,w*-tej komórce macierzy przechowywana jest decyzja o liczbie obsłużonych klientów, gdyby pracownik obługiwał i klientów przy czasie pracy równym w jednostek czasu pracy. *i=0,...,4* - liczba klientów do obsługi przy danym czasie pracy *w=0,...,4*.

Algorytm chodzi po tablicy wierszowo. Zatem na początku sprawnie rozpoznał, że przy liczbie 0 klientów nie będzie mógł obsłużyć nikogo, w żadnym z możliwych czasów pracy mniejszych bądź równych 4, a zatem w wierszu pozostawił zera. Następnie, sprawdził sytuację, w której pracownik miałby do obsłużenia tylko pierwszego klienta, który wymaga czasu równego 2 do obsłużenia. Dla czasów 0,1 algorytm zostawił w tablicy 0, natomiast już dla czasu 2 zauważył, że jest możliwa obsługa tego klienta, zatem wartość komórki zmieniła się na 1, zgodnie z warunkiem:
$y_{i,w} = \max(y_{i-1,w}, y_{i-1,w-times_{i-1}} + 1)$.

Oczywiście, przy zwiększeniu liczby jednostek czasu do 3 i 4 nie zmieniamy kolejnych wartości komórek, ponieważ jedyny rozpatrywany przez nas klient, został już obsłużony. Następnie dla dwóch pierwszych klientów przy czasach pracy 0,1,2,3 pracownik nie mógłby obsłużyć kolejnego, ponieważ ich czas obsługi to wartości [2,2]. Jednak już dla czasu równego 4 algorytm zauważył, że jest możliwe obsłużenie tych dwóch klientów i zmienił wartość komórki na 2. Dalej, analogicznie, sprawdzamy możliwość obsługi obecnych klientów wraz z trzecim i czwartym, i widzimy, że nie jest możliwe obsłużenie większej liczby klientów niż 2. Wartość komórki w ostatnim wierszu i ostatniej kolumnie jest zawsze odpowiedzią na pytanie, ilu klientów możemy obsłużyć przy danej ich liczbie i w danym czasie. Warto również zauważyć, że algorytm podaje zawsze bardziej optymalną z możliwości doboru klientów. Dla czasu równego 3 i przy rozpatrywaniu wszystkich czterech klientów, odpowiada on, że można obsłużyć aż dwóch klientów (pierwszego i ostatniego, o czasach obsługi odpowiednio 2 i 1). Dzieje się tak, ponieważ algorytm skutecznie sprawdza (zauważmy, że czwarty klient jest obsługiwany w czasie równym 1), że dla poprzedniej puli klientów i dla czasu równego 3 było możliwe obsłużenie klienta pierwszego. (Wprowadzono już do komórki wcześniej cyfrę 1). W ten sposób algorytm liczy się z konsekwencją swojego wyboru z przeszłości i stwierdza, że mimo iż pracownik mógł już obsłużyć klienta numer 1, to w aktualnie rozpatrywanej sytuacji może obsłużyć także klienta numer 4.

