Algorytmy i Struktury Danych 2

Michał Okurowski

Wydział Matematyki i Nauk Informacyjnych
Politechnika Warszawska

Spis treści

1 Programowanie dynamiczne

Programowanie dynamiczne ma zastosowanie w problemach wykazujących własność **optymalnej podstruktury** – to znaczy, kiedy optymalne rozwiązanie problemu łatwo jest uzyskać znając optymalne rozwiązania podproblemów.

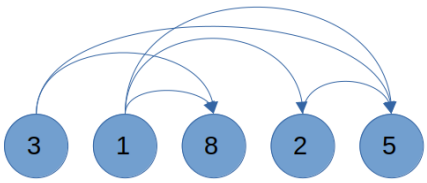
1.1 Znajdowanie najdłuższego podciągu rosnącego

Rozważmy następujący problem:

> **Dane:** Pewien ciąg $c = c_1, c_2, \ldots, c_n$
>
> **Szukane:** Dowolny najdłuższy podciąg $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ ciągu c, taki, że $i_1 < i_2 < \ldots < i_k$ oraz $c_{i_1} < c_{i_2} < \ldots < c_{i_k}$.

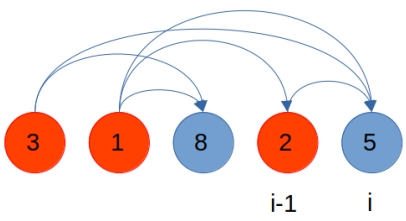
Przykładowo, dla wejściowego ciągu $(3, 1, 8, 2, 5)$ odpowiedzią będzie ciąg $(1, 8, 5)$.

W pierwszym kroku rozwiązywania problemów programowania dynamicznego, szukamy odpowiedniego podproblemu. Dla ułatwienia, zareprezentujmy ciąg jako graf skierowany G, w którym wierzchołkami są elementy ciągu c oraz istnieje krawędź z wierzchołka c_i do wierzchołka c_j jeśli $x < y$ oraz $i < j$. Opisany graf dla powyższego przykładu wygląda tak jak na rysunku **??**.



Rys. 1

Uprośćmy tymczasowo analizowany problem do problemu znajdowania długości najdłuższego rosnącego podciągu. Utwórzmy tablicę T (numerujemy od 1), w której na indeksie i będziemy przechowywać długość najdłuższego podciągu, dla którego c_i jest jego ostatnim elementem. Przypuśćmy, że wypełniliśmy tablicę T do $(i-1)$-tego elementu. Wtedy aby poznać wartość na i-tym indeksie wystarczy przeanalizować wszystkie te wierzchołki w grafie G które mają skierowaną krawędź na i-ty wierzchołek.



Rys. 2

Niech I to zbiór indeksów wszystkich elementów ciągu c, które są mniejsze niż c_i, wtedy

$$T[i] = \max_{j \in I}\{T[j]\} + 1.$$

Jeśli wypełnimy całą tablicę T wg. powyższej procedury, odpowiedziedzą będzie maksymalny element tablicy T.

Algorithm 1 Znajdowanie najdłuższego rosnącego podciągu.

```
1:  procedure MaxIncreasingSubseqenceLength(c = c_1, c_2, ..., c_n)
2:      Utwórz tablicę n-elementową T
3:      Ustaw wartość każdej komórki tablicy T na 1
4:      i ← 2
5:      while i ⩽ n do
6:          j ← 1
7:          while j ⩽ i − 1 do
8:              if c_i ⩾ c_j then continue
9:              if T[i] ⩽ T[j] + 1 then
10:                 T[j] ← T[i]
11:     return max_{1⩽i⩽n} {T[i]}
```

Złożoność czasowa powyszego rozwiązania jest rzędu n^2.

Powyższy algorytm można bardzo łatwo zmodyfikować w taki sposób, aby możliwym było zwrócenie podciągu stanowiącego rozwiązanie (problem pierwotny). Wystarczy utworzyć tablicę P, która na indeksie i będzie zapisywać indeks ostatniego elementu, który spełnił warunek $T[i] \leqslant T[j] + 1$. Ustalmy, że -1 będzie oznaczało, że takiego elementu nie ma (tzn. c_i jest pierwszym elementem podciągu).

Algorithm 2 Znajdowanie najdłuższego rosnącego podciągu.

```
1: procedure MaxIncreasingSubseqence(c = c_1, c_2, ..., c_n)
2:     Utwórz tablicę n-elementową T
3:     Ustaw wartość każdej komórki tablicy T na 1
4:     Utwórz tablicę n-elementową P
5:     Ustaw wartość każdej komórki tablicy P na −1
6:     i ← 2
7:     while i ⩽ n do
8:         j ← 1
9:         while j ⩽ i − 1 do
10:            if c_i ⩾ c_j then continue
11:            if T[i] ⩽ T[j] + 1 then
12:                T[j] ← T[i]
13:                P[i] ← j
14:    m = max_{1⩽i⩽n} {T[i]}
15:    Utwórz tablicę m-elementową A reprezentującą kolejne wartości szukanego podciągu
16:    i ← m
17:    j ← indeks elementu o wartości m
18:    while i ⩾ 1 do
19:        A[i] ← c_j
20:        j ← P[j]
21:        i ← i − 1
22:    return A
```

1.2 Problem wydawania reszty

Rozważmy następujący problem:

> **Dane:** Nominały monet n_1, n_2, …, $n_k \in \mathbb{N}$ oraz kwota $K \in \mathbb{N}$
>
> **Szukane:** Ciąg liczb naturalnych $a = a_1$, a_2, … a_k taki, że
>
> $$\sum_{i=1}^{k} a_i n_i = K \text{ oraz suma } \sum_{i=1}^{k} a_i \text{ jest najmniejsza możliwa,}$$
>
> innymi słowy, szukamy sposobu na wydanie kwoty K, z użyciem jak najmniejszej liczby monet.

Na początek zaobserwujmy, że jeśli $a = a_1$, a_2, … a_k to rozwiązanie optymalne powyższego problemu, to zmniejszenie $a_i \geqslant 1$ ($i \in [k]$) o 1, powinno zaskutkować powstaniem rozwiązania problemu dla kwoty $K - n_i$. To stwierdzenie musi być prawdziwe, bo inaczej moglibyśmy skonstruować rozwiązanie o mniejszej liczbie monet niż optymalne co jest sprzeczne.

Przykładowo, dla $K = 13$ i nominałów $(1, 2, 5)$, optymalnym rozwiązaniem jest ciąg $(1, 1, 2)$. Wtedy ciąg $(0, 1, 2)$ jest optymalnym rozwiązaniem dla $K = 12$, $(1, 0, 2)$ jest optymalnym rozwiązaniem dla $K = 11$ oraz $(1, 1, 1)$ dla $K = 8$.

Z powyższego stwierdzenia wynika, że otrzymanie optymalnego rozwiązania dla kwoty K jest możliwe po przeanalizowaniu **wszystkich** optymalnych rozwiązań dla kwot $K - n_i$, co oznacza że udało nam się znaleźć optymalną podstrukturę.

Tak samo jak w przypadku problemu znajdowania najdłuższego podciągu rosnącego, rozważmy uproszczoną wersję problemu, w której interesować nas będzie nie ciąg a, ale liczba monet w optymalnym rozwiązaniu.

Zdefiniujmy tablicę T (indeksowaną od 0) jako tablicę rozmiaru $K + 1$, w której komórka $T[i]$ $(i \in 0, 1, \ldots, K)$ przechowuje liczbę monet potrzebną do optymalnego wydania kwoty równej i. Z powyższych rozważań wiadomo, że jeśli tablica jest wypełniona do $(i-1)$-tego elementu $(i \in 0, 1, \ldots, K)$ włącznie, to

$$T[i] = \min_{1 \leqslant j \leqslant k} \{T[i - n_j]\} + 1, \tag{1}$$

ponadto przy takiej definicji tablicy, możemy przyjąć, że $T[0] = 0$.

Na koniec zauważmy, że w badanym problemie istnieją dane wejściowe, dla których nie istnieje rozwiązanie (w szczególności szukane rozwiązanie optymalne), np. nie możemy wydać $K = 3$ dla nominałów $(2, 4)$. Aby ułatwić zapis rozwiązania jak i wykrywanie sytuacji, w których ono nie istnieje, każda komórka tablicy T będzie inicjowana wartością ∞. Jeśli po wykonaniu się algorytmu, $T[K] = \infty$ możemy zwrócić informację, że rozwiązanie nie istnieje.

Algorithm 3 Znajdowanie liczby monet optymalnego rozwiązania w problemie wydawania reszty.

```
1:  procedure MinCoinsCountChangeMaking(n_1, n_2, ..., n_k ∈ ℕ, K ∈ ℕ)
2:      Utwórz tablicę (K + 1)-elementową T
3:      Ustaw wartość każdej komórki tablicy T na ∞
4:      T[0] = 0
5:      for i = 1, 2, ... K do
6:          for j = 1, 2, ... k do
7:              if i < n_j then continue
8:              c = T[i − n_j] + 1
9:              if c < T[i] then T[i] ← c + 1
10:     if T[K] = ∞ then return null
11:     return T[K]
```

Powyższy algorytm ma złożoność pamięciową rzędu K oraz czasową $K \cdot k$. Gdybyśmy chcieli ten sam problem rozwiązać rekurencyjną metodą otrzymalibyśmy złożność czasową rzędu 2^K, przez rozwiązywanie tych samych podproblemów kilkukrotnie.

Aby rozwiązać problem pierwotny, podobnie jak w znajdowaniu najdłuższego podciągu rosnącego, utworzymy tablicę P o długości $K+1$, która będzie przechowywać w komórce o indeksie $i \in \{0, 1, \ldots K\}$, indeks j nominału, takiego, że zależność (**??**) będzie prawdziwa.

Algorithm 4 Znajdowanie liczby monet optymalnego rozwiązania w problemie wydawania reszty.

```
1: procedure ChangeMaking(n_1, n_2, ..., n_k ∈ ℕ, K ∈ ℕ)
2:     Utwórz tablicę (K+1)-elementową T
3:     Ustaw wartość każdej komórki tablicy T na ∞
4:     Utwórz tablicę (K+1)-elementową P
5:     T[0] ← 0
6:     for i = 1, 2, ... K do
7:         for j = 1, 2, ... k do
8:             if i < n_j then continue
9:             c = T[i − n_j] + 1
10:            if c < T[i] then
11:                T[i] ← c + 1
12:                P[i] ← j
13:    if T[K] = ∞ then return null
14:    Utwórz tablicę T[K]-elementową A reprezentującą kolejne wartości ciągu.
15:    i ← K
16:    while i > 0 do
17:        A[P[i]] ← A[i] + 1
18:        i ← i − n_{P[i]}
19:    return A
```

1.3 Skreślanie ciągów

2 Programowanie dynamiczne – Zadania

2.1 Zadanie 1 – Szukanie podzbioru o zadanej sumie elementów

Treść: Zaprojektuj algorytm, który dla zadanego zbioru liczb naturalnych S i liczby naturalnej N rozstrzygnie, czy S zawiera podzbiór o sumie równej N w czasie $O(N|S|)$.

Wskazówka: wyznacz wszystkie możliwe sumy podzbiorów zbioru S

Rozwiązanie:

Niech $S = \{s_0, s_1, \ldots, s_{|S|-1}\}$ to zbiór wejściowy, oraz niech N to liczba, dla której sprawdzimy czy istnieją elementy, które się do niej sumują.

Niech T to tablica dwuwymiarowa o wymiarach $|S|$ na $N+1$, przyjmująca wartości *True* lub *False* w każdej komórce tablicy T. Komórka tablicy $T[i, j]$ odpowiada na pytanie, czy da się otrzymać sumę równą i z j-elementowego podzbioru $\{s_1, s_2, \ldots, s_j\} \subseteq S$.

Algorytm polega na uzupełnianiu kolejnych wierszy tablicy. Najpierw rozważamy problem oparty na zbiorze $\{s_1\}$. Nastepnie aby otrzymać rozwiązanie dla $\{s_1, s_2\}$, wystarczy od każdej komórki przechowującej T przemieścić się o s_2 komórek w prawo i zmienić wartość na T i tak do momentu uzupełnienia tablicy T.

