Notatki z AiSD. Nr 3. 7 marca 2022

Algorytmy Zachłanne.

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

1 Schemat ogólny.

Typowe zadanie rozwiązywane metodą zachłanną ma charakter optymalizacyjny. Mamy dany skończony zbiór C. Rozwiązaniami zadania są pewne podzbiory zbioru C. Znamy kryterium pozwające na porównywanie jakości rozwiązań. Chcemy znaleźć rozwiązanie, które jest optymalne względem tego kryterium.

Przykład
Problem:
Dane: liczba naturalna R oraz zbiór liczb naturalnych $c_1, c_2, \ldots, c_k$;
(interpretacja: R jest kwotą, którą chcemy rozmienić, a c_i są nominałami monet).
Zadanie: rozmienić kwotę R na możliwie najmniejszą liczbę monet (przy założeniu, że dysponujemy nieograniczoną liczbą monet każdego nominału).

W tym przykładzie zbiorem C jest zbiór monet (zauważ, że zbiór ten jest skończony, ponieważ możemy przyjąć, że należy do niego nie więcej niż R monet o nominale c_i, dla każdego $i = 1, \ldots, k$). Rozwiązaniami problemu są te podzbiory zbioru C, których elementy sumują się do kwoty R. Kryterium jakości rozwiązania jest liczba jego elementów. □

Algorytm zachłanny konstruuje rozwiązanie, nazwijmy je S, stopniowo (zwykle startując od zbioru pustego). W każdym z kolejnych kroków, algorytm rozważa jeden element z C, powiedzmy x. Element ten jest umieszczany w S, jeśli algorytm uzna, że $S \cup \{x\}$ da się rozszerzyć do rozwiązania. Niezależnie od decyzji, x jest usuwany z C. Wybór x-a dokonywany jest na podstawie lokalnego (zachłannego) kryterium optymalności.

Algorytmy zachłanne nie podejmują żadnych (lub prawie żadnych) działań sprawdzających czy konstruowany zbiór ma szansę być optymalnym rozwiązaniem. W efekcie algorytmy zachłanne są proste i szybkie, ale mogą dawać rozwiązania nieoptymalne, a nawet nie dawać rozwiązań. Dlatego ważnym zadaniem jest analiza poprawności algorytmu zachłannego. Jeśli algorytm produkuje rozwiązania nieoptymalne, warta rozważenia może być kwestia "jak dalece nieoptymalne": czasami mogą one być akceptowalnie bliskie optymalnym. Z takimi zagadnieniami zapoznamy się, gdy będziemy omawiać algorytmy aproksymacyjne.

Przykład
Strategia zachłanna dla problemu wydawania reszty może polegać na tym, by w kolejnych krokach do konstruowanego rozwiązania wstawiać monetę o największym nominale nie przekraczającym kwoty pozostałej do wydania.
Można łatwo wykazać, że dla zbioru nominałów $\{1, 2, 5, 10, 20, 50, 100\}$ taka strategia prowadzi zawsze do rozwiązania optymalnego. Natomiast dla zbioru $\{1, 2, 5, 9, 10\}$ czasami daje rozwiązania nieoptymalne: np. dla $R = 14$ znajdzie rozwiązanie $10, 2, 2$, chociaż wystarczają dwie monety. □

1.1 Konstrukcja minimalnego drzewa rozpinającego

Rozważamy grafy nieskierowane $G = (V, E; c)$, gdzie V oznacza zbiór wierzchołków, E - zbiór krawędzi, a $c : E \to R_+$ jest funkcją wagową. Wagą podgrafu $G' = (V', E')$ grafu G nazywamy sumę wag krawędzi z E'.

Definicja 1 Drzewem rozpinającym *grafu* $G = (V, E; c)$ *nazywamy dowolne drzewo* $T = (V, E')$, *takie, że* $E' \subseteq E$. *Drzewo rozpinające* T *nazywamy* minimalnym, *jeśli ma minimalną wagę spośród wszystkich drzew rozpinających grafu* G.

PROBLEM:
Dane: graf $G = (V, E; c)$
Zadanie: Wyznaczyć minimalne drzewo rozpinające $T = (V, E')$ grafu G.

Jak łatwo zauważyć zadanie sprowadza się do wyznaczenia zbioru krawędzi drzewa rozpinającego. Wiele znanych algorytmów rozwiązujących ten problem opartych jest na strategiach zachłannych. Poniżej przedstawiamy trzy z nich.

