

**WYDZIAŁ PODSTAWOWYCH PROBLEMÓW TECHNIKI
POLITECHNIKI WROCŁAWSKIEJ**

**ALGORYTMY
WYZNACZANIA NAJKRÓTSZYCH ŚCIEŻEK
W RZECZYWISTYCH SIECIACH DROGOWYCH**

TOMASZ STRZAŁKA

Praca magisterska napisana
pod kierunkiem
dr hab. Pawła Zielińskiego, prof. PWr



Politechnika
Wrocławska

WROCŁAW 2014

Spis treści

1 Podstawy	5
1.1 Grafy w sieciach drogowych	5
1.2 Reprezentacje grafu	5
1.2.1 Macierz incydencji	6
1.2.2 Macierz sąsiedztwa	7
1.2.3 Listy incydencji	8
1.2.4 Pęki	8
1.2.5 Złożoność pamięciowa i czasowa	10
1.3 Problem najkrótszych ścieżek	14
1.3.1 Reprezentacja problemu	15
1.3.2 Podstawowe operacje	17
1.3.3 Właściwości najkrótszych ścieżek	19
1.3.4 Algorytm Bellmana-Forda	20
1.4 Uwagi do rozdziału	23
2 Najkrótsze ścieżki z jednym źródłem	25
2.1 Sortowanie topologiczne	25
2.1.1 Algorytm Khana	25
2.1.2 Przeszukiwanie w głąb	28
2.1.3 Sortowanie topologiczne	29
2.2 Generyczny algorytm Dijkstry	30
2.2.1 Algorytm Dijkstry	30
2.2.2 Złożoność obliczeniowa	31
2.2.3 Ujemne koszty krawędzi	32
2.2.4 Poporawność działania	33
2.3 Podstawowe struktury danych	34
2.4 Struktury oparte na kopcach	34
2.4.1 Kopiec R-arny	34
2.4.2 Kopiec Fibonacciego	37
2.5 Struktury oparte na kubełkach	40
2.5.1 Pierwsze podejście	40
2.5.2 Z przepeleniem	41
2.5.3 Dial	43
2.5.4 Aproksymacja zakresu	45
2.5.5 Kubełki wielopoziomowe	48
2.5.6 Kopce pozycyjne	52
3 Inne algorytmy	57
3.1 Kombinacje struktur	57
3.1.1 Algorytm przyrostowy	57
3.1.2 Algorytm przyrostowy z dwoma kolejkami	59
3.2 Algorytm progowy	60

SPIS TREŚCI

4 Biblioteka: Take Me Home	63
4.1 Opis	63
5 Zakończenie	65
A Instrukcja	67
A.1 Wymagania i instalacja	67
A.2 Użytkowanie	67
A.2.1 konfiguracja	67
A.2.2 API	67
B Dowody twierdzeń	69
C Trochę matmy	71
C.1 Złożoność obliczeniowa	71
C.1.1 Analiza asymptotyczna	71
C.1.2 Analiza amortyzacyjna	71

Podstawy

W tym rozdziale zostaną omówione wszystkie ważniejsze pojęcia, którymi będziemy się od tej pory posługiwać. Przedstawimy przede wszystkim pojęcie **grafu** i sposoby jego reprezentacji w algorytmach, które będziemy omawiać w późniejszej części pracy. Przedyskutujemy postawiony przed nami problem wyszukiwania **najkrótszych ścieżek** w rzeczywistych sieciach drogowych, a także przyjrzymy się dokładnie własnościom, z jakich przyjdzie nam niejednokrotnie skorzystać podczas analizy kolejnych rozwiązań tego problemu. Na końcu tego rozdziału przedstawimy algorytm Bellmana-Forda jako podstawową ideę rozwiązywania tego typu problemów, zastanowimy się nad tym co można w nim usprawnić, aby uzyskiwać rozwiązania problemu w znacznie krótszym czasie.

1.1 Grafy w sieciach drogowych

Grafem będziemy nazywać taką parę $G = (V, E)$, gdzie każde $v \in V$ jest **wierzchołkiem** tego grafu, zaś każdy element $e \in E$ jest jego **krawędzią**, łączącą dowolne dwa wierzchołki.

W naszym przypadku krawędzie (E) grafu będziemy rozumieć jako dowolny odcinek drogi na jezdni między dwoma dowolnie wybranymi punktami v_p oraz v_k , gdzie przez v_i dla $i \in \{1, \dots, |V|\}$ będziemy od tej pory oznaczać dowolny wierzchołek w grafie G (w odniesieniu do rzeczywistego ruchu drogowego może to być skrzyżowanie, dowolny punkt drogi na wysokość jakiegoś specyficznego budynku, miejsca wystąpienia znaku drogowego itp.). Taką krawędź będziemy zwykle oznaczać przez e_{pk} , gdzie kolejność wypisania indeksów określa nam zwrot danego łuku. Stosowanie grafowej reprezentacji w odniesieniu do sieci dróg determinuje w pewnym stopniu właściwości grafu z jakim będziemy mieli do czynienia. Nie będzie w nim na pewno krawędzi, których **koszt** jest mniejszy lub równy zero. W ogólności nie będziemy także mogli nic powiedzieć o acykliczności grafu, ani rozstrzygnąć, czy będziemy pracować na grafach skierowanych czy nie - zdecydowana większość rozważanych sieci drogowych posiada jednak w swojej topologii liczne cykle, a także wiele dróg jednokierunkowych i właśnie na taki model grafu - skierowany z cyklami - się zdecydujemy.

Każda krawędź w grafie posiada swoją **wagę**, podobnie jak każda droga z punktu v_p do v_k ma zdefiniowaną odległość między tymi dwoma punktami, wyrażoną w dowolnych jednostkach długości. Aby uprościć nasz model grafu założymy, że **koszt** (waga) każdego łuku¹ będzie wyrażony przez jedną liczbę naturalną, będącą odzwierciedleniem odległości między punktami, które dana krawędź $e \in E$ łączy. W rzeczywistych warunkach drogowych takie połączenie mogłoby posiadać cały szereg atrybutów takich jak np. intensywność ruchu ulicznego, rodzaj nawierzchni, nachylenie terenu, ograniczenia prędkości na danym odcinku drogi, które miałyby bezpośredni wpływ na wybór najkrótszej ścieżki od punktu v_p do v_k , a które my w swoich rozważaniach, dla zachowania ich prostoty, pominiemy.

1.2 Reprezentacje grafu

W informatyce istnieje kilka sposobów na efektywne przedstawienie struktury grafu. Jak później pokażemy, wybór ten w znaczny sposób może wpływać na efektywność algorytmów wyszukiwania najkrótszych ścieżek, zarówno na ich złożoność czasową jak i pamięciową. W niniejszym podrozdziale pokażemy trzy różne podejścia

¹ w odniesieniu do połączeń pomiędzy węzłami w grafie $G = (V, E)$ będziemy wymiennie stosować nazwy: krawędź, łuk, połączenie.

do problemu przedstawienia grafu jako struktury w programie, gdzie pierwsze z nich zakłada tworzenie dla grafu wejściowego **macierzy incydencji** - jednego z dwóch wariantów reprezentacji macierzowej grafu jakie będziemy rozróżniać.

1.2.1 Macierz incydencji

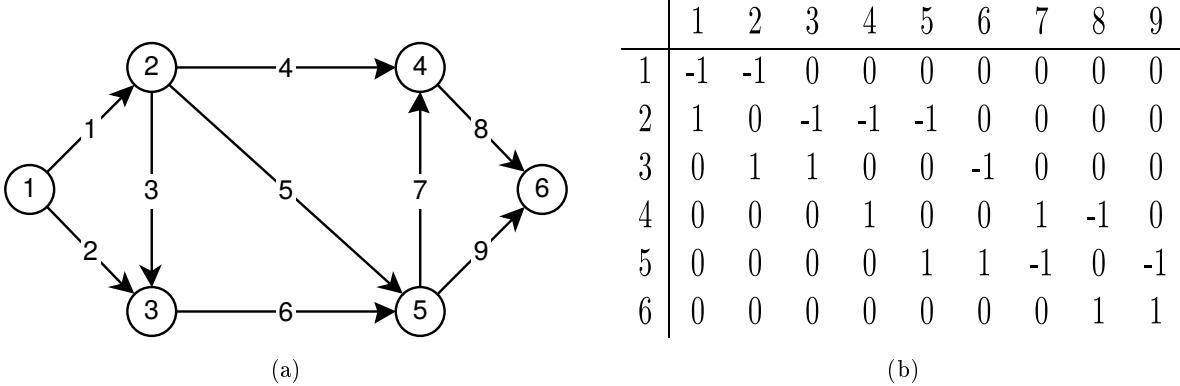
Macierz incydencji dla grafu $G = (V, E)$ jest to macierz o wymiarach $|V| \times |E|$, gdzie każda komórka jest zdefiniowana następująco:

$$a_{ij} = \begin{cases} -1 & \text{jeżeli krawędź } j \text{ wychodzi z wierzchołka } i, \\ 1 & \text{jeżeli krawędź } j \text{ wchodzi z wierzchołka } i, \\ 0 & \text{w przeciwnym wypadku} \end{cases} \quad (1.1)$$

gdzie:

$$\begin{aligned} i &\in \{1, \dots, |V|\}, \\ j &\in \{1, \dots, |E|\}. \end{aligned} \quad (1.2)$$

Innymi słowy każde połączenie w grafie między dwoma wierzchołkami v_p i v_k jest traktowane jako takie, które uaktywniamy, pobierając jedną jednostkę „energii” (stąd w macierzy na odpowiednim miejscu wartość -1), którą następnie przekazujemy do docelowego wierzchołka v_k (co odnotowujemy w macierzy wartością $+1$).



Rysunek 1.1: **Macierz incydencji** (a) Graf skierowany $G = (V, E)$ z identyfikatorami węzłów 1 – 6 i łuków 1 – 9 (dla znanych identyfikatorów łuków będziemy stosować oznaczenie e_i zamiast e_{pk}). (b) Macierz incydencji $|V| \times |E|$ grafu G .

Operacje, jakie będziemy chcieli - jak się później okaże - wykonywać na tak zdefiniowanej macierzy (i na każdej następnej strukturze z tego podrozdziału) to procedury wyszukiwania wszystkich bezpośrednich następców danego węzła v_i (oznaczać będziemy taki zbiór za pomocą symbolu $A(i)$) i odnajdywania każdego takiego łuku, wychodzącego z danego węzła do wszystkich $v_k \in A(i)$. Dla pierwszego podejścia:

- koszt identyfikacji wszystkich łuków jest ściśle powiązany z ilością krawędzi w grafie - wystarczy, że dla wierzchołka v_i przejrzymy cały jeden wiersz o indeksie i , aby odnaleźć identyfikatory wszystkich krawędzi, bezpośrednio wychodzących z danego wierzchołka (wszystkie komórki o wartości mniejszej od zera). Możemy ograniczyć ilość potrzebnych skanowań poprzez wprowadzenie licznika krawędzi wychodzących, lecz w najgorszym możliwym przypadku takie skanowanie wciąż będzie wymagało $O(m)$ porównań, gdzie m to oczywiście liczba krawędzi w grafie.
- Koszt wyszukiwania wszystkich następców węzła v_i , obejmuje koszt wyszukania wszystkich łuków, prowadzących do tych wierzchołków oraz odnalezienie ich identyfikatorów - co dla każdego odnalezionej łuków e_j zmusza nas do przeszukania wszystkich elementów macierzy, znajdujących się w tej samej, j tej kolumnie. Takich kolumn, jak wspomnieliśmy wcześniej, będzie $|A(i)|$, przeszukanie każdej j tej kolumny

wymagać będzie, w najgorszym przypadku, $n - 1$ porównań dla każdej z nich tak więc ostatecznie otrzymujemy $O(|A(i)| \cdot n)$ dla wyszukiwania wszystkich następców węzła v_i (wraz z wyszukiwaniem łuków $O(m + |A(i)| \cdot n)$).

