Teoria algorytmów i obliczeń
Problem prostokątów - dokumentacja

Krzysztof Tabeau Kacper Słowikowski

Październik 2020

Spis treści

1 Wstęp

Poniższa praca opisuje problem i rozwiązanie zadania, które można porównać do popularnej gry Tetris. Zadanie polega na ułożeniu w zamkniętym obszarze lub przestrzeni elementów o różnych kształtach. Występuje tu wiele wariantów problemu. Jednym z nich jest poszukiwanie ułożenia, w którym uda nam się zmieścić w wyznaczonym obszarze jak najwięcej zadanych na wejściu elementów. Rozwiązanie takiego problemu może być przydatne w sytuacji w której trzeba zaplanować przeprowadzkę mając do dyspozycji dużo rzeczy i mało miejsca bagażowego.
Nasz problem definiujemy w ten sposób, że chcemy dopasować zadane elementy do minimalnego obszaru w jakim się zmieszczą wykonując jak najmniej operacji na elementach. Realnym przykładem naszego problemu może być ułożenie prostokątnego placu z kostek o różnych kształtach. Na placu nie mogą pozostawać puste miejsca, a kostki można ciąć (koszt przecięcia jest niezerowy).

2 Opis problemu

Problem optymalizacyjny ścieżki podstawowej - Tetris.
W naszej wersji na wstępie otrzymujemy zadaną liczbę klocków, złożonych z 5 kwadratów (bloków), o różnych predefiniowanych kształtach. Każdy blok jest kwadratem 1x1. Bloki w klockach muszą stykać się całą krawędzią. Należy je ułożyć w taki sposób, aby bloki na siebie nie zachodziły oraz wszystkie zawierały się w prostokącie o polu równym sumie pól wszystkich kawałków. Prostokąt w którym zawierają się kawałki ma kształt jak najbardziej zbliżony do kwadratu (na przykład prostokąt o polu 24 powinien mieć boki 6x4, a nie 24x1, 12x2 czy 8x3).
Informacja o klockach przekazywana jest w pliku tekstowym w postaci kolejnych liczb mówiących ile razy występuje dany kawałek.
Na każdym z klocków, jeśli jest to konieczne, można wykonać cięcia tak, aby otrzymać klocki o innych kształtach, składające się z mniejszej liczby bloków. Cięcia możemy wykonywać w linii prostej, wzdłuż brzegów bloków od pustej przestrzeni do pustej przestrzeni. Rozwiązanie tego problemu wykonamy w dwóch wariantach:
a) algorytmem znajdującym optymalne rozwiązanie - rozwiązanie, w którym do ułożenia klocków musieliśmy wykonać najmniej cięć kawałków
b) wykorzystując algorytm heurystyczny - czyli algorytm nie dający gwarancji znalezienia rozwiązania optymalnego, ale umożliwiający znalezienie rozwiązania w rozsądnym czasie.

3 Algorytm dokładny

3.1 Opis

Ogólnym pomysłem na rozwiązanie problemu jest sprawdzenie wszystkich możliwości i wybraniu z nich najlepszych. Osiągniemy to w następujących krokach

1. Znalezienie wymiarów prostokąta do którego będziemy dopasowywać elementy
2. Stworzyć kolejkę zbiorów elementów, które będziemy rozpatrywać z etykietą ile operacji wymagał zbiór, aby go utworzyć
3. Do kolejki dodać zbiór wejściowy z liczbą potrzebnych operacji równą 0
4. Dopóki kolejka jest niepusta
   (a) Jeśli liczba potrzebnych operacji do uzyskania zbioru na początku kolejki jest większa niż dla znalezionego już rozwiązania lub układ był już rozważany, zbiór jest usuwany z kolejki i nie jest dalej rozważany.
   (b) Rozważyć wszystkie możliwe ułożenia elementów dla zbioru znajdującego się na początku kolejki za pomocą backtrackingu
   (c) Jeśli udało się znaleźć rozwiązanie to zapisać liczbę operacji potrzebnych do osiągnięcia danego zbioru i dodać wszystkie rozwiązania tego zbioru do zbioru rozwiązań.
   (d) W przeciwnym przypadku stworzyć nowe zbiory z elementami rozważanego zbioru po zastosowaniu jednej operacji dla jednego elementu tego zbioru i dodać każdy taki zbiór na koniec kolejki z oznaczeniem, że liczba operacji potrzebnych do osiągnięcia tych zbiorów jest o jeden większa od rozważanego zbioru.

Backtracking osiągamy za pomocą rekurencji. Funkcja ta działa w następujący sposób:

1. Przy pierwszym uruchomieniu tej funkcji definiujemy: listę rozważanych elementów, obszar w którym będziemy umieszczać elementy (na początku obszar jest wypełniony wartościami -1 tzn. dla algorytmu jest to puste miejsce), oraz podajemy numer elementu w liście elementów który będziemy w tym kroku backtrackingu podawać (na początku równy 0).
2. Dla każdego pola w rozważanym układzie rozważamy każde ułożenie rozważanego elementu pokrywającego to pole za pomocą funkcji *dopasuj*
3. Dla każdego pomyślnego umieszczenia elementu, uruchamiamy funkcję backtracking z tą samą listą elementów, nowym układem obszaru oraz kolejnym elementem na liście do rozważenia.
4. Pomyślne rozwiązania łączymy w jedną listę i zwracamy.