- Strategia 1: Algorytm Kruskala
  Rozpoczynamy od pustego E'. Zbiór C jest początkowo równy E. W kolejnym kroku rozpatrujemy krawędź z C o minimalnej wadze. Dodajemy ją do E', o ile nie powoduje to powstania cyklu.

- Strategia 2: Algorytm Prima
  Inicjujemy E' wstawiając do niego minimalną krawędź spośród krawędzi incydentnych z wierzchołkiem v (v - wybierany jest arbitrarnie). Podobnie jak poprzednio zbiór C jest początkowo równy E. W kolejnym kroku rozpatrujemy minimalną krawędź z C incydentną z jakąś krawędzią z E' . Dodajemy ją do E', o ile drugi z jej końców nie jest incydentny z E'.

- Strategia 3: Algorytm Boruvki
  Strategia ta nieco odbiega od ogólnego schematu. Zbiór E' budujemy fazami. W każdej fazie wykonujemy dwa kroki:

  - Dla każdego wierzchołka z G znajdujemy najkrótszą incydentną z nim krawędź; krawędzie te dołączamy do zbioru E'.
  - Tworzymy nowy graf G'. Wierzchołki w G' (nazwijmy je superwierzchołkami) odpowiadają spójnym składowym w E'. Dwa superwierzchołki S_1 i S_2 łączymy krawędzią wtedy i tylko wtedy, gdy jakiś wierzchołek z S_1 był połączony w G krawędzią z jakimś wierzchołkiem z S_2. Jako wagę tej krawędzi przyjmujemy minimalną wagę krawędzi w G pomiędzy wierzchołkami z S_1 i S_2.
    Za G przyjmujemy G' i przechodzimy do nowej fazy.

  UWAGI:

  - algorytm Boruvki jest szczególnie przystosowany do implementacji na maszynach równoległych;
  - algorytm ten działa poprawnie, gdy wszystkie krawędzie mają różne wagi (to jednak zawsze potrafimy zagwarantować).

Dowody poprawności tych algorytmów są podobne. Łatwo zauważyć, że wszystkie algorytmy znajdują drzewa rozpinające. Strategie Kruskala i Boruvki gwarantują bowiem, że w każdym momencie krawędzie z E' tworzą las drzew, a strategia Prima gwarantuje, że krawędzie z E' tworzą drzewo. O ile graf G jest spójny, to po zakończeniu działania algorytmu graf (V, E') także jest spójny (a więc jest drzewem rozpinającym). W przeciwnym razie minimalna krawędź spośród krawędzi o końcach w różnych składowych nie byłaby dołączona przez algorytmy Kruskala i Boruvki do E'; a w przypadku algorytmu Prima taką niedołączoną krawędzią byłaby minimalna krawędź indydentna jednym końcem z E'. W obydwu przypadkach otrzymujemy sprzeczność.

Minimalność wyznaczonych drzew wynika z faktu, że w każdym momencie konstrukcji zbioru E' jest on rozszerzalny do minimalnego drzewa rozpinającego.

1.2 Szeregowanie zadań

Rozważymy teraz dwa przykłady prostych problemów teorii szeregowania zadań. Obydwa dotyczą szeregowania dla pojedynczego procesora i dają się rozwiązać algorytmami zachłannymi. Pod tym względem są wyjątkowe, ponieważ większość problemów szeregowania jest NP-trudna.

1.2.1 Szeregowanie zadań dla pojedynczego procesora

SCENARIUSZ: System z jednym serwerem (procesorem) ma do obsłużenia n zleceń. Zawczasu znany jest czas obsługi każdego zlecenia. Przez *czas przebywania zlecenia w systemie* rozumiemy czas jaki upłynął od momentu zgłoszenia zlecenia systemowi do momentu zakończenia jego obsługi. Zakładamy, że wszystkie zlecenia zgłoszone zostały jednocześnie i że obsługa zleceń odbywa się bez przerwań, więc czas przebywania zlecenia w systemie jest równy sumie czasu oczekiwania na zlecenie i czasu obsługi. Za czas jaki zlecenia przebywają w systemie, system płaci karę proporcjonalną do tego czasu, dlatego naszym zadaniem jest ustawienie zleceń w kolejności minimalizującej karę.