1.2.2 Macierz sąsiedztwa

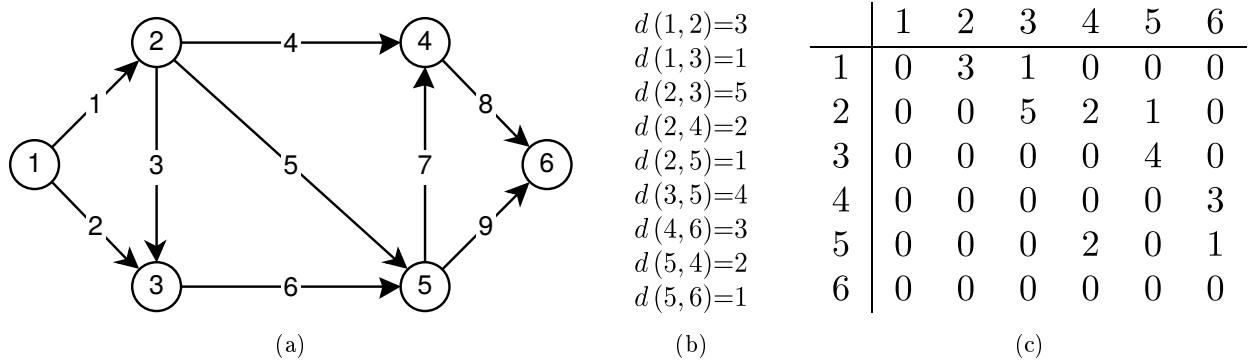
Oprócz macierzowej reprezentacji typu wierzchołek-krawędź możliwe jest także reprezentowanie struktury grafu za pomocą macierzy typu wierzchołek-wierzchołek. **Macierz sąsiedztwa** dla grafu $G = (V, E)$ jest to macierz o wymiarach $|V| \times |V|$, gdzie wartość każdej komórki jest zdefiniowana następująco:

$$b_{ij} = \begin{cases} 1 & \text{jeżeli istnieje ścieżka z wierzchołka } v_i \text{ do } v_j, \\ 0 & \text{w przeciwnym wypadku} \end{cases} \quad (1.3)$$

O ile, w przypadku takiego przedstawienia grafu, jesteśmy w stanie uzyskać informacje o wszystkich bezpośrednich następcach dowolnego węzła w czasie liniowym (dla węzła v_i wystarczy przeszukać wiersz o indeksie i) to macierz w takiej postaci nie niesie ze sobą istotnych informacji o krawędziach grafu - takich, które umożliwiłyby ich szybką identyfikację, nie zmuszając nas do ponownego przeglądania wszystkich krawędzi grafu w poszukiwaniu łuku, który ma swój początek i koniec w - danych nam przez macierz - wierzchołkach. Umiejętność szybkiej identyfikacji takich krawędzi będzie nam w późniejszych rozważaniach nieodzowna, jako że chcemy się skupić na wyszukiwaniu najkrótszych ścieżek, gdzie kluczowych informacji dla tego problemu będąemy szukać właśnie w krawędziach między wierzchołkami. Założymy, że każda krawędź będzie określana za pomocą trzech cech: wierzchołka startowego, końcowego oraz odległości między tymi dwoma punktami. Zauważmy, że w takim przypadku wszystkie informacje o krawędzi możemy umieścić bezpośrednio w macierzy sąsiedztwa, zastępując starą definicję nową:

$$b_{ij} = \begin{cases} d(i, j) & \text{jeżeli istnieje ścieżka z wierzchołka } v_i \text{ do } v_j, \\ 0 & \text{w przeciwnym wypadku} \end{cases} \quad (1.4)$$

gdzie przez $d(i, j)$ zwykle będziemy oznaczać długość ścieżki (odległość między wierzchołkami, które łączy). Takie podejście pozwala nam na nie tworzyć dodatkowych struktur, przechowujących informacje o krawędziach. Jeżeli chcielibyśmy wzbogacać naszą strukturę krawędzi wygodniej (a także bezpieczniej) zamiast wartości $d(i, j)$ w danych komórkach b_{ij} będzie wstawić numer identyfikatora danej krawędzi. Zapewni nam to dodatkowo jednoznaczność w przypadku chęci identyfikacji krawędzi (zwróciemy uwagę, że oba łuki, wchodzące do węzła v_4 mają ten sam koszt $d(2, 4) = d(5, 4) = 2$, co uniemożliwia nam ich rozróżnienie).



Rysunek 1.2: **Macierz sąsiedztwa** (a) Graf skierowany $G = (V, E)$ z identyfikatorami węzłów 1–6 i łuków 1–9. (b) Wagi/koszty łuków grafu G . (c) Macierz sąsiedztwa $|V| \times |V|$ grafu G . Dla każdej wagi łuku $d(v_p^{ID}, v_k^{ID}) = c$ odpowiednie komórki na przecięciu wiersza o numerze v_p^{ID} i kolumny v_k^{ID} mają wartość równą c , gdzie v_k^{ID} oznacza identyfikator węzła v_k ($v_k^{ID} = k$).

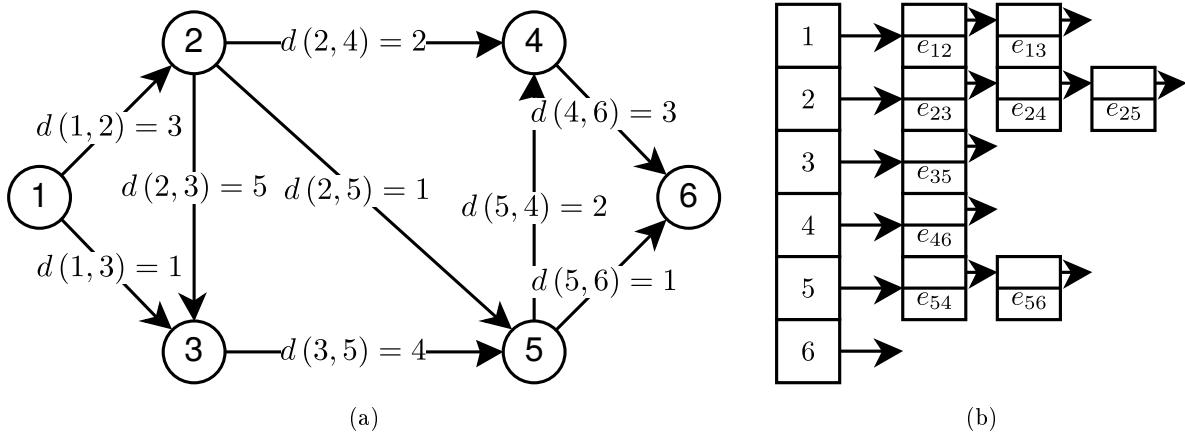
Zatem ostatecznie otrzymamy:

- koszt identyfikacji wszystkich łuków o złożoności liniowej $O(n)$, gdzie $n = |V|$,

- koszt wyszukiwania wszystkich następców węzła v_i w tym przypadku jest tożsamy z odnalezieniem wszystkich łuków wychodzących z danego węzła, co zajmuje $O(n)$.

1.2.3 Listy incydencji

Wprowadzone wcześniej przez nas oznaczenie $A(i)$, oznaczające wszystkich bezpośrednich następców danego węzła v_i od teraz będzie dla nas oznaczało jednokierunkową listę łuków, wychodzących z węzła v_i oraz wchodzących do każdego węzła $v_k \in A(i)$ (ang. *Adjacency list*), a operacja wyszukania takich łuków będzie dla nas tożsama ze znalezieniem wszystkich wierzchołków v_k . Słownem - połączymy poprzednio rozdzielane operacje identyfikacji łuków i węzłów w jedną. Da to nam w najgorszym przypadku liniowy czas dostępu do dowolnego wierzchołka, będącego bezpośredniem nastepnikiem badanego elementu (gdy będziemy musieli przejść przez całą listę $A(i)$).



Rysunek 1.3: **Lista sąsiedztwa** (a) Grafi skierowany $G = (V, E)$ z identyfikatorami węzłów 1 – 6. Zamiast identyfikatorów łuków na krawędziach zostały naniesione wagi każdego z nich. (b) Listy sąsiedztw grafu G . Dla każdego węzła $v_i : i \in \{1, \dots, 6\}$ jest stworzona lista jednokierunkowa, której każdy element zawiera informację o pojedynczym łuku e_{ij} .

Każdy taki łuk, oprócz swojej długości (dalej zwanej ogólniej: **kosztem**) będzie zatem posiadał dodatkowy atrybut, jednoznacznie wskazujący na węzeł, do którego prowadzi. Formalnie:

$$v_k \in A(i) \Leftrightarrow \exists e_{ik} = (v_i, v_k) \in E$$

Taką strukturę będziemy dalej nazywać zamiennie listą sąsiedztwa lub **incydencji**².

1.2.4 Pęki

Na podobnym pomyśle, co listy sąsiedztwa, bazuje rozwiązywanie z użyciem tzw. pęków. Tutaj także dla każdego węzła tworzyć będziemy „listę” węzłów, które są jego bezpośredniimi następcami z tą różnicą, że te informacje będziemy zapisywać w jednej tablicy, nie na osobnych listach.

W niniejszym podrozdziale omówimy najbogatszą wersję reprezentacji z jednoczesnym wykorzystaniem **pęków wyjściowych** jak i **wejściowych** (ang. *Forward and Reverse Star Representation*), koncentrując się po kolei na pierwsze i drugiej części, które równie dobrze mogą stanowić samodzielne reprezentacje grafów.

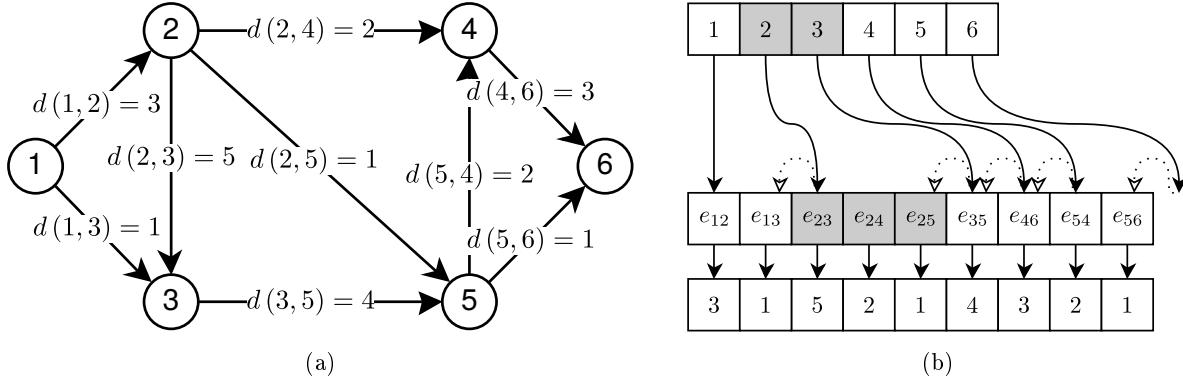
Pęki wyjściowe

Załóżmy, że nasza reprezentacja grafu zawiera tablicę indeksowaną od 1 do $|V|$ - nazwijmy ją $vtab$. Stwórzmy drugą tablicę, która będzie konstruowana następująco:

dla każdego elementu tablicy $vtab$ o indeksie $vIdx$ (indeksy tej tablicy odpowiadają identyfikatorom wszystkich węzłów w grafie) będziemy chcieli, aby wartość, przechowywana w tej komórce, była równa minimalnemu indeksowi $aIdx$ w tablicy $atab$, dla którego element $atab[aIdx]$ będzie zawierał informacje o łuku,

²Mówimy, że dwa wierzchołki są incydentne, jeżeli istnieje łącząca je krawędź.

wychodzącym z wierzchołka o identyfikatorze $vIdx$, zaś wartość $vtab[vIdx + 1]$ będzie zawierała ostatni taki indeks, powiększony o jeden. Innymi słowy będziemy chcieli aby każdy element z tablicy $vtab$ zawierał informację o tym, od którego miejsca w tablicy $atab$ są przechowywane kolejno wszystkie informacje o łukach, wychodzących z węzła, przechowywanego w badanym elemencie pierwnej tablicy (rys. 1.4).



Rysunek 1.4: **Pęki wyjściowe** (a) Grafo skierowany $G = (V, E)$ z identyfikatorami węzłów 1 – 6. (b) Pęki łuków dla grafu G . Dla każdego węzła $v_i \in vtab$ element $vtab[i]$ zawiera indeks pierwszej, wychodzącej krawędzi z węzła v_i w tablicy $atab$. Elementy w $atab$ są z kolejnymi wskazaniami na odpowiednie krawędzie grafu.

Na początku musimy stworzyć obie tablice, pierwszą ($vtab$) zainicjować samymi jedynkami (numer pierwszego indeksu łuku), zaś drugą pozostawić pustą - będziemy ją wypełniać w trakcie działania algorytmu. Następnie, iterując po tablicy z wierzchołkami $vtab$ odnajdujemy wszystkie łuki wychodzące z danego wierzchołka i wstawiamy dane takiej krawędzi (zakładamy, że są nimi identyfikatory łuków) do kolejnych elementów tablicy $atab$, jednocześnie zwiększając licznik $aIdx$. Gdy przeglądniętyśmy wszystkie łuki wychodzące z pierwszego wierzchołka v_{vIdx} , wstawiamy do następnego elementu z tablicy $vtab$ ($vtab[vIdx + 1]$) wartość licznika $aIdx$ - jest to najmniejszy indeks w tablicy $atab$, dla którego element $atab[aIdx]$ będzie zawierał identyfikator łuku, wychodzącego z wierzchołka v_{vIdx+1} , a tym samym łatwo będziemy mogli policzyć, który element w tablicy zawiera ostatni łuk, wychodzący z wierzchołka poprzedniego (o indeksie $vIdx$). Algorytm kontynuujemy, dopóki nie przeglądnięliśmy wszystkich elementów w tablicy $vtab$. W efekcie otrzymamy tablicę $atab$, której elementy będą posortowane rosnąco względem identyfikatorów wierzchołków, z których wychodzą, a każda para elementów ($vtab[i], vtab[i + 1]$) będzie zawierać indeksy, dla których od $atab[vtab[i]]$ do $atab[vtab[i + 1] - 1]$ znajdują się łuki wychodzące z wierzchołka o indeksie i .