Ostatecznie, cała tablica jest już zapełniona i teraz w kolejnej pętli (2) wybieramy tych klientów, których wybraliśmy jako optymalnych. Tablica employerTimes przechowuje czasy obsługi, które już dodaliśmy do obecnego pracownika. Na początku jest ona oczywiście pusta. Wybranie klienta jest optymalne, jeżeli obecna wartość w wierszu dla czasu równego *w* różni się od wartości w tym wierszu dla czasu równego *w-1*. Dodatkowo pętla idzie kolumnowo (a więc rozpatruje najpierw obsługę kolejnych klientów od ostatniego oraz przy maksymalnej liczbie czasu) od końca tablicy dynamicznej dp (aby maksymalnie wypełnić czas pracy pracowników maksymalną liczbą klientów). Sprawdzany jest dodatkowo warunek, czy nie wyszliśmy poza czas pracy pracownika równy 4. Jeżeli wyjdziemy poza ten czas, to pętla skończy dodawanie kolejnych pracowników. Zatem, dla naszego przykładu, do tablicy employerTimes dodany zostanie klient numer 2 oraz klient numer 1, ponieważ są to pierwsze wartości, które różnią się w kolumnie czas=4, oraz nie wyszliśmy poza czas pracy równy 4, a obsługa tych dwóch klientów trwa odpowiednio czasy 2 i 2. Kończąc, dodajemy tych klientów do tablicy employerShedule i kończymy obliczenia dla tego pracownika zerując jeszcze czasy obsługi wybranych klientów, aby oznaczyć że są oni już obsłużeni. Dzięki temu, przy rozpatrywaniu kolejnego pracownika, tablica czasów obsługi klientów jest postaci [0,0,4,1] oraz algorytm zawsze omija wartości czasów obsługi które są niedodatnie.

Przy kolejnym pracowniku następuje analogiczny system selekcji klientów, tablica dynamiczna dp jest postaci:

	Czas = 0	Czas = 1	Czas = 2	Czas = 3	Czas = 4
0 klientów	0	0	0	0	0
1 klient	0	0	0	0	0
2 klientów	0	0	0	0	0
3 klientów	0	0	0	0	1
4 klientów	0	1	1	1	1

Tabela 2: Tablica dynamiczna dla drugiego pracownika

Tym razem widzimy, że jedynie klient numer 3 został zakwalifikowany jako dobry wybór do obsługi przez drugiego pracownika. Istotnie sprawdźmy, że czas jego obsługi wynosi 4, co również jest najbardziej optymalnym wyborem z punktu widzenia maksymalnego czasu pracy naszego pracownika. Analogicznie jak poprzednio dodajemy klienta do grafiku pracownika i kończymy program.

Ostateczny wynik dla tego przykładu to: [[3],[2,1]],
co odpowiada czasom obsługi tych klientów [[4],[2,2]].

6 Złożoność algorytmu

Pokażemy, że zaprezentowany wyżej algorytm jest o złożoności *O(Tnm)*. Rozpatrzmy najpierw część (1) algorytmu, a więc tą, w której tworzymy tablicę dynamiczną. W pętli wewnętrznej w = 0,...,T mamy operację tylko jednego przypisania w zależności od warunku $times_{i-1}$. Tą operację wykonujemy jeszcze n razy, dla każdego klienta. Stąd złożoność pierwszej, podwójnej pętli wewnętrznej to T(nT)

$$\sum_{i=1}^{n+1}\sum_{i=0}^{T} c = \sum_{i=0}^{n} Tc = nTc \tag{5}$$

Następnie przypisujemy dwie zmienne, oczywiście jest to operacja stała liniowo, zatem możemy zapisać, że koszt tej operacji to łącznie c_w, c_j Dalej w (2) wpisanie obliczonych czasów zamówienień jako numery ich klientów w grafik to złożoność O(n-1), bo:

$$\sum_{i=1}^{n}\sum_{i=0}^{T} c_1(c_2 + (c_3c_4)) = (n-1)c_1(c_2 + (c_3c_4)) \tag{6}$$

Gdzie c_1,c_2,c_3,c_4 to stałe odpowiedzialne za kolejno: zewnętrzny warunek if, przypisanie wartości do w oraz dwa najbardziej wewnętrzne operatory porównania if. Pozostało nam obliczyć złożoność usuwania obsłużonych klientów z listy $times_i$ (zerowania czasów zamówień, które zostały już zrealizowane). W tym kroku tworzymy tablicę, która ma tyle samo elementów co liczba obsłużonych klientów, a następnie iterujemy po tej tablicy. Możemy zauważyć, że w najgorszej możliwej sytuacji za każdym razem liczba obsłużonych klientów wynosić będzie T (wtedy, gdy każde zamówienie ma wartość 1 oraz przy założeniu, że mamy wystarczającą liczbę klientów). Zatem złożoność tej pętli, to:

$$\sum_{i=0}^{T-1} c_5 + c_6 = (T-1)(c_5 + c_6) \tag{7}$$

Ostatecznie każdą z powyższych procedur powtarzamy tyle razy, ile obróci się pętla zewnętrzna iterująca po liczbie klientów. Stąd ostateczne równanie wyraża sie poprzez:

$$\sum_{e=0}^{m-1}(nTc + (n-1)c_1(c_2 + (c_3c_4)) + (T-1)(c_5 + c_6)) = \tag{8}$$

$$= m(nTc + (n-1)c_1(c_2 + (c_3c_4)) + (T-1)(c_5 + c_6)) = \tag{9}$$

$$= mnTc + mn(c_2 + (c_3c_4) - mc_1(c_2 + (c_3c_4) + m(T-1)(c_5 + c_6) \tag{10}$$

Ostatecznie uwzględniając stałe otrzymaliśmy całkowitą złożoność algorytmu:

$$O(mnT + mn - m + mT) = O(mnT) \tag{11}$$

7 Testy

Sprawdźmy teraz, czy zaimplementowany algorytm działa poprawnie również dla bardziej złożonych problemów. W tym celu ustalmy liczbę pracowników na 4 i zobaczmy, jak algorytm przydziela pracownikom zadania dla liczby klientów ze zbioru {20, 40} oraz czasu pracy ze zbioru {4, 6, 8}. Przejdziemy przez wszystkie sześć kombinacji zmiennych. Wylosowane czasy realizacji zamówień to:

$$[4, 2, 4, 3, 1, 4, 2, 1, 2, 4, 3, 2, 1, 4, 3, 3, 4, 1, 4, 4]$$

dla 20 klientów;

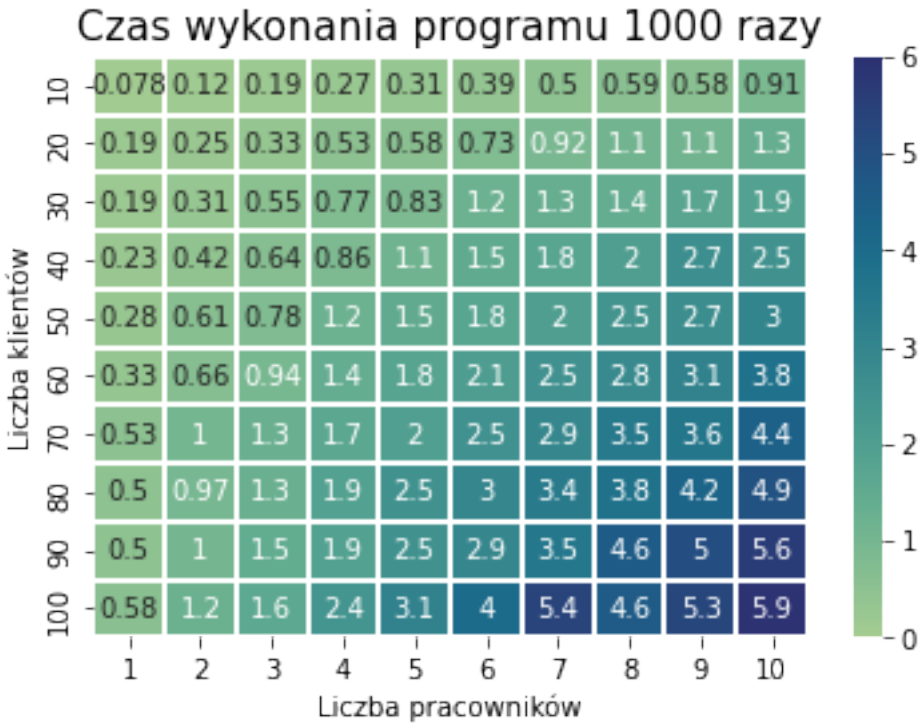
$$[2, 4, 3, 4, 2, 3, 1, 1, 1, 2, 1, 4, 4, 4, 1, 2, 3, 4, 2, 4, 2, 4, 3, 2, 2, 3, 1, 3, 4, 4, 3, 3, 4, 1, 1, 2, 3, 4, 2, 2]$$