Funkcja *dopasuj* znajduje wszystkie możliwe umieszczenia danego ułożenia elementu w obszarze pokrywającego dane pole

1. Dla każdego pola elementu zakładamy, że pokrywa rozważane pole w obszarze
2. Sprawdzamy czy pozostałe pola w elementu umieszczone w obszarze nie będą kolidować.
3. Jeśli nie będą kolidować to dodajemy umieszczenie do zbioru udanych dopasowań
4. Zwracamy zbiór dopasowań elementu

3.2 Pseudokod

Funkcja główna:
begin
stwórz pustą listę *rozwiązania*;
/* do tej listy będziemy dodawać optymalne rozwiązania */
stwórz pustą listę *rozpatrzone*;
/* do tej listy będziemy rozpatrzone zbiory elementów */
min <- $+\infty$;
/* minimalna liczba operacji potrzebnych do znalezienia rozwiązania */
użyj funkcji *dopasuj_prostokąt*, aby znaleźć *wysokość* i *szerokość* prostokąta;
/* prostokąta do którego będziemy dopasowywać elementy */
stwórz pustą kolejkę l;
/* w tej kolejce będą przetrzymywane zbiory elementów do rozpatrzenia oraz
liczba operacji potrzebna do ich osiągnięcia */
dodaj na koniec l zbiór początkowy z liczbą 0;
while(*l* nie jest puste)
elementy <- zbiór na początku l;
elementy_min <- liczba operacji potrzebna do osiągnięcia zbioru *elementy*;
usuń zbiór z na początku l;
if(*elementy_min* < *min* && *rozpatrzone* nie zawiera *elementy*)
stwórz *nowy_układ* o wysokości *wysokość* i szerokości *szerokość*;
/* układ w którym będziemy umieszczać elementy */
$nowy\_ukad_{ij}$ <- (-1) for i=1,2...*wysokość* for j=1,2...*szerokość*;
rozwiązania_elementów <- użyj funkcji *backtrack*(*elementy*, *0*, *nowy_układ*);
/* funkcja *backtrack* zwraca listę rozwiązań dla danego zbioru elementów */
if(*rozwiązania_elementów* nie jest puste)
dołącz *rozwiązania_elementów* do *rozwiązania*;
min <- *elementy_min*;
else
for(*element* in *elementy*)
temp <- lista elementów z *elementy* bez *element*;
for((*a*, *b*) in *potnij*(*element*))
/* *potnij* zwraca listę wszystkich możliwych pocięć elementu na dwa kawałki
- wykonanie w czasie stałym */
dodaj do *temp* elementy a i b;
dodaj na koniec kolejki l zbiór *temp* oraz
($elementy\_min + 1$) jako liczbę potrzebnych operacji do osiągnięcia zbioru *temp*;
end
end
end
dodaj *elementy* do *rozpatrzone*;
end
end

```
    wypisz min;
    wypisz rozwiązania;
end

backtrack(zbiór elementów elementy, numer rozpatrywanego elementu nr, układ do którego dopasowujemy elementy układ)
begin
    stwórz pustą listę rozwiązania; /* do tej listy będziemy dodawać rozwiązania */
    id <- id elementu elementy_nr;
    wysokość <- wysokość układu układ;
    szerokość <- szerokość układu układ;
    if(nr jest mniejszy niż liczba elementów w elementy)
        for(i <- 1,2...wysokość)
            for(j <- 1,2...szerokość)
                if(ukad_ij == -1) /* to znaczy, że jest pusty*/
                    ułożenia <- rotacje(elementy_nr);
                    /* funkcja rotacje zwracają wszystkie rotacje elementu - wykonanie w czasie stałym */
                    for(ułożenie in ułożenia)
                        układy_z_elementem <- dopasuj(ułożenie, układ, i, j, id));
                        /* funkcja dopasuj zwraca listę wszystkich układów możliwych
                            po włożeniu zadanego elementu do zadanego układu */
                        for(nowy_układ in układy_z_elementem)
                            nowe_rozwiązania <- backtrack(elementy, nr + 1, nowy_układ));
                            /* rekurencyjne wywołanie funkcji dla kolejnego elementu */
                            dołącz nowe_rozwiązania do rozwiązania;
                        end
                    end
                end
            end
        end
    end
    zwróć rozwiązania;
end

dopasuj(konkretne ułożenie elementu ułożenie, układ do którego dopasowujemy element układ, wysokość planszy wysokość, szerokość planszy szerokość, id elementu id)
begin
    stwórz pustą listę układy_z_elementem;
    /* w tej liście będziemy trzymać układy z dopasowanym elementem */
    for(pole in ułożenie)
        jest_ok <- prawda;
        nowy_układ <- sklonuj układ;
        pole_wysokość <- współrzędna odpowiadająca ułożenia w pionie pola pole;
```

```
        pole_szerokość <- współrzędna odpowiadająca ułożenia w poziomie pola pole;
        for(sąsiednie_pole in ułożenie)
            sąsiednie_pole_wysokość <- współrzędna odpowiadająca ułożenia w pionie pola sąsiednie_pole;
            sąsiednie_pole_szerokość <- współrzędna odpowiadająca ułożenia w poziomie pola sąsiednie_pole;
            w <- sąsiednie_pole_wysokość - pole_wysokość + wysokość;
            s <- sąsiednie_pole_szerokość - pole_szerokość + szerokość;
            if(nowy_ukad_ws == -1)
            /* jeśli pole jest puste */
                nowy_ukad_ws <- id;
            else
                jest_ok <- fałsz;
            end
        end
        if(jest_ok jest prawda)
            dodaj nowy_układ do układy_z_elementem;
        end
    end
    zwróć układy_z_elementem;
end

dopasuj_prostokąt(zbiór elementów elementy)
begin
    suma <- 0;
    /* w suma będziemy przechowywać sumę pól wszystkich elementów */
    for(element in elementy)
        rozmiar <- suma pól elementu element;
        suma += rozmiar;
    end
    max <- zaokrąglenie w dół do liczby całkowitej pierwiastka z suma;
    for(i <- max...0)
        if(rozmiar%i == 0)
            zwróć i oraz rozmiar/i;
        end
    end
end
```