Jeżeli okaże się, że jakiś wierzchołek v_i nie ma żadnych krawędzi wychodzących, wtedy $vtab[i] = vtab[i + 1]$, w szczególności $vtab[1] = 1$ (pierwszy element, jako że indeksujemy od jedynki).

Algorithm 1: CREATE-FORWARD-STAR-REPRESENTATION

```

Data:  $G = (V, E)$ 
Result:  $atab[1 \dots |E|]$ 
begin
1    $aIdx \leftarrow 1$ 
2    $vtab[1 \dots |V|] \leftarrow aIdx$ 
3   for  $vIdx \in 1 \dots |V|$  do
4       for każdy łuk  $e$ , prowadzący z  $v_{vIdx}$  do wierzchołka  $\in A(v_{vIdx})$  do /* Dla każdego węzła w  $vtab$  */
5            $atab[aIdx] = e$ 
6            $aIdx \leftarrow aIdx + 1$ 
7    $vtab[vIdx + 1] \leftarrow aIdx$ 
8

```

Pęki wejściowe

Mamy sytuację odwrotną: chcemy mieć możliwość szybkiego zidentyfikowania wszystkich wierzchołków wchodzących do danego węzła v_k . Algorytm jest taki sam jak w poprzednim przypadku z tą różnicą, że kolejność występowania łuków w tablicy $atab$ będzie inna - będą one występować w kolejności rosnących identyfikatorów węzłów, do których prowadzą, a nie mają początku. W obu przypadkach kolejność występowania krawędzi, wchodzących/prowadzących do tych samych wierzchołków nie ma znaczenia - w algorytmach wyszukiwania najkrótszych ścieżek, jeżeli zapytamy o dany węzeł to będziemy chcieli poznać wszystkich jego bezpośrednich następców/poprzedników, a nie tylko wybranego z nich.

Pęki wejścia-wyjścia

W pewnych przypadkach warto abyśmy dysponowali możliwością nie tylko szybkiego przeglądania tablic incydencji w poszukiwaniu następców, ale chcielibyśmy także mieć taką możliwość w odniesieniu do wszystkich poprzedników dowolnego węzła w sieci. Aby zrobić to oszczędnie, będziemy wykorzystywać fakt, że we wcześniejszych wersjach w tablicy $atab$ celowo trzymaliśmy tylko identyfikatory łuków, nie zaś ich wszystkie atrybuty (np. dla łuku o identyfikatorze i o atrybutach: węzeł początkowy, końcowy oraz koszt, wartości te trzymalibyśmy w elementach osobnych tablic: $head[i]$, $tail[i]$ oraz $cost[i]$). Jako podstawę dla naszego algorytmu weźmiemy pierwszą z omawianych reprezentacji, a więc zakładamy, że mamy tablice:

- $vtab[1 \dots |V|]$ - przechowującą informacje o łukach dla węzłów o identyfikatorach od 1 do $|V|$,
- $atab[1 \dots |E|]$ - przechowującą krawędzie o identyfikatorach od 1 do $|E|$, posortowane rosnąco według identyfikatorów wierzchołków początkowych, gdzie dla każdego węzła v_k wszystkie identyfikatory łuków wchodzących z tego węzła znajdują się w elementach od $atab[vtab[k]]$ do $atab[vtab[k+1]-1]$ włącznie

i do nich chcemy dodać dwie nowe:

- $rtab[1 \dots |V|]$ - analogicznie do $vtab$ przechowującą informacje o łukach dla węzłów o identyfikatorach od 1 do $|V|$ (dla reprezentacji odwróconej - *Reverse Star Representation*),
- $mtab[1 \dots |E|]$ - mapującą krawędzie o identyfikatorach od 1 do $|E|$.

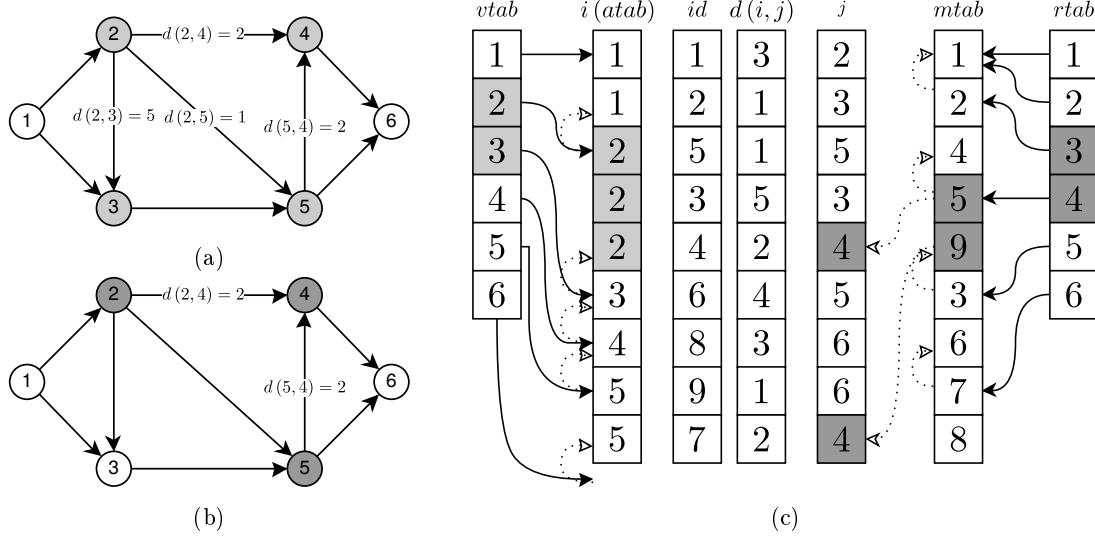
Druga z nich - $mtab$ - jest tablicą wejściową dla metody, opisanej w sekcji „Pęki wejściowe”. Przechowuje ona w swoich kolejnych elementach indeksy tablicy $atab$ w taki sposób, aby ciąg $atab[mtab[1]], atab[mtab[2]], \dots, atab[mtab[|E|]]$ tworzył ciąg identyfikatorów łuków, które są posortowane rosnąco wedle wierzchołków wejściowych. Wówczas tablica $rtab$ razem z tablicą $mtab$ zachowuje się dokładnie tak samo jak druga para tablic.

Oczywiście taki sposób reprezentacji danych jest bardzo wrażliwy na wszelkie zmiany w strukturze sieci - każde usunięcie czy dodanie krawędzi powoduje konieczność aktualizacji wszystkich tablic, co jest bardzo pracochłonne, w porównaniu z reprezentacjami macierzowymi (gdzie taka operacja mogła być wykonana w czasie stałym) czy przy wykorzystaniu list sąsiedztwa (gdzie koszt dodawania krawędzi jest także stały, zaś koszt jej usunięcia to czas rzędu $O(|E|)$). Taka reprezentacja natomiast pozwala nam na błyskawiczne - bo polegające na odjęciu wartości $vtab[i]$ od $vtab[i+1]$ - policzenie **stopnia** wierzchołka v_i , a także uzyskać dostęp do wszystkich elementów $v \in A(i)$ dla dowolnego wierzchołka v_i w tym samym, liniowym, zależnym od ilości elementów w $A(i)$, czasie, jednocześnie przy zachowaniu stosunkowo małego - bo także liniowego - zapotrzebowania na pamięć (rys. 1.5).

1.2.5 Złożoność pamięciowa i czasowa

Innym kryterium wyboru najodpowiedniejszego sposobu przedstawienia grafu jest ilość pamięci, jakiej wymagają implementacje poszczególnych rozwiązań. Dla obu implementacji macierzowych będą to wymagania rzędu $O(|V| \dots |E|)$ lub $O(|V| \dots |V|)$, odpowiednio dla macierzy incydencji i sąsiedztwa. Ile tak naprawdę z tej pamięci jest przez nas marnowanej najlepiej widać w przypadku, gdy mamy do czynienia z **grafem rzadkim**³, gdzie w takiej sytuacji większość macierzy jest wypełniona zerami (macierz rzadka). Stosując

³graf, którego stosunek posiadanych krawędzi do ilości węzłów w danym grafie jest niewielki



Rysunek 1.5: **Pęki wejścia-wyjścia** **(a)** Reprezentacja grafu $G = (V, E)$ z oznaczonymi następcami węzła v_2 . **(b)** Ten sam graf skierowany z oznaczonymi poprzednikami węzła v_2 . **(c)** Graf przedstawiony w formie tablic.

odpowiednie kodowanie, możemy wpływać na tą niepożądaną własność np. macierz incydencji, której elementy a_{ij} posiadają tylko trzy wartości: $-1, 0, 1$ możemy przedstawić za pomocą dwóch macierzy, z której jedna - nazwijmy ją macierzą wyjścia - będzie zawierać jedynki na tych samych pozycjach, co poprzednia macierz, lecz w miejscach wystąpienia wartości ujemnej będzie miała wartość równą zero. Dla macierzy wejścia z kolei będziemy wstawiać jedynki w miejscach, gdzie uprzednio znajdowały się wartości przeciwnie.

$$a_{ij}^{IN} = \begin{cases} 1 & \text{jeżeli } a_{ij} = -1, \\ 0 & \text{w przeciwnym wypadku} \end{cases} \quad (1.5)$$

$$a_{ij}^{OUT} = \begin{cases} 1 & \text{jeżeli } a_{ij} = 1, \\ 0 & \text{w przeciwnym wypadku} \end{cases} \quad (1.6)$$

Następnie zamieniamy każdą macierz na ciągi binarne długości $|V| \dots |E|$ każdy. Jest to prosta metoda na zgromadzenie potrzebnych nam informacji na jak najmniejszym fragmencie pamięci (jedna informacja - 1 bit), dodatkowo umożliwiająca nam bardzo szybkie przeszukiwanie takiej macierzy za pomocą operacji bitowych, a wykonanie kopii tak zgromadzonych danych to już nie koszt skopiowania wartości każdego elementu macierzy, a przepisanie jednej, potencjalnie olbrzymiej, liczby. Jednakże taka metoda wpływa tylko na obniżenie stałej i asymptotycznie nie daje żadnych widocznych różnic, jeżeli mówimy o złożoności pamięciowej, gdyż nadal pamiętamy $|V| \cdot |E|$ elementów.

W przypadku macierzy sąsiedztwa złożoność pamięciowa wynosi $O(|V|^2)$, co nie powinno być dla nas zaskoczeniem, gdyż na obu osiach macierzy znajdują się wszystkie wierzchołki grafu. Oczywistym także jest, że zapotrzebowanie na pamięć będzie wzrastać im więcej informacji będziemy chcieć przechowywać w komórkach macierzy.

Dla samych list sąsiedztwa będziemy potrzebowali $O(|E|)$ pamięci, gdzie stała znowu może się różnić, w zależności od tego ile danych będziemy chcieli przechowywać w elementach list. Podczas tworzenia $|V|$ list incydencji mamy zagwarantowane, że żadnego łuku nie dodamy dwa razy oraz, że wszystkie łuki w grafie zostaną do nich dodane (łuk z definicji ma tylko jeden początek i koniec, więc jeśli dany łuk $e \in A(i)$ dla węzła v_i to na pewno nie należy do żadnego $A(j)$, gdzie $j \neq i$). Stąd elementów na wszystkich $|V|$ listach sąsiedztwa jest dokładnie $|E|$ elementów. Łącznie zatem, aby zaimplementować rozwiązania oparte na listach sąsiedztwa, potrzebujemy $O(|V| + |E|)$ pamięci.