dla 40 klientów.
Dla czytelności testów, algorytm wyjątkowo zwróci nie numery zamówień, a czasy ich realizacji. Pozwala to lepiej zobrazować jego działanie. Wyniki testów przedstawia Tabela 3.

Parametry	$n = 20$, $T = 4$	$n = 20$, $T = 6$	$n = 20$, $T = 8$	$n = 40$, $T = 4$	$n = 40$, $T = 6$	$n = 40$, $T = 8$
Pracownik 1	[4]	[3, 3]	[4, 4]	[2, 2]	[2, 2, 2]	[2, 3, 3]
Pracownik 2	[2, 2]	[3, 3]	[3, 3, 2]	[2, 2]	[2, 2, 2]	[2, 2, 2, 2]
Pracownik 3	[2, 2]	[2, 2, 2]	[3, 2, 3]	[1, 1, 1, 1]	[1, 1, 2, 2]	[2, 2, 2, 2]
Pracownik 4	[1, 1, 1, 1]	[1, 1, 1, 1, 2]	[1, 1, 1, 2, 1, 2]	[1, 1, 1, 1]	[1, 1, 1, 1, 1, 1]	[1, 1, 1, 1, 1, 1, 1, 1]
Zostało	[4, 3, 4, 4, 3, 4, 3, 3, 4, 4, 4]	[4, 4, 4, 4, 4, 4, 4, 4]	[4, 4, 4, 4, 4, 4]	[4, 3, 4, 3, 4, 4, 4, 3, 4, 2, 4, 2, 4, 3, 2, 2, 3, 3, 4, 4, 3, 3, 4, 2, 3, 4, 2, 2]	[4, 3, 4, 3, 4, 4, 4, 3, 4, 4, 4, 3, 3, 3, 4, 4, 3, 3, 4, 2, 3, 4, 2, 2]	[4, 4, 4, 4, 4, 3, 4, 4, 4, 3, 3, 3, 4, 4, 3, 3, 4, 3, 4, 2, 2]
Obsłużono klientów	9	12	14	12	16	19

Tabela 3: Wyniki testu poprawności

Jak widać, algorytm działa poprawnie. Zawsze maksymalnie wykorzystuje miejsce w grafiku pracownika. Zwiększenie czasu pracy, przy tej samej liście zamówień, wiąże się z obsłużeniem większej liczby klientów, co jest bardzo pożądanym wynikiem. Ponadto, procedura rozpoznaje, że bardziej opłaca się w pierwszej kolejności obsługiwać zadania o najmniejszym czasie realizacji. Tego również należało się spodziewać od dobrego algorytmu.