Po uzupełnieniu T odpowiedzią na zadane pytanie jest $T[k, N]$.

Rozważmy przykład: $S = \{1, 3, 5, 10\}$ (tzn. $s_0 = 1$, $s_1 = 3$, $s_2 = 5$, $s_3 = 10$), $N = 9$. Tablica prezentuje się wtedy tak jak poniżej.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1 $\{1\}$	T	T	F	F	F	F	F	F	F	F
2 $\{1, 3\}$	T	T	F	T	T	F	F	F	F	F
3 $\{1, 3, 5\}$	T	T	F	T	T	T	T	F	T	T
4 $\{1, 3, 5, 10\}$	T	T	F	T	T	T	T	F	T	T

Tabela 1: Elementy $s_1, s_2 \ldots, s_k \in S$ nie muszą być posortowane.

Algorithm 5 Rozwiązanie zadania 1.1

```
1:  procedure SubsetSum(S = {s_0, s_1, ..., s_{|S|-1}} ⊆ ℕ, N ∈ ℕ)
2:      Utwórz tablicę dwuwymiarową T o |S| wierszach i N + 1 kolumnach
3:      Wypełnij tablicę T wartością False
4:      T[0, 0] ← True
5:      T[0, s_1] ← True
6:      i ← 1
7:      for i = 1, 2 ... |S| − 1 do
8:          for j = 0, 1, 2 ... N do
9:              if T[i − 1, j] = True then
10:                 T[i, j] ← True
11:                 if j + s_j ⩽ N then
12:                     T[i, j + s_j] ← True
13:     return T[k, N]
```

2.2 Zadanie 2 – Szukanie najdłuższego wspólnego podciągu

Treść: Zaprojektuj algorytm, który znajdzie długość najdłuższego wspólnego podciągu dwoch zadanych ciągów o nie więcej niż n wyrazach w czasie $O(n^2)$. Przykładowo, najdłuższym wspólnym podciągiem ciągów „abcdef" i „eacgd" jest „acd".

Wskazówka: zastosuj programowanie dynamiczne; wyznacz rozwiązanie dla każdej pary prefiksów zadanych napsów.

Rozwiązanie: Oznaczmy ciągi wejściowe jako $A = a_1, a_2, \ldots a_n$ oraz $B = b_1, b_2, \ldots b_m$. Niech C to największy możliwy wspólny podciąg A i B. Zauważmy, że jeśli znamy rozwiązanie nastepujących podproblemów:

1. Największy możliwy wspólny podciąg C_1 dla ciągów $A_1 = a_1, a_2, \ldots a_n$ oraz $B_1 = b_1, b_2, \ldots b_{m-1}$,
2. Największy możliwy wspólny podciąg C_2 dla ciągów $A_2 = a_1, a_2, \ldots a_{n-1}$ oraz $B_2 = b_1, b_2, \ldots b_m$,

3. Największy możliwy wspólny podciąg C_3 dla ciągów $A_3 = a_1, a_2, \ldots a_{n-1}$ oraz $B_3 = b_1, b_2, \ldots b_{m-1}$

to jesteśmy w stanie znaleźć C w następujący sposób: jeżeli $a_n = b_m$, to rozwiązaniem jest podciąg C_3 z dodatkowym elementem a_n, w przeciwnym przypadku rozwiązaniem musi być większy z ciągów C_1 oraz C_2 (★).

Aby zaimplementować to rozumowanie, zastosujemy tablicę dwuwymiarową T (indeksujemy od 0) o $n+1$ wierszach oraz $m+1$ kolumnach. Jako, że w zadaniu mowa jest tylko o długości szukanego podciągu C, każda z komórek tablicy T o indeksach i oraz j ($i \in \{0, 1, \ldots, n\}$, $j \in \{0, 1, \ldots, m\}$) będzie przechowywać długość szukanego podciągu dla ciągów $a_1, a_2, \ldots, a_i$ oraz $b_1, b_2, \ldots, b_j$.

Dla przykładu jeśli $A =$ „abcdef" i $B =$ „eacgd", to tablica wygląda tak jak tabela nr. **??**.

	""	**a**	**b**	**c**	**d**	**e**	**f**
""	0	0	0	0	0	0	0
e	0	0	0	0	0	1	1
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
g	0	1	1	2	2	2	2
d	0	2	2	2	3	3	3

Tabela 2

Na początku zerową kolumnę oraz zerowy wiersz tablicy T wypełniamy zerami. W każdym kroku algorytmu, wypełniając $T[i, j]$ ($i \in \{0, 1, \ldots, n\}$, $j \in \{0, 1, \ldots, m\}$) będziemy analizowali $T[i-1, j]$, $T[i, j-1]$ oraz $T[i-1, j-1]$ wg. (★).

Algorithm 6 Rozwiązanie zadania 1.2

```
1:  procedure SubsetSum(A = a_1, a_2, ..., a_n, B = b_1, b_2, ..., b_m)
2:      Utwórz tablicę dwuwymiarową T o n+1 wierszach i m+1 kolumnach.
3:      Wypełnij zerową kolumnę i zerowy wiersz tablicy T zerami
4:      i ← 1
5:      while i ⩽ n do
6:          j ← 1
7:          while j ⩽ m do
8:              if a_i = b_j then
9:                  T[i, j] = T[i − 1, j − 1] + 1
10:             else
11:                 T[i, j] = max{T[i − 1, j], T[i, j − 1]}
12:             j ← j + 1
13:         i ← i + 1
14:     return T[n, m]
```

2.3 Zadanie 3 – Problem łamania tekstu

Treść: W problemie łamania tekstu dany jest tekst składający się z n słów o długościach $d_1, d_2, \ldots, d_n$ oraz szerokość wiersza s. Szukamy rozmieszczenia słów w wierszach w taki sposób, żeby każde słowo mieściło się w całości w wierszu, w żadnym wierszu nie bylo więcej niż s znaków oraz suma kwadratów liczb pozostałych wolnych miejsc we wszystkich wierszach była jak najmniejsza. Zaprojektuj jak najszybszy algorytm rozwiązujący problem łamania tekstu.

2.4 Zadanie 4 – Szukanie odległości edycyjnej napisów

Treść: Przez odległość edycyjną napisów x i y rozumiemy najmniejszą liczbę operacji wstawienia lub usunięcia pojedynczego znaku, które pozwalają przekształcić napis x w napis y. Zaprojektuj algorytm, który znajdzie odległość edycyjną dwóch zadanych napisów.

Rozwiązanie: Na początek warto wspomnieć, że dodanie jeszcze jednej operacji - zamiany znaków z dwóch różnych słów powoduje powstanie metryki którą nazywamy odległością Levenshtein'a.

Rozważmy najpierw naiwne podejście rekurencyjne: badamy oba napisy od tyłu, oznaczmy m jako długość napisu x oraz n jako długość napisu y.

1. Jeśli znaki są takie same to wywołujemy rekurencyjnie nasz algorytm dla napisu x skróconego do $m-1$ oraz napisu y skróconego do $n-1$.

2. Jeśli znaki są różne to robimy dwa wywołania rekurencyjne - jedno dla usunięcia znaku z napisu x a drugie dla dodania znaku z napisu x, po to by wybrać z tych dwóch wywołań mniejszy koszt.

Podejście programowania dynamicznego: Tworzymy tablicę $T[,]$ o $m+1$ kolumnach oraz $n+1$ wierszach gdzie kolumny oznaczają kolejne litery napisu x a wiersze kolejne litery napisu y. $T[i,j]$ będzie oznaczało odległość podnapisów: i-elementowego podnapisu x licząc od początku oraz j-elementowego podnapisu y licząc od początku.

Wtedy możemy wypełniać od lewego górnego rogu tablicy.. to do

Przykład: rozważmy dwa napisy $x = piesek$ oraz $y = kotek$.

	""	**p**	**i**	**e**	**s**	**e**	**k**
""	0	1	2	3	4	5	6
k	1	0	0	0	0	0	0
o	2	0	0	0	0	0	0
t	3	0	0	0	0	0	0
e	4	0	0	0	0	0	0
k	5	0	0	0	0	0	0

Tabela 3

3 Reprezentacje grafów

4 Przeszukiwanie grafów – Zadania

4.1 Zadanie 1 – Struktura danych reprezentująca graf skierowany

Treść: Zaprojektuj strukturę danych reprezentującą graf skierowany o n wierzchołkach i m krawędziach, z następującymi operacjami:

- dodanie krawędzi,
- usunięcie krawędzi,
- sprawdzenie istnienia krawędzi,
- przejrzenie wszystkich krawędzi wychodzących z zadanego wierzchołka v,

która spełnia następujące wymagania:

a) Operacje na pojedynczej krawędzi w czasie $O(logn)$, przejrzenie krawędzi wychodzących z v w czasie $O(deg + (v))$[1], złożoność pamięciowa $O(m)$.

b) Dodawanie krawędzi w czasie $O(1)$, przejrzenie krawędzi wychodzących z v w czasie $O(deg + (v))$, złożoność pamięciowa $O(m)$.

c) Operacje na pojedynczej krawędzi w czasie $O(1)$, przejrzenie krawędzi wychodzących z v w czasie $O(deg + (v))$.

d) Operacje na pojedynczej krawędzi w oczekiwanym czasie $O(1)$, przejrzenie krawędzi wychodzących z v w czasie $O(deg + (v))$, złożoność pamięciowa $O(m)$.

Rozwiązanie

a) Wystarczy zastosować reprezentację listową, w której listy zastępujemy drzewami zrównoważonymi (CC, AVL).

b) Lista nieuporządkowana dla każdego wierzchołka

c) Zastosować reprezentację macierzową, która przechowuje referencję na node w liscie incydencji (mamy liste incydencji gdzie sa wartosci i tablice referencji)

d) Tablica haszująca o m elementach (haszowanie łańcuchowe)

4.2 Zadanie 2 – Nierekurencyjne przeszukiwanie w głąb (DFS)

Treść: Zaproponuj nierekurencyjną implementację przeszukiwania grafu w głąb.

Rozwiązanie: Poniższy algorytm działa zarówno dla grafów skierowanych jak i prostych.

Algorithm 7 Rozwiązanie zadania 2

```
1: procedure DFSNR(G - graf, v - wierzchołek startowy)
2:     Zainicjalizuj stos (last in, first out) S
3:     Zainicjalizuj tablicę bool'i T o |V(G)| elementach wartością false
4:     S.Push(v)
5:     T[v] ← false
6:     while S.IsEmpty() = false do
7:         x ← S.Pop()
8:         while y ∈ N(x) do
9:             if T[y] = false then
10:                S.Push(y)
11:                T[y] ← true
```

Wersja rekurencyjna: Zakładamy, że dysponujemy globalną tablicą bool'i T o $|V(G)|$ elementach zainicjowaną wartością *false*.

Algorithm 8 Algorytm przeszukiwania DFS - wersja rekurencyjna

```
1: procedure DFS(G - graf, v - wierzchołek startowy)
2:     while x ∈ N(v) do
3:         if T[x] = false then
4:             T[x] ← true
5:             DFS(G, x)
```

4.3 Zadanie 3 – Implementacja przeszukiwania wszerz (BFS)

Treść: Zaproponuj implementację przeszukiwania grafu wszerz.

Rozwiązanie: Poniższy algorytm działa zarówno dla grafów skierowanych jak i prostych.

Algorithm 9 Rozwiązanie zadania 2

```
1: procedure BFS(G - GRAF, v - WIERZCHOŁEK STARTOWY)
2:     Zainicjalizuj kolejkę (first in, first out) Q
3:     Zainicjalizuj tablicę bool'i T o |V(G)| elementach wartością false
4:     Q.PushBack(v)
5:     T[v] ← false
6:     while Q.IsEmpty() = false do
7:         x ← Q.PopFront()
8:         while y ∈ N(x) do
9:             if T[y] = false then
10:                Q.PushBack(y)
11:                T[y] ← true
```

4.4 Zadanie 4 – Szukanie liczby składowych spójności grafu

Treść: konstruuj algorytm znajdujący liczbę składowych spójności zadanego grafu w czasie $O(m)$

Rozwiązanie: Stosujemy BFS lub DFS z dowolnego wierzchołka i zapisujemy odwiedzone wierzchołki (tablica bool). Jeżeli po wyszukiwaniu istnieje nieodwiedzony wierzchołek to wykonujemy na nim BFS lub DFS ponownie. Powtarzamy aż wszystkie wierzchołki będą odwiedzone.