Mamy zatem:

	Macierze Incydencji	Sąsiedztwa	Listy Sąsiedztwa	Pęki Wejścia-wyjścia
Potrzebna pamięć	$O(V \cdot E)$	$O(V ^2)$	$O(V + E)$	$O(V + E)$
Przegląd $v \in A(i)$	$O(E + A(i) \cdot V)$	$O(V)$	$O(A(i))$	$O(A(i))$
Dodawanie krawędzi ⁴	$O(1)$	$O(1)$	$O(1)$	$O(V + E)$
Dodawanie nowej krawędzi	$O(V \cdot E)$	$O(1)$	$O(1)$	$O(V + E)$
Usuwanie krawędzi ⁵	$O(V)$	$O(V)$	$O(E)$	$O(V + E)$
Trwałe usuwanie krawędzi	$O(V \cdot E)$	$O(V \cdot E)$	$O(E)$	$O(V + E)$
Stopień wierzchołka	$O(E)$	$O(V)$	$O(A(i))$	$O(1)$

Jak widać rozdzieliłyśmy operacje dodawania i usuwania elementów grafu, wyróżniając takie, które mają na celu tylko „zakryć” obecność danego elementu, bądź z powrotem przywrócić jego widoczność oraz na taki, których celem jest permanentna modyfikacja, przechowywanego w pamięci, grafu. Różnicę między tymi operacjami bardzo wyraźnie widać w przypadku korzystania z reprezentacji macierzowych grafu, gdzie „usunięcie” krawędzi możemy przeprowadzić w czasie zdecydowanie krótszym, niż byśmy mieli zmieniać rozmiar samych macierzy poprzez usuwanie/dodawanie elementów na którejkolwiek z ich osi. Chcąc „usunąć” z grafu daną krawędź wystarczy abyśmy zlokalizowali węzły, z którymi ma ona połączenie i w odpowiednich komórkach macierzy zamazali zapis o istniejącym połączeniu (założyliśmy, że w nasze grafy są skierowane tak więc znając łuk, który chcemy usunąć, mamy informację tylko o węźle, do którego dany łuk prowadzi - w obu przypadkach reprezentacji macierzowych zmusza nas to do dodatkowego przeszukania jednej z wybranych kolumn, w celu odszukania węzła, z którego łuk, który chcemy usunąć, wychodzi). Krawędź w grafie nadal będzie istnieć, lecz stanie się ona bezużyteczna, a dla algorytmów niezauważalna. Należy jednak tutaj podkreślić, że za takie traktowanie danych przyjdzie nam zapłacić, utrzymującym się na stałym poziomie, kosztem przeglądania macierzy w poszukiwaniu nastęników interesujących nas węzłów, zaś poza macierzowymi reprezentacjami różnice pomiędzy trwałym, a tymczasowym usuwaniem elementów nie ma żadnego wpływu na złożoność obliczeniową bez dodatkowych modyfikacji przedstawionych struktur.

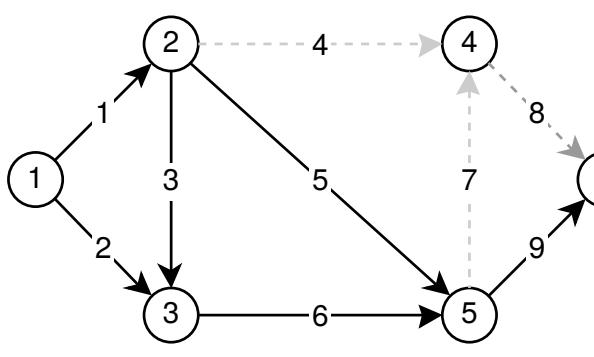
	Macierze Incydencji	Sąsiedztwa	Listy Sąsiedztwa	Pęki Wejścia-wyjścia
Dodawanie węzła	$O(V \cdot E)$	$O(1)$	$O(1)$	$O(V + E)$
Dodanie nowego węzła	$O(V \cdot E)$	$O(V ^2)$	$O(1)$	$O(V)$
Przysłonięcie węzła	$O(V \cdot E)$	$O(V)$	$O(E)$	$O(V + E)$
Trwałe usunięcie węzła	$O(V \cdot E)$	$O(V ^2)$	$O(E)$	$O(V + E)$

Poza takimi operacjami jak: dodawanie, usuwanie krawędzi grafu, wyznaczanie stopnia wierzchołka, jego nastęników możemy także chcieć na bieżąco modyfikować ilość węzłów, jakie znajdują się w grafie. Poniżej przedstawiono zestawienie czasów trwania wymienionych operacji dla wszystkich omówionych sposobów reprezentacji grafu. Podobnie jak w poprzednim przypadku, dla macierzy incydencji potrafimy wykonać operację „usuwania” węzła bez faktycznej zmiany rozmiarów tych macierzy, jednak w tym wypadku zysk z takiego postępowania jest niewielki, by wręcz nie powiedzieć: asymptotycznie żaden - podstawowy pomysł na wymazanie informacji o danym węźle jest usunięcie wszystkich danych o łukach, które do niego prowadzą tak, aby węzeł przestał być osiągalny, lecz (jak pokazaliśmy wyżej) każdorazowa operacja samego wymazania informacji o pojedynczej krawędzi już kosztuje nas $O(|V|)$. W najgorszym przypadku musielibyśmy usunąć wszystkie krawędzie w grafie, co daje nam łącznie złożoność operacji tymczasowego usuwania węzła $O(|V| \cdot |E|)$.

Nieco lepsze rezultaty jesteśmy w stanie osiągnąć z macierzami sąsiedztw, gdyż w tym przypadku odnalezienie wiersza/kolumny z danym węzłem, który chcemy usunąć, jest równoznaczne z odnalezieniem wszystkich łuków, które do niego prowadzą i, gdybyśmy zignorowali konieczność zmiany rozmiaru całej macierzy (chcieli tylko ukryć jej elementy), to otrzymalibyśmy górną ograniczenie na złożoność tej operacji na poziomie $O(|V|)$.

⁴W przypadku dodawania krawędzi znamy identyfikatory obu węzłów, z którymi połączony ma być dodawany łuk.

⁵Poza ostatnim przypadkiem, znany jest tylko identyfikator łuku i węzła, do którego dana krawędź prowadzi. Nie dotyczy to reprezentacji opartej na pęckach wejścia-wyjścia, gdzie znajomość identyfikatora, jaki posiada łuk, daje nam dostęp do identyfikatorów obu węzłów, które ta krawędź łączy.



(a)

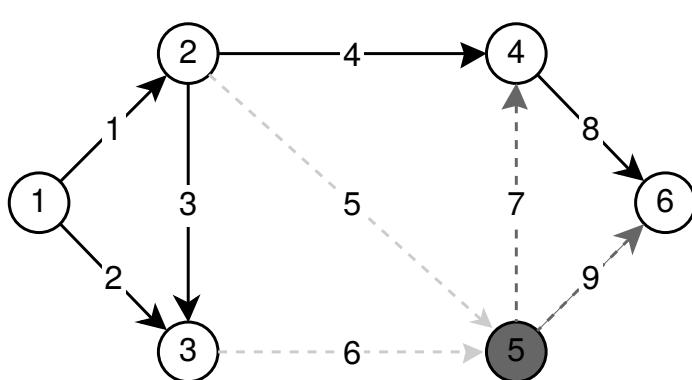
	1	2	3	4	5	6
1	0	1	2	0	0	0
2	0	0	3	4	5	0
3	0	0	0	0	6	0
4	0	0	0	0	0	8
5	0	0	0	7	0	9
6	0	0	0	0	0	0

(b)

Rysunek 1.6: **Ukrycie węzła w macierzy sąsiedztwa** (a) Graf skierowany $G = (V, E)$. (b) Macierz sąsiedztwa dla grafu G . Chcemy ukryć w niej informację o v_4 tak, aby nie był on osiągalny z żadnego innego węzła w grafie.

Dla takiej koncepcji, dodanie na powrót węzła do grafu jest tylko kwestią dodania jakiegokolwiek łuku, który będzie łączył istniejący w sieci węzeł z tym, który chcemy dodać, więc koszty tej operacji będą identyczne, do dla dodawania krawędzi z poprzedniej tabeli.

W przypadku pozostałych dwóch struktur znowu nie odnotujemy żadnej zmiany - aby usunąć dany węzeł z list sąsiedztwa będziemy musieli odnaleźć wszystkie łuki, które prowadzą do usuwanego węzła, a na koniec usunąć całą listę jego sąsiedztwa, wraz z łukami, które się na niej znajdują. W najgorszym przypadku będzie od nas to wymagało przeglądnięcia wszystkich krawędzi, występujących w grafie, co uczynimy w czasie $O(|E|)$. Dla pęków operacja usunięcia węzła jest jeszcze bardziej skomplikowana, gdyż nie istnieje w niej możliwość zaznaczenia konkretnego węzła, który chcielibyśmy ukryć, zaś jego usunięcie wymaga od nas odnalezienia obu zbiorów krawędzi (wychodzących z usuwanego węzła oraz do niego wchodzących), usunięcie ich z tablic, przechowujących o nich informacje (co uczynimy w czasie $O(A(i) + A^R(i))$ ⁶, zmiany ich rozmiarów ($O(|E|)$), usunięcia danego węzła z dwóch pozostałych tablic, indeksowanych od $1 \dots |V|$ (oraz zmiana ich rozmiarów - $O(|V|)$)) oraz na końcu aktualizacji tych ostatnich ($O(|V|)$) wraz z pomocniczą tablicą $mtab$, której rozmiar także trzeba zaktualizować ($O(|E|)$).



(a)

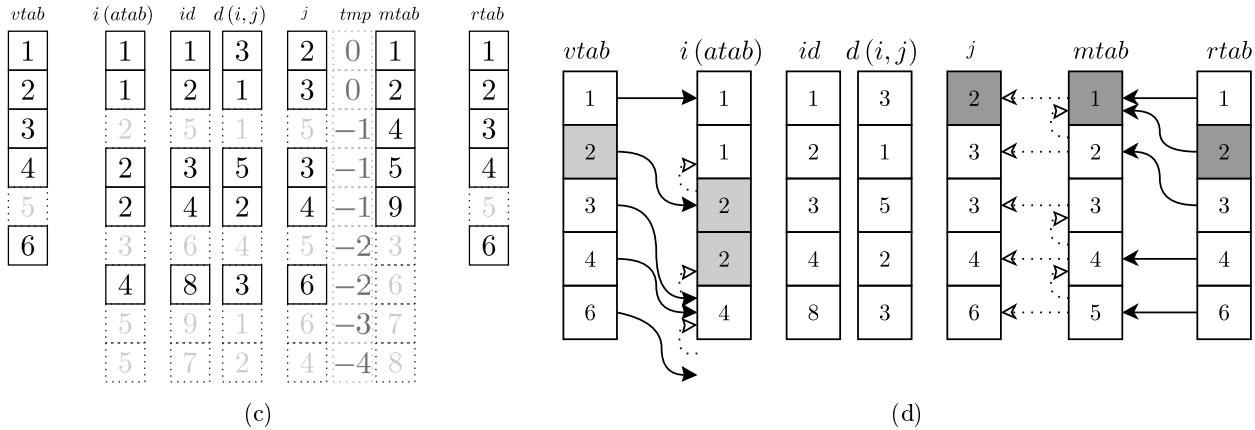
vtab	i (atab)	id	d(i, j)	j	mtab	rtab
1	1	1	3	2	1	1
2	1	2	1	3	2	2
3	2	5	1	5	3	3
4	2	3	5	3	4	4
5	2	4	2	4	5	5
6	3	6	4	5	6	6
	4	8	3	6	7	7
	5	9	1	6	8	8
	5	7	2	4		
	5	2	1			

(b)

Rysunek 1.7: **Usunięcie węzła dla pęków wejścia-wyjścia** (a) Graf skierowany $G = (V, E)$. Usuwamy węzeł v_5 oraz wszystkie łuki wychodzące (e_7, e_9) i wchodzące do danego węzła (e_5, e_6). (b) Pęki z oznaczonymi łukami do usunięcia, wyznaczonymi odpowiednio w czasie $O(A(5))$ dla krawędzi wychodzących oraz $O(A^R(5))$ dla przychodzących. Węzeł, który należy usunąć wyznaczamy w czasie $O(1)$.

⁶Poprzez $A^R(i)$ oznaczać będziemy zbiór sąsiadów węzła v_i , lecz będą to węzły bezpośrednio ten węzeł poprzedzające, z których krawędzie prowadzą do tego węzła.

1.3. PROBLEM NAJKRÓTSZYCH ŚCIEŻEK



Rysunek 1.7: (c) Tablice z usuniętymi powiązańiami - po usunięciu chcianych elementów odtworzymy je w czasie liniowym. Rekonstrukcja związków, zachodzących między tablicami $vtab$, a $atab$ wymagać będzie przejścia przez ich wszystkie elementy, wyjawszy krawędzie, wychodzące z ostatniego węzła, gdyż jego lista nastęników naturalnie kończy się wraz z końcem tablicy (po usunięciu wszystkich elementów tablice $vtab$ i $atab$ nadal będą prawidłowo posortowane). (d) Stan tablic po usunięciu węzła wraz z wszystkimi łukami. Odtworzenia własności tablicy $rtab$ oraz $mtab$ wymaga od nas większego nakładu pracy (ciąg wartości $j[mtab[1]], \dots, j[mtab[[E]]]$ musi tworzyć ciąg niemalejący). W tym celu wprowadzamy pomocniczą tablicę, w której zapiszemy różnice indeksów elementów, jakie nastąpią w tablicy j w trakcie usuwania krawędzi, a następnie aktualizujemy wszystkie elementy tablicy $mtab$ tak, aby $mtab[i] = mtab[i] + tmp[mtab[i]]$ (poza tymi, które wskazują na łuki, które będziemy usuwać - tutaj cztery ostatnie). (d) Sytuacja po usunięciu węzła v_5 i zaktualizowaniu wartości w tablicach. Szarym kolorem zaznaczono węzeł v_2 i powiązane z nim łuki.