3.3 Dowód poprawności

Plan dowodu:

1. Dowód, że algorytm zawsze się skończy i zwróci rozwiązanie
2. Dowód, że algorytm znajdzie optymalne rozwiązanie pod względem liczby operacji

Algorytm działa za pomocą kolejki w której są umieszczone zbiory elementów, które będzie rozpatrywał. Warto zauważyć, że w kolejce zawsze będzie posortowana pod względem liczby operacji potrzebnych do ich osiągnięcia danego zbioru.

1. Algorytm się skończy kiedy kolejka będzie pusta. Jako że w każdej iteracji pętli pierwszy zbiór w kolejce jest usuwany, to jedynym warunkiem opróżnienia kolejki jest nie dodawanie kolejnych zbiorów na jej koniec. Algorytm jest tak napisany, że jak znajdzie jedno rozwiązanie to, nie dodaje już nowych zbiorów na koniec kolejki.

Jeśli algorytm znajdzie rozwiązanie to kończy dowód. Załóżmy, że nie znajdzie rozwiązania.

W takim razie, aby dowiedzieć się kiedy zbiory nie będą dodawane do kolejki trzeba rozważyć w jakich przypadkach będą dodawane.
Będzie się tak działo kiedy funkcja *potnij* (która zwraca wszystkie możliwe podziały po wykonaniu jednej operacji na elemencie) zwróci niepusty zbiór przynajmniej dla jednego elementu w zbiorze. Co z kolei się zdarzy wtedy i tylko wtedy jeśli jakiś element w zbiorze można pociąć na mniejsze części. Element można pociąć wtedy i tylko wtedy kiedy jego pole jest większe od 1.
Jako że w każdej iteracji obsługi kolejki na koniec kolejki są dodawane zbiory o liczbie operacji o jeden większej od zbioru zdjętego z początku kolejki, to dojdzie do momentu, kiedy na koniec kolejki zostanie włożony zbiór, którego nie można podzielić na mniejsze elementy.
Co za tym idzie, kolejne zbiory nie będą dodawane na koniec kolejki.

To dowodzi, że algorytm się skończy.

Rozważmy teraz ten ostatni zbiór dodany do kolejki. Jeśli nie można go pociąć na mniejsze kawałki to znaczy, że składa się on z pewnej liczby elementów o polu 1. W takim razie ten zbiór jest rozwiązaniem problemu, gdyż taki zbiór zawsze można dopasować do obszaru prostokąta o całkowitych bokach i o tym samym polu co suma pól elementów w zbiorze.

Dochodzimy do sprzeczności, czyli algorytm zawsze znajdzie rozwiązanie

2. Najpierw dowiedziemy, że funkcja backtrack rozważa każde możliwe ułożenie elementów w zadanym obszarze.
Dla każdego pola w obszarze, do którego chcemy dopasować elementy rozważamy każde ułożenie elementu pokrywające to pole. Rozważamy zarówno różne umiejscowienia względem tego pola jak i rotacje. Jeśli element udało się dopasować, to w ten sam sposób rozważamy dopasowanie kolejnych elementów, które w zadaniu mamy dopasować. W ten sposób funkcja rozważa wszystkie możliwe ułożenia elementów w obszarze.

