Analiza algorytmów – zadanie projektowe AAL-11-LS *kartony*

Semestr 2018Z

Omówienie rozwiązania problemu

Michał Belniak

Prowadzący: dr Łukasz Skonieczny

1. Treść zadania.

   Ortodyksyjny kolekcjoner tekturowych kartonów zaczyna narzekać na brak miejsca do przechowywania swoich cennych zdobyczy. Postanowił oszczędzić miejsce przez wkładanie kartonów jeden w drugi. W trosce o zachowanie dobrego stanu kartonów, umieszcza tylko jeden karton wewnątrz większego, a wolną przestrzeń wypełnia materiałem ochronnym. Tak zabezpieczony karton może następnie umieścić wewnątrz innego większego kartonu, ale nie może umieścić dwóch kartonów obok siebie w jednym kartonie. Dla danego zbioru kartonów należy znaleźć najlepsze upakowanie kartonów, tzn. takie, które zwalnia najwięcej miejsca.

2. Omówienie zadania wraz z założeniami.

   Zadanie polega na znalezieniu takiego upakowania kartonów, które zwalnia jak najwięcej objętości, co jest równoznaczne minimalizowaniu sumy objętości kartonów niespakowanych, czyli takich, których już nie można nigdzie spakować. Przez spakowanie będziemy rozumieli umieszczenie jednego kartonu w drugi.

   Głównym założeniem w tym zadaniu będzie przyjęcie, że każdy karton jest prostopadłościanem. Ułatwi to określenie jego objętości oraz tego, czy dany karton zmieści się w innym danym kartonie.
   Kolejnym założeniem jest to, że wkładając karton w drugi karton, odpowiadające ściany będą do siebie równoległe. Nie uwzględniamy więc przypadku, gdzie wkładamy karton pod skosem. Ułatwi to sprawdzanie jakie kartony mieszczą się w danym, większym kartonie.

   Oznaczmy zbiór kartonów jako:

   $$K=\{k_1, k_2, k_3, \ldots, k_n\},$$

   gdzie k_i jest trójwymiarowym wektorem postaci (x_i, y_i, z_i), opisującym wymiary i-tego kartonu. Przyjmujemy także, że $x_i \geq y_i \geq z_i$ – ułatwi to sprawdzanie, czy karton mieści się w innym kartonie. Z każdym elementem k_i powiązana jest wartość V_i oznaczająca jego objętość, co pozwala utworzyć zbiór

   $$Vk=\{V_1, V_2, V_3, \ldots, V_n\}, \text{ gdzie } V_i=x_i*y_i*z_i.$$

   Przy tak zdefiniowanych zbiorach, zadanie polega na maksymalizacji funkcji celu:

   $$max\ f_z=V_b,$$

gdzie $V_b=\sum_{i=1}^{n} V_i \cdot c_i,\ c_i = \begin{cases} 1, & \text{jeśli } k_i \text{ spakowany w inny karton} \\ 0, & \text{jeśli } k_i \text{ niespakowany} \end{cases}$ – suma objętości kartonów spakowanych w jakiś inny karton. Innymi słowy, chcemy aby objętość kartonów niespakowanych w żaden inny była jak najmniejsza.

Niech możliwość spakowania będzie zdefiniowana przez wektor (k_i, k_j), $i \neq j$, który oznacza, że karton k_j może być spakowany w karton k_i. Warto zauważyć, że jest to relacja przechodnia, tzn.:

$$(k_i,k_j) \wedge (k_m,k_i) \Rightarrow (k_m,k_j),\ i \neq j \neq m$$

Karton k_j może być spakowany w karton k_i wtedy i tylko wtedy, gdy:

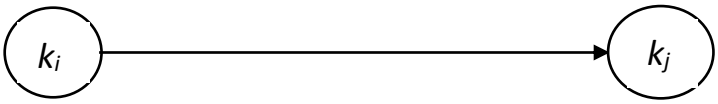
$$x_i > x_j \wedge y_i > y_j \wedge z_i > z_j \qquad (1.1)$$