Liczba wywołań BFS lub DFS to liczba składowych w grafie G.

4.5 Zadanie 5 – Sprawdzanie dwudzielności grafu

Treść: Skonstruuj algorytm sprawdzający, czy zadany graf o m krawędziach jest dwudzielny, w czasie $O(m)$.

Rozwiązanie: Zauważmy, najpierw, że graf G jest dwudzielny $\Leftrightarrow$ istnieje 2-kolorowanie poprawne grafu G (dowód w prawą stronę - kolorujemy wierzchołki obu zbiorów dwudzielności na przeciwne kolory, dowód w lewą stronę - kolorujemy poprawnym kolorowaniem, niebieskie wierzchołki wyznaczają jeden zbiór, a czerwone drugi zbiór dwudzielności).

Aby rozwiązać ten problem stosujemy kolorowanie zachłanne z przeszukiwaniem BFS (dowód że to działa - to do). Podczas wyszukiwania, wierzchołek od którego zaczynamy wyszukiwanie kolorujemy bez straty ogólności na czerwono, natomiast jego wszystkich sąsiadów na niebiesko.

Powtarzamy wywołanie BFS tak długo aż nie będzie wierzchołków niepokolorowanych.

4.6 Zadanie 6 – Sortowanie topologiczne

Treść: Skonstruuj algorytm znajdujący sortowanie topologiczne acyklicznego grafu skierowanego o m krawędziach w czasie $O(m)$. Przez sortowanie topologiczne rozumiemy

uporządkowanie wierzchołków w taki sposób, że dla każdej krawędzi uv, wierzchołek u znajduje się przed wierzchołkiem v

Rozwiązanie: Zakładamy, że dysponujemy globalną tablicą bool'i T o $|V(G)|$ elementach zainicjowaną wartością *false*.

Ponadto zakładamy, że mamy globalną listę L, w której będą znajdować się posortowane wierzchołki.

Algorithm 10 Rozwiąznie zadania 5

```
1: procedure TSort(G - graf, v - wierzchołek startowy)
2:     while x ∈ N(v) do
3:         if T[x] = false then
4:             T[x] ← true
5:             TSort(G, x)
6:     L.PushFront(x)
```

4.7 Zadanie 7 – Szukanie cyklu Eulera

Treść: Skontruuj algorytm znajdujący cykl Eulera w zadanym grafie o m krawędziach w czasie $O(m)$. **Rozwiązanie:**

5 Problem najkrótszej ścieżki

5.1 Problem najkrótszej ściezki w grafach nieważonych

5.1.1 Wykorzystanie DFS dla drzew

5.1.2 Wykorzystanie BFS dla dowolnych grafów

5.2 Problem najkrótszej ściezki w grafach ważonych

W tym rozdziale poprzez *spacer* będziemy rozumieli dowolny ciąg wierzchołków v_0, v_1, ... v_n, w którym dla każdego $i \in \{0, 1, n-1\}$, istnieje krawędź (może być skierowana) v_iv_{i+1}. Poprzez *ścieżkę* będziemy rozumieli dowolny spacer, w którym żadne wierzchołki nie powtarzają się.

Lemat 5.1. *Niech (G, ω) będzie skierowanym grafem ważonym bez ujemnych cykli oraz niech $u, v \in V(G)$ będą takie, że $u \neq v$. Wtedy każdy spacer od u do v ma wagę nie mniejszą niż waga najkrótszej ścieżki od u do v.*

Dowód. Przypuśćmy, że istnieje taki u-v-spacer S, którego waga jest mniejsza niż waga najkrótszej u-v-ściezki P.

Zauważmy, że spacer S nie może być ścieżką, bo wtedy P nie jest najkrótszą ścieżką co byłoby sprzeczne z doborem P.

Skoro, S nie jest ścieżką, to musi zawierać w sobie $k \geqslant 1$ cykli C_1, C_2, ..., C_k.

Niech Q to ścieżka utworzona z S w taki sposób, że każdy cykl jest pominięty. (np. dla $S = uabcdbgv$, $Q = uabgv$)

Zauważmy, że

$$\omega(P) > \omega(S) \geqslant \omega(Q),$$

gdzie pierwsza nierównwność to założenie o tym, że S jest mniejszego kosztu, a druga wynika, z faktu że w G nie ma ujemnych cykli.

Ale przecież, założyliśmy, że P jest najkrótszą ścieżką, co prowadzi do sprzecności.

□

Lemat 5.2. *Niech (G, w) będzie skierowanym grafem ważonym bez ujemnych cykli oraz niech $u, v \in V(G)$, będą takie, że $u \neq v$. Niech P to najkrótsza ścieżka od u do v. Oznaczmy przez x przedostatni wierzchołek P oraz przez P' fragment ścieżki P od u do x.*

Wówczas śieżka P' jest najkrótszą ścieżką od u do x.

Dowód. Przypuśćmy, że tak nie jest, tzn., że istnieje u-x-ścieżka P'', taka, że $\omega(P'') < \omega(P')$ (waga ścieżki P'' jest mniejsza niż waga ścieżki P').

Zdefiniujmy u-v-spacer R jako $P'' + vx$ (tzn. ścieżka P'' z krawędzią vx dodaną na końcu). Wtedy prawdą jest, że

$$\omega(R) = \omega(P'') + \omega(vx) < \omega(P') + \omega(vx) = \omega(P),$$

zatem udało nam się skonstruować u-v-spacer R, który ma wagę mniejszą niż najmniejsza u-v-ścieżka P, co jest sprzeczne z lematem **??**. □

5.2.1 Algorytm Bellmana-Forda

Algorytm Bellmana-Forda służy do znajdowania najkrótszej ścieżki w grafie skierowanym ważonym (G, w), nieposiadającym cykli o ujemnej długości, przy dowolnych wagach krawędzi, z wierzchołka startowego $s \in V(G)$ do dowolnego osiągalnego wierzchołka.

Algorithm 11 Algorytm Bellmana-Forda

```
1:  procedure BellmanFord((G, w), s ∈ V(G)))
2:      odległość = tablica liczb, rozmiaru V[G]
3:      for v ∈ V(G) do
4:          odległość[v] = ∞
5:      odległość[s] = 0
6:      for i = 1, 2, ..., n − 1 do
7:          for uv ∈ E(G) do
8:              if odległość[v] > odległość[u] + w(uv) then
9:                  odległość[v] = odległość[u] + w(uv)
10:     return odległość
```

Powyższą implementację można rozbudować o znajdowanie (i zwracanie) najkrótszych ścieżek (algorytm **??**).

Złożoność pesymistyczna powyższego algorytmu to $O(nm)$. Najgorsza złożoność, czyli $O(n^3)$ zajdzie dla grafów gęstych lub dla dowolnych grafów zareprezentowanych z użyciem reprezentacji macierzowej, co wynika z potrzeby przejścia po całej macierzy sąsiedztwa w pętli w linii nr. 7.

Twierdzenie 5.1. *(Poprawność algorytmu Bellmana-Forda) Jeśli (G, w) jest ważonym grafem skierowanym bez ujemnych cykli oraz $s \in V(G)$, to algorytm* **??** *poprawnie rozwiązuje problem najkrótszej ścieżki.*

Dowód. Dla wierzchołka $v \in V(G)$ znaczmy przez $d(v)$ odległość w grafie (G, w) od s do v. Ponadto, dla dowolnej liczby naturalnej k i wierzchołka $v \in V(G)$ niech $d^{(k)}(v)$ oznacza minimalną długość spaceru o co najwyżej k krawędziach od s do v. W przypadku kiedy s-v-spacer nie istnieje przyjmujemy, że $d(v), d^{(k)}(v)$ są nieskończone.

Zaczniemy od pokazania, że odległości od wierzchołka s wyznaczane przez algorytm nigdy nie są mniejsze niż faktyczne odległości w (G, w).

Stwierdzenie: W każdym momencie działania algorytmu dla każdego wierzchołka $v \in V(G)$ zachodzi

$$\text{odległość}[v] \geqslant d(v).$$

Dowód stwierdzenia: Wykorzystujemy indukcję matematyczną po liczbie wywołań linii nr. 9 algorytmu.

Zauważmy, że dla zerowej liczby wywołań teza jest spełniona, bo tablica odległości jest wypełniona nieskończonościami (formalnie: liczbami większymi niż największa waga w (G, ω)) oraz $\text{odległości}[s] = d(s) = 0$, co oznacza, że baza indukcyjna jest spełniona.

Przypuśćmy, że stwierdzenie jest prawdziwe przed pewnym wywołaniem lini nr. 9, co oznacza, że $\text{odległość}[u] \geqslant d(u)$.

Korzystając z lematu **??**, wiemy, że spacer składający się z najkrótszej ścieżki od s do u z dodaną na końcu krawędzią uv jest nie krótszy niż najkrótsza ścieżka od s do v, co możemy wyrazić jako:

$$d(u) + \omega(uv) \geqslant d(v),$$

łącząc z poprzednią nierównością otrzymujemy

$$\text{odległość}[u] + \omega(uv) \geqslant d(v),$$

co gwarantuje nam prawdziwość stwierdzenia po rozważanym wykonaniu linii nr. 9. Na mocy indukcji stwierdzenie musi być prawdziwe.

Z powyższego stwierdzenia wnioskujemy, że OPT $\leqslant$ REZULTAT. Wprowadźmy oznaczenie, że odległość_i to stan tablicy po i iteracjach. Pokażmy prawdziowość następującego niezmiennika.

Niezmiennik: Dla każdego $i \in \{0, 1, \ldots, n-1\}$ po i iteracjach pętli w linii 6 prawdą jest, że dla każdego wierzchołka $v \in V(G)$ zachodzi

$$\text{odległość}_i[v] \leqslant d^{(i)}(v).$$

Dowód niezmiennika: Stosujemy indukcję po liczbie iteracji i.

Zauważmy, że skoro w grafie G nie ma ujemnych cykli to $d(s) = 0$, możemy zapisać, że

$$\text{odległość}_0[s] = 0 = d(s) \leqslant d^{(0)}(s),$$

co oznacza, że baza indukcji ($i = 0$) jest spełniona.

Przypuśćmy, że niezmiennik jest zachowany dla $i = j$, gdzie $j \in \{0, 1, \ldots, n-2\}$, tzn., że po j iteracjach pętli w linii nr. 6 dla każdego wierzchołka v zachodzi

$$\text{odległość}_j[v] \leqslant d^{(j)}(v).$$

Ustalmy dowolny wierzchołek v. Pokażemy, że $\text{odległość}_{j+1}[v] \leqslant d^{(j+1)}(v)$. Jeśli $\text{odległość}_j[v] \leqslant d^{(j+1)}(v)$ to koniec, ponieważ wartości elementów tablicy odległość nie mogą się zwiększać w trakcie działania algorytmu.

Przypuśćmy, że $\text{odległość}_j[v] > d^{(j+1)}(v)$. Niech P będzie najkrótszą ścieżką od s do v o nie więcej niż $j+1$ krawędziach, x przedostatnim wierzchołkiem P, a P' fragmentem tej ścieżki zaczynającym się w s i kończącym się w x, wtedy

$$\omega(P) = d^{(j+1)}(v) \wedge \omega(P') = d^{(j)}(x),$$

gdzie drugi fakt wynika z lematu **??** (założenie o maksymalnej liczbie krawędzi nie unieważnia tego lematu). Korzystając z założenia indukcyjnego

$$\text{odległość}_j[x] \leqslant d^{(j)}(x) = \omega(P'),$$

dodając $\omega(xv)$ obustronnie, otrzymamy

$$\text{odległość}_j[x] + \omega(xv) \leqslant \omega(P') + \omega(xv) = \omega(P) = d^{(j+1)}(v).$$

Zauważmy, że z założenia $\text{odległość}_j[v] > d^{(j+1)}(v)$, wynika, że warunek linii nr. 8 zostanie spełniony, a więc napewno wykona się linia nr. 9, czyli

$$\text{odległość}_{j+1}[v] \leqslant \text{odległość}_j[x] + \omega(xv),$$

więc ostatecznie

$$\text{odległość}_{j+1}[v] \leqslant d^{(j+1)}.$$

Na mocy indukcji matematycznej niezmiennik jest prawdziwy.