Załóżmy, że algorytm nie znajdzie optymalnego rozwiązania, tzn. znajdzie rozwiązanie, do którego potrzeba

więcej operacji na elementach niż dla optymalnego rozwiązania. W takim wypadku, jako że zbiory elementów są dodawane do kolejki w kolejności rosnącej liczby operacji, to znaczy że zbiór elementów z optymalnym rozwiązaniem został przez algorytm rozpatrzony. Skoro zbiór został rozpatrzony to funkcja *backtrack* rozważyła wszystkie możliwe ułożenia elementów i zwróciła rozwiązanie.

Dochodzimy do sprzeczności, więc algorytm zawsze zwraca optymalne rozwiązanie

3.4 Złożoność

Oznaczenia:

- n - liczba elementów wejściowych
- k - liczba pól elementu
- p - pole prostokąta do którego dopasowujemy elementy
- r - liczba rotacji elementu (maksymalnie 4)
- Z - łączna suma możliwych do wykonania operacji dla zbioru początkowego

W pesymistycznym przypadku algorytm obsłuży każdy możliwy podział zbioru początkowego oraz zbiorów pochodzących od zbioru początkowego. W takim przypadku, główna pętla kontrolująca stan kolejki odbędzie następującą liczbę iteracji:

$$1 + Z + Z * (Z - 1) + Z * (Z - 1) * (Z - 2) + ... \tag{1}$$

gdzie 1 przedstawia zbiór początkowy, Z oznacza liczbę zbiorów rozważanych po jednej operacji na dowolnym elemencie, $Z * (Z - 1)$ oznacza liczbę zbiorów rozważanych po dwóch operacjach na dowolnych elementach itd.. Dzieje się tak, ponieważ po rozpatrzeniu pierwszego zbioru można dokonać Z operacji i tyle powstanie kolejnych zbiorów. W kolejnych zbiorach można wykonać co najwyżej $Z - 1$ operacji. Z racji tego, że rozpatrujemy przypadek pesymistyczny założymy, że będzie to właśnie $Z - 1$.
Powyższy wzór można zapisać następująco :

$$\sum_{i=0}^{Z} \binom{Z}{i} * i\,! \tag{2}$$

Z powyższego wzoru wynika, że po zastosowaniu Z operacji będziemy mieć $Z\,!$ zbiorów. Jest to prawda gdyż zbiór elementów o polu jeden można otrzymać na $Z\,!$ sposobów. Algorytm zamiast rozpatrywać możliwości zbioru już analizowanego, będzie je pomijać, ale wciąż musi je rozważyć w kontekście porównania czy były już rozpatrzone.
Dla każdego rozpatrywanego zbioru elementów mogą być wykonywane następujące działania:

- pewna liczba operacji ze stałym czasem wykonania (usunięcie z początku kolejki, sprawdzenie czy są rozwiązania itp.) - w rozważaniach nad złożonością uznamy, że sumą liczb tych operacji będzie C.
- stworzenie pustego układu, do którego będziemy dopasowywać elementy - złożoność: p
- rozpatrzenie czy układ był już rozważany. Złożoność: $liczba\,elementw\,w\,zbiorze * liczba\,rozpatrzonych\,zbiorw$

- użycie backtrackingu, czyli dla każdego pola w układzie (mamy p), dla każdej rotacji elementu (r), dla każdego pola elementu(k) próbujemy dopasować element tak aby pokrywał rozważane pole w układzie i jeśli jest taka możliwość to uruchamiamy ponownie backtracking dla kolejnego elementu. Złożoność: $(p * r * k)^{liczba\,elementw\,w\,zbiorze}$
- dla każdego elementu w zbiorze, dla każdego możliwego podziału dodanie nowego zbioru po wykonaniu operacji na koniec kolejki. Złożoność: $liczba\,elementw\,w\,zbiorze * liczba\,moliwych\,podziaw\,elementu$

Ograniczenie dolne złożoności pesymistycznej: Zakładając, że rozwiązanie znajdziemy w ostatnim kroku, wiemy że dla $\sum_{i=0}^{Z} \binom{Z}{i} * i\,!$ zbiorów rozpatrzymy wszystkie możliwe działania. Większość z nich niestety zależy od akurat rozważanej sytuacji. Aby znaleźć ograniczenie dolne, wszystkie operacje wymagające znajomości rozważanej sytuacji zastąpimy ich dolnym ograniczeniem następująco:

- $liczba\,elementow\,w\,zbiorze$ - n
- $liczba\,rozpatrzonych\,zbiorow$ - 0
- $liczba\,mozliwych\,podzialow\,elementu$ - 0
- r - 1
- k - 1
- p - n