Wszystkie wektory możliwych zapakowań tworzą zbiór W. Poprawnym rozwiązaniem zadania jest wyznaczony podzbiór *W'* zbioru *W*, w którym dla każdego elementu (k_i, k_j) spełnione są warunki:

1. Jeśli w podzbiorze *W'* istnieje (k_i, k_m), $m \neq j$ to musi istnieć także (k_j, k_m), lub (k_m, k_j). Wynika to z ograniczenia, że nie można umieszczać dwóch kartonów obok siebie.
2. Jeśli w podzbiorze *W'* istnieje (k_k, k_j), $k \neq i$ to musi istnieć także (k_i, k_k), lub (k_k, k_i). Oznacza to, że karton może być spakowany w dwa różne kartony, tylko wtedy, gdy te dwa kartony są wzajemnie spakowane.

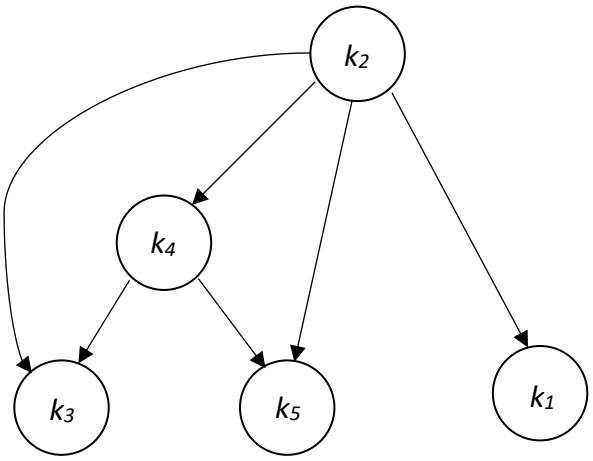
3. Tworzenie odpowiedniej struktury danych.

Wektory oznaczające relację spakowania można reprezentować graficznie. Zbiór możliwych zapakowań można więc reprezentować za pomocą grafu, którego wierzchołkami będą kartony k_i, a krawędziami – wektory (k_i,k_j) ze zbioru W. Pozwoli to uzyskać graf reprezentujący wszystkie możliwe relacje spakowania. Z warunku *(1.1)* wynika, że graf będzie skierowany oraz acykliczny. Poniższy rysunek oznacza, że karton k_j może zostać spakowany w karton k_i.



Dla kilku kartonów przykładowy graf może wyglądać jak na rysunku 1.1.

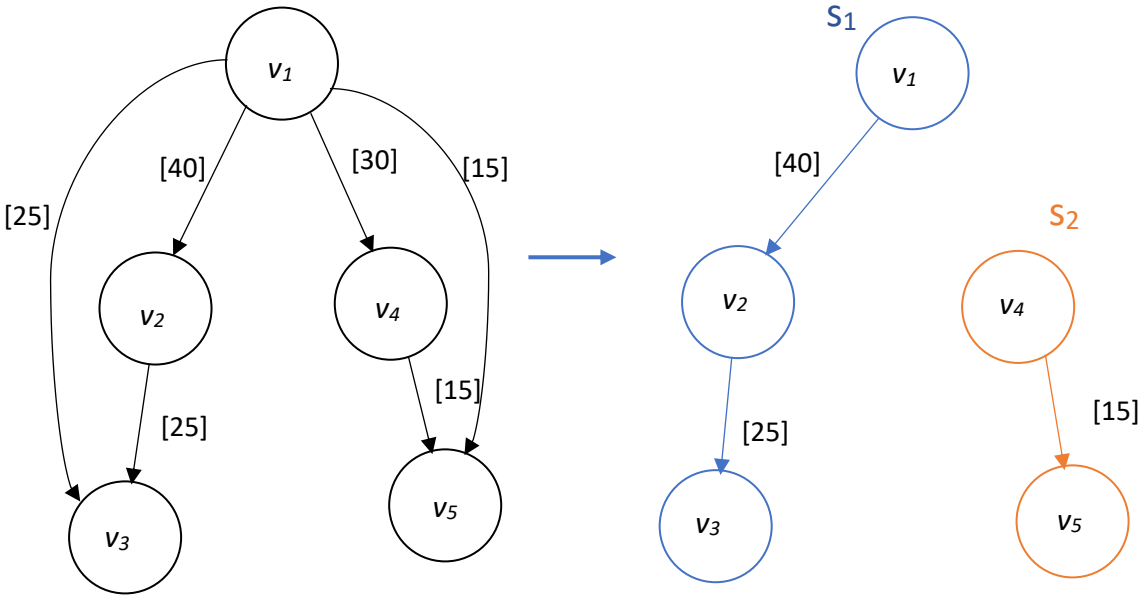
Wyznaczenie podzbioru *W'*, czyli poprawnego rozwiązania polega na wyborze odpowiednich krawędzi grafu, jednak aby znaleźć rozwiązanie optymalne, trzeba najpierw powiązać krawędzie z wartościami objętości kartonów pakowanych. Niech wagą krawędzi (k_i, k_j) będzie objętość kartonu k_j, a więc V_j. Wtedy wybranie krawędzi (k_i, k_j) do podzbioru *W'* będzie oznaczać zaoszczędzenie przestrzeni o objętości V_j, a właśnie sumę zaoszczędzonej objętości należy maksymalizować, aby znaleźć rozwiązanie optymalne.