W następnych rozdziałach spojrzymy już na sam problem najkrótszej ścieżki od strony zarówno formalnej jak i praktycznej - omówimy podstawowe własności problemu, zdecydujemy się na jeden ze, omówionych w powyższych rozdziałach, sposobów reprezentacji grafu, przedstawimy dodatkowe założenia, które ułatwiają nam rozwiązanie problemu, jaki przed sobą postawiliśmy. Na koniec rozdziału - tak jak pisaliśmy na samym jego początku - przedstawimy krótko praktyczne zastosowanie zdobytej przez nas wiedzy w postaci algorytmu Bellmana-Forda.

1.3 Problem najkrótszych ścieżek

Omówiliśmy sposoby w jaki efektywnie możemy reprezentować dane, potrzebne nam do wyznaczenia najkrótszej ścieżki z punktu v_p do węzła v_k w skierowanym grafie z cyklami $G = (V, E)$, nie definiując przy tym formalnie samego problemu najkrótszej ścieżki, gdyż do tej pory, do opisu wszystkich reprezentacji grafu G , wystarczały nam intuicje, podparte prostą logiką. W dalszych rozważaniach będziemy opierać się o następujące oznaczenia i definicje:

- **Ścieżką** od węzła v_p do węzła v_k będziemy nazywać każdą krawędź $e \in E$ w grafie $G = (V, E)$ taką, że ma ona swój początek w wierzchołku $v_p \in V$ i jest skierowana w stronę wierzchołka $v_k \in V$, gdzie ścieżka ta ma swój koniec. Z każdą ścieżką powiązana jest jej **waga** - dalej zwana również **kosztem** - c_{pk} , gdzie indeksy p i k odpowiadają indeksom węzłów: początkowego v_p oraz końcowego v_k , a którego wartość jest obliczana na podstawie, poniżej zdefiniowanej, **funkcji wagi**.
- **Funkcja wagi** - jest funkcją, na podstawie której jest obliczany **koszt** danej ścieżki e_{ij} . W pseudokodach będziemy ją zwykle oznaczać przez $d(v, u, \dots)$, gdzie ostatni argument (...) oznacza, że **koszt** danej ścieżki może być zależny od wielu dodatkowych parametrów. My będziemy koszt każdej ścieżki e_{ij} utożsamiać po prostu z jej długością i będziemy się do takiego kosztu odwoływać poprzez c_{ij} (ang. cost). Parametry funkcji $d(v, u, c)$ będą kolejno oznaczały:

v - węzeł początkowy o indeksie i ,

- u - węzeł początkowy o indeksie j ,
- c - koszt c_{ij} związany z łukiem e_{ij} , łączącym węzły v i u .

- Najkrótszą ścieżką ze źródła v_p do węzła v_k nazywać będziemy takim zbiorem $P = \langle v_0, v_1, \dots, v_k \rangle$, że suma kosztów c_{ij} ścieżek jest najmniejsza:

$$\sum_{e_{ij} \in P'} c_{ij} = \min : e_{ij} \in P' \Leftrightarrow v_i, v_j \in P \wedge v_i \rightsquigarrow v_j = e_{ij} \ni E. \quad (1.7)$$

gdzie przez $v_i \rightsquigarrow v_j$ będziemy oznaczać pojedynczą ścieżkę z węzła v_i do węzła v_j . Wprowadzimy także zapis $v_i \xrightarrow{k} v_k$, przez który będziemy rozumieć drogę złożoną z k ścieżek, prowadzących od punktu v_i do v_k (użyty symbol „ \ast ” zamiast liczby ścieżek oznacza, że nie interesuje nas konkretna ich ilość, tylko fakt istnienia ścieżki o zadanych właściwościach - z danym punktem początkowym i końcowym).

- Źródłem v_S będziemy nazywać węzeł, z którego rozpoczynamy wyszukiwanie najkrótszych ścieżek do wszystkich pozostałych węzłów w grafie i będzie to nasz podstawowy cel przy konstruowaniu wszystkich algorytmów, rozwiązywających problem najkrótszych ścieżek.

1.3.1 Reprezentacja problemu

Oprócz, omawianych już, własności naszej struktury oraz jej elementów składowych, takich jak koszt ścieżek c_{ij} , wspomnianych list sąsiedztwa, wprowadzimy także dodatkowe parametry dla węzłów, którymi będą:

- **identyfikator węzła (ID)** jednoznacznie określa dany węzeł.
- **poprzednik węzła (pred/Π)**, dalej zwany również jego **rodzicem**, który będzie determinował poprzedni węzeł na najkrótszej ścieżce (dla węzła v_i będzie wyznaczał v_{i-1} w $P = \langle v_0, v_1, \dots, v_{i-1}, v_i, \dots, v_k \rangle$),
- **waga najkrótszej ścieżki do węzła ($d(i)$)**, który dla węzła v_i będzie przyjmował zawsze wartość najmniejszego znanego, kosztu przejścia ze źródła do tego węzła - dalej będziemy mówić o górnym ograniczeniu na koszt najkrótszej ścieżki, co okaże się równoważne (jeśli węzeł v_i za wartość $d(i)$ przyjmie k to znaczy, że każda ścieżka $v_S \xrightarrow{*} v_i$, aby być tą najkrótszą musi mieć łączny koszt nie większy niż $d(i)$, czyli mniejszy lub równy k). Ponadto założymy, że dla każdego węzła j , do którego nie istnieje żadna ścieżka (w tym najkrótsza), bądź nie jest ona jeszcze znana, wartość $d(j) = \infty$ - wtedy dowolna ścieżka, która będzie nam pozwalała osiągnąć węzeł j natychmiastowo stanie się najkrótszą ścieżką, do niego prowadzącą. Formalnie:

$$d(i) \geq \delta(s, i) = \delta(v_s, v_i) = \begin{cases} \min \{c(s, i) : v_s \xrightarrow{*} v_i\} & \text{jeśli } \exists v_s \xrightarrow{*} v_i \\ \infty & \text{w przeciwnym przypadku} \end{cases} \quad (1.8)$$

gdzie:

$$c(p, k) = c(v_p, v_k) = \sum_{e_{ij} \in P} c_{ij} : P = \langle v_p, v_1, \dots, v_k \rangle \quad (1.9)$$

Do wszystkich atrybutów węzłów będziemy odwoływać się w dalszej części albo umieszczając jego nazwę w górnym indeksie, jak to robiliśmy do tej pory (np. v_i^{ID} oznaczał identyfikator węzła i), albo pisząc ją za operatorem kropki w przypadku, gdy górny indeks będzie potrzebny nam do czego innego (np. zapis $v_i^{(k)}.\Pi$ będzie oznaczał k 'tego rodzica⁷ węzła v_i).

Aby efektywnie móc rozwiązywać problem najkrótszych ścieżek musimy przyjąć kilka założeń, które wynikają zarówno ze specyfikacji samego problemu, z przyjętego modelu (rzeczywistych sieci drogowych) oraz

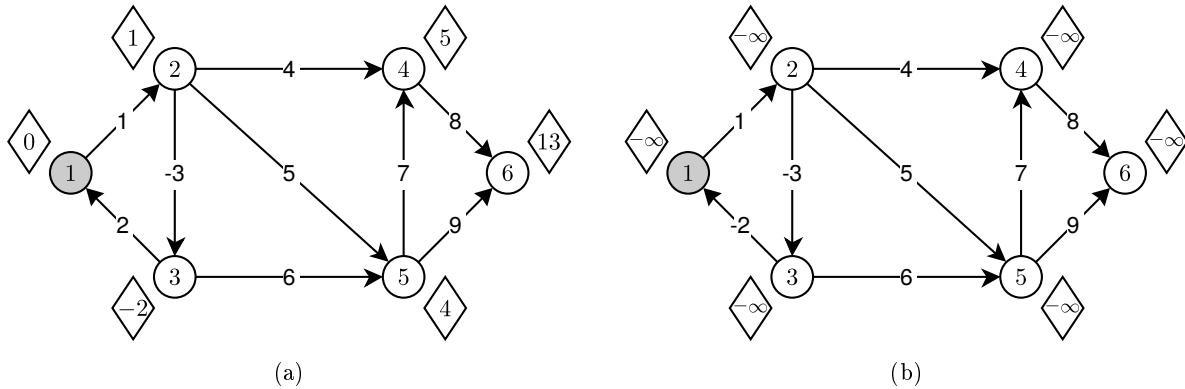
⁷W sensie takim, że dla $k = 1$ (domyślnie) wyrażenie $v_i^{(k)}.\Pi$ oznacza rodzica podanego węzła, dla $k = 2$ dziadka, dla $k = 3$ pradziadka itd.

1.3. PROBLEM NAJKRÓTSZYCH ŚCIEŻEK

ograniczeń, jakie nakłada na nas konieczność reprezentacji wszystkich danych w sposób zrozumiały dla komputera.

Na samym początku należy wspomnieć o sposobie numerowania podstawowych elementów grafu $G = (V, E)$. Do tej pory milcząco zakładaliśmy, że każdy wierzchołek $v \in V$ w grafie G ma przypisany swój własny, unikalny identyfikator, będący dodatnią liczbą całkowitą. Co więcej, identyfikatory te były przypisywane do tych wierzchołków w kolejności rosnącej ze skokiem o 1 tak, aby identyfikator ostatniego węzła równocześnie był liczbą wszystkich węzłów w grafie. Podobnie numerowaliśmy wszystkie krawędzie $e \in E$ - pozostało nam przy tych oznaczeniach dla prostoty omawianego problemu, lecz nic nie stoi na przeszkodzie, by zamiast zwykłych tablic (bo temu właśnie ma służyć taka numeracja elementów grafu) wykorzystać bardziej złożone struktury, bądź *tablice z haszowaniem*, które pozwoliłyby nam nazywać węzły dowolnie (mapując ich nazwy na liczby naturalne od 1 do $|V|$).

Bardziej restrykcyjnym założeniem o podobnym charakterze jest ograniczenie wartości wag wszystkich krawędzi (a co za tym idzie wag najkrótszych ścieżek w węzłach) do liczb całkowitych dodatnich. O ile jego obejście także nie stanowi większego problemu (wystarczy podnieść rząd wielkości wszystkich wartości tak, aby otrzymać liczby całkowite) to brak spełnienia tego warunku uniemożliwi nam efektywną konstrukcję pewnych wariantów algorytmów Dijkstry, które w dużej mierze opierają swoje działanie o te właśnie wartości (wspomniane algorytmy omówimy w rozdziale 2.5). Dodatkowo w międzyczasie przemyciliśmy kolejne bardzo ważne założenie, które poczynimy - żadna z wag krawędzi w naszym grafie nie będzie mogła przybrać wartości ujemnej. Założenie to nie tylko jest słusne z siecią drogową jako modelem, który sobie obraliśmy, lecz także bezpośrednio z niego wynika inna własność, którą chcielibyśmy, aby miała nasza sieć - brak cykli o ujemnej długości tj. brak takich zamkniętych ścieżek, którymi da się przejść, a których koszt całkowity jest niedodatni ($c(i, i) < 0$, gdzie v_i to odpowiednio „początek” i „koniec” cyklu).



Rysunek 1.8: **Graf skierowany z ujemnymi wagami na krawędziach** (a) Węzeł v_1 jest źródłem, na krawędziach umieszczono ich wagi, a w rombach przy węzłach, do których prowadzą odpowiednio koszty najkrótszych ścieżek do tych węzłów ($d(1) = 0$). W samych węzłach umieszczono ich identyfikatory. Pomimo wystąpienia krawędzi e_{23} o wadze $c_{23} < 0$ to długości najkrótszych ścieżek dla każdego z węzłów nadal są poprawnie wyliczone i przedstawiają faktyczny koszt najkrótszych ścieżek. (b) Wraz z pojawiением się cyklu o ujemnej wadze ($v_1 \rightsquigarrow v_2 \rightsquigarrow v_3 \rightsquigarrow v_1 \rightsquigarrow \dots$) wszystkie koszty w wierzchołkach stają się nieskończoność małe (z każdym kolejnym cyklem koszt dotarcia do węzłów w tym cyklu maleje, jednocześnie wpływając na koszt we wszystkich wierzchołkach, które są z tych węzłów osiągalne - dalej proces przebiega lawinowo, gdyż $-\infty + n = -\infty$). Oczywiście jest, że takie wyniki są dla nas bezwartościowe, gdyż informują co najwyżej o wystąpieniu w grafie ujemnego cyklu oraz o wierzchołkach, do których możemy dojść z jego wykorzystaniem.

