# Dokumentacja

## Treść zadania

AAL-11-LS kartony

Ortodoksyjny 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.

## Doprecyzowanie treści zadania

Zadanie polega na znalezieniu optymalnego upakowania kartonów. Możemy zauważyć, że skoro dwa kartony nie mogą być jednocześnie spakowane do jednego, większego kartonu, to jeśli dla danego upakowania kartonów spróbujemy narysować graf, w którym zbiór wierzchołków tworzą kartony, a zbiór krawędzi tworzą relacje zawierania się kartonów (zakładamy, że graf jest skierowany); to graf ten jest grafem składającym się z pewnej liczby podgrafów, gdzie każdy z podgrafów jest listą i pomiędzy tymi podgrafami nie ma krawędzi.

Zadanie polega na zminimalizowaniu funkcji $V(\sigma, Kartony)$, która przyjmuje za argumenty permutację

$$\sigma = \begin{pmatrix} 0 & \dots & n-1 \\ a_0 & \dots & a_{n-1} \end{pmatrix} \in S_n,\ gdzie\ n = |Kartony|$$

oraz zbiór kartonów $Kartony$.

Jako, że dla danego przypadku testowego zbiór $Kartony$ jest stałą, w takim razie poszukujemy takiej permutacji $\Sigma$, która minimalizuje funkcję $V$ dla danego zbioru kartonów.

Wartość funkcji $V$ określona jest wzorem:

$$V(\sigma, Kartony) = v\Big(Karton\big(\sigma^{-1}(a_{n-1})\big)\Big) + \sum_{i=0}^{n-2} 0^{f(a_i, a_{i+1})} v\Big(Karton\big(\sigma^{-1}(a_i)\big)\Big) \quad (1)$$

gdzie

$$f(a_i, a_j) = \begin{cases} 1,\ dla\ MIESCI\_SIE\Big(Karton\big(\sigma^{-1}(a_i)\big), Karton\big(\sigma^{-1}(a_j)\big)\Big) \\ 0,\ w\ p.p. \end{cases}$$

$$v(karton) = x_{karton} y_{karton} z_{karton}$$

Warto zauważyć, że zawsze spełnione będzie poniższe twierdzenie.

$$\forall_{i,j \in \{0,\dots,n-1\}} : i + 1 = j$$

$$\Downarrow$$

$$\Bigg[\Big(MIESCI\_SIE\Big(Karton\big(\sigma^{-1}(a_i)\big), Karton\big(\sigma^{-1}(a_j)\big)\Big) \wedge JEST\_W\Big(Karton\big(\sigma^{-1}(a_i)\big), Karton\big(\sigma^{-1}(a_j)\big)\Big)\Big)$$

$$\vee$$

$$\Big(\neg MIESCI\_SIE\Big(Karton\big(\sigma^{-1}(a_i)\big), Karton\big(\sigma^{-1}(a_j)\big)\Big) \wedge \neg JEST\_W\Big(Karton\big(\sigma^{-1}(a_i)\big), Karton\big(\sigma^{-1}(a_j)\big)\Big)\Big)\Bigg]$$

## Dane wejściowe

Zestaw testowy składał się będzie z :

- liczby $n \in \mathbb{N}$ określającej ilość kartonów;

- $n$ lini, a w każdej z nich trójka $(x, y, z) \in \mathbb{N}^3$ taka, że $\forall_{i \in \{0,\ldots,n-1\}} : x_i, y_i, z_i$ określają kolejno długość, szerokość oraz wysokość $i$-tego kartonu.

# Dane wyjściowe

Wyjście będzie się składało z dwóch linii.
W pierwszej program wypisze ciąg liczb (numerów kartonów z wejścia)

$$a_1, \ldots, a_{n-1}$$

taki, że:

$$\forall_{i,j \in \{0,\ldots,n-1\}} : a_i \in \{0,\ldots,n-1\} \wedge a_i \in \{0,\ldots,n-1\} \wedge \left(a_i = a_j \Rightarrow i = j\right)$$

oraz taki, że

$$\Sigma = \begin{pmatrix} 0 & \ldots & n-1 \\ a_0 & \ldots & a_{n-1} \end{pmatrix} \in S_n$$

będzie permutacją minimalizującą funkcję $V$.
W drugiej linii wypisze sumaryczną minimalną zajętą przestrzeń przez kartony

$$V_{min} = V(\Sigma, Kartony).$$

# Przyjęte założenia

Zakładam, że :

- można swobodnie obracać kartony np.

  Niech karton *a* ma wymiary (1, 1, 100), natomiast karton *b* ma wymiary (2 , 101, 2). Widzimy, że próbując je włożyć w siebie, karton *a* nie zmieści się w karton *b*, jednak po obrocie takim, że wymiary kartonu *a* to (1, 100, 1) widzimy, że karton *a* mieści się do kartonu *b;*

- $\forall_{a,b \in Kartony} : MIESCI\_SIE(a, b) \Leftrightarrow \left(x_{a'} < x_b \wedge y_{a'} < y_b \wedge z_{a'} < z_b\right)$, $gdzie\ a'\ jest\ pewnym\ obrotem\ a$

# Koncepcja rozwiązania

Rozwiązanie będzie się składało z kilku algorytmów rozwiązujących ten sam problem. Pierwszym zaimplementowanym algorytmem będzie algorytm typu brut-force, który będzie generował kolejne permutacje kartonów i będzie obliczał za pomocą wzoru (1) objętość zajmowaną przez dane upakowanie. Algorytm ten będzie służył tylko i wyłącznie jako tester innych algorytmów, dla małej ilości kartonów - ze względu na złożoność obliczeniową $O(n!)$.