Rysunek 1: Heatmapa czasu wykonania

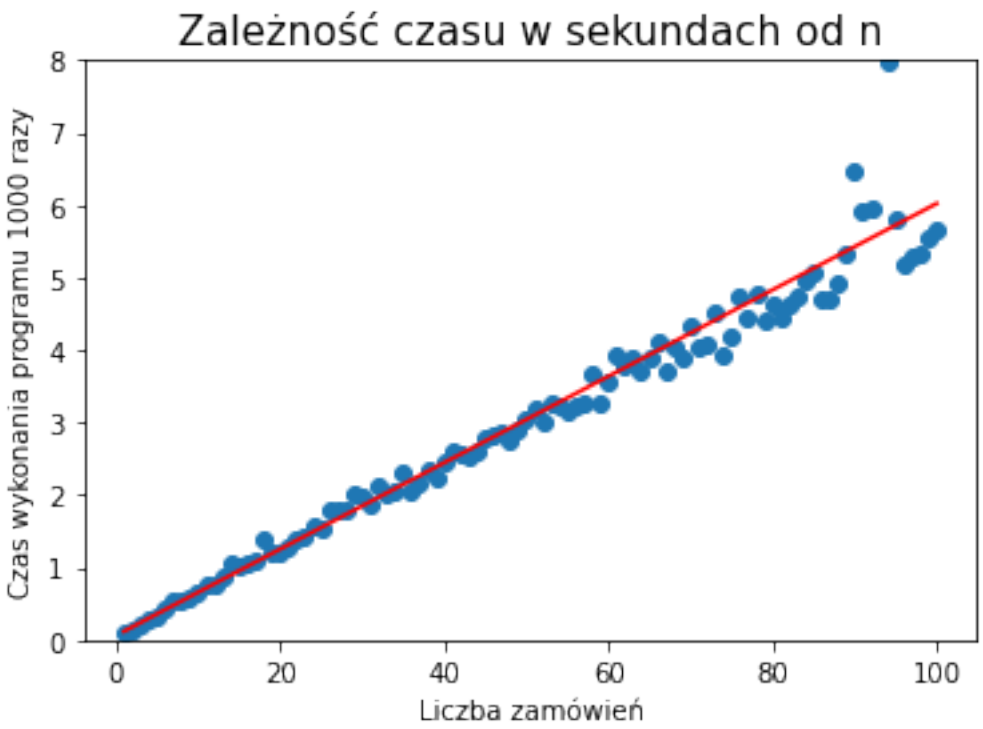
8 Wizualizacje

8.1 Czas wykonania programu

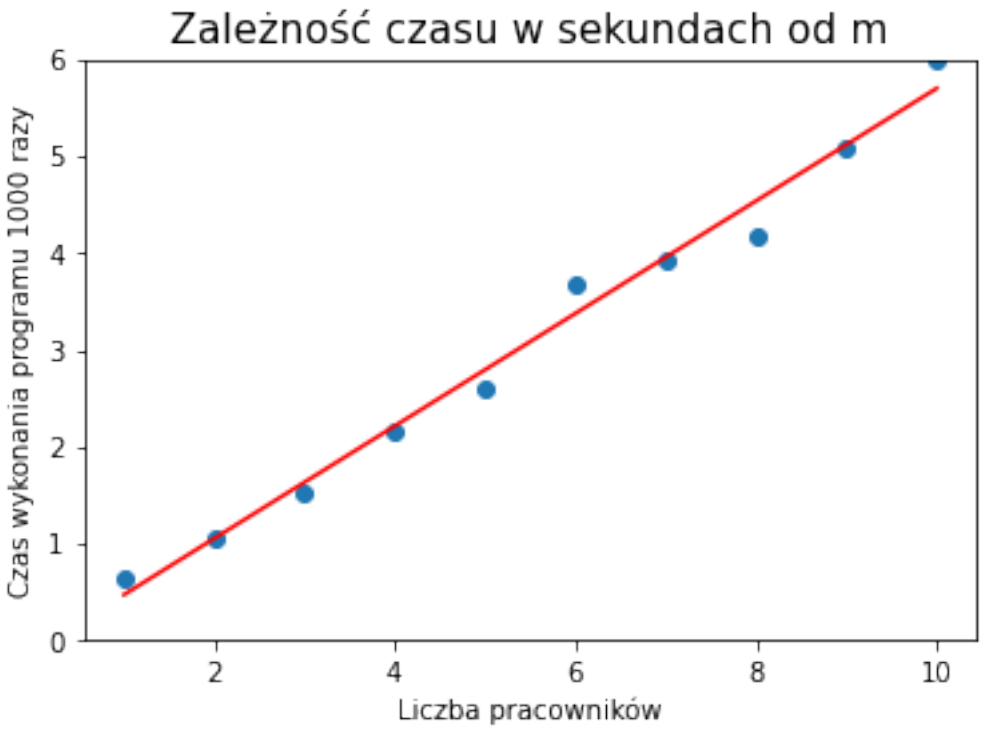
Z powyższych rozważań znamy już teoretyczną czasową złożoność algorytmu. Pokażmy ją jednak w praktyce. Czas wykonania programu raz jest na tyle mały, że nie warto rozważać pojedynczego jego wykonania. Zajmiemy się więc czasem obliczeń 1000 iteracji. Rysunek 1 przedstawia zależność czasu od dwóch najważniejszych zmiennych, a mianowicie, liczby klientów i liczby pracowników. Obserwujemy na nim to, co zgodne z intuicją, czyli że im więcej zamówień i pracowników, tym dłużej wykonuje się program.