Dzięki czemu uzyskujemy następujący wzór:

$$(\sum_{i=0}^{Z} \binom{Z}{i} * i\,!) * (C + n + n^n) \tag{3}$$

Jako że C jest pewną nieprzybliżoną stałą i rozważamy ograniczenie dolne to możemy złożoność algorytmu ograniczyć poprzez funkcję:

$$(\sum_{i=0}^{Z} \binom{Z}{i} * i\,!) * n^n \tag{4}$$

Ograniczenie górne złożoności pesymistycznej: Zakładając, że rozwiązanie znajdziemy w ostatnim kroku, wiemy że dla $\sum_{i=0}^{Z} \binom{Z}{i} * i\,!$ zbiorów rozpatrzymy wszystkie możliwe działania. Większość z nich niestety zależy od akurat rozważanej sytuacji. Aby znaleźć ograniczenie górne, wszystkie operacje wymagające znajomości rozważanej sytuacji zastąpimy ich górnym ograniczeniem następująco:

- $liczba\,elementow\,w\,zbiorze$ - $n * (k - 1)$
- $liczba\,rozpatrzonych\,zbiorow$ - $\sum_{i=0}^{Z} \binom{Z}{i} * i\,!$
- $liczba\,mozliwych\,podziaow\,elementu$ - $k - 1$
- r - 4
- p - n * k
- Z - n * (k - 1)

Dzięki czemu uzyskujemy następujący wzór:

$$(\sum_{i=0}^{n*(k-1)} \binom{n*(k-1)}{i} * i\,!) * (C + n*k + n*k*(\sum_{i=0}^{n*(k-1)} \binom{n*(k-1)}{i} * i\,!) + (4*n*k^2)^{n*k} + n*k*n*(k-1)) \quad (5)$$

Po jeszcze większym ograniczeniu niektórych składników otrzymujemy:

$$((n*k+1)\,!)^2 * (2*k*n)^{2*k*n} \quad (6)$$

Złożoność całkowita Dolne ograniczenie możemy ograniczyć jeszcze bardziej do n^n, a górne do $(2*k*n)^{4*k*n}$, gdzie k jest z góry ustaloną wartością.
W ten sposób możemy stwierdzić, że nasz algorytm jest złożoności **wykładniczej**.

3.5 Optymalizacja

Aby potencjalnie skrócić działanie funkcji zastosowaliśmy następujące techniki:

1. Jako, że poprzez zastosowanie różnych operacji na elementach można dojść do tego samego zbioru, postanowiliśmy zapisywać rozważone zbiory i nie powtarzać dla nich obliczeń

2. W momencie znalezienia pierwszego rozwiązania zapisujemy liczbę operacji potrzebnych do wykonania, aby osiągnąć optymalne rozwiązanie i nie rozważamy zbiorów potrzebujących większej liczby operacji.

4 Algorytm heurystyczny

4.1 Opis

Algorytm heurystyczny oprzemy na algorytmie zachłannym - w danym momencie będziemy wybierać potencjalnie najlepsze rozwiązanie dla postawionego problemu/zbioru kawałków. Kroki wykonywania algorytmu:

1. Znalezienie wymiarów prostokąta, do którego należy dopasować zadane elementy
2. Na wejściu dodajemy do listy elementów wszystkie zadane elementy
3. Dopóki lista jest niepusta:
   (a) Bierzemy pierwszy element z listy i usuwamy go z listy.
   (b) Próbujemy dopasować element do prostokąta.
   (c) Jeśli nie udało się dopasować elementu to dokonujemy na nim cięć i wybieramy, to cięcie po którym jeden z otrzymanych kawałków ma pole = 1. Jeśli nie jest to możliwe bierzemy ostatnie cięcie. Otrzymane kawałki dodajemy do końca listy
4. Jeśli lista jest pusta zwróć rozwiązanie oraz liczbę potrzebnych cięć.

4.2 Pseudokod

Funkcja główna:
begin
stwórz pustą listę l;
/*do tej listy będziemy dodawać elementy do ułożenia*/
użyj funkcji *dopasuj_prostokąt*, aby znaleźć *wysokość* i *szerokość* prostokąta;
/* prostokąta do którego będziemy dopasowywać elementy */
liczba_cięć <- 0;
for(*element* in zbiór elementów wejściowych)
dodaj na koniec listy l element *element*;
/* dodawanie do listy kolejnych elementów ze zbioru wejściowego */
end
stwórz *nowy_układ* o wysokości *wysokość* i szerokości *szerokość*;
/* układ w którym będziemy umieszczać elementy */
$nowy\_ukad_{ij}$ <- (-1) for i=1,2...*wysokość* for j=1,2...*szerokość*;
while(l nie jest puste)
elem <- element na początku l;
usuń element z początku listy l;
trzeba_ciąć <- *prawda*;
/*ustawiamy czy trzeba dokonać cięcia na elemencie*/
for(i<- 1,2,3...*wysokość*) /*wysokość prostokąta */
for(j <- 1,2,3...*szerokość*) /*szerokość prostokąta*/
if($ukad_{ij}$ == **-1**) /* to znaczy, że jest pusty*/
ułożenia <- *rotacje*($elementy_{nr}$);
/* funkcja rotacje zwracają wszystkie rotacje elementu - wykonanie w czasie stałym */

```
                for(ułożenie in ułożenia)
                    (nowy_uklad, dodano) <- dopasuj(ułożenie, nowy_uklad, i, j, bieżący_element.id));
                    /* funkcja dopasuj zwraca układ z dopasowanym elementem bądź nie oraz informację
                        czy element został dodany */
                    if(dodano jest prawdą)
                        trzeba_ciac <- fałsz;
                        break; /*Jeśli element został dodany pomiń resztę przypadków i wyjdź z pętli for*/
                    end
                end
            end
        end
        if(trzeba_ciac jest prawdą) /*element nie został dodany i trzeba go podzielić */
            a <- pusty element;
            b <- pusty element;
            for((elem1,elem2) in potnij(elem))
                /* potnij zwraca listę wszystkich możliwych pocięc elementu na dwa kawałki */
                a<-elem1;
                b<-elem2;
                if(elem1.pole==1 lub elem2.pole==1)
                    /*Sprawdza czy któryś element jest pojedynczym kwadratem */
                    break; /* Jeśli tak to przerwij sprawdzanie kolejnych przypadków */
                end
            end
            dodaj na koniec listy l elementy a i b
            liczba_cięć += 1; /* Zwiększ liczbę cięć o 1 */
        end
        wypisz liczba_cięć;
        wypisz nowy_układ;
    end
end
```