Jak pokazano na rysunku 1.8, za poprawną ścieżkę będziemy traktować tylko takie ścieżki, które nie zawierają w sobie cykli o ujemnej długości. Dodatkowo też za takie ścieżki będziemy uważały wszystkie, które zawierają cykl także o koszcie dodatnim (z analogicznego powodu). Założymy, że istnieje najkrótsza ścieżka $P = \langle v_0, v_1, \dots, v_k \rangle$ taka, że $c(0, k) = D$ i zawiera ona cykl dowolnej, niezerowej długości (powiedzmy $c = \langle v_i, v_{i+1}, \dots, v_j \rangle$, gdzie $v_i = v_j$ oraz $c(i, j) \neq 0$). Nasza ścieżka zatem wygląda tak: $P = \langle v_0, v_1, \dots, v_i, v_{i+1}, \dots, v_j, \dots, v_k \rangle$ (bez straty ogólności założymy, że cykl nie znajduje się na żadnym z końców ścieżki) i ma ona koszt $c(0, k) = D$. Jeżeli chcielibyśmy usunąć teraz węzły cyklu od v_{i+1} do v_j to

otrzymamy następującą ścieżkę: $P' = \langle v_0, v_1, \dots, v_i, v_{j+1}, \dots, v_k \rangle$ o tym samym węźle początkowym i końcowym⁸, co poprzednia ścieżka (a więc „tą samą” ścieżkę), tyle że o zmienionym koszcie. Jeżeli usuneliśmy cykl o długości dodatniej, to nowa ścieżka będzie miała mniejszą wagę od ścieżki P , co czyni tą drugą dłuższą od P' - czyli ścieżka P nie mogła być tą najkrótszą. Analogicznie możemy postępować dla cyklu o ujemnej długości, tyle że w tym przypadku zamiast usuwać cykle, będziemy je dodawać, by dojść do tej samej sprzeczności, co w poprzednio.

Z tego faktu dodatkowo wynika, że każda najkrótsza ścieżka może posiadać maksymalnie $|V| - 1$ składowych krawędzi - gdyby posiadała ich więcej, znaczyłoby to, że na ścieżce występuje cykl, a wykluczyliśmy już taką możliwość (nie zajmowaliśmy się cyklami długości zero, gdyż takie zawsze możemy eliminować z najkrótszych ścieżek).

Ostatnimi założeniami, jakie przyjmiemy, będą te związane z poprawną reprezentacją obliczeń, jakie będziemy wykonywać podczas wykonywania algorytmów (a które jedno już przedstawiliśmy). Jako, że dopuszczać będziemy sytuacje, w których dla danego węzła v_i jego $v_i.d$ ⁹ będzie się równać $-\infty$ albo ∞ , konieczne jest zdefiniowanie operacji przy wykorzystaniu tych symboli nieoznaczonych. W związku z tym, będziemy przyjmować, że dla dowolnej liczby $n \neq \pm\infty$ zachodzić będą równości: $a + (\mp\infty) = (\mp\infty) + a = \mp\infty$.

W dalszej części, przy omawianiu algorytmów, będziemy posługiwać się jedną z, przedstawionych wcześniej, reprezentacji grafu - w naszym przypadku będą to listy sąsiedztwa, gdyż w najbardziej naturalny (a zarazem najprostszy) sposób wyrażają one wszystkie własności, z jakich przyjdzie nam korzystać podczas konstruowania algorytmów wyszukiwania najkrótszych ścieżek.

1.3.2 Podstawowe operacje

Omówimy teraz podstawowe operacje, z których będziemy bardzo często korzystać w trakcie budowania kolejnych algorytmów wyszukujących najkrótsze ścieżki.

Jedną z takich operacji jest niewątpliwie procedura inicjalizująca graf, na którym dany algorytm będzie pracować. Jak wspomnieliśmy w poprzednich rozdziałach, w przypadkach, gdy nie istnieje najkrótsza ścieżka, bądź nie mamy informacji o takowej, która by prowadziła do danego węzła i , wtedy wartość parametru tego węzła $d(i) = \infty$ - przed rozpoczęciem działania algorytmu o ścieżkach nie wiemy nic, zatem każdy wierzchołek inicjalizujemy w ten sposób, dodatkowo upewniając się, że żaden z nich nie posiada informacji o swoim rodzicu (jako, że takie informacje posłużą nam później do odtworzenia znalezionych, najkrótszych ścieżek). Źródło v_S - wierzchołek w grafie, od którego zaczniemy poszukiwania najkrótszych ścieżek - będzie miało ustawiony dystans ($d(S)$) na wartość równą zeru („najkrótszą ścieżką” do węzła v_S z węzła v_S jest ścieżka długości zero - założyliśmy brak ujemnych cykli oraz brak jakichkolwiek krawędzi o takim koszcie).

Algorithm 2: INIT-GRAF (G, s)

```

1 begin
2   for  $vIdx \in 1 \dots |V|$  do                                /* Dla każdego węzła w  $vtab$  */
3      $vtab[vIdx].d \leftarrow \infty$ 
4      $vtab[vIdx].Pi \leftarrow NULL$ 
5    $vtab[s].d \leftarrow 0$ 

```

Dwa rozdziały temu wprowadziliśmy kilka nowych oznaczeń dla atrybutów węzłów, z których od tamtej pory mieliśmy korzystać. Mówiliśmy, że dla każdego węzła v_i jego wartość $d(i)$ przyjmuje zawsze koszt równy najmniejszemu, znanemu kosztowi ścieżki, prowadzącej ze źródła do danego węzła, co formalnie możemy w skróconej formie wyrazić:

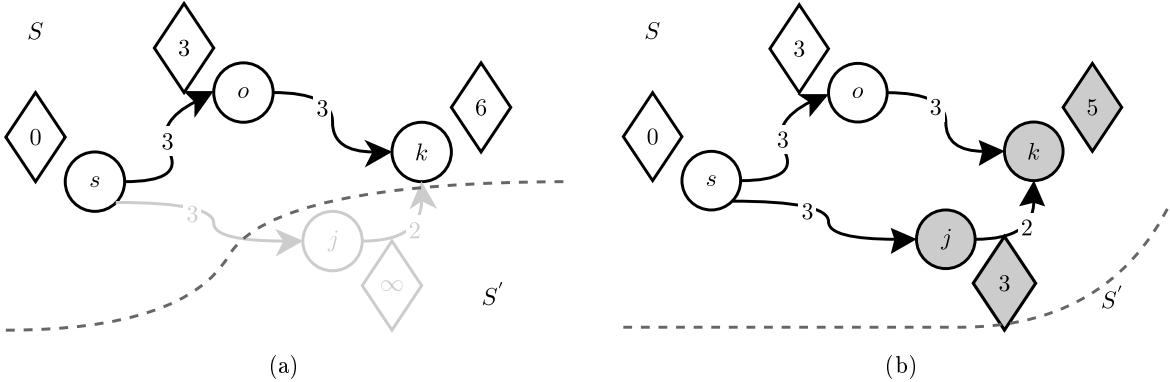
$$d(i) = \sum_{e_{jk} \in P} c_{jk} \rightarrow \min : P = \langle e_{Sj^{(1)}}, e_{j^{(1)}j^{(2)}}, \dots, e_{j^{(m)}i} \rangle \quad (1.10)$$

⁸Nie usuwamy pierwszego węzła cyklu, więc jeżeli cykl znajdowałby się na początku ścieżki to w oczywisty sposób węzeł v_p na ścieżce $v_p \xrightarrow{*} v_k$ nie uległby zmianie. Analogicznie w przypadku, gdyby cykl znajdował się na końcu tj. $P = \langle v_0, v_1, \dots, v_i, v_{i+1}, \dots, v_j \rangle$ i $v_j = v_k$, gdzie usunięcie cyklu spowodowałoby powstanie ścieżki $P' = \langle v_0, v_1, \dots, v_i \rangle$. Pamiętając, że węzły v_i oraz v_j były odpowiednio początkiem i końcem cyklu c , a co za tym idzie: $v_i = v_j = v_k$.

⁹ $v_i.d \equiv v_i^d \equiv d(i)$

gdzie P jest zbiorem krawędzi dla istniejącej ścieżki $v_S \xrightarrow{m+1} v_i$.

W czasie działania wszystkich algorytmów wyszukiwania najkrótszych ścieżek będziemy chcieli zachować tę własność, by w momencie zakończenia ich działania otrzymywać poprawne wyniki (czyli by $\forall v \in V d(i) = \delta(s, i)$). Aby to było możliwe, musimy wprowadzić kolejną operację, którą będziemy nazywać operacją relaksacji krawędzi. Przyjrzymy się sytuacji, która zobrazuje jej działanie.



Rysunek 1.9: **Relaksacja krawędzi** (a) Sytuacja przed dodaniem wierzchołka v_j do zbioru wierzchołków o optymalnych wartościach $d(i)$, gdzie $i \in \{i' : v_{i'} \in S\}$. (b) Dodanie do zbioru S wierzchołka v_j spowodowało powstanie nowej ścieżki $v_s \xrightarrow{*} v_k$, której koszt jest mniejszy od dotychczasowej $v_s \rightsquigarrow v_o \rightsquigarrow v_k$ i w efekcie reorganizację najkrótszej ścieżki węzła v_k . Etykieta $d(k)$ uległa zmniejszeniu, a $v_k.PI$ wskazuje teraz na węzeł v_j (poprzednio v_o).

Na rysunku 1.9 przedstawiona jest sytuacja w trakcie działania pewnego algorytmu wyszukiwania najkrótszych ścieżek, gdzie wyszczególniono wszystkie wierzchołki (oraz powiązane z nimi krawędzie), o których algorytm jeszcze nic nie wie (przyjmijmy, że zbiór tych wierzchołków będziemy oznaczać krótko jako S' - rys. 1.9a), gdyż zaczął je przeglądać od danego węzła v_S - źródła. W tej chwili wszystkie atrybuty $d(i)$ węzłów nie należących do zbioru S' (nazwijmy go zbiorem S) mają wartości, odpowiadające kosztom najkrótszych ścieżek od źródła do każdego z nich (są optymalne) i jest to sytuacja, którą chcemy utrzymać. Przyjmijmy teraz, że do zbioru wierzchołków S dołączamy wierzchołek $v_j \in S'$ (jednocześnie go stamtąd usuwając - rys. 1.9b). Po dodaniu wierzchołka v_j widzimy, że do v_k da się dojść krótszą ścieżką, niż to było przedstawione na rysunku 1.9a, a zatem $d(k)$ nie jest już długością najkrótszej ścieżki dla tego węzła. Podobnie węzeł $v_o = v_k^{\text{II}}$ nie należy już do zbioru węzłów, leżących na najkrótszej ścieżce $v_S \xrightarrow{*} v_k$. Łatwo zauważyc, że jedyną możliwą alternatywą dla najkrótszej ścieżki do węzła v_k jest przed chwilą powstała ścieżka, tak więc nowa wartość parametru $d(k) = \min\{d(o) + c_{ok}, d(k)\}$ ¹⁰.

Algorithm 3: RELAX (j, k, d)

```

1 begin
2   if  $k.d > j.d + d(j, k)$  then
3      $k.d \leftarrow j.d + d(j, k)$ 
4      $k.PI \leftarrow j$ 

```

/* Oznaczenia węzłów zachowane z rysunku 1.9 */
/* $d(j, k, \dots)$ - funkcja wagowa */

Algorytm relaksacji wierzchołków sprawdza, czy nowo dodany wierzchołek zaburza, interesującą nas, właściwość grafu. Jeżeli nie to następne kroki są pomijane. W przeciwnym przypadku aktualizowane jest wskazanie na rodzica d oraz wartość $d(i)$ dla wierzchołka, który tego wymaga. Metoda relaksacji przyjmuje trzy parametry, z którego dwa są wierzchołkami, a trzeci funkcją wagową krawędzi, którą zdefiniowaliśmy w podrozdziale 1.3.

Ostatnią operacją, którą chcielibyśmy móc wykonywać jest operacja przeglądania wierzchołków, które znajdują się na dowolnej najkrótszej ścieżce w grafie $G = (V, E)$, jako że sama informacja o długości takiej

¹⁰O tym, że tak jest w istocie przekonamy się w następnym rozdziale.