PROBLEM:
Dane: ciąg $t_1, \ldots, t_n$ dodatnich liczb rzeczywistych;
(interpretacja: t_j - czas obsługi j-tego zadania w systemie).
Zadanie: ustawić zadania w kolejności minimalizującej wartość:
$T = \sum_{i=1}^{n}$ (*czas przebywania i-tego zadania w systemie*)

Strategia zachłanna: Zadania ustawiamy w kolejności rosnących czasów obsługi.

DOWÓD POPRAWNOŚCI: Zauważamy , że jeśli zadania realizowane są w kolejności zadanej permutacją $\pi = (i_1, i_2, \ldots, i_n)$ liczb $(1, 2, \ldots, n)$, to związany z tym koszt wynosi:

$$T(\pi) = t_{i_1} + (t_{i_1} + t_{i_2}) + \ldots + (t_{i_1} + \ldots + t_{i_n}) = \sum_{k=1}^{n} (n - k + 1) t_{i_k}$$

Załóżmy, że π jest optymalnym porządkiem oraz że istnieją x, y takie, że $x < y$ oraz $t_{i_x} > t_{i_y}$. Zamieniając i_x oraz i_y miejscami otrzymujemy nowy porządek π' o koszcie:

$$T(\pi') = (n - x + 1) t_{i_y} + (n - y + 1) t_{i_x} + \sum_{k=1, k \neq x, y}^{n} (n - k + 1) t_{i_k}.$$

Nowy porządek jest lepszy, ponieważ

$$T(\pi) - T(\pi') = (n - x + 1)(t_{i_x} - t_{i_y}) + (n - y + 1)(t_{i_y} - t_{i_x}) = (y - x)(t_{i_x} - t_{i_y}) > 0$$

co jest sprzeczne z założeniem optymalności π. □

1.2.2 Szeregowanie z terminami

SCENARIUSZ: System z jednym procesorem ma do wykonania n zadań. Każde z nich wymaga jednej jednostki czasu procesora. Dla każdego zadania znany jest zysk, jaki otrzyma system za jego wykonanie oraz termin. Za wykonanie zadania po terminie system nie otrzymuje żadnego zysku. Naszym celem jest ustawienie zadań w kolejności maksymalizującej zysk.

PROBLEM:
Dane: ciąg $d_1, \ldots, d_n$ liczb naturalnych oraz
ciąg $g_1, \ldots, g_n$ dodatnich liczb rzeczywistych;
(Interpretacja: d_i-termin dla i-tego zadania; g_i-zysk za i-te zadanie).
Zadanie: znaleźć wykonalny podzbiór (patrz definicja poniżej) zadań $S \subseteq \{1, \ldots, n\}$ maksymalizujący sumę $\sum_{i \in S} g_i$.

Definicja 2 *Ciąg zadań* $\langle i_1, i_2 \dots i_n \rangle$ *taki, że* $\forall_{k=1\dots n}\ k \leq d_{i_k}$ *nazywamy* wykonalnym. *Zbiór zadań jest* wykonalny, *jeśli wszystkie jego elementy można ustawić w ciąg wykonalny.*

Strategia zachłanna: Startując od zbioru pustego konstruujemy wykonalny zbiór zadań, na każdym kroku dodając do niego zadanie o największym g_i spośród zadań jeszcze nie rozważonych (pod warunkiem, że zbiór pozostaje wykonalny).

DOWÓD POPRAWNOŚCI (SZKIC): Załóżmy, że nasz algorytm wybrał zbiór zadań I, podczas gdy istnieje zbiór optymalny $J \neq I$. Pokażemy, że dla obydwu zbiorów zysk za wykonanie zadań jest taki sam. Niech π_I, π_J-wykonalne ciągi zadań z I i J. Dowód przebiega w dwóch etapach:

1. Wykonując przestawienia otrzymujemy ciągi π'_I oraz π'_J takie, że wszystkie wspólne zadania (tj. zadania z $I \cap J$) wykonują się w tym samym czasie.
2. Pokazujemy, że w pozostałych jednostkach czasowych π'_I oraz π'_J mają zaplanowane wykonanie zadań o tym samym zysku.