Funkcje pomocnicze:
dopasuj(konkretne ułożenie elementu *ułożenie*, układ do którego dopasowujemy element *układ*, wysokość planszy *wysokość*, szerokość planszy *szerokość*, id elementu *id*)
begin
for(*pole* in *ułożenie*)
jest_ok <- *prawda*;
nowy_układ <- sklonuj *układ*;
pole_wysokość <- współrzędna odpowiadająca ułożenia w pionie pola *pole*;
pole_szerokość <- współrzędna odpowiadająca ułożenia w poziomie pola *pole*;
for(*sąsiednie_pole* in *ułożenie*)
sąsiednie_pole_wysokość <- współrzędna odpowiadająca ułożenia w pionie pola *sąsiednie_pole*;
sąsiednie_pole_szerokość <- współrzędna odpowiadająca ułożenia w poziomie pola *sąsiednie_pole*;
w <- *sąsiednie_pole_wysokość* - *pole_wysokość* + *wysokość*;
s <- *sąsiednie_pole_szerokość* - *pole_szerokość* + *szerokość*;
if($nowy\_ukad_{ws}$ == **-1**)
/* jeśli pole jest puste */
$nowy\_ukad_{ws}$ <- *id*;
else
jest_ok <- *fałsz*;
break; /* Przerwanie pętli */
end
end
end
zwróć *nowy_układ* oraz *jest_ok*; /* Zwraca układ i informację czy element został dodany */
end

4.3 Złożoność

Oznaczenia:

- n - liczba elementów wejściowych
- k - liczba pól elementu

Złożoność naszego algorytmu heurystycznego ma być co najwyżej wielomianowa. Wszystkie pętle w głównej pętli while mają złożoności wielomianowe, więc aby algorytm miał złożoność wielomianową to pętla while musi być co najwyżej wielomianowa.
Wyjaśnienie:
Na wejściu otrzymujemy **n** elementów na liście. Każdy element staramy się dopasować do prostokąta. Jeśli się nie uda to każda operacja dodaje nam dodatkowy element na liście. Maksymalnie możemy uzyskać $k * n$ elementów w liście. Do uzyskania takiej liczby kawałków, pętla musi się wykonać $(k-1) * n$ razy i na koniec jeszcze $k * n$, aby dopasować kawałki do prostokąta. Stąd otrzymujemy, że główna pętla wykona się **maksymalnie** $(k + k - 1) * n$ razy.
Dzięki temu możemy stwierdzić, że nasz algorytm ma złożoność wielomianową.

5 Zmiany w implementacji

Zmiany, które nastąpiły względem powyższych implementacji pseudokodów zostały wyszczególnione pomiędzy znakami "+++++". Wartości te oznaczają dodanie kodu w odpowiadającym im miejscu. Wprowadzone zmiany nie wpływają na złożoność algorytmów.

1. Alg. optymalny Funkcja Bactrack:
Dodane zostało sprawdzenie czy wypełniona tablica jest rozwiązaniem oraz czy lista rozwiązań zawiera już to rozwiązanie.