8.2 Zależność czasu od danych wejściowych

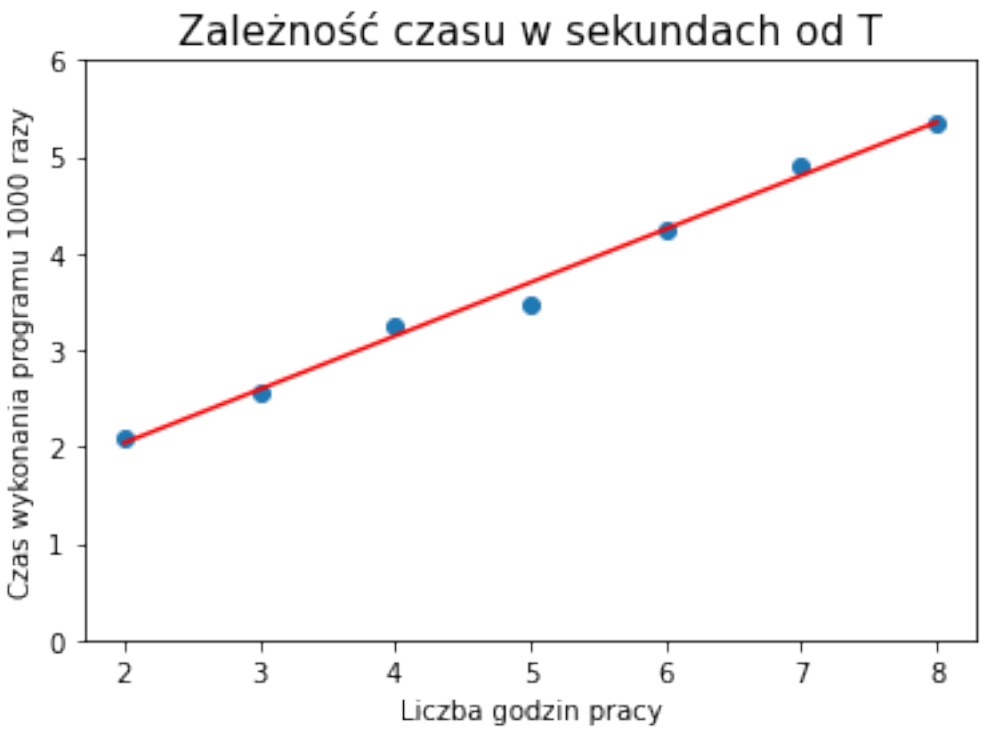
Jeżeli przyjrzymy się uważnie Rysunkowi 1, nożemy zauważyć, że przy przy ustalonej drugiej zmiennej, zależność od tej pierwszej jest liniowa. Spróbujmy to jednak bardziej unaocznić. Rysunki 2, 3, 4 zawierają wyniki eksperymentów badania złożoności czasowej algorytmu ze względu na jedną zmienną przy ustalonych pozostałych z dopasowanym modelem regresji liniowej.