ścieżki nie zawsze okazuje się wystarczająca. Przy omawianiu dwóch poprzednich algorytmów na równi z operowaniem na atrybutach węzłów $d(i)$ pozwalaliśmy sobie zmieniać także atrybut, odpowiadający wskazaniu na rodzica danego węzła (Π), doprowadzając zawsze do sytuacji, w której rodzicem danego węzła, leżącego na najkrótszej ścieżce, zawsze był węzeł, który także na niej się znajdował, będąc jednocześnie o jedną krawędź bliżej źródła, od którego dana ścieżka się rozpoczęła. Innymi słowy dbaliśmy o to, by dla każdej najkrótszej ścieżki $P = \langle v_0, v_1, \dots, v_k \rangle$ dla każdego $v_i : i \in \{1, \dots, k\}$ zachodziła prawidłowość: $v_i.\Pi = v_{i-1}$ (dla $i = 0$ w oczywisty sposób $v_0.\Pi = \text{NULL}$), a zatem nasz algorytm, pozwalający nam na wypisanie po kolejnych węzłach na danej, najkrótszej ścieżce $v_0 \xrightarrow{k} v_k$, wyglądać mógłby dla przykładu tak:

Algorithm 4: WRITE-PATH (v)

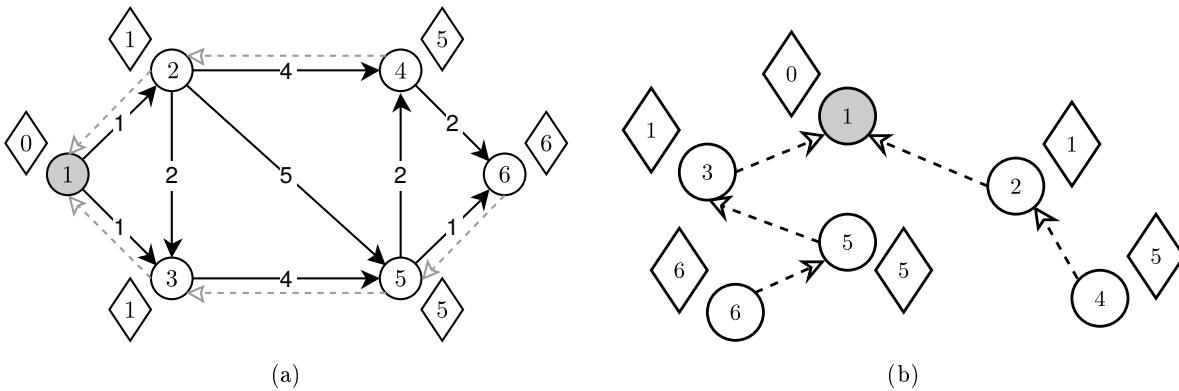
Result: Ścieżka, wypisane w kolejności od węzła docelowego do źródła.

```

1 begin
2   while  $v.\Pi \neq \text{NULL}$  do
3     Wypisz  $v$ 
4      $v \leftarrow v.\Pi$ 
5   Wypisz  $v$ 

```

gdzie na wejściu musielibyśmy podać węzeł, do którego ścieżkę chcemy wypisać, pamiętając, że będzie to ścieżka wypisana w odwrotnej kolejności i że nie jesteśmy w stanie wypisać najkrótszych ścieżek innych, niż te, które swój początek mają w źródle (zbiór wszystkich takich ścieżek tworzy swego rodzaju drzewo, którego korzeniem jest właśnie ten węzeł - rys. 1.10).



Rysunek 1.10: **Poddrzewo najkrótszych ścieżek** (a) Graf $G = (V, E)$ z obliczonymi odległościami najkrótszych ścieżek dla wszystkich węzłów, gdzie przerywanymi liniami zaznaczone są wskazania na poprzedników każdego z nich. W przypadku braku takiej krawędzi dla danego węzła v zakładamy, że $v.\Pi = \text{NULL}$. (b) Poddrzewo grafu, zawierające najkrótsze ścieżki od źródła do wszystkich pozostałych węzłów w grafie G , wraz z ich kosztami.

Oczywiście nic nie stoi na przeszkodzie, byśmy zastosowali jedną z technik, by odwrócić kolejność takiego wypisywania (np. wpisać wyniki do kolejki FIFO, by później z niej odczytać wartości w kolejności od v_0 do v_k czy też zastosować rekurencję).

1.3.3 Właściwości najkrótszych ścieżek

Algorytmy, które będziemy omawiać w następnych rozdziałach, poza czerpaniem pomysłów na ich implementacje z wielu innych zagadnień informatycznych (jak się przekonamy), w głównej mierze oparte będą na właściwościach najkrótszych ścieżek, które przytoczymy w tym podrozdziale bez dowodów (Czytelnik może je wszystkie odnaleźć w dodatku TODO), wierząc, że pomoże się to skupić Czytelnikowi na samych algorytmach i dowodach ich poprawności, zwłaszcza, że zdecydowana większość właściwości najkrótszych ścieżek jest naturalna i łatwa do zrozumienia.

Lemat 1.3.1 (Własność trójkąta) Dla każdej krawędzi $e_{ij} \in E$ w zachodzi $\delta(v_k, v_j) \leq \delta(v_k, v_i) + c_{ij}$.

Lemat 1.3.2 (Własność optymalnej podstruktury) Jeśli w ważonym grafie skierowanym $G = (V, E)$ z funkcją wagową $w : E \leftarrow \mathbb{R}$ (w naszym przypadku wagi są utożsamiane z kosztem przejścia między jednym węzłem, a drugim, wyrażonym - bez straty ogólności - w liczbach naturalnych, gdyż zajmujemy się rzeczywistymi sieciami drogowymi, w których odległości między dowolnymi dwoma punktami mierzymy w całkowitych wielokrotnościach przyjętej jednostki długości) najkrótszą ścieżką z wierzchołka v_p do v_k jest ścieżka $P = \langle v_p, v_{p+1}, \dots, v_k \rangle$ to dla każdych i, j takich, że $p \leq i \leq j \leq k$ podścieżka $P_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ jest najkrótszą ścieżką z v_i do v_j .

Lemat 1.3.3 (Własność braku ścieżki) Jeśli nie istnieje ścieżka z s do v_i to zawsze $v_i.d = \delta(s, i) = \infty$

Lemat 1.3.4 (Własność ścieżki z ujemnym cyklem) Jeśli istnieje ścieżka $P = \langle v_p, v_{p+1}, \dots, v_k \rangle$ z v_p do v_i , na której znajduje się cykl o ujemnej wadze $P = \langle v_i, v_{i+1}, \dots, v_j \rangle$, gdzie $p \leq i \leq j \leq k$ to zawsze $v_k.d = \delta(p, k) = -\infty$

Lemat 1.3.5 (Własność górnego ograniczenia) Dla każdego wierzchołka $v_i \in V$ zachodzi $v_i.d \geq \delta(s, v_i)$, gdzie wartość $v_i.d$ monotonicznie maleje i w momencie osiągnięcia swojego dolnego ograniczenia $\delta(s, v_i)$ przestaje ulegać zmianie.

Lemat 1.3.6 (Własność zbieżności) Jeśli w grafie ważonym $G = (V, E)$ istnieje najkrótsza ścieżka $P = \langle v_p, v_{p+1}, \dots, v_{k-1}, v_k \rangle$ i w dowolnym momencie przed relaksacją krawędzi e_{kk-1} zachodzi $v_{k-1}.d = \delta(v_p, v_{k-1})$ to po tej relaksacji $v_k.d = \delta(v_p, v_k)$.

Lemat 1.3.7 (Własność relaksacji dla ścieżki) Jeśli $P = \langle v_p, v_{p+1}, \dots, v_k \rangle$ jest najkrótszą ścieżką z $s = v_p$ do v_k i wykonamy szereg relaksacji jej krawędzi w kolejności od e_{pp+1} do e_{k-1k} to $v_k.d$ będzie równe $\delta(s, v_k)$ niezależnie od kolejności wykonywania pozostałych relaksacji.

Lemat 1.3.8 (Własność podgrafu poprzedników) Jeśli dla każdego węzła v_i w grafie $G = (V, E)$ zachodzi $v_i.d = \delta(s, v_i)$ to podgrafem poprzedników grafu G jest drzewo o korzeniu w węźle s i krawędziach, będących odwzorowaniem wszystkich najkrótszych ścieżek w tym grafie.

1.3.4 Algorytm Bellmana-Forda

Ostatnim naszym krokiem w tym rozdziale będzie przedstawienie prostego algorytmu, wykorzystującego całą naszą, dotychczas zdobytą, wiedzę, do wyznaczenia najkrótszych ścieżek w zadanym, skierowanym grafie acyklicznym $G = (V, E)$ z nieujemnymi wagami na krawędziach. Algorytm, o którym będziemy mówić w tym rozdziale, jest oparty na właściwościach najkrótszych ścieżek, które omówiliśmy powyżej i bezpośrednio z nich wynika dowód jego poprawności, który także przytoczymy. Bardzo prostą ideą omawianego algorytmu jest wykonanie operacji relaksacji krawędzi w rundach aż „do skutku”. Pod pojęciem rundy będziemy rozumieli przeprowadzenie takiej operacji dla każdej krawędzi w grafie dokładnie jeden raz, zaś wykonywanie rund będziemy powtarzać do momentu, w którym wartości $d(i)$ każdego wierzchołka V_i w grafie osiągną wartości równe rzeczywistym wagom najkrótszych ścieżek $v_s \xrightarrow{*} v_i$ - dalej, dla zachowania prostoty, będziemy te wartości zapisywać w skrócie jako: $\delta(s, i)$. Naszym celem zatem staje się już tylko znalezienie takiej minimalnej ilości rund, po których dla każdego wierzchołka $v_i \in G$ zachodzi równość $d(i) = \delta(s, i)$ (dla źródła s). W tym momencie warto przypomnieć sobie, że w naszym modelu rzeczywistych sieci drogowych przyjęliśmy brak krawędzi, których koszt byłby mniejszy lub równy 0, a co za tym idzie najkrótsze ścieżki, które będziemy chcieli odnaleźć, nie będą składać się z więcej niż $|V| - 1$ elementów i przechodzić przez więcej niż $|V| - 2$ wierzchołków, nie wliczając to wierzchołka początkowego i końcowego - zwróciliśmy już uwagę na tę własność najkrótszych ścieżek pod koniec podrozdziału 1.3.1.

Powyżej została przedstawiona podstawowa wersja algorytmu Bellmana-Forda, której pesymistyczna złożoność, jeżeli chodzi o ilość wykonywanych operacji, da się w oczywisty sposób oszacować przez $O(|V| \cdot |E|)$ - wykonujemy $O(|V|)$ razy instrukcję (3) = (4), gdzie każda z nich wykonuje dokładnie $|V|$ operacji RELAX. Druga pętla, którą widzimy, przegląda raz jeszcze wszystkie krawędzie w grafie, upewniając się, że nie

Algorithm 5: BELLMAN-FORD (G, s)

```

1 begin
2   for  $i = 1$  to  $|V| - 1$  do
3     forall the  $(u, v) \in E$  do
4        $\quad\quad\quad$  RELAX( $u, v$ )
5   forall the  $(u, v) \in E$  do
6     if  $v.d > u.d + c_{uv}$  then
7        $\quad\quad\quad$  return FALSE
8   return TRUE

```

można wykonać na którejkolwiek z nich dodatkowej relaksacji, czyli innymi słowy - czy nasz algorytm zakończył swoje działanie i dał poprawny wynik. Jeżeli po wykonaniu pierwszej z pętli znajdziemy taką krawędź $(u, v) \in E$, że spełniony będzie warunek $v.d > u.d + c_{uv}$ to niechybny znak, że w gafie, w którym szukaliśmy najkrótszych ścieżek od źródła s do wszystkich pozostałych węzłów, występuje cykl o ujemnej długości. Do takiego wniosku dojdziemy, wykonując proste rozumowanie:

Załóżmy, że mamy w grafie cykl $C = \langle v_0, v_1, \dots, v_k \rangle$, gdzie $v_0 = v_k$, a ponadto, że - mimo jego występowania - algorytm po wykonaniu drugiej pętli zwróci nam wartość pozytywną **TRUE** (czyli, że dla każdej krawędzi $(u, v) \in E$ zaszła nierówność $v.d \leq u.d + c_{uv}$ w tym i dla wszystkich krawędzi, należących do cyklu). Jak wspomnieliśmy w podrozdziale 1.3.1, cyklem ujemnym nazywamy taką zamkniętą ścieżkę, po której jesteśmy w stanie przejść, a całkowity koszt przejścia po wszystkich krawędziach tego cyklu wynosi $c(v_0, v_k) < 0$ (nierówność 1.9). Dokładniej:

$$\exists C = \langle v_0, v_1, \dots, v_k \rangle \wedge v_0 = v_k \wedge \sum_{i=1}^k c(v_{i-1}, v_i) = c(v_0, v_k) < 0 \Leftrightarrow C - \text{cykl o ujemnej długości} \quad (1.11)$$

Kluczową dla naszego rozumowania okaże się nierówność $\sum_{i=0}^{k-1} c(v_i, v_{i+1}) < 0$ oraz fakt, że z definicji cyklu $v_0 = v_k$ gdzie $v_0, v_k \in C$. Z założenia, że algorytm Bellmana-Forda zwraca wartość **TRUE** mamy $\forall_{v_i \in C} v_i.d \leq v_{i-1}.d + c_{v_{i-1}v_i}$, gdzie sumując po $v_1, \dots, v_k \in C$ otrzymujemy:

$$\sum_{i=1}^k v_i.d \leq \sum_{i=1}^k (v_{i-1}.d + c_{v_{i-1}v_i}) = \sum_{i=1}^k v_{i-1}.d + \sum_{i=1}^k c(v_{i-1}, v_i) = \sum_{i=1}^k v_{i-1}.d + c(v_0, v_k) \quad (1.12)$$

Z własności cyklu: $v_0 = v_k$ możemy wyprowadzić taki ciąg równości:

$$\sum_{i=0}^k v_i.d = v_0.d + \sum_{i=1}^{k-1} v_i.d + v_k.d = \sum_{i=1}^k v_{i-1}.d + v_k.d = v_0.d + \sum_{i=1}^k v_i.d \quad (1.13)$$

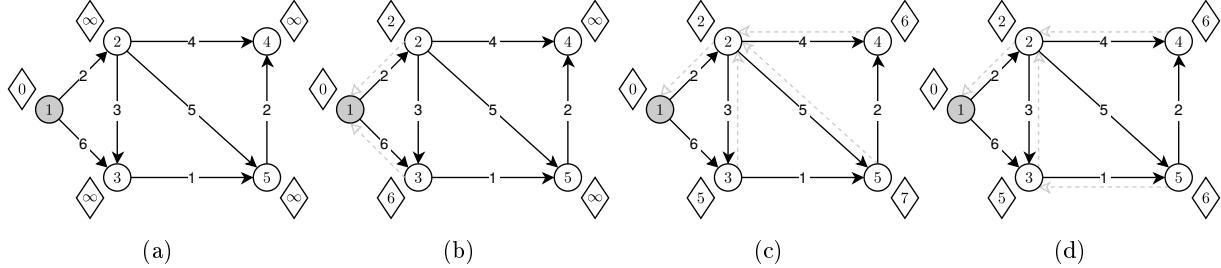
zaś z obu wyprowadzeń w ostateczności otrzymujemy:

$$\begin{aligned} \sum_{i=1}^k v_i.d &\leq \sum_{i=1}^k v_{i-1}.d + c(v_0, v_k) \\ \sum_{i=1}^k v_i.d &= \sum_{i=1}^k v_{i-1}.d \\ 0 &\leq c(v_0, v_k) \end{aligned}$$

co stoi w sprzeczności z tym, że na naszej ścieżce znajduje się cykl o ujemnej długości ($c(v_0, v_k) < 0$). Widzimy więc, że algorytm Bellmana-Forda, podobnie zresztą jak wszystkie, omawiane w tej pracy algorytmy,

nie radzi sobie z wyszukiwaniem najkrótszych ścieżek w grafach, gdzie występują cykle o długości ujemnej (jak pokazaliśmy na przykładzie 1.8).