Z powyższego niezmiennika wnioskujemy, że $\text{odległość}_{n-1}[v] \leqslant d^{(n-1)}(v) = d(v)$ dla dowolnego $v \in V(G)$, co daje REZULTAT $\leqslant$ OPT, więc ostatecznie OPT = REZULTAT, co kończy dowód.

□

5.2.2 Algorytm Dijkstry

Algorytm Dijkstry stanowi algernatywę do algorytmu Bellmana-Forda, pod warunkiem, że graf wejściowy jest grafem (skierowanym) ważonym (G, ω), gdzie każda krawędź ma nieujemną wagę. Tak samo jak algorytm Belmmana-Forda, Algorytm Dijkstry znajduje wszystkie ścieżki do osiągalnych wierzchołków z wierzchołka wejściowego s.

Założenie o nieujemnych wagach krawędzi pozwala na sformuowanie poniższego lematu.

Lemat 5.3. *Jeśli (G, w) jest ważonym grafem skierowanym bez ujemnych wag krawędzi oraz $P = v_1 v_2 \dots v_k$ jest najkrótszą ścieżką od v_1 do v_k w G, wówczas odległość w grafie G od v_1 do v_{k-1} jest nie większa niż odległość od v_1 do v_k.*

Dowód. Rozważmy pewną najkrótszą ścieżkę $P = v_1 v_2 \dots v_k$ w G. Na mocy lematu **??** $P' = v_1 v_2 \dots v_{k-1}$ jest najkrótszą ścieżką od v_1 do v_{k-1}. Zatem $\omega(P) = \omega(P') + v_1 v_{k-1}$, więc z założenia, że wagi są nieujemne wynika, że $\omega(P) \geqslant \omega(P')$. □

Zauważmy, że lemat **??** implikuje, że dowolna podścieżka pewnej ścieżki P musi mieć wagę nie większą niż ścieżka P.

Algorithm 12 Algorytm Dijkstry

```
1:  procedure DIJKSTRA((G, ω), s ∈ V(G)))
2:      Niech odległość to tablica liczb, rozmiaru V[G]
3:      Niech Q to pusta kolejka priorytetowa
4:      for v ∈ V(G) do
5:          odległość[v] ← ∞
6:      odległość[s] ← 0
7:      Q.Push(s, odległość[s])
8:      while Q.IsEmpty = False do
9:          u = Q.ExtractMin()
10:         for v ∈ N(u) do
11:             if odległość[u] + ω(uv) < odległość[v] then
12:                 odległość[v] ← odległość[u] + ω(uv)
13:                 Q.DecreaseKey(v, odległość[v]);
14:     return odległość
```

Algorytm **??**, tak samo jak algorytm Belmmana-Forda, możemy rozbudować o zwracanie ścieżek.

Zakładając, że kolejka priorytetowa jest oparata na kopcach Fibonacciego otrzymujemy złożoność $O(n \log n + m)$.

Twierdzenie 5.2. *(Poprawność algorytmu Dijkstry). Jeśli (G, ω) jest ważonym grafem (skierowanym) bez ujemnych wag krawędzi, wówczas algorytm Dijkstry, dla danych (G, ω) i dowolnego $s \in V(G)$, poprawnie rozwiązuje problem najkrótszej ścieżki.*

Dowód. Przez $d(v)$ będziemy oznaczać odległość w grafie (G, w) od s do v. Na potrzeby notacji będziemy utożsamiać kolejkę Q ze zbiorem wierzchołków, które się w niej znajdują (i na przykład zapis $v \in Q$ będzie dla nas oznaczał, że wierzchołek v znajduje się w kolejce). Poprawność algorytmu wyniknie z poniższego niezmiennika.

Niezmiennik: Na początku oraz po każdej iteracji pętli w linii nr. 8 prawdziwe są następujące własności:

(a) Dla każdego wierzchołka $w \notin Q$ zachodzi odległość$[w] = d(w)$

(b) Dla każdego wierzchołka $w \in Q$, odległość$[w]$ jest długością najkrótszej ścieżki od s do w, której pierwszy wierzchołek i wszystkie wierzchołki wewnętrzne nie znajdują się w Q

Dowód niezmiennika: Indukcja po iteracjach pętli w linii nr. 8.

Warunki (a), (b) są spełnione przed pierwszą iteracją pętli, ponieważ, Q nie posiada żadnych wierzchołków poza kolejką Q, co oznacza, że baza indukcyjna jest spełniona.

Od teraz poprzez tablicę odległość będziemy rozumieli stan tablicy przed wykonaniem się i-tej iteracji, natomiast przez odległość$'$ po wykonaniu się i-tej iteracji. Tak samo Q to stan kolejki przed i-tą iteracją, a Q' to stan po tej iteracji.

Chcemy pokazać, że jeśli Q oraz odległość spełniają warunki (a) i (b), to Q' oraz odległość$'$ spełniają warunki (a$'$) i (b$'$).

Najpierw pokażemy, że (a$'$), czyli, że dla każdego wierzchołka $w \notin Q'$ zachodzi odległość$'[w] = d(w)$. Niech u to wierzchołek zdjęty z kolejki priorytetwowej w linijce nr. 9. Zauważmy, że odległość$'[u]$ = odległość$[u]$. Aby wykazać prawdziwość tego warunku wystarczy pokazać, że odległość$[u] = d(u)$, ponieważ u to jedyny wierzchołek, który został zdjęty z kolejki (wszystkie pozostałe wierzchołki spełniają (a$'$) na mocy (a)).

Przypuśćmy, że istnieje s-u-ścieżka P taka, że $\omega(P) <$ odległość$[u]$. Z (b) wynika, że odległość$[u]$ jest odległością najmniejszej ścieżki wzdłuż wierzchołków z poza Q, co oznacza, że P musi mieć co najmniej jeden wierzchołek w Q, który nie jest wierzchołkiem u. Oznaczmy pierwszy wierzchołek z Q na ścieżce P jako x.

Z (b) wynika, że $\omega(sPx)$ = odległość$[x]$, natomiast z warunku kolejki, wiemy, że odległość$[x] \geqslant$ odległość$[u]$. Aby warunek $\omega(P) <$ odległość$[u]$ był spełniony w (G, ω) musiałyby istnieć krawędzie z ujemnymi wagami (z lematu **??** implikujemy, że dowolna podścieżka P o początku w s musi mieć nie większą wagę niż ścieżka P) co jest sprzeczne z założeniem, czyli odległość$[u] = d(v)$ co oznacza, że (a$'$) jest spełnione.

Teraz pokażemy, że (b$'$), czyli, że dla każdego wierzchołka $w \in Q'$, odległość$'[w]$ jest równa minimalnej długości ścieżki z s do w, której pierwszy wierzchołek i wszystkie wierzchołki wewnętrzne nie znajdują się w Q'. Ponownie przez u oznaczamy wierzchołek zdjęty z kolejki priorytetwowej w linijce nr. 9.

Z przebiegu pętli, możemy wyodrębnić następujące przypadki:

1. $w \notin N(u) \Rightarrow$ odległość$'[w]$ = odległość$[w]$, a więc (b) implikuje (b'), bo nie powstaje żadna nowa ścieżka, której pierwszy wierzchołek i wszystkie wierzchołki wewnętrzne nie znajdują się w Q

2. $w \in N(u) \Rightarrow$ powstaje dokładnie jedna nowa ścieżka P, której pierwszy wierzchołek i wszystkie wierzchołki wewnętrzne nie znajdują się w Q. Z (a′) wynika, że odległość′$[u] = d(u)$, więc

$$\omega(P) = d(u) + \omega(uw) = \text{odległość}'[u] + \omega(uw) = \text{odległość}[u] + \omega(uw).$$

Algorytm w linijce nr 11. dokonuje porównania, które skutkuje zapisaniem $\omega(P)$, w przypadku kiedy możliwe jest poprawienie odległości dla w. Z faktu, że P to jedyna taka ścieżka, która może poprawić odległość implikujemy, że (b′) jest spełnione

Na mocy indukcji matematycznej niezmiennik musi być prawdziwy.

Zauważmy, że po ostatniej iteracji głównej pętli kolejka Q musi być pusta, zatem część (a) niezmiennika pociąga za sobą poprawność algorytmu. □

5.2.3 Algorytm A*

Algorytm A* działa w sposób analogiczny do algoyrtmu Dijkstry, ale służy do znajdywania ścieżki pomiędzy dwoma zadanymi wierzchołkami s i t, a nie jak w przypadku algorytmu Dijkstry pomiędzy s i wszystkimi innymi wierzchołkami w grafie.

Algorytm A* wykorzysuje heurystykę h (jedna z wartości wejściowych), gdzie dla wierzchołka $v \in G$, $h(v)$ definiujemy jako oszacowanie na odległość od v do t. Założenia danych wejściowych są następujące

- nieujemne wagi krawędzi,
- dla każdego wierzchołka v zachodzi $h(v) \leqslant \text{dist}(v, t)$,
- dla każdej krawędzi uv zachodzi $h(u) \leqslant h(v) + \omega(uv)$ (warunek zgodności).

Ideą algorytmu jest przeszukiwanie wierzchołków v w kolejności rosnącego $\text{dist}(s, v)+h(v)$.

Algorithm 13 Algorytm A*

```
1: procedure AStar((G, ω), s, t, h)
2:     Niech odległość to tablica liczb, rozmiaru V[G]
3:     for v ∈ V(G) do
4:         odległość[v] ← ∞
5:     odległość[s] ← 0
6:     Q - kolejka priorytetowa inicjalizowana zbiorem {V(G)}, gdzie priorytet każdego
       wierzchołka v to odległość[v] + h(v)
7:     while Q.IsEmpty = False do
8:         u = Q.ExtractMin()
9:         if u = t then
10:            break
11:        for v ∈ N(u) do
12:            if odległość[u] + ω(uv) < odległość[v] then
13:                odległość[v] ← odległość[u] + ω(uv)
14:                Q.DecreaseKey(v, odległość[v] + h(v))
15:    return odległość[t]
```

Zauważmy, że w przypadku kiedy $h(v) = 0$ dla każdego $v \in G$ to algorytm A* jest równoważny algorytmowi Dijkstry, co oznacza, że złożoność pesymistyczna tego algorytmu jest taka sama jak złożoność pesymistyczna algorytmu Dijsktry.

Okazuje się, że przy dobrej heurystyce, mamy szansę na przeszukanie $o(n)$ wierzchołków. Przebieg konstruowania dobrej heurystyki, zależy od problemu z którym się mierzymy. Przykładowo, wyobraźmy sobie, że szukamy odległości z miasta A do miasta B na mapie samochodowej. Każde połączenie jakąś drogą jest krawędzią z wagą oznaczającą liczbę kilometrów, natomiast każde miasto to wierzchołek grafu. Jako heurystykę w tym problemie możemy przyjąć odległość euklidesową od każdego miasta do miasta docelowego, jako że droga pomiędzy miastami nie może mieć mniejszej długości niż odległość euklidesowa, oba warunki heurystyki są spełnione.

Twierdzenie 5.3. *(Poprawność algorytmu A*) Jeśli (G, w) jest ważonym grafem skierowanym z wagami nieujemnymi, $s, t \in V(G)$, oraz heurystyka $h(v)$ spełnia następujące założenia:*

- *dla każdego wierzchołka v zachodzi $h(v) \leqslant dist(v, t)$,*
- *dla każdej krawędzi uv zachodzi $h(u) \leqslant h(v) + \omega(uv)$,*

to algorytm **??** *poprawnie rozwiązuje problem najkrótszej ścieżki pomiędzy wierzchołkami s i t.*

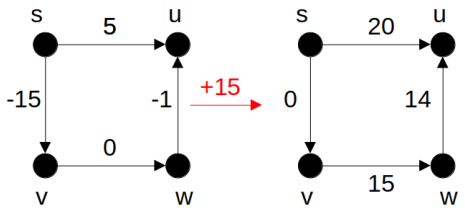
Dowód. to do (nie obowiązuje na kolokwium) □

5.2.4 Algorytm Johnsona

Algorytmy Johnsona jak i Floyda-Warshalla różnią się od poprzedników, tym, że ścieżki znajdowane są pomiędzy każdą parą wierzchołków z tego grafu.