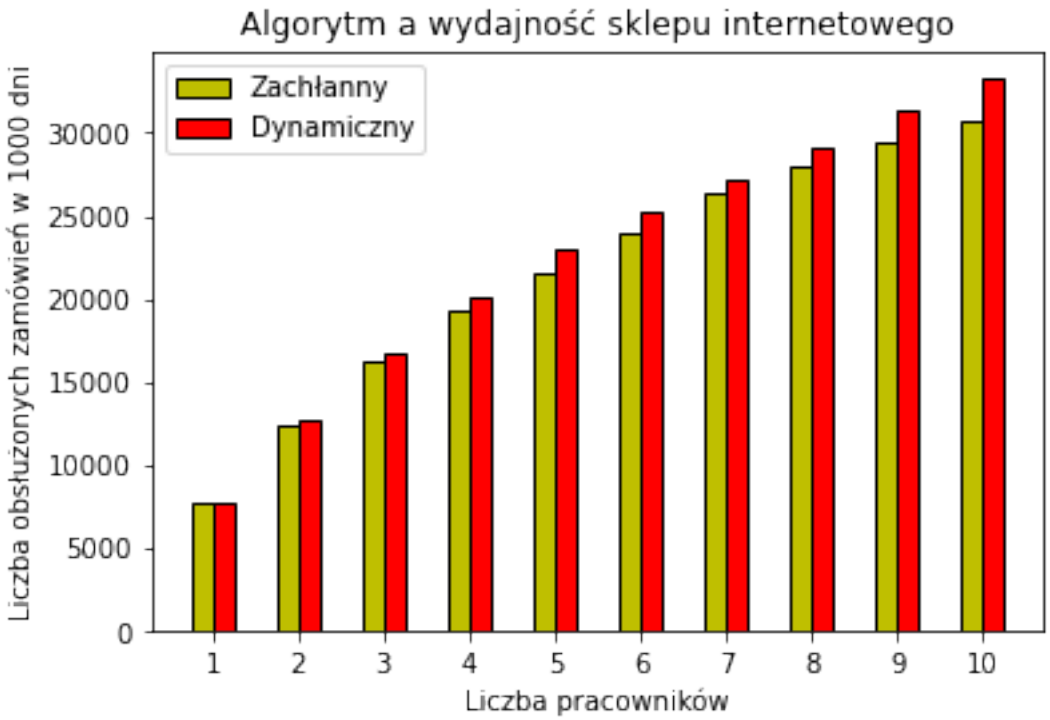
Rysunek 2: Zależność czasu od liczby zamówień