Ad.1. Niech $a \in C$ będzie zadaniem umieszczonym na różnych pozycjach w π_I oraz π_J. Niech będą to pozycje odpowiednio i oraz j. Bez zmniejszenia ogólności możemy założyć, że $i < j$. Zadanie a w ciągu π_I możemy zamienić miejscami z zadaniem znajdującym się na pozycji j, nazwijmy to zadanie b. Otrzymamy ciąg wykonalny, gdyż:

- $d_a \geq j$, ponieważ a znajduje się na pozycji j w π_J,
- $d_b > i$, ponieważ b znajduje się na pozycji j w π_I.

Liczba zadań z $I \cap J$ rozmieszczonych na różnych pozycjach w π_J i nowym ciągu π_I zmniejszyła się o co najmniej 1. Iterując postępowanie otrzymujemy tezę.

Ad.2. Rozważmy dowolną pozycję i, na której różnią się π'_I oraz π'_J.
Zauważamy, że zarówno π'_I jak i π'_J mają na tej pozycji umieszczone jakieś zadanie, nazwijmy je a i b odpowiednio. Gdyby bowiem π'_J miało tę pozycję wolną, to moglibyśmy umieścić na niej a otrzymując ciąg wykonalny dla $J \cup \{a\}$, co przeczy optymalności J. Z drugiej strony, gdyby π'_I miało tę pozycję wolną to algorytm zachłanny umieściłby b w rozwiązaniu.

Wystarczy teraz pokazać, że $g_a = g_b$:

- gdyby $g_a < g_b$, to algorytm zachłanny rozpatrywałby wcześniej b niż a; ponieważ zbiór $I \setminus \{a\} \cup \{b\}$ jest wykonalny (a więc także i ten jego podzbiór, który był skonstruowany w momencie rozpatrywania b), więc b zostałoby dołączone do rozwiązania.
- gdyby $g_a > g_b$, to $J \setminus \{b\} \cup \{a\}$ dawałby większy zysk niż J?!

□

Pozostaje problem: jak można ustalać, czy dany zbiór J złożony z k zadań jest wykonalny (oczywiście sprawdzanie wszystkich $k!$ ciągów nie jest najlepszym pomysłem). Poniższy prosty lemat mówi, że wystarczy sprawdzać wykonalność tylko jednego ciągu.

Lemat 1 *Niech J będzie zbiorem k zadań i niech $\sigma = (s_1, s_2 \dots s_k)$ będzie permutacją tych zadań taką, że $d_{s_1} \leq d_{s_2} \leq \dots \leq d_{s_k}$. Wówczas J jest wykonalny iff σ jest wykonalny.*

1.2.3 Prosta implementacja

Zakładamy, że zadania ułożone są według malejących zysków, tj. $g_1 \geq g_2 \geq \dots \geq g_n$.

Prosta implementacja polega na pamiętaniu skonstruowanego fragmentu wykonywalnego ciągu zadań w początkowym fragmencie tablicy. Zadania umieszczane są tam według rosnących wartości terminów w sposób podobny jak w procedurze sortowania przez wstawianie.

Fakt 1 *Taka implementacja działa w czasie* $\Omega(n^2)$.

1.2.4 Szybsza implementacja

Kluczem do szybszej implementacji jest następujący lemat.

Lemat 2 *Zbiór zadań J jest wykonalny iff następująca procedura ustawia wszystkie zadania z J w ciąg wykonalny:*

> *$\forall_{i \in J}$ ustaw i-te zadanie na pozycji t, gdzie t jest największą liczbą całkowitą, taką że $0 < t \leq \min(n, d_i)$ i na pozycji t nie ustawiono jeszcze żadnego zadania.*

Efektywna realizacja tej procedury używa struktur do pamiętania zbiorów rozłącznych. Poznamy je, gdy będziemy omawiać problem UNION-FIND. Uzyskany czas działania algorytmu będzie bliski liniowego.

UWAGA: Trzeba jednak pamiętać, że jeśli zadania nie są, jak to założyliśmy, wstępnie uporządkowane według wartości g_i, to czas działania algorytmu zostanie zdominowany przez czas sortowania.

1.3 Pokrycie zbioru

PROBLEM:

Dane: rodzina $S = \{S_1, S_2, \ldots, S_k\}$ podzbiorów n-elementowego uniwersum U
funkcja kosztu $c : S \to \mathcal{R}_+$

Zadanie: Znaleźć najtańszą podrodzinę S pokrywającą U, tj.
znaleźć $S' \subseteq S$ taką, że $\bigcup_{X \in S'} X = U$ i żadna inna podrodzina nie ma kosztu mniejszego od $Koszt(S')$, gdzie koszt podrodziny Z jest zdefiniowany jako

$$Koszt(Z) = \sum_{X \in Z} c(X).$$

1.3.1 Strategie zachłanne

Można pomyśleć o kilku różnych strategiach zachłannych dla tego problemu. Przykładowo:

- Strategia 1: wybierz podzbiór pokrywający najwięcej elementów.
- Strategia 2: wybierz najtańszy podzbiór.
- Strategia 3: wybierz podzbiór, który najtaniej pokrywa elementy.
- ...

Dla każdej z nich łatwo można podać dane, dla których nie znajduje ona rozwiązania optymalnego. Nie ma w tym nic dziwnego, ponieważ problem należy do klasy problemów $\mathcal{NP}$-trudnych, więc szansa na istnienie algorytmu o złożoności wielomianowej jest znikoma.[1]

1.3.2 Algorytm aproksymacyjny

Pokażemy jednak, że trzecia strategia gwarantuje znalezienie rozwiązania, które jest niezbyt odległe od rozwiązania optymalnego.

Najpierw sprecyzujemy tę strategię. W kolejnych krokach:

(a) dla każdego podzbioru S_i określamy cenę za pokrywany element (w skrócie *cne*):

$$cne(S_i) = \frac{c(S_i)}{|S_i \setminus C|}$$

gdzie C oznacza zbiór dotychczas pokrytych elementów.

[1] O problemach $\mathcal{NP}$-trudnych będziemy mówić na jednym z kolejnych wykładów.

(b) do rozwiązania wybieramy ten z podzbiorów, dla którego wartość *cne* jest minimalna.

Twierdzenie 1 *Opisana powyżej strategia zachłanna znajduje rozwiązanie o koszcie nie większym niż* $(\log n) \cdot OPT$*, gdzie* OPT *jest kosztem rozwiązania optymalnego.*

Niech $e_1, e_2, \ldots, e_n$ będzie ciągiem elementów uniwersum wypisanych w kolejności pokrywania ich przez algorytm. Jeśli pokrywanym elementom przypiszemy cenę, równą wartości *cne* podzbioru, który je pokrywa, to koszt pokrycia możemy wyrazić jako sumę cen elementów. Zauważamy następujący fakt, z którego wynika dowód twierdzenia.

Fakt 2 *Dla każdego* $i = 1, \ldots, n$

$$cena(e_i) \leq \frac{OPT}{n - i + 1}$$

Wynika on z następujących spostrzeżeń:

- W momencie pokrywania elementu e_i niepokrytych było co najmniej $n - i + 1$ elementów uniwersum.
- Te niepokryte elementy można by pokryć podzbiorami, które należą do rozwiązania optymalnego. Sumaryczny koszt tych podzbiorów jest nie większy od OPT. Koszt ten zostałby rozłożony na ceny tych elementów, więc istnieje element, którego cena jest nie większa od średniej, tj. od $\frac{OPT}{n-i+1}$.

Algorytmy działające w czasie wielomianowym, które dają gwarancję, że otrzymane rozwiązanie ma wartość nie większą (w przypadku problemu minimalizacyjnego i nie mniejszą w przypadku problemu maksymalizacyjnego) niż $c \cdot OPT$ nazywamy *algorytmami aproksymacyjnymi*. Wartość c nazywamy *współczynnikiem aproksymacji*. Może to być wartość stała lub określona funkcją, tak jak w przypadku rozważanego tu algorytmu.

Powstają naturalne pytania:

- czy nie można poprawić analizy naszego algorytmu i wykazać, że ma on lepszy (tj. mniejszy) współczynnik aproksymacji?
- czy nie istnieje inny algorytm dla problemu pokrycia zbioru, o lepszym współczynniku aproksymacji?
- czy istnieje jakieś nietrywialne ograniczenie dolne na wartość współczynnika aproksymacji algorytmów dla problemu pokrycia zbioru?

Odpowiedź na pierwsze pytanie jest dość prosta i była prezentowana na wykładzie. Odpowiedź na pozostałe pytania istotnie wykracza poza zakres wykładu.

Algorytmom aproksymacyjnym będziemy chcieli poświęcić jeszcze jeden wykład po koniec semestru.