backtrack(zbiór elementów *elementy*, numer rozpatrywanego elementu *nr*, układ do którego dopasowujemy elementy *układ*)
begin
stwórz pustą listę *rozwiązania*; /* do tej listy będziemy dodawać rozwiązania */
id <- id elementu $elementy_{nr}$;
wysokość <- wysokość układu *układ*;
szerokość <- szerokość układu *układ*;
if(*nr* jest mniejszy niż liczba elementów w *elementy*)
for(i <- 1,2...*wysokość*)
for(j <- 1,2...*szerokość*)
if($ukad_{ij}$ == **-1**) /* to znaczy, że jest pusty*/
ułożenia <- *rotacje*($elementy_{nr}$);
/* funkcja rotacje zwracają wszystkie rotacje elementu - wykonanie w czasie stałym */
for(*ułożenie* in *ułożenia*)
układy_z_elementem <- *dopasuj*(*ułożenie*, *układ*, *i*, *j*, *id*));
/* funkcja *dopasuj* zwraca listę wszystkich układów możliwych
po włożeniu zadanego elementu do zadanego układu */
for(*nowy_układ* in *układy_z_elementem*)
nowe_rozwiązania <- *backtrack*(*elementy*, $nr + 1$, *nowy_układ*));
/* rekurencyjne wywołanie funkcji dla kolejnego elementu */
+++++
for(*rozw* in *nowe_rozwiązania*)
if(*rozwiązania* nie zawierają *rozw*)
/*Sprawdzamy czy rozwiązanie zostało już dodane*/
dołącz *nowe_rozwiązania* do *rozwiązania*;
+++++
end
end
end
end
end
end
+++++
else

```
        for(i <- 1,2...wysokość)
            for(j <- 1,2...szerokość)
                if(ukad_ij == -1) /*Sprawdzenie czy któreś z miejsc nie zostało uzupełnione*/
                    zwróć rozwiązania;
            end
        end
        dołącz nowe_rozwiązania do rozwiązania;
    end
    +++++
    zwróć rozwiązania;
end
```

2. Alg. optymalny Funkcja główna: Przy dodawaniu rozwiązań do listy rozwiązań sprawdzane jest czy rozwiązanie jest już na liście.

```
...
if(rozwiązania_elementów nie jest puste)
    +++++
    for(rozw in rozwiązania_elementów)
        if(rozwiązania nie zawierają rozw)
        /*Sprawdzamy czy rozwiązanie zostało już dodane*/
            dołącz rozw do rozwiązania;
    +++++
    min <- elementy_min;
else
...
```

6 Testy

Aby sprawdzić poprawność i czasy wykonania napisanych algorytmów wykonaliśmy szereg testów na różnej liczbie kawałków.
Dla n (liczba kawałków) losowo wybranych kawałków wykonaliśmy obliczenie algorytmem optymalnym oraz heurystycznym. Najpierw kawałki były o polu 5 następnie losowaliśmy kawałki o polu 6.

6.1 Parametry komputera

Testy zostały wykonane na komputerze Asus UX430U
System operacyjny: Windows 10 Home
Procesor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz - 1.80GHz
RAM: 7,86 GB
Dysk: 256 GB SSD

6.2 Wyniki

Wartości jakie porównywaliśmy to czas oraz liczba potrzebnych cięć.
Czas został zapisany w formacie h:mm:ss

6.2.1 5-elementowe kawałki:

Lp.	Liczba kawałków	Cięcia heurystyczn-nych	Czas heurystyczny	Cięcia optymalny	Czas Optymalny
1.	1	2	000:0000:00.0053	1	00:00:00.0058
2.	1	3	00:00:00.0051	3	00:00:00.0090
3.	2	4	00:00:00.0049	1	00:00:00.0288
4.	2	2	00:00:00.0078	2	00:00:00.3865
5.	2	4	00:00:00.0054	2	00:00:00.4877
6.	3	1	00:00:00.0058	1	00:00:00.8823
7.	3	3	00:00:00.0113	1	00:00:00.7674
8.	3	3	00:00:00.0053	1	00:00:00.8675
9.	4	3	00:00:00.0072	1	00:00:34.0204
10.	4	5	00:00:00.0057	2	0:01:14.6344
11.	4	4	00:00:00.0051	1	00:00:47.6956
12.	5	4	00:00:00.0056	1	0:38:48.1316
13.	5	3	00:00:00.0053	2	0:52:42.7670
14.	6	3	00:00:00.0042	1	07:09:47.7226
15.	10	8	00:00:00.0129	-	-
16.	15	9	00:00:00.0144	-	-
17.	20	8	00:00:00.0193	-	-
18.	25	10	00:00:00.0187	-	-
19.	30	13	00:00:00.0651	-	-
20.	35	15	00:00:00.1158	-	-
21.	40	19	00:00:00.1685	-	-
22.	45	16	00:00:00.2116	-	-
23.	50	23	00:00:00.2928	-	-
24.	100	40	00:00:2.3459	-	-
25	500	3	00:03:29.0977	-	-