Algorytm Johnsona opiera się na sprytnym wykorzystaniu algorytmu Belmmana-Forda oraz algorytmu Dijkstry. Na wejściu przyjmowane są grafy (skierowane) o dowolnych wagach, bez cyklów o ujemnej długości. Ideą tego algorytmu jest skonstruowanie równoważnego grafu o nieujemnych wagach na krawędziach, w celu zastosowania algorytmu Dijsktry.

W tym miejscu warto zaznaczyć, że skonstruowanie takiego grafu nie może odbyć się przez zwyczajne dodanie do wagi każdej krawędzi, wartości $|\min_{v \in V(G)} \omega(v)|$. Na rysunku (**??**) znajduje się kontrprzykład - w grafie wejściowym optymalna s-u-ścieżka to $svwu$, natomiast w grafie wyjściowym, otrzymanym po dodaniu $|\min_{v \in V(G)} \omega(v)| = 15$ do każdej wagi, optymalną s-u-ścieżką jest su.



Rys. 3

Konstrukcja grafu równoważnego (G', ω') do grafu wejściowego (G, ω) w algorytmie Johnsona polega na dodaniu do G wierzchołka q, oraz dla każdego wierzchołka $v \in V(G)$ krawędzi skierowanych qv z wagą 0.

W następnym kroku wywołujemy algorytm Belmmana-Forda na grafie (G', ω') z wierzchołkiem startowym q, po to aby utworzyć nową funkcję wagową, na G, która zawsze będzie przyjmować wartości nieujemne, co umożliwi zastowanie algorytmu Dijsktry.

Algorithm 14 Algorytm Johnsona

1: **procedure** JOHNSON(G, ω)
2: Niech $V' = V(G) \cup \{q\}$ oraz $E' = E(G) \cup \{qv : v \in V(G)\}$
3: Skonstruuj graf skierowany (G', V')
4: Zdefiniuj funkcję $\omega' : E' \to \mathbb{R}$, gdzie

$$\omega'(e) = \begin{cases} \omega(e), \text{ jeśli } e \in E(G) \\ 0, \text{ jeśli } e \notin E(G) \end{cases}$$

5: $h = \text{BellmanFord}((G', \omega'), q)$
6: Zdefiniuj funkcję $\omega'' : E(G) \to \mathbb{R}$, gdzie

$$\omega''(uv) = \omega(uv) + h(u) - h(v)$$

7: **for** $u \in V(G)$ **do**
8: $d_u = \text{Dijsktra}((G, \omega''), u)$
9: **for** $v \in V$ **do**
10: odległości$[u, v] = d_u[v] - h[u] + h[v]$
11: **return** odległości

Zakładając, że kolejka priorytetowa zastosowana w algorytmie Dijsktry jest oparata na kopcach Fibonacciego otrzymujemy złożoność $O(n^2 \log n + nm)$. Algorytm ten jest szybszy niż później omawiany algorytm Floyda-Warshalla, dla graów rzadkich.

Twierdzenie 5.4. *(Poprawność algorytmu Johnsona) Jeśli (G, ω) jest ważonym grafem skierowanym bez ujemnych cykli, to algorytm Johnsona dla danych wejściowych (G, ω) poprawnie rozwiązuje problem najkrótszej ścieżki pomiędzy dowolnymi parami wierzchołków.*

Dowód. Przez $dist_\chi(u, v)$ oznaczmy wagę minimalnej ścieżki z wierzchołka u do v przy ważeniu χ domyślając się, że chodzi o graf G lub G' w zależności od dziedziny χ.

Aby wykazać poprawność aglorytmu Johnsona musimy udowodnić poniższe obserwacje:

1. Dane wejściowe do algorytmu Bellmana-Forda są poprawne.
2. Dane wejściowe do algorytmu Dijkstry są poprawne.
3. Niech P to dowolna u-v-ścieżka w grafie G oraz niech h to odwzorowanie zwrócone przez algorytm Belmmana-Forda w linii nr. 5, wtedy

$$\omega''(P) = \omega(P) + h(u) - h(v).$$

4. Niech $u, v \in V(G)$, wtedy

$$dist_\omega(u, v) = dist_{\omega''}(u, v) - h(u) + h(v).$$

Obserwacja 1. jest prawdziwa, ponieważ (G, ω) z założenia nie posiada ujemnych cykli oraz dodanie do G wierczchołka q z krawędziami skierowanymi od q do każdego wierzchołka z G nie może utworzyć żadnego nowego cyklu, a więc w szczególności nie może w ten sposób powstać żaden ujemny cykl, co oznacza, że (G', ω') oraz q to poprawne dane wejściowe do algorytmu Bellmana-Forda.

Dowód obserwacji 2: Niech $uv \in E(G')$, wtedy prawdą jest, że

$$dist_{\omega'}(q, v) \leqslant dist_{\omega'}(q, u) + \omega'(uv),$$

co w połączeniu z poprawnością algorytmu Belmmana-Forda (tw. **??**) oraz faktem, że $uv \in E(G)$ daje nam

$$h(v) \leqslant h(u) + \omega(uv).$$

Po przeniesieniu $h(v)$ na prawą stronę otrzymujemy

$$0 \leqslant \omega(uv) + h(u) - h(v) = \omega''(uv),$$

więc każda krawędź grafu ważonego (G, ω'') ma nieujemną wagę, co oznacza, że wywołanie algorytmu Dijkstry jest poprawne.

Dowód obserwacji 3: Niech $P = v_1 v_2 \dots v_k$, gdzie $v1 = u$ oraz $v_k = v$ to dowolna u-v-ścieżka w grafie G, wtedy

$$\omega''(P) = \omega''(v_1v_2) + \omega''(v_2v_3) + \cdots + \omega''(v_{k-1}v_k),$$

korzystając z definicji ω'' otrzymamy

$$\omega''(P) = \omega(v_1v_2) + h(v_1) - (h_2) + \omega(v_2v_3) + h(v_2) - (h_3) + \cdots + \omega(v_{k-1}v_k) + h(v_{k-1}) - (h_k),$$

po uporządkowaniu powyższego dostaniemy

$$\omega''(P) = \omega(P) + \omega(v_1) - \omega(v_k) = \omega(P) + h(u) - h(v),$$

co należało dowieść.

Dowód obserwacji 4: Niech P_ω oraz $P_{\omega''}$ to najkrótsze u-v-ścieżki kolejno przy ważeniu ω oraz ω''. Korzystając z Obserwacji 3 otrzymujemy

$$\omega''(P_\omega) = \omega(P_\omega) + h(u) - h(v) \leqslant \omega(P_{\omega''}) + h(u) - h(v) = \omega''(P_{\omega''}),$$

co wynika, z założenia, że P_ω to najmniejsza ścieżka w (G, ω).

Z powyższej nierówności oraz z założenia, że $P_{\omega''}$ to najmniejsza ścieżka w (G, ω'') otrzymujemy

$$\omega''(P_{\omega''}) \leqslant \omega''(P_\omega) \leqslant \omega''(P_{\omega''}),$$
$$\omega''(P_{\omega''}) = \omega''(P_\omega),$$

a zatem po ponownym zastosowaniu obserwacji 3

$$\omega''(P_{\omega''}) = \omega(P_\omega) + h(u) - h(v),$$
$$\omega(P_{\omega''}) = \omega''(P_{\omega''}) - h(u) + h(v),$$

co dzięki poprawności algorytmu Dijkstry (tw. **??**), daje nam

$$dist_\omega(u, v) = dist_{\omega''}(u, v) - h(u) + h(v).$$

Obserwacje 1, 2 i 4 implikują, że tablica odległości zostanie wypełniona poprawnie. Jako, że algorytm na pewno zakończy pracę dowód poprawności jest kompletny. □

5.2.5 Algorytm Floyda-Warshalla

Algorithm 15 Algorytm Floyda-Warshalla

```
1:  procedure FloydWarshall(G, ω)
2:      Utwórz tablicę odległości o wymiarach V(G) × V(G)
3:      for i ∈ V(G) do
4:          for j ∈ V(G) do
5:              if ij ∈ E(G) then
6:                  odległości[i, j] = ω(ij)
7:              else
8:                  odległości[i, j] = ∞
9:      for i ∈ V(G) do
10:         odległości[i, i] = 0
11:     for k ∈ V(G) do
12:         for i ∈ V(G) do
13:             for j ∈ V(G) do
14:                 if odległości[i, j] > odległości[i, k] + odległości[k, j] then
15:                     odległości[i, j] = odległości[i, k] + odległości[k, j]
16:     return odległości
```

Złożoność powyższego algorytmu to $O(n^3)$.

Twierdzenie 5.5. *(Poprawność algorytmu Floyda-Warshalla) Jeśli (G, ω) jest ważonym grafem skierowanym bez ujemnych cykli, to algorytm Floyda-Warshalla dla danych wejściowych (G, ω) poprawnie rozwiązuje problem najkrótszej ścieżki pomiędzy dowolnymi parami wierzchołków.*

Dowód. Oznaczmy przez $d^{(m)}$(i, j) długość najkrótszej i-j-ścieżki, której wewnętrzne wierzchołki należą do zbioru $\{0, 1, 2, \ldots, m-1\}$. Ponadto, przez odległości$_m[i, j]$ będziemy rozumieli stan tablicy po wykonaniu się m-tej iteracji.

Niezmiennik: Po l iteracjach pętli for w linii nr. 11, dla każdych wierzchołków $i, j \in V(G)$ zachodzi odległości$[i, j] \leqslant d^{(l)}(i, j)$.

Dowód niezmiennika: Indukcja po liczbie iteracji l.

Wartości początkowe spełniają warunek, zatem baza indukcyjna jest prawdziwa.

Przypuśćmy, że odległości$_l[i, j] \leqslant d^{(l)}(i, j)$, chcemy pokazać, że odległości$_{l+1}[i, j] \leqslant d^{(l+1)}(i, j)$. W przypadku kiedy $d^{(l)}(i, j) = d^{(l+1)}(i, j)$, to

$$d^{(l+1)}(i, j) = d^{(l)}(i, j) \geqslant \text{odległości}_l[i, j] \geqslant \text{odległości}_{l+1}[i, j],$$

gdzie pierwsza nierówność wynika z założenia indukcyjnego, natomiast druga z 14 i 15 linii algorytmu, które implikują, że w każdej iteracji odległości mogą się tylko zmniejszyć lub nie uledz żadnej zmianie.

Rozważmy przypadek, kiedy $d^{(l)}(i,j) > d^{(l+1)}(i,j)$. W takiej sytuacji musiała powstać co najmniej jedna i-j-ścieżka. Każda z nowopowstałych i-j-ścieżek musi zawierać w sobie wierzchołek l, niech P to najkrótsza z nich. Zauważmy, że

$$\omega(P) = d^{(l+1)}(i,j) = d^{(l+1)}(i,l) + d^{(l+1)}(l,j),$$

gdzie druga równość wynika z wielokrotnego zastosowania lematu **??**.

Ponadto, możemy zauważyć, że $d^{(l+1)}(i,l) = d^{(l)}(i,l)$ oraz $d^{(l+1)}(l,j) = d^{(l)}(l,j)$, co wynika z fatku, że najkrótsza ścieżka, której wnętrze składa się z wierzchołków $\{0,1,2,\ldots,l-1\}$, zaczynająca się w l nie być gorsza niż najkrótsza ścieżka której wnętrze składa się z wierzchołków $\{0,1,2,\ldots,l-1,l\}$.

Zatem ostatecznie:

$$\text{odległości}_{l+1}[i,j] \leqslant \text{odległości}_l[i,l] + \text{odległości}_l[l,j] \leqslant$$
$$\leqslant d^{(l)}(i,l) + d^{(l)}(l,j) = d^{(l+1)}(i,l) + d^{(l+1)}(l,j) = d^{(l+1)}(i,j),$$

gdzie pierwsza nierówność wynika z 14 oraz 15 linii algorytmu, natomiast druga - z założenia indukcyjnego.

Na mocy indukcji matematycznej niezmiennik jest prawdziwy.

Stwierdzenie: W każdym momencie działania algorytmu dla każdego wierzchołka $v \in V(G)$ zachodzi

$$\text{odległość}[v] \geqslant d(v).$$

Dowód stwierdzenia: Indukcja po liczbie iteracji l.