Rysunek 3: Zależność czasu od liczby pracowników



Rysunek 4: Zależność czasu od liczby godzin pracy



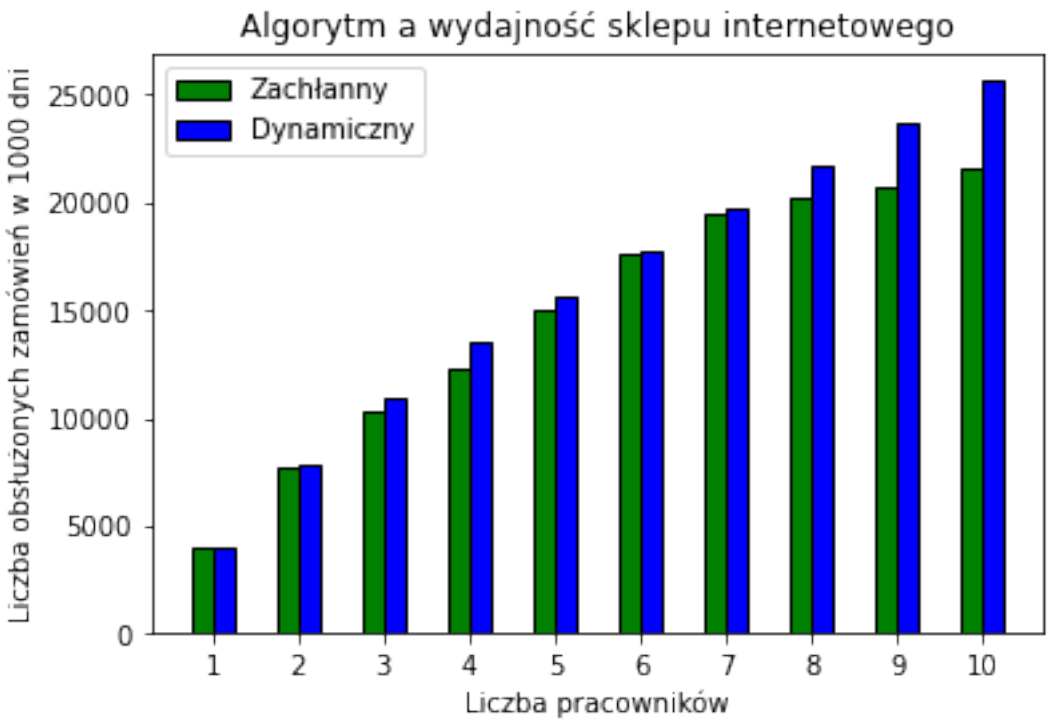
Rysunek 5: Porównanie algorytmów

Jak widać, teoretycznie wyznaczona złożoność zgadza się z tą zbadaną empirycznie.

8.3 Porównanie skuteczności z innym algorytmem

Podczas pracy nad optymalnym algorytmem rozważone zostało również podejście zachłanne. Okazało się ono jednak niewystarczające, gdyż często wiązało się ono ze stratą w łącznej liczbie zrealizowanych zamówień, które dało się lepiej porozdzielać między pracowników. Algorytm zachłanny nie zawsze jednak wypada gorzej. Spróbujmy w jakiś sposób pokazać, ile tracimy używając go, zamiast zaproponowanego algorytmu dynamicznego. Zauważmy, że im większą liczbą pracowników dysponujemy, tym większa jest liczba kombinacji przypisań klientów do pojedynczych pracowników, a co za tym idzie, algorytm dynamiczny ma szanse pokazać swoją wyższość. Tak jest w istocie, co pokazuje Rysunek 5.

Zauważmy też, że im mniej zamówień o czasie wykonania 1, tym trudniej będzie algorytmowi zachłannemu rozdzielić klientów między pracowników. Spróbujmy wykonać analogiczny eksperyment, z tą tylko różnicą, że wszystkie zamówienia mają czas wykonania co najmniej 2. Wyniki przedstawione są na Rysunku 6.



Rysunek 6: Porównanie algorytmów, gdy zamówienia są większe

Po dokonaniu tej nieznacznej modyfikacji, okazuje się, że algorytm dynamiczny radzi sobie znacznie lepiej od zachłannego, szczególnie przy dużej liczbie pracowników.

9 Podsumowanie

Podsumowując, zagadnienie przydzielenia zadań pracownikom sklepu, okazało się być szczególnym przypadkiem dobrze znanego problemu plecakowego, a dokładniej, wielowymiarowym problemem plecakowym. Rozwiązanie wymagało użycia programowania dynamicznego. Uzyskany algorytm cechował się liniową złożonością czasową ze względu na każdy z trzech najważniejszych jego parametrów. Poprawnie przydzielał zadania pracownikom i maksymalnie wykorzystywał miejsce w grafiku. W porównaniu z algorytmem zachłannym radził sobie znacznie lepiej, tym lepiej, o ile bardziej skomplikowany był przykład.

Literatura

[1] $http://www.cs.put.poznan.pl/mszachniuk/mszachniuk\_files/lab\_aisd/Szachniuk-ASD-t5.pdf$