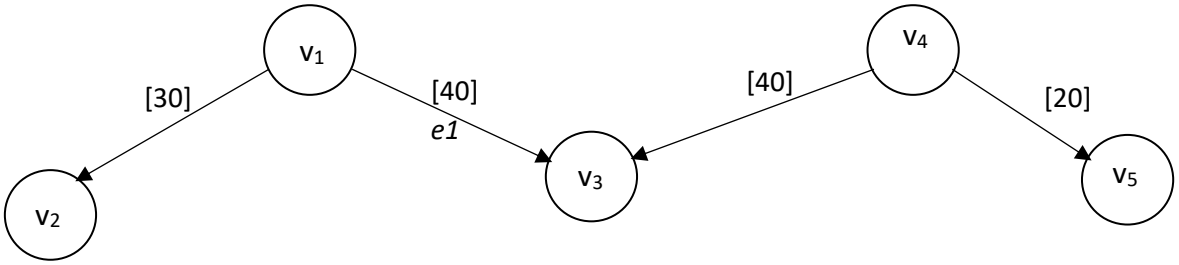
Rysunek 1.1

Analizując postawione warunki można dojść do wniosku, że znalezienie rozwiązania optymalnego polega na wyznaczeniu takich ścieżek, które są wierzchołkami rozłączne oraz razem obejmują wszystkie wierzchołki grafu, a suma wag krawędzi należących do ścieżek jest największa. Co więcej, dzięki temu, że znajdujemy ścieżki w grafie, które są wierzchołami rozłączne, na pewno spełnione zostaną warunki poprawnego rozwiązania z poprzedniej strony.

(V, E) – rozważany graf.
S_o – zbiór ścieżek wyznaczających optymalne rozwiązanie.
$s_i = (v_{i0}, v_{i1}, v_{i2}, \ldots, v_{im})$, $s_i \in S_o$ - i-ta ścieżka.
$s_1 \cap s_2 \cap \ldots \cap s_n = \emptyset$ oraz $s_1 \cup s_2 \cup \ldots \cup s_n = S$



Przeanalizujmy możliwe wybory z perspektywy wierzchołka, z którego wychodzi kilka krawędzi (np. v_1 na rysunku poniżej). Intuicja podpowiada, aby wybrać krawędź z największą wagą – zaoszczędzimy w ten sposób najwięcej objętości. Oznaczmy tę krawędź jako e_1. Trzeba jednak spojrzeć na ten wybór z perspektywy wierzchołka, do którego wchodzi ta krawędź. Nazwijmy ten wierzchołek v_3. Być może do wierzchołka v_3 wchodzi kilka krawędzi. Wtedy wybierając krawędź e_1, uniemożliwiamy wybranie jakiejkolwiek innej krawędzi wchodzącej do v_3, zmuszając w ten sposób wybór alternatywnej ścieżki dla wierzchołków połączonych z v_3. W złośliwym przypadku doprowadzi to do nieprawidłowego rozwiązania. Dlatego należy wprowadzić dodatkowy warunek w przypadku, gdy rozważana krawędź prowadzi do wierzchołka, do którego wchodzi wiele krawędzi. Z tego powodu, trudnym przypadkiem będzie na przykład sytuacja, gdy mamy 10 kartonów o tych samych wymiarach o raz 10 innych kartonów o tych samych wymiarach, ale innych niż pierwsza 10. Wtedy tworzy się graf dwudzielny pełny i algorytm musi przejrzeć relatywnie dużą ilość wierzchołków.



W powyższym przypadku, wybór krawędzi (v_1, v_3) wymusi wybranie krawędzi (v_4, v_5), co prowadziłoby do rozwiązania nieoptymalnego. Wybrana powinna zostać krawędź (v_1, v_2).

4. Opis algorytmu.

Najpierw należy ustalić kolejność, w jakiej będziemy analizowali wierzchołki i wychodzące z nich krawędzie. Sortując je malejąco po prostu ze względu na ich objętość, otrzymamy listę wierzchołków, którą na pewno rozpoczyna wierzchołek bez wchodzących do niego krawędzi (rodziców). Dla każdego wierzchołka z listy (po kolei) wykonujemy następujące kroki. Tworzymy nową, pustą ścieżkę *s*. Zapisujemy wierzchołek w ścieżce *s* i usuwamy go listy wierzchołków . Następnie znajdujemy najbardziej kosztowną krawędź wychodzącą z tego wierzchołka (krawędź prowadzi do dziecka). Jeśli wierzchołek nie ma żadnego dziecka, to znaczy że jest końcem ścieżki – rozpoczynamy algorytm dla kolejnego wierzchołka z listy.
(a)Dla znalezionego dziecka sprawdzamy, czy ma on więcej niż jednego rodzica. Jeśli nie, dodajemy dziecko do ścieżki *s*, usuwamy je z listy wierzchołków i powtarzamy algorytm dla dziecka. Jeśli tak, to dla każdego rodzica dziecka (nazwijmy v_k) innego niż analizowany obecnie wierzchołek sprawdzamy, jaka jest maksymalna waga wśród dzieci wierzchołka v_k (poza analizowanym dzieckiem*)* i spośród tych wartości zapamiętujemy najmniejszą. Jeśli ta zapamiętana wartość jest mniejsza niż maksymalna wartość pozostałych dzieci analizowanego wierzchołka, usuwamy najbardziej kosztowną krawędź wychodzącą z analizowanego wierzchołka i znajdujemy kolejną największą krawędź, dla której powtarzamy (a). Jeśli zaś ta

wartość jest większa lub równa, dodajemy dziecko do ścieżki *s*, usuwamy je z listy wierzchołków i powtarzamy algorytm dla dziecka.
Kończymy algorytm, gdy lista wierzchołków stanie się pusta.

Wszelkie nieścisłości i wątpliwości powinien rozwiać bardziej zwięzły opis:

0. Stwórz listę wierzchołków grafu posortowaną malejąco ze względu na objętość oraz stwórz zbiór znalezionych ścieżek.
1. Dla każdego wierzchołka *v* z listy:
    a. Stwórz nową ścieżkę s.
    b. Wykonaj:
        i. Dodaj *v* do ścieżki *s*.
        ii. Znajdź wierzchołek v_k, do którego prowadzi najbardziej kosztowna krawędź z *v*. Jeśli wierzchołek *v* nie ma dzieci, usuń *v* z listy wierzchołków oraz *v* i krawędzie z nim związane z grafu. Powróć z rekurencji lub zakończ punkt b.
        iii. Jeśli v_k ma tylko jednego rodzica, usuń *v* z listy wierzchołków oraz z grafu wraz z krawędziami z nim związanymi i wykonaj pkt. b. dla v_k (rekurencja).
        iv. Jeśli v_k ma więcej niż 1 rodzica, dla każdego jego rodzica v_r znajdź wartość najdroższej krawędzi z v_r nieprowadzącej do v_k (0 jeśli nie ma). Spośród znalezionych wartości wybierz najmniejszą = x_{min}. Znajdź najdroższą krawędź wychodzącą z *v* nieprowadzącą do v_k i zapamiętaj jako x_{alt} (0 jeśli nie ma). Jeśli $x_{alt} > x_{min,}$ usuń krawędź (v, v_k) z grafu i powtórz pkt. ii. W przeciwnym wypadku usuń *v* z listy wierzchołków oraz *v* i krawędzie z nim związane z grafu. Wykonaj pkt. i. dla v_k.
    c. Dodaj ścieżkę *s* do zbioru znalezionych ścieżek.
2. Jeśli lista wierzchołków jest pusta, zakończ algorytm. Zbiór ścieżek wyznacza zbiór *W'*, który jest rozwiązaniem zadania.

Jeśli poddamy algorytm analizie złożoności, może się wydawać, że działa on ze złożonością obliczeniową O(n^4) o małym współczynniku. Jednak zastosowanie presortowania powoduje, że zwracanie najlepszego kartonu danego rodzica działa ze złożonością stałą. To daje nam złożoność O(n^3): musimy uwzględnić każdy wierzchołek co daje nam n; dla każdego wierzchołka musimy w pesymistycznym przypadku przejrzeć n/2 wierzchołków co daje n^2; dla każdego dziecka musimy sprawdzić alternatywy każdego rodzica co może wynieść nawet n-2.

5. Implementacja i dokumentacja kodu źródłowego.

Algorytm został zaimplementowany w języku C++. Na projekt składają się 4 pliki źródłowe: *main.cpp, Graph.h, MPacking.h, EfficiencyTest.h.* Jako pierwszy omówię Graph.h, ponieważ jest najprostszy.

1. Graph.h

W tym pliku znajduje się definicja klasy Graph, która jest implementacją struktury danych – grafu skierowanego ważonego, jednak takiego, który jest przystosowany do naszego problemu. Każdy wierzchołek ma określoną objętość, co opowiada wadze krawędzi kończącej się w tym wierzchołku. Warto zwrócić uwagę na to, że krawędzie są zaimplementowane jako wskaźniki na inne wierzchołki i co więcej, przechowywane są wskaźniki zarówno na wierzchołki-dzieci, jak i na wierzchołki-rodziców. Pomaga to w rozwiązaniu problemu.

Omówienie pól i metod:

a) *struct Vertice – struktura realizująca wierzchołek. Składają się na nie następujące pola:*
   a. *unsigned number* – numer wierzchołka w grafie, unikalny dla każdego wierzchołka. Pozwala identyfikować karton.
   b. *float width, length, height, volume* – zmienne określające wymiary i objętość kartonu.
   c. *std::vector<Vertice*> children, parents* – wektor wskaźników na wierzchołki-dzieci (kartony, które można zmieścić) oraz na wierzchołki-rodziców (kartony, do których można zmieścić).
   d. *bool isDeleted* - flaga oznaczająca, czy wierzchołek został usunięty z grafu, bez usuwania go z wektora wierzchołków. Zastosowano to rozwiązanie, aby zapobiec zmianie pozycji w wektorze innych wierzchołków w przypadku usunięcia danego wierzchołka z wektora.

b) *std::vector<Vertice> vertices* – wektor przechowujący wszystkie wierzchołki grafu.

c) *float v_total* – całkowita objętość kartonów opisanych tym grafem.

d) *int findRoot()* – zwraca pozycję w wektorze pierwszego wierzchołka, który nie ma rodziców. Graf jest z założenia acykliczny, więc musi istnieć taki wierzchołek.

e) *static bool compare(Vertice first, Vertice second)* – metoda porównująca dwa wierzchołki, potrzebna dla funkcji std::sort(). Wierzchołek większy to taki, który ma większą objętość.

f) *void addVertice(float dims[3])* – dodaje do grafu wierzchołek-karton o wymiarach podanych w *dims[3]*.

g) *void findParentsAndChildren()* – wyszukuje wszystkie relacje rodzic-dziecko w grafie, a więc krawędzie. Dla każdego wierzchołka w wektorze *children* są zapisywane wskaźniki do wierzchołków, które mają takie wymiary, że

mieszczą się w danym wierzchołku(kartonie). Analogicznie w wektorze *parents* – kartony, do których się mieści dany karton.