Dla zerowej liczby wywołań teza jest spełniona, a zatem baza indukcyjna jest prawdziwa.

Przyjmijmy, że jeśli $\text{odległości}_l[i,j] \geqslant d(i,j)$ jest prawdą to $\text{odległości}_{l+1}[i,j] \geqslant d(i,j)$ również musi być prawdziwe.

Jeśli $\text{odległość}_{l+1}[i,j] = \text{odległość}_l[i,j]$, to korzystając z założenia indukcyjnego możemy zakończyć dowód.

Rozważmy przypadek, kiedy $\text{odległość}_{l+1}[i,j] < \text{odległość}_l[i,j]$. Aby ten przypadek zaszedł musiały wykonać się linijki 14 i 15 co oznacza, że

$$\text{odległość}_{l+1}[i,j] = \text{odległość}_l[i,l+1] + \text{odległość}_l[l+1,j] \geqslant d(i,l+1) + d(l+1,j) \geqslant d(i,j),$$

gdzie pierwsza nierówność wynika, z założenia indukcyjnego, natomiast druga - z faktu, że $d(i,j)$ oznacza długość najkrótszej ścieżki.

Na mocy indukcji matematycznej, stwierdzenie jest prawdziwe.

Stwierdzenie oraz niezmiennik implikują, że dla każdej pary wierzchołków $i,j \in V(G)$ prawdą jest, że

$$\text{odległość}_n[i,j] \leqslant d^{(n)}(i,j) = d(i,j) \leqslant \text{odległość}_n[i,j],$$

a zatem

$$\text{odległość}_n[i,j] = d(i,j),$$

co kończy dowód.

□

5.2.6 Podsumowanie

6 Problem najkrótszej ścieżki – Zadania

6.1 Zadanie 1 – Algorytm Bellmana-Forda z odtwarzaniem ściezki

Treść: Zmodyfikuj algorytm Bellmana-Forda tak, aby znajdował najkrótszą ścieżkę między zadanymi dwoma wierzchołkami.

Rozwiązanie:

Algorithm 16 Algorytm Bellmana-Forda z odtwarzaniem ściezki

```
1: procedure BellmanFord((G, w), s, x ∈ V(G)))
2:     odległość = tablica liczb, rozmiaru V[G]
3:     poprzednik = tablica wierzchołków, rozmiaru V[G]
4:     for v ∈ V(G) do
5:         odległość[v] ← ∞
6:         poprzednik[v] ← −1
7:     odległość[s] ← 0
8:     for i = 1, 2, ..., n − 1 do
9:         for uv ∈ E(G) do
10:            if odległość[v] > odległość[u] + w(uv) then
11:                odległość[v] ← odległość[u] + w(uv)
12:                poprzednik[v] ← u
13:    ścieżka = pusta lista wierzchołków
14:    ściezka.PushFront(x)
15:    while x ← poprzednik[x] ≠ −1 do
16:        ścieżka.PushFront(x)
17:    return ścieżka
```

6.2 Zadanie 2 – Szukanie ujemnego cyklu

Treść: Zaprojektuj algorytm, który znajdzie w ważonym grafie skierowanym ujemny cykl, jeśli taki istnieje. Wskazówka: zmodyfikuj algorytm Bellmana-Forda

Rozwiązanie: Rozwiązanie polega na sprawdzeniu, czy wykonanie jeszcze jednej nadmiarowej iteracji w algorytmie Belmmana-Forda, oraz sprawdzenie czy pewien wierzchołek został poprawiony.

Jeżeli poniższy warunek jest spełniony

$$\exists_{v \in V(G)} \text{odległość}_n[v] < \text{odległość}_{n-1}[v],$$

to w grafie G musi istnieć ujemny cykl. Dzieje się tak ponieważ zawsze istnieje co najmniej jedna krawędź w cyklu ujemnym C, która spełnia warunek

$$\text{odległość}[v] > \text{odległość}[u] + w(uv),$$

zatem będzie wtedy istniał wierzchołek, który zostanie poprawiony.

Jako, że graf wejściowy może być niespójny, musimy jeszcze dodać dodatkowy wierzchołek s, który będzie połączony z każdym innym wierzchołkiem przy pomocy krawędzi skierowanej. Jako że dodane krawędzie to krawędzie skierowane od wierzchołka s, możemy mieć pewność że żadne nowe cykle nie powstaną, a więc w szczególności nie powstaną nowe ujemne cykle.

Algorithm 17 Znajdowanie ujemnego cyklu

```
1:  procedure BellmanFord((G, w)))
2:      G' - graf G z dodanym wierzchołkiem s oraz krawędziami skierowanymi od s do
    każdego innego wierzchołka o wadze 0
3:      odległość = tablica liczb, rozmiaru V[G']
4:      for v ∈ V(G') do
5:          odległość[v] ← ∞
6:      odległość[s] ← 0
7:      for i = 1, 2, ..., n − 1 do
8:          for uv ∈ E(G') do
9:              if odległość[v] > odległość[u] + w(uv) then
10:                 odległość[v] ← odległość[u] + w(uv)
11:     for uv ∈ E(G') do
12:         if odległość[v] > odległość[u] + w(uv) then
13:             return True
14:     return False
```

6.3 Zadanie 3 – Szukanie liczby spacerów o zadanej liczbie krawędzi

Treść: Zaprojektuj algorytm, który dla danego grafu prostego G, wierzchołków $u, v \in V(G)$ i liczby $k \in \{1, 2, \ldots, n\}$ znajdzie liczbę spacerów od u do v o dokładnie k krawędziach.

Rozwiązanie:

6.4 Zadanie 4 – Wyznaczanie odległości w drzewie ważonym

Treść: Zaprojektuj algorytm, ktory dla ważonego acyklicznego grafu skierowanego G oraz wierzchołków $s, t \in V(G)$ wyznaczy odległość od s do t w czasie $O(m)$. Wskazówka: wykorzystaj sortowanie topologiczne (zadanie 1.5)

Rozwiązanie: Dijkstra tylko, że zamiast kolejki Q stosujemy sortowanie topologiczne (średnia złożoność $O(m)$), po czym iter

Algorithm 18 Znajdowanie ujemnego cyklu

```
1: procedure FindDistanceInTree((T, w))
2:     Niech odległość to tablica liczb, rozmiaru V[G]
3:     Niech kolejność to tablica liczb, rozmiaru V[G], na i-tej komórce znajduje się
       wierzchołek, który ma i-te miejsce w posortowanej topologicznie tablicy wierzchołków
4:     kolejność ← TopologicalSort(T)
5:     for i = 0, 1, ... n − 1 do
6:         u ← kolejność[i]
7:         for v ∈ N(u) do
8:             if odległość[u] + ω(uv) < odległość[v] then
9:                 odległość[v] = odległość[u] + ω(uv)
10:    return False
```

6.5 Zadanie 5 – Naiwna wersja algorytmu A*

Treść: Znajdź przykład, dla którego algorytm A* nie zadziała zgodnie z oczekiwaniami, w którym spełnione są wszystkie założenia poza "dla każdej krawędzi uv zachodzi $h(u) \leqslant h(v) + \text{waga}(uv)$". **Rozwiązanie:**

6.6 Zadanie 6 – Poprawność algorytmu A*

Treść: Udowodnij poprawność algorytmu A* i wskaż, w których miejscach wykorzystywane są jego założenia. Wskazówka: zainspiruj się dowodem poprawności algorytmu Dijkstry. **Rozwiązanie:** Dowód twierdzenia **??**.

6.7 Zadanie 7 – Poprawność algorytmu Floyda-Warshalla

Treść: Udowodnij poprawność algorytmu Floyda-Warshalla. Wskazówka: niech $d^{(m)}(i,j)$ będzie długością najkrótszej ścieżki od i do j, której wewnętrzne wierzchołki należą do zbioru $\{0,1,2,...,m-1\}$; udowodnij, że po l iteracjach zewnętrznej pętli dla każdych $i,j \in V(G)$ zachodzi odległość$[i,j] \leqslant d^{(l)}(i,j)$.

Rozwiązanie: Dowód twierdzenia **??**.

7 Przepływy

7.1 Wstęp

Definicja 7.1. *Sieć przepływowa to graf skierowany G z funkcją wag $c : E(G) \to \mathbb{R}_{\geqslant 0} \cup \{\infty\}$ oraz wyróżnionymi wierzchołkami s i t. Mówimy, że*

- *$c(e)$ to przepustowość krawędzi $e \in E(G)$,*
- *s to źródło (ang. source),*

- *t to ujście (ang. target).*

Definicja 7.2. *Przepływem nazywamy funkcję $f : E(G) \to \mathbb{R}_{\geqslant 0}$ taką, że*

- *$f(e) \leqslant c(e)$ dla każdej krawędzi $e \in E(G)$,*
- *$f^+(v) = f^-(v)$ dla każdego wierzchołka $v \in V(G) \setminus \{s, t\}$,*

gdzie $f^+(v) := \sum_{u:vu \in E(G)} f(vu)$ oraz $f^-(v) := \sum_{u:uv \in E(G)} f(uv)$, tak samo jeśli $S \subset V(G)$, to $f^+(S) := \sum_{uv \in E(G), u \in S \wedge v \notin S} f(uv)$ oraz $f^-(S) := \sum_{uv \in E(G), u \notin S \wedge v \in S} f(uv)$

Definicja 7.3. *Wartością przepływu nazywamy różnicę $val(f) = f^+(s) - f^-(s)$.*

7.2 Problem maksymalnego przepływu

Dana jest sieć przepływowa (G, c, s, t) szukamy przepływu f o maksymalnej możliwej wartości $\text{val}(f)$.

Definicja 7.4. *Sieć rezydualna dla danej sieci przepływowej (G, c, s, t) oraz przepływu f to graf skierowany R o zbiorze wierzchołków $V(G)$, funkcji c_f oraz zbiorze krawędzi równym $\{uv : c_f(uv) > 0\}$, gdzie funkcja c_f jest zdefiniowana następująco*

- *$c_f(uw) = c(uw) - f(uw)$, gdy $uw \in E(G)$ oraz $wu \notin E(G)$ (krawędź w przód)*
- *$c_f(uw) = f(wu)$, gdy $uw \notin E(G)$ oraz $wu \in E(G)$ (krawędź wstecz)*
- *$c_f(uw) = c(uw) - f(uw) + f(wu)$, gdy $uw \in E(G)$ oraz $wu \in E(G)$*

Intuicyjnie $c_f(uw)$ możemy rozumieć jako określenie ile jednostek przepływu jesteśmy w stanie "przepuścić"z u do w, co możemy osiągnąć albo zwiększając przepływ na krawędzi uw albo zmniejszając przepływ na krawędzi wu.

Warto zauważyć, że w sieci rezydualnej rozpatrujemy tylko te krawędzie, dla których przepustowość rezydualna jest większa niż 0.

Definicja 7.5. *Ścieżką powiększającą nazywamy dowolną ścieżką od s do t w sieci rezydualnej.*

Zauważmy, że jeśli znajdziemy ścieżkę powiększającą P w sieci rezydualnej, to wartość przepływu f jesteśmy w stanie powiększyć o $\min_{uw \in E(P)} \{c_f(uw)\}$.

7.2.1 Algorytm Forda-Fulkersona

Algorithm 19 Algorytm Forda-Fulkersona

1: **procedure** FordFulkerson($G, w,\ s,\ t$)
2: Niech f to zerowy przepływ w sieci (G, c, s, t)
3: Utwórz sieć rezydualną (R, c_f) dla wejściowej sieci przepływowej oraz dla f
4: **while** Istnieje ścieżka powiększająca P w R **do**
5: Oznaczmy $P = v_1 v_2 \dots v_k$ (gdzie $v_1 = s$ oraz $v_k = t$)
6: $a \leftarrow \min_{i \in \{1,2,\dots,k-1\}} r(v_i v_{i+1})$
7: Powiększ f o a na wszystkich krawędziach P
8: Uaktualnij (R, c_f) na wszystkich krawędziach P
9: **return** f

Twierdzenie 7.1. *(Twierdzenie Forda-Fulkersona) Przepływ ma maksymalną wartość wtedy i tylko wtedy, gdy nie istnieje ścieżka powiększająca.*

Dowód. MD2 □

Uwaga: zwykle zakładamy, że przepustowości krawędzi są liczbami całkowitymi, ponieważ, wówczas po każdym wykonaniu pętli while, wartość przepływu f zwiększy się przynajmniej o 1, co umożliwia oszacowanie złożoności algorytmu.