6.2.2 6-elementowe kawałki

Lp.	Liczba kawałków	Cięcia heurystyczn-nych	Czas heurystyczny	Cięcia optymalny	Czas Optymalny
1.	1	1	00:00:00.0078	1	00:00:0077
2.	1	3	00:00:00.0008	1	00:00:00.0065
3.	2	4	00:00:00.0023	1	00:00:00.0233
4.	2	5	00:00:00.0023	2	00:00:05.2636
5.	3	3	00:00:00.0011	2	00:02:57.3776
6.	3	4	00:00:00.0007	1	00:00:01.1217
7.	4	3	00:00:00.0061	2	01:46:44.8529
8.	4	5	00:00:00.0019	2	02:10:47.6128
9.	10	9	00:00:00.0124	-	-
10.	15	15	00:00:00.0619	-	-
11.	20	17	00:00:00.0764	-	-
12.	25	16	00:00:00.1206	-	-
13.	30	19	00:00:00.1417	-	-
14.	35	27	00:00:00.3582	-	-
15.	40	23	00:00:00.5418	-	-
16.	50	35	00:00:00.6543	-	-
17.	75	49	00:00:01.7779	-	-
18	100	63	00:00:04.4164	-	-
19	250	144	00:00:59.8414	-	-
20	500	278	00:07:19.0671	-	-

6.3 Analiza

Otrzymane wyniki są satysfakcjonujące. Wartości, które chcieliśmy zbadać, uzyskaliśmy we wszystkich testach. Możemy zauważyć, że algorytm heurystyczny dla wszystkich przypadków wykonuje się szybciej niż algorytm optymalny. Jednak dla więcej niż 4 5-elementowych oraz x 6-elementowych kawałków czas wykonania się obliczeń w algorytmie optymalnym znacząco wzrasta. Dlatego też dla większej liczby zaprzestaliśmy wykonywania obliczeń.
W tych samych przypadkach w algorytmie optymalnym zauważamy, że liczba potrzebnych cięć nie przekracza liczby 3, a heurystycznym 5. Największa różnica w liczbie cięć jaką zanotowaliśmy to 3.
Algorytm heurystyczny nie zawsze znajduje optymalne rozwiązanie. Możemy stwierdzić, że jego wyniki nie odbiegają znacząco od rozwiązania optymalnego. Dopiero dla 500 kawałków czas algorytmu heurystycznego wynosi ponad 3 min (5-elementowe) oraz ponad 7 min (6-elementowe).

6.4 Wnioski

Po analizie możemy śmiało stwierdzić, że dla małej liczby elementów (1-4) lepiej wykorzystać algorytm optymalny, chyba że na kawałkach można wykonać dużej liczby cięć lub kawałki się powtarzają - wtedy czas wykonania algorytmu optymalnego znacząco wzrasta. Dla większej liczby elementów lepszym rozwiązaniem jest algorytm heurystyczny, który w krótkim czasie daje nam wynik zbliżony do optymalnego.

7 Instrukcja obsługi

Program działa w systemie Windows. Plik binarny **taio.exe** należy umieścić w dowolnym katalogu i w tym samym katalogu należy stworzyć katalog o nazwie **tests**. W tym katalogu można umieścić dowolną liczbę plików tekstowych, i tylko tekstowych, o dowolnej nazwie spełniających poniższy schemat.

- W pierwszej linijce wielkość dla jakich klocków algorytm działa (5 lub 6)
- W drugiej linijce jedna z następujących wartości: 'op', 'hp' (małymi literami) oznaczające jaki algorytm powinien się uruchomić, kolejno:
  - op - problem prostokąta, algorytm optymalny
  - hp - problem prostokąta, algorytm heurystyczny
- W trzeciej linijce może znajdować się jedna liczba, lub lista liczb oddzielonych spacjami:
  - jeśli jest jedna liczba to informuje nas ona dla ilu losowo wybranych klocków powinniśmy uruchomić nasz algorytm,
  - jeśli jest więcej niż jedna liczba to jest to lista ilości kolejnych klocków dla których powinniśmy uruchomić nasz algorytm (ignorujemy zera końcowe) np. 2 5 1 1 0 0 5 3 oznacza, że powinniśmy wziąć 2 klocki o numerze 1, 5 klocków o numerze 2, 1 klocek o numerze 3 itd.

Jeden plik może zawierać klika wyżej przestawionych sekwencji, wtedy powinniśmy uruchomić algorytmy dla kolejnych sekwencji.
Program jest uruchamiany podwójnym kliknięciem (tak jak większość programów w systemie Windows)
Po uruchomieniu pokaże się terminal w którym na bieżąco będę wypisywane wyniki dla kolejnych zadanych problemów w następujący sposób:

- Zostaną wypisane elementy z ich rozmiarami, id nadanymi przez program oraz krótki subiektywny opis elementu.
- Dla obliczeń algorytmu dokładnego, pokaże się sekcja w której będzie wypisywany progres algorytmu w postaci informacji o liczbie operacji potrzebnych do uzyskania obecnie przetwarzanych zbiorów
- Wypisany zostanie wynik w postaci: liczby potrzebnych operacji, czasu wykonania algorytmu oraz planszy reprezentującej rozwiązanie problemu. Odpowiednie pola będą zaznaczone odpowiednimi id elementów.
- Dla algorytmu dokładnego zostanie wypisana liczba pozostałych optymalnych rozwiązań z pytaniem czy użytkownik chce je wyświetlić. Po kliknięciu przycisku 'y' na klawiaturze zostaną wypisane pozostałe wyniki.