Sam zaś algorytm można nieco usprawnić, by nie wykonywał niepotrzebnych operacji i przez to działał szybciej od swojego pierwotnego. Nie są to jednak zmiany na tyle duże, by miały jakikolwiek wpływ na pesymistyczną złożoność algorytmu.



Rysunek 1.11: **Działanie algorytmu Bellmana-Forda** (a) Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAPH ze źródłem $v_s.id = 1$. (b) Warunek $v.d > u.d + c_{uv}$ dla krawędzi (u, v) spełniony jest tylko dla krawędzi: $(1, 2)$ i $(1, 3)$ dla tych węzłów (v_2 i v_3) zostały zaktualizowani ich poprzednicy (zaznaczeni szarymi strzałkami) oraz etykietę d . Dla pozostałych algorytm nie wprowadzi żadnych zmian w trakcie iterowania po wszystkich $A(i) : i \in \{1, \dots, 5\}$. (c) Przyjęliśmy kolejność iterowania po wszystkich łukach (pętla 3–4) zgodną z kolejnością ponumerowania węzłów na rysunkach. Przyjmijmy ponadto rosnącą kolejność identyfikatorów węzłów, do której łuki prowadzą tj. podczas drugiej iteracji algorytm wykonuje operację RELAX na krawędziach w kolejności: $(1, 2)$, $(1, 3)$ (dla których relaksacja nie wprowadzi żadnych zmian), $(2, 3)$ (zostaje zaktualizowany węzeł v_3 - jego wartość d przyjmie długość odnalezionej, krótszej ścieżki oraz otrzyma nowego rodzica), $(2, 4)$, $(2, 5)$, (d) $(3, 5)$ i $(5, 2)$. Dla normalnej wersji algorytmu powinniśmy wykonać jeszcze 3 iteracje (z $|V| - 1$) po wszystkich krawędziach, jednak wprowadziliśmy modyfikację, która przerywa działanie algorytmu, jeżeli podczas pełnej iteracji nie nastąpi w grafie G żadna zmiana.

Usprawnienie algorytmu

Pierwsze co możemy zauważyć to fakt, że jeśli uporządkujemy wszystkie krawędzie e_{ij} względem pierwszego indeksu (czyli tak, aby kolejność przeglądania łuków wierszach 3–4 była podyktowana przeglądanyymi węzłami, z których dane łuki wychodzą) to możemy pomijać relaksacje tych wszystkich krawędzi, które wychodzą z węzła v , do którego jeszcze nie ma wyznaczonej ścieżki ($v.d = \infty$), gdyż warunek na jej wykonanie nigdy nie zajdzie ($k.d > \infty + d(j, k)$). W wybranym przez nas sposobie prezentowania danych grafu (jako listy sąsiedztwa) taka kolejność przeglądania krawędzi jest naturalna (chcąc przejść przez wszystkie krawędzie w grafie będziemy kolejno przeglądać zawartość list $A(i) : v_i \in G$). Dodatkowo - jak już zaznaczono na rysunku 1.11d - jeżeli w czasie wykonywania relaksacji wszystkich krawędzi w grafie nie nastąpi żadna zmiana to nie wystąpią one także podczas następnych iteracji głównej pętli algorytmu, a zatem możemy jej wykonanie przerwać, nie czekając aż wykona się ona dokładnie $|V| - 1$ razy.

Poprawność działania

Dowód poprawności działania takiego algorytmu dla grafu $G = (V, E)$, który nie zawiera ujemnych cykli (dla których pokazaliśmy już, że omawiany algorytm nie działa) jest natychmiastowy, jeżeli powołamy się na odpowiednie własności najkrótszych ścieżek.

Z lematu 1.3.7 wiemy, że jeśli w grafie istnieje najkrótsza ścieżka $P = \langle v_p, v_{p+1}, \dots, v_k \rangle$ i wykonamy dla jej krawędzi relaksację w odpowiedniej kolejności to $v_k.d$ będzie się równać $\delta(s, v_k)$. Wiemy także, że każda najkrótsza ścieżka w grafie składać się może maksymalnie z $|V| - 1$ krawędzi. Z każdym nawrotem pętli głównej algorytmu Bellmana-Forda wykonywana jest relaksacja wszystkich krawędzi w grafie G , zaś pętla ta jest powtarzana dokładnie $|V| - 1$ razy. Podczas każdej i -tej iteracji w szczególności wykonamy relaksację dla wszystkich krawędzi na ścieżce P , a zwłaszcza dla e_{pp+1} (w pierwszej iteracji), dla e_{p+1p+2} i dla każdej następnej. W najgorszym przypadku, zależnym od faktycznej kolejności przeglądania krawędzi, ostatnią z nich na ścieżce P będziemy relaksować w iteracji $k - p + 1$ (w przypadku, gdy dla k pierwszych i -tych iteracji

$i = 1, 2, \dots |V| - 1$ będziemy wykonywali relaksację tylko jednej krawędzi ze ścieżki P : łączącej węzeł $v_{p+(i-1)}$ z następnym węzłem $v_{p+(i-1)+1}$, gdzie po ich wykonaniu $v_k.d = \delta(s, v_k)$. Analogiczne rozumowanie możemy przeprowadzić dla każdej najkrótszej ścieżki w grafie.

1.4 Uwagi do rozdziału

W tym rozdziale zapoznaliśmy Czytelnika z wszystkimi, podstawowymi pojęciami, które mają, przy najmniej taką mamy nadzieję, pomóc mu w pełniejszym zrozumieniu dalszej części, gdzie skupimy się na omawianiu kilkunastu algorytmów wyszukiwania najkrótszej ścieżki wraz z ich modyfikacjami, które pozwolą im na jeszcze szybsze działanie. Większą częścią z nich będą algorytmy oparte na podstawowym pomyśle holenderskiego informatyka Edsgera Dijkstry, którym poświęcimy cały następny dział. Podobnie jak algorytm Bellmana-Forda, będą one także służyły do znajdowania najkrótszej ścieżki z pojedynczego źródła w grafie bez ujemnych cykli, lecz tym razem nie będą mogły w nim występować krawędzie o takim koszcie ze względu na sposób realizacji algorytmu.

1.4. UWAGI DO ROZDZIAŁU

Najkrótsze ścieżki z jednym źródłem

W poprzednim rozdziale omówiliśmy prosty algorytm wyszukiwania najkrótszych ścieżek w charakterze przykładu na wykorzystanie w praktyce wcześniej omówionych zagadnień. Doszliśmy do wniosku, że algorytm wykonuje ogromną liczbę operacji, w tym większość z nich niepotrzebnie (jak na przykład próby relaksacji krawędzi, wychodzących z wierzchołków, do których algorytm jeszcze nie potrafił dojść, wykorzystując wcześniej obliczone ścieżki), starając się zminimalizować ilość tych ostatnich, poprzez wprowadzenie dodatkowych warunków do naszej implementacji. Ich obecność pozwalała mieć nadzieję na efektywniejsze działanie algorytmu, jednak asymptotycznie nie uzyskaliśmy żadnej poprawy, nadal oszacowując czas działania algorytmu na ograniczony przez $O(V \cdot E)$. Nie umknął też naszej uwadze fakt, iż od kolejności wykonywanych relaksacji głównie zależy ilość operacji, jakie algorytm musi wykonać, aby zwrócić poprawny wynik i zakończyć pracę. Nic więc dziwnego, że rozwój algorytmiki zaowocował zaproponowaniem wielu innych rozwiązań tego samego problemu, skupiając się przede wszystkim na wymuszeniu takiej kolejności operacji, aby algorytm wykonywał ich jak najmniej.

2.1 Sortowanie topologiczne

Aby przekonać się o skuteczności takiego podejścia, przedstawimy prosty algorytm **sortowania topologicznego**, z którego pomocą będziemy mogli odnaleźć wszystkie najkrótsze ścieżki w skierowanym grafie ważonym $G = (V, E)$ w czasie liniowym (co jest ogromnym skokiem wydajnościowym, jeżeli chodzi o kwadratową złożoność algorytmu Bellmana-Forda)! Niestety, jak się będziemy mieli okazję przekonać, algorytm **sortowania topologicznego** narzuca nam bardzo silne ograniczenie na postać grafu, dla którego będziemy chcieli dokonywać obliczeń, przez co algorytm ten nie będzie już taki atrakcyjny, jakim wydawał się na początku, jednak będzie nadal wystarczająco skuteczny, by zastosować go w celu wyszukiwania najkrótszych ścieżek.

Sortowanie topologiczne polega na takim posortowaniu wierzchołków, aby dla każdej pary (v_i, v_j) , w przypadku istnienia krawędzi pomiędzy tymi wierzchołkami, prowadzącej z V_i do v_j , w posortowanym ciągu v_i znajdował się przed wierzchołkiem, do którego dana krawędź prowadzi. Innymi słowy sortowanie topologiczne prowadzi do ustalenia możliwej kolejności odwiedzeń wszystkich wierzchołków w grafie. Jako przykład w literaturze najczęściej można spotkać problem stworzenia listy kolejno wykonywanych czynności na podstawie posiadanego grafu, przedstawiającego zależności między poszczególnymi czynnościami (na przykładzie pieczenia ciasta, bądź kolejności zakładania ubrań).

Mówiąc o kolejności w grafie oczywistym więc jest, że nie uda nam się ustalić odpowiedniego porządku dla grafów, które będą zawierały cykle, lecz - jak się przekonamy - dla takich danych jesteśmy w stanie ustalić wystarczająco dobry porządek wśród wierzchołków, aby bardzo szybko obliczyć wszystkie najkrótsze ścieżki w zadanym grafie. Poniżej przedstawimy dwie popularne metody wyznaczania takiego porządku, gdzie pierwsza z nich okazać całkowicie bezradna w obliczu wystąpienia cyklu, zaś druga będzie je po prostu ignorować. Należy wyraźnie podkreślić, że w tym drugim przypadku dla grafu zawierającego cykle jako wynik nie otrzymamy porządku topologicznego, lecz uporządkowanie do niego podobne.

2.1.1 Algorytm Khana

Jednym z naturalnych sposobów na wyznaczenie opisanej przez nas kolejności wierzchołków jest rozpoznanie ich spisywania od tych, do których nie prowadzą żadne krawędzie - powiedzmy, że takie wierzchołki

2.1. SORTOWANIE TOPOLOGICZNE

nie mają żadnych „wymagań” by mogły być odwiedzone. Skoro odwiedziliśmy już wszystkie takie wierzchołki, możemy założyć, że pewne „wymagania”, które te wierzchołki sobą reprezentują, zostały spełnione i posiadamy nieco większe „możliwości”, by czynić zadość „wymaganiom” pozostałych wierzchołków. Takie rozumowanie, przeprowadzone dla wszystkich wierzchołków w grafie bez cykli da nam listę, na którą zostały spisane wierzchołki w porządku topologicznym. Łatwo sobie wyobrazić analogiczne rozumowanie w przypadku wystąpienia cyklu: gdy wierzchołek v_A na swojej „liście wymagań” posiada takie, by przed jego odwiedzeniem był odwiedzony węzeł v_B (co przedstawilibyśmy na grafie w postaci krawędzi z v_B do v_A), a wierzchołek v_B - by przed jego odwiedzeniem był odwiedzony węzeł v_A . Widzimy, że żadnego z tych warunków nie da się spełnić. Algorytm, opisujący takie działanie, wyglądać mógłby następująco:

Algorithm 6: KHAN-TOPOLOGICAL-SORT (G)