Możemy oszacować, że algorytm Forda-Flukersona zakończy się w czasie $O(mf_{\max})$, gdzie przez $f_{\max}$ oznaczamy wartość maksymalnego przepływu. Z tego oszacowania wnioskujemy, że dla sieci, w których maksymalna wartość przepływu jest duża, nie warto stosować tego algorytmu.

7.2.2 Algorytm Edmondsa-Karpa

Algorytm Edmondsa-Karpa różni się od algorytmu Forda-Fulkersona tylko tym, że w pętli while operujemy na najkrótszej ścieżce powiększającej zamiast na dowolnej.

Okazuje, się że taka modyfikacja daje nam gwarancję, że algorytm zakończy się w czasie wielomianowym.

Algorithm 20 Edmonds-Karp

1: **procedure** EdmondsKarp($G, w,\ s,\ t$))
2: Niech f to zerowy przepływ w sieci (G, c, s, t)
3: Utwórz sieć rezydualną (R, c_f) dla wejściowej sieci przepływowej oraz dla f
4: **while** Istnieje ścieżka powiększająca w R **do**
5: Niech $P = v_1 v_2 \dots v_k$ będzie **najkrótszą** ścieżką powiększająca (gdzie $v_1 = s$ oraz $v_k = t$)
6: $a \leftarrow \min_{i \in \{1,2,\dots,k-1\}} r(v_i v_{i+1})$
7: Powiększ f o a na wszystkich krawędziach P
8: Uaktualnij (R, c_f) na wszystkich krawędziach P
9: **return** f

Twierdzenie 7.2. *Algorytm Edmondsa-Karpa dla zadanej sieci przepływowej (G, c, s, t) zakończy się w czasie $O(nm^2)$, gdzie n i m to dopowiednio liczba wierzchołków i liczba krawędzi grafu skierowanego G.*

Dowód. Dla dowolnego przepływu f w sieci (G, c, s, t) i wierzchołków $u, v \in V(G)$ przez $d_f(u, v)$, będziemy oznaczali najmniejszą możliwą liczbę krawędzi na u-v-ścieżce w sieci rezydualnej odpowiadającej przepływowi f (inaczej: odległość od u do v w sensie liczby krawędzi w sieci rezydualnej dla f).

Niech $P_1, P_2, \ldots P_k$ to wszystkie najkrótsze s-v-ścieżki w sieci rezydualnej odpowiadającej przepływowi f, wtedy

$$L_f := |\bigcup_{i \in [k]} E(P_i)|.$$

Stwierdzenie 1: Niech f będzie przepływem przed pewną iteracją pętli w linii nr. 4, a f' przepływem po tej iteracji. Wówczas dla każdego v zachodzi $d_{f'}(s, v) \geqslant d_f(s, v)$,

Dowód stwierdzenia 1: Przypuśćmy, że tak nie jest, tzn., że istnieje taki wierzchołek $v \in V(G)$, że $d_{f'}(s, v) < d_f(s, v)$. Bez straty ogólności przyjmijmy, że v minimalizuje $d_{f'}(s, v)$ spośród takich problematycznych wierzchołków - tzn., że dla każdego wierzchołka x, dla którego $d_{f'}(s, x) < d_{f'}(s, v)$ prawdą jest, że $d_{f'}(s, x) \geqslant d_f(s, x)$.

Niech P' to najkrótsza w sensie liczby krawędzi s-v-ścieżka w sieci rezydualnej odpowiadającej f'. Przez w oznaczmy przedostatni wierzchołek na ścieżce P'. Rozpatrzmy teraz dwa przypadki:

1. Krawędź wv należy do sieci rezydualnej odpowiadającej przepływowi f, tzn. $c_f(wv) > 0$. W takiej sytuacji możemy zapisać, że

   $$d_f(s, v) \leqslant d_f(s, w) + 1 \leqslant d_{f'}(s, w) + 1 = d_{f'}(s, v),$$

   gdzie pierwsza nierówność wynika z faktu, że sieć rezydualna f ma krawędź od w do v, druga z minimalności v (tzn. w znajduje się bliżej źródła niż v), a ostatnia równość wynika stąd, że w poprzeda v na najkrótszej ścieżce w sieci rezydualnej dla f' (z doboru P'). Powyższy ciąg nierówności stoi w sprzeczności z $d_{f'}(s, v) < d_f(s, v)$.

2. Krawędź wv nie należy do sieci rezydualnej odpowiadającej przepływowi f, tzn. $c_f(wv) = 0 \Rightarrow c(wv) = f(wv)$. Skoro $c_{f'}(wv) > 0$, to oznacza, że przepływ na krawędzi wv podczas wykonywania się iteracji zmienił się, co w połączeniu z faktem, że $c_f(wv) = 0$ implikuje, że krawędź vw musiała być częścią ścieżki powiększającej wybranej w linii nr. 5 w tej iteracji. Zatem

   $$d_f(s, w) = d_f(s, v) + 1,$$

   co doprowadza nas do

   $$d_f(s, v) = d_f(s, w) - 1 \leqslant d_{f'}(s, w) - 1 = d_{f'}(s, v) - 2 < d_{f'}(s, v)$$

   Powyższy ciąg nierówności stoi w sprzeczności z $d_{f'}(s, v) < d_f(s, v)$.

Zatem stwierdzenie 1 jest prawdziwe.

Stwierdzenie 2: Niech f będzie przepływem przed pewną iteracją pętli w linii nr. 4, a f' przepływem po tej iteracji. Wówczas jeżeli $d_{f'}(s,t) = d_f(s,t)$, to $L_{f'} < L_f$.

Dowód stwierdzenia 2: Najpierw pokażemy, że każda krawędź licząca się do $L_{f'}$, musi zaliczać się również do L_f. W tym celu rozważmy pewną najkrótszą (względem liczby krawędzi) ścieżkę $P' = \omega_0\omega_1 \ldots \omega_k$, w sieci rezydualnej dla f', gdzie $s = \omega_0$, $t = \omega_k$ oraz $k = d_{f'}(s,t)$. Rozważmy odległości kolejnych wierzchołków na ścieżce od s w sieci rezydualnej dla f.

Rozważmy dwa przypadki dla krawędzi $\omega_i\omega_{i+1}$:

1. Krawędź $\omega_i\omega_{i+1}$ była obecna w sieci rezydualnej dla przepływu f, wtedy musi zachodzić
$$d_f(\omega_{i+1}) \leqslant d_f(\omega_i) + 1.$$

2. Krawędź $\omega_i\omega_{i+1}$ nie była obecna w sieci rezydualnej dla przepływu f, wtedy musi zachodzić
$$d_f(\omega_{i+1}) \leqslant d_f(\omega_i) - 1,$$
ponieważ krawędź $\omega_i\omega_{i+1}$ została dodana do sieci rezydualnej f' wskutek powiększenia przepływu wzdłuż ścieżki P wybranej w linii nr. 5.

Kumulując powyższe przypadki, otrzymamy

$$d_f(s,t) \leqslant d_f(s,\omega_{k-1}) \pm 1 \leqslant (d_f(s,\omega_{k-2}) \pm 1) \pm 1 \leqslant \ldots \leqslant A - B,$$

gdzie A oznacza liczbę wystąpień przypadku 1., z kolei B - liczbę wystąpień przypadku 2. Ale z założenia $d_{f'}(s,t) = d_f(s,t)$, więc ścieżka P' ma dokładnie $d_f(s,t)$ krawędzi, co pociąga za sobą $B = 0$ (bo wyrażenie $A - B$ jest maksymalnie równe $d_f(s,t)$), z czego wynika, że żadna z krawędzi ścieżki P' nie może realizować przypadku 2. Z tego rozumowania wynika, że każda krawędź na ścieżce P' znajdowała się w sieci rezydualnej dla f. Z dowolności wyboru P' każda krawędź licząca się do L_f musi też zaliczać się do $L_{f'}$, co chcieliśmy pokazać.

Teraz zauważmy, że przynajmniej jedna krawędź wliczająca się do L_f nie wlicza się do $L_{f'}$, jest to krawędź o przepustowości a (linijka nr. 6), która znajduje się na ścieżce P wybranej w linii nr. 5. Zatem ostatecznie

$$L_{f'} < L_f,$$

a więc dowód stwierdzenia 2. jest kompletny.

Z obu powyższych stwierdzeń możemy wywnioskować, dwa przypadki

1. $d_{f'}(s,t) > d_f(s,t)$,
2. $d_{f'}(s,t) = d_f(s,t) \Rightarrow L_{f'} < L_f$.

Oba powyższe przypadki implikują, że z sieci rezydualnej została usunięta co najmniej jedna krawędź, co pozwala nam dokonać oszacowania.

W pesymistycznym przypadku mamy $n-1$ s-t-ścieżek oraz pesymistycznie usuwamy dokładnie jedną krawędź w każdej iteracji co ostatecznie daje nam nie więcej niż nm wywołań pętli w nr. 4. Ponadto w każdej iteracji wywołujemy BFS w celu znalezienia najkrótszej ścieżki, z kosztem $O(m)$. Zatem złożoność algorytmu Edmondsa-Karpa to $O(nm^2)$.

□

7.3 Problem MinCostMaxFlow

Dane:

- Sieć przepływowa (G, c, s, t)
- Funkcja kosztu $k : E(G) \to \mathbb{R}$

Szukane: Maksymalny przepływ f, który minimalizuje koszt

$$k(f) := \sum_{e \in E(G)} k(e)f(e).$$

Ten problem możemy rozumieć jako wybranie takiego przepływu f ze wszystkich maksymalnych przepływów, dla którego $k(f)$ jest najmniejsze.

Definicja 7.6. *Siecią rezydualną z kosztami dla danej sieci przepływowej (G, c, s, t), funkcji kosztów k i przepływu f nazywamy graf skierowany R o zbiorze wierzchołków $V(G)$, funkcji wag c_f, funkcji kosztów k_f oraz zbiorze krawędzi równym $\{uv : c_f(uv) > 0\}$, gdzie funkcje c_f i k_f są zdefiniowane następująco:*

- $c_f(uw) = c(uw) - f(uw)$ *i* $k_f(uw) = k(uw)$, *jeśli* $uw \in E(G)$ *oraz* $wu \notin E(G)$
- $c_f(uw) = f(uw)$ *i* $k_f(uw) = -k(uw)$, *jeśli* $uw \notin E(G)$ *oraz* $wu \in E(G)$

W powyższej definicji nie uwzględniamy przypadku kiedy $uw \in E(G)$ oraz $wu \in E(G)$, ponieważ w takiej sytuacji sieć rezydualna jest multigrafem, który utrudnia reprezentację. Jako, że taką sytuację można łatwo rozwiązać tworząc podpodział krawędzi uw lub wu, przyjmujemy, że operujemy na grafach, w których takie sytuacje nie występują.

Twierdzenie 7.3. *(Twierdzenie o minimalnym koszcie) Niech f będzie maksymalnym przepływem w sieci (G, c, s, t) z funkcją kosztu k. Wówczas f jest maksymalnym przepływem o minimalnym koszcie wtedy i tylko wtedy, gdy odpowiadająca mu sieć rezydualna z kosztami nie zawiera cyklu o ujemnym koszcie.*

Dowód. Przez (R_f, c_f, k_f), będziemy oznaczali sieć rezydualną z kosztami odpowiadającą przepływowi f, gdzie c_f jest funkcją przepustowości, a k_f funkcją kosztów. Zauważmy, że (R_f, c_f) oraz (R_f, k_f) są ważonymi grafami skierowanymi.