2. MPacking.h

Ten plik zawiera definicję klasy *MPacking*, która realizuje rozwiązanie problemu. Nazwa to skrócona forma *Matrioshka Packing*, ponieważ forma problemu przypomina pakowanie na zasadzie matrioszki. Klasa *MPacking* dziedziczy po klasie *Graph* i rozszerza ją o funkcjonalność pozwalającą znaleźć rozwiązanie problemu i wyświetlić je w czytelnej formie.

Omówienie pól i metod:

a) *float v_gained* – całkowita objętość zaoszczędzona dzięki pakowaniu.
b) *std::vector<std::vector<int>> paths* – wektor znalezionych do tej pory ścieżek. Liczby w wektorze odpowiadają numerom wierzchołków.
c) *void sortDims(float dims[3])* – metoda sortująca wymiary pudełka, aby width>=lenght>=height.
d) *void rebuild()* – usuwa i wyszukuje ponownie rodziców i dzieci każdego wierzchołka.
e) *float getMaxAlternative(Vertice* parent_analyzed, Vertice* child)* – zwraca objętość największego kartonu, jaki może się zmieścić w *parent_analyzed* poza *child*.
f) *int findChildIndex(Vertice* child)* – zwraca pozycję wierzchołka *child* w wektorze *vertices* grafu.
g) *void removeChild(Vertice* parent, int child_number)* – usuwa z wektora *children* wskaźnik na dziecko o numerze *child_number* w *vertices*.
h) *void continuePath(std::vector<int>* path, int rootIndex)* – dodaje do ścieżki *path* wierzchołek *rootIndex* i wyszukuje następnego wierzchołka, który będzie składał się na optymalne rozwiązanie problemu. Jeśli nie znajdzie takiego wierzchołka, funkcja się kończy, a jeśli znajdzie, wywołuje sama siebie dla *rootIndex*=znaleziony wierzchołek.
i) *MPacking()* – domyślny konstruktor. Tworzy graf na podstawie standardowego wejścia, w którym podaje się kolejne wymiary kolejnych pudełek.
j) *MPacking(std::string file_name*) – konstruktor, który tworzy graf na podstawie wymiarów podanych w pliku.
k) *MPacking(int complexity, bool hardCases)* – konstruktor, który tworzy losowy graf o *complexity* wierzchołkach. Jeśli *hardCases*=true, co jakiś czas generuje w grafie takie wierzchołki, które są trudne do rozwiązania dla algorytmu.
l) *void findBestPaths()* – uruchamia algorytm szukający optymalnego rozwiązania.
m) *void writePaths()* – wypisuje znalezione rozwiązanie na standardowe wyjście.
n) *void regenerateGraph()* – przywraca graf do stanu sprzed wykonania algorytmu.
o) *void writeVertices()* – wypisuje informacje o każdym wierzchołku.
p) *bool isCreated()* – zwraca true, jeśli w grafie jest przynajmniej 1 wierzchołek.

3. EfficiencyTest.h

   Ten plik zawiera definicję klasy *EfficiencyTest*, która korzystając z klasy *MPacking* tworzy jeden lub kilka obiektów tej klasy o odpowiedniej wielkości i dokonuje pomiarów czasu wykonania algorytmu szukającego rozwiązania. Korzysta z zegara *std::chrono::high_resolution_clock.* Wypisuje rozwiązanie na standardowe wyjście.

   Omówienie metod:

   a) *EfficiencyTest(int portions, int complexities[], int c_count)* – wykonuje pomiarów dla *portions* problemów dla każdej złożoności z tablicy *complexities. c_count* jest rozmiarem tablicy *complexities.*
   b) *EfficiencyTest(int complexities[], int c_count, bool hardCases)* – wykonuje pojedynczych pomiarów dla każdej wielkości problemu z tablicy *complexities*. Jeśli *hardCases* jest true, problemy będą zawierać trudne przypadki.

4. Main.cpp

   W tym pliku dokonuje się obsługi wywołania programu i uruchamia żądane funkcje programu. Program pozwala na uruchomienie go w 5 trybach:

   ./kartony -p wywołanie programu, jeśli chcemy podawać wymiary kolejnych pudełek z klawiatury lub innego, standardowego wejścia.
   Dobre rozwiązanie dla sprawdzania poprawności działania algorytmu.

   ./kartony -p <file_name.txt> wywołanie programu z wczytaniem danych z pliku tekstowego. Plik powinien zawierać tylko i wyłącznie wymiary kolejnych kartonów rozdzielone enterem, z podwójnym enterem między kolejnymi kartonami.

   ./kartony -c <complexity> wywołanie programu z losowym problemem o wielkości <complexity> kartonów.

   ./kartony -t <portions_per_step> <step1> [step2] [step3] ... wywołanie programu, który dla każdej wielkości problemu <step1>, <step2>... wykonuje <portions_per_step> losowych problemów i podaje czasy dla każdego wykonania algorytmu.

   ./kartony -t -g [-n] <step1> [step2] [step3] ... wywołanie programu, który dla każdej wielkości problemu <step1>, <step2>... wykonuje losowy problem i tworzy tabele ze zgodnościami czasu wykonania z oszacowaniem teoretycznym problemu. Dodanie flagi -n powoduje, że wymiary będą całkowicie losowe, bez generowania trudnych przypadków.

5. Konwencja związana z danymi wejściowymi i wyjściowymi.

Dane wejściowe to trójki liczb zmiennoprzecinkowych oznaczających wymiary pudełek, np.:

123
152.2
124.5

12.2
14.42
12.75

oznacza dwa pudełka o podanych wymiarach.

Dane wyjściowe są postaci:

5<-1<-3

2

4

co oznacza, że pudełko nr 3 powinniśmy umieścić w pudełku nr 1, a to zaś w pudełku nr 5. Pudełka 2 i 4 powinny pozostać niezapakowane. Dodatkowo program podaje całkowitą objętość kartonów oraz objętość zaoszczędzoną dzięki optymalnemu zapakowaniu.

Rozmiar problemu, z którym dobrze sobie radzi algorytm to 1-1200 (czas rozwiązania poniżej 5 sekund na laptopie Dell Inspiron 7570 Intel Core i7-8550U (4 rdzenie, od 1.80 GHz do 4.00 GHz), 8gb RAM).

6. Podsumowanie.

Udało się znaleźć algorytm o złożoności obliczeniowej T(n) Є O(n^3). Pozostaje kwestia sprawdzenia jak rzeczywiste pomiary czasu mają się do teoretycznej złożoności. Program może generować tabel, w której współczynnik q(n) wskazuje na zgodność czasu wykonania z oszacowaniem teoretycznym. Jeśli q(n) jest w pobliżu 1, zgodność jest wysoka, jeśli q(n) maleje wraz ze wzrostem n to znaczy, że złożoność jest przeszacowana, a jeśli rośnie – niedoszacowana. Dla losowych grafów o podanych złożonościach z generacją trudnych przypadków otrzymujemy taką tabelę:

Algorytm z asymptota O(n^3)

n	t(n)[ms]	q(n)
80	4.7895	3.72352
90	6.1224	3.34293
100	10.6313	4.23174
200	35.3413	1.75843
300	105.579	1.55649
400	181.75	1.13039
500	306.92	0.977347
600	529.039	0.974915
700	745.779	0.865463
800	1108.07	0.861447
900	1464.25	0.799506
1000	2178.33	0.867073

Widać, że wartość(n) lekko maleje, jednak nie odbiega mocno od 1, co sugeruje, że złożoność teoretyczna całkiem dobrze oddaje naturę algorytmu.