Drugi algorytm będzie korzystał z faktu mieszczenia się, bądź nie, kartonów jeden w drugi. Możliwe jest posortowanie topologiczne grafu składającego się z wierzchołków będącymi kartonami i krawędzi utworzonymi na podstawie relacji mieszczenia się. Utworzenie takiego grafu ma złożoność $O(n^2)$ - musimy sprawdzić czy miedzy każdą parą wierzchołków istnieje krawędź. Następnym krokiem będzie wykonanie sortowania topologicznego, które, przy pewnych modyfikacjach standardowego algorytmu DFS, da rozwiązanie w czasie $O(n + m)$ - gdzie $n$-liczbą wierzchołków, a $m$-liczbą krawędzi, możemy zauważyć, że liczba krawędzi w grafie nie przekroczy $\frac{n(n-1)}{2}$ co daje złożoność czasową algorytmu $O(n^2)$.

Kolejny algorytm będzie zawierał usprawnienie w budowaniu grafu. Będzie on korzystał z przechodniości relacji mieszczenia się:

$$\forall_{a,b,c \in Kartony} : MIESCI\_SIE(a, b) \wedge MIESCI\_SIE(b, c) \Rightarrow MIESCI\_SIE(a, c).$$

# Implementacja

Program będący rozwiązaniem zadania i jednocześnie implementującym algorytm problemu ortodoksyjnego kolekcjonera kartonów, napiszę w języku C++17, co spełni warunki zadania dotyczące przenośności. Program nie będzie posiadał graficznego interfejsu użytkownika.

Rozwiązanie będzie testowane przez generator danych testowych oraz testerkę, które ze względu na wygodę i szybkość rozwiązania prawdopodobnie zostaną napisane w Bash'u.

# Dokumentacja końcowa

Rozwiązanie zadania poprzez modyfikację sortowania topologicznego okazało się niemożliwe. W związku z tym rozwiązaniem zadania są dwa algorytmy typu brut force o złożoności czasowej $O(n!)$ i liniowej złożoności pamięciowej. Zaimplementowany został także algorytm heurystyczny popełniający błędy średnio w 10% testów. Nie potrafi on rozwiązać jednej klasy problemów, jednak działa ze złożonością $O(n^2)$.

Rozważałem wiele algorytmów i wiele podejść jednak zawsze byłem w stanie wskazać klasę podproblemów, których dane rozwiązanie nie jest w poprawnie rozwiązać. Problemem okazuje się, to że nie jest możliwe podjęcie decyzji o pakowaniu poprzez zwykłe przejście krawędzi grafu bądź sortowanie topologiczne. By poprawnie podejmować decyzje musimy analizować cały graf.

Grafu nie da się także dwukolorować, co znacząco ułatwiłoby rozwiązanie.

Próby korzystania z przechodniości mieszczenia się pudełek także zawiodły, ponieważ bardzo łatwo można znaleźć przypadek, w którym usunięcie krawędzi, wydających się jako zbędne, gubi prawidłowe rozwiązania.

Rozważałem napisanie algorytmu z nawrotami, jednak jego pesymistyczna złożoność sięgnęła $O(n!)$, przez co zaniechałem napisania tego algorytmu.

# Algorytm naiwny

Algorytm naiwny sprawdza wszystkie możliwe permutacje pudełek i wybiera najlepsze.

Czasy wykonania algorytmu naiwnego

| n | t | q |
|---|---|---|
| **4** | 1 | 105 |
| **5** | 2 | 42 |
| **6** | 2 | 7 |
| **7** | 3 | 1,5 |
| **8** | 16 | 1 |
| **9** | 120 | 0,833333333333333 |
| **10** | 1320 | 0,916666666666667 |
| **11** | 15700 | 0,991161616161616 |
| **12** | 193500 | 1,01799242424242 |

# Poprawiony algorytm naiwny

Ten algorytm w stosunku do algorytmu naiwnego przeprowadza szybkie poszukiwanie przypadków trywialnych do rozwiązania - gdy do pudełka *a* mieści się tylko jedno inne pudełko - *b*. W tym przypadku wiadome jest, że w rozwiązaniu pudełko *b* znajdzie się w środku pudełka *a*. Do pozostałych wierzchołków stosujemy algorytm naiwny, lecz na zmniejszonym problemie.

# Algorytm zachłanny

Algorytm heurystyczny potrafi dość szybko rozwiązać problem, lecz nie daje pełnej poprawności. Idea polega na przetwarzaniu największego nierozpatrzonego wierzchołka i włożeniu go do jednego z kartonów, w który się mieści, w taki sposób aby minimalizował prawdopodobieństwo złego ułożenia.

| n | t | q |
|---|---|---|
| **12** | 1 | 6,8620992534036 |
| **100** | 9 | 0,889328063241107 |
| **500** | 253 | 1 |
| **1000** | 1400 | 1,38339920948617 |

# Wyniki

Czasy wykonania programów były mierzone systemowym narzędziem time, które pozwala odfiltrować czas uśpienia programu i czas, w którym oddawał on sterowanie systemowi. Niestety narzędzie podaje czas wykonania z dokładnością do milisekundy.