Najpierw pokażemy, że jeśli (R_f, k_f) zawiera ujemny cykl, to maksymalny przepływ f nie minimalnego kosztu. Niech C będzie ujemnym cyklem w (R_f, k_f), tzn. $k_f(C) < 0$. Niech a to najmniejsza przepustowość w (R_f, c_f, k_f), pewnej krawędzi na C, tzn.

$$a := \min_{uw \in C} c_f(uw) > 0,$$

gdzie nierówność wynika z definicji sieci rezydualnej. Rozważmy przepływ f', taki, że, dla każdego $e \in E(G)$

$$f'(e) = \begin{cases} f(e) + a, \text{jeśli } e \in C \\ f(e), \text{jeśli } e \notin C \end{cases}$$

Zauważmy, że wartość przepływu f' jest taka sama jak wartość przepływu f, zatem f' również jest maksymalnym przepływem. Ponadto zachodzi

$$k(f') = k(f) + ak_f(C),$$

skoro $k_f(C)$ jest ujemne, to

$$k(f') > k(f),$$

czyli znaleźliśmy inny przepływ maksymalny o mniejszym koszcie co kończy tę część dowodu.

Pokażemy teraz, że jeśli maksymalny przepływ f nie jest minimalnego kosztu, to w (R_f, k_f) musi istnieć ujemny cykl. Rozważmy przepływ f', który jest maksymalnym przepływem o minimalnym koszcie różniącym się od f na najmniejszej możliwej liczbie krawędzi tzn. dla każdego maksymalnego przepływu f'' prawdą jest, że

$$|\{e \in E(G) : f'(e) \neq f(e)\}| \leqslant |\{e \in E(G) : f''(e) \neq f(e)\}|.$$

Niech S będzie zbiorem krawędzi sieci rezydualnej dla f takich, że przepuszczenie dodatkowego (dodatniego) przepływu wzdłuż tych krawędzi przybliży f do f', tzn.

$$S = \{uw : f(uw) < f'(uw) \vee f(wu) > f'(wu)\}.$$

Przez S' oznaczmy krawędzie sieci rezydualnej dla f' odwrotne do S.

Zauważmy, że wartości przepływów f i f' są równe (bo oba są maksymalnymi przepływami), a więc dla każdego wierzchołka $v \in V(G)$ otrzymujemy

$$f^+(v) - f^-(v) = f'^+(v) - f'^-(v),$$

bo dla $v \in \{s, t\}$ otrzymujemy po obu stronach wartość przepływu, która jest taka sama, a dla $v \notin \{s, t\}$ otrzymujemy zero z definicji przepływu. Przekształcając tę równość otrzymujemy

$$f'^+(v) - f^+(v) = f'^-(v) - f^-(v),$$

więc jeśli do v wchodzi jakaś krawędź z S, to z powyższej równości oraz definicji S wynika, że musi z niego wychodzić jakaś krawędź z S. Jako, że liczba wierzchołków jest skończona powyższy fakt musi pociągać za sobą istnienie cyklu wewnątrz zbioru S (bo nie możemy wychodzić do nieodwiedzonego wierzchołka w nieskończoność).

Niech C będzie cyklem w S, a więc również cyklem w sieci rezydualnej f; przez C' oznaczmy odwrotność cyklu C, tzn. $C' := \{wu : uw \in C\}$. Zauważmy, że C' jest cyklem w S', a więc jest również cyklem w sieci rezydualnej f'.

Ponadto oznaczmy przez a najmniejszą wartość z $f'(uw) - f(uw)$ po krawędziach $uw \in C$ spełniających $f(uw) < f'(uw)$ oraz z $f(wu) - f'(wu)$ po krawędziach $wu \in C$ spełniających $f(wu) > f'(wu)$. Intuicyjnie, a jest taką wartością, że możemy powiększyć przepływ f wzdłuż cyklu C o a nie przekraczając przepustowości krawędzi oraz takie powiększenie spowoduje, że któraś z krawędzi C przestanie „zaliczać się" do zbioru S.

Zauważmy, że gdyby $k_f(C) > 0$, wówczas $k_{f'}(C) < 0$ co jest sprzeczne, bo f' jest maksymalnym przepływem o minimalnym koszcie a już pokazaliśmy, że to implikuje brak ujemnych cykli w sieci rezydualnej z kosztem.

W przypadku kiedy $k_f(C) = 0$ to oznaczałoby że $k_{f'}(C) = 0$, a więc przepływ $f'' := f' + aC'$, czyli przepływ otrzymany z f' poprzez powiększenie wzdłuż cyklu C' o a również jest maksymalnym przepływem o minimalnym koszcie, jednakże ze sposobu wyboru a wynika, że f'' różni się od f na mniejszej liczbie krawędzi niż f', co stoi w sprzeczności ze sposobem wyboru f'.

Oznacza to, że $k_f(C) > 0$ co kończy dowód.

□

Algorithm 21 Algorytm „Przez usuwanie cykli"

```
1: procedure CycleCanceling((G, c, s, t), k)
2:     f ← maksymalny przepływ dla (G, c, s, t)
3:     (R, c_f, k_f) ← (G, c, k) (sieć rezydualna dla f)
4:     while Istnieje ujemny cykl C w (R, k_f) do
5:         a ← min{c_f(uw) : uw ∈ C}
6:         Powiększ f o a na wszystkich krawędziach C
7:         Uaktualnij (R, c_f, k_f) na wszystkich krawędziach C
8:     return f
```

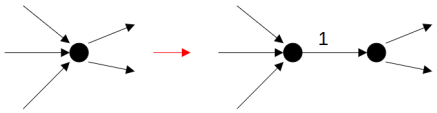
8 Przepływy – Zadania

8.1 Zadanie 3 – Szukanie najliczniejszego zbioru wewnętrznie rozłącznych ścieżek

Treść: Zaprojektuj algorytm, który dla zadanego grafu G oraz wierzchołków $v, w \in V(G)$ znajdzie najliczniejszy zbiór wewnętrznie rozłącznych wierzchołkowo ścieżek od v do w (innymi słowy, dla każdych dwóch ścieżek P_1 i P_2 z tego zbioru jedynymi wspólnymi wierzchołkami P_1 i P_2 mogą być v i w).

Rozwiązanie: Graf wejściowy G przekształcamy do grafu G' w taki sposób, że dla każdego wierzchołka $x \in V(G)$, tż. $x \notin \{v, w\}$, tworzymy wierzchołek x', następnie prze-

pinamy krawędzie wyjściowe z x do x' oraz dodajemy krawędź xx'. Rysunek **??** wizualizuje wyżej opisany zabieg.



Rys. 4

Wywołujemy Forda-Fulkersona na grafie G', który zwróci nam wartość maksymalnego przepływu oraz przepływ f. Aby zapisać ścieżki tworzymy graf G'', w którym usuwamy wszystkie krawędzie $e \in E(G)$, tż., $f(e) = 0$. Dla każdego sąsiada v idziemy do w, odkładając napotkane wierzchołki do na listę. Zwracamy wszystkie listy.

8.2 Zadanie 4 – Znajdowanie najmniejszego przekroju w sieci

Treść: Przekrojem sieci (G, c, s, t) nazywamy podział zbioru $V(G)$ na rozłączne zbiory S i T taki, że $s \in S$ oraz $t \in T$. Przepustowość takiego przekroju to suma $\sum_{v \in S, w \in T} c(vw)$. Zaprojektuj algorytm, który znajdzie przekrój o najmniejszej przepustowości w danej sieci przepływowej.

Rozwiązanie: Rozwiązanie polega na uruchomieniu algorytmu Floyda-Fulkersona, a następnie przebadaniu sieci rezydualnej R dla maksymalnego przepływu f zwróconego przez ten algorytm.

Uruchamiamy DFS lub BFS w celu znalezienia wszystkich osiągalnych wierzchołków z s w sieci rezydualnej R. Zbiór osiągalnych wierzchołków S oraz $\overline{S} = V(G) \setminus S$ stanowią podział o najmniejszej przepustowości.

8.3 Zadanie 5 – Algorytm Dinic'a

Treść: Niech R będzie siecią rezydualną dla pewnej sieci przepływowej (G, c, s, t) i przepływu f; niech d będzie odległością (w sensie liczby krawędzi) od s do t w R. Przez przepływ blokujący w R rozumiemy przepływ f' w R taki, że każda ścieżka od s do t w f' ma dokładnie d krawędzi oraz dla każdej ścieżki o d krawędziach od s do t w R, przynajmniej jedna krawędź tej ścieżki jest nasycona przez f'.

a) Zaprojektuj algorytm, który znajdzie przepływ blokujący w czasie $O(nm)$

b) Skonstruuj algorytm rozwiązujący problem maksymalnego przepływu w czasie $O(n^2m)$

Rozwiązanie:

Definicja 8.1. *Niech R będzie siecią rezydualną dla pewnej sieci przepływowej (G, c, s, t) oraz przepływu f. Niech d będzie odległością w sensie liczby krawędzi od s do t w R.*

Przez przepływ blokujący w R rozumiemy przepływ b w sieci (R, c_f, s, t), taki, że każda ścieżka od s do t w b ma dokładnie d krawędzi oraz dla każdej ścieżki o d krawędziach od s do t w R, przynajmniej jedna krawędź tej ścieżki jest nasycona przez b.

BFS + usuwanie niepotrzebnych krawędzi w R w celu utworzenia tzw. grafu warstwowego, potem DFS w celu przejscia po kazdej ścieżce i utworzenia przepływów. **Rozwiązanie b:**

Algorithm 22 Algorytm Dinic'a

```
1: procedure DINIC(G, w, s, t)
2:     Niech f to zerowy przepływ w sieci (G, c, s, t)
3:     Utwórz sieć rezydualną (R, c_f) dla wejściowej sieci przepływowej oraz dla f
4:     while Istnieje ścieżka powiększająca w R do
5:         Znajdź przepływ blokujący f' w sieci (R, c_f, s, t)
6:         Powiększ f o f'
7:         Uaktualnij (R, c_f)
8:     return f
```

8.4 Zadanie 6 – Wyznaczanie pesymistycznej wydajności w sieci

Treść: Przypuśćmy, że mamy daną sieć przewodową składającą się z n węzłów (w tym jeden serwer) i m połączeń między węzłami, przy czym każde połączenie ma pewną przepustowość wyrażoną w kB/s. Zakładamy, że każdy węzeł może jedynie odbierać dane i przesyłać je dalej (w szczególności, zabronione są modyfikacje danych przed ich przesłaniem). Przez pesymistyczną wydajność sieci rozumiemy najwiekszą wartość w taką, że serwer jest w stanie wysyłać jednocześnie do każdego innego węzła dane w ilości w kB/s.

Zaprojektuj algorytm, który wyznaczy pesymistyczną wydajność zadanej sieci.

Rozwiązanie: Rozwiązanie polega na połączeniu wszystkich routerów z ujściem t krawędziami o przepustowości 1. Jeżeli po wywołaniu algorytmu znajdującego maksymalny przepływ każda z krawędzi do ujścia t jest nasycona, to zwiększamy przepustowość każdej z tych krawędzi o 1 i ponawiamy próbę. Ostatnia liczba dla której wszysktkie krawędzie były nasycone to szukane w.

Powyższe rozumowanie można zoptymalizować korzystając z binarnego przeszukiwania. Jako maximum przyjmujemy sumę przepustowości na krawędziach wychodzących z s (bo valf_max jest nie większe niż wartość dowolnego przekroju), a jako minimum - 1.

Algorithm 23 Wyznaczanie pesymistycznej wydajności

```
1: procedure FindPesimisticEfficiency(G, c, s)
2:     Niech V' = V(G) + {t}
3:     Niech E' = E(G) + {vt : v ∈ V' \ {s, t}}
4:     Niech G' = (V', E')
5:     R ← Σ_{v∈N(s)} c(sv)
6:     L ← 1
7:     while L < R do
8:         w = ⌊(L+R)/2⌋
9:         for e ∈ E' do
10:            c(e) = w
11:        f ← Ford-Fulkerson(G', c, s, t)
12:        if ∀_{e∈E'} f(e) = w then
13:            L ← w + 1
14:        else
15:            R ← w − 1
16:    return w
```

8.5 Zadanie 8

Treść: Rozważmy pociąg, który zatrzymuje się na stacjach numerowanych liczbami od 1 do n. Pociag może przewozić jednocześnie P pasażerów. Dla każdej pary stacji i, j, gdzie $i < j$, mamy daną liczbę b_{ij} pasażerów chcących przejechać od stacji i do j oraz koszt jednego takiego biletu c_ij . Zaprojektuj algorytm, który określi, komu należy sprzedać bilety tak, aby zmaksymalizować zysk (tzn. sumę kosztów sprzedanych biletów).

9 Algorytmy z nawrotami (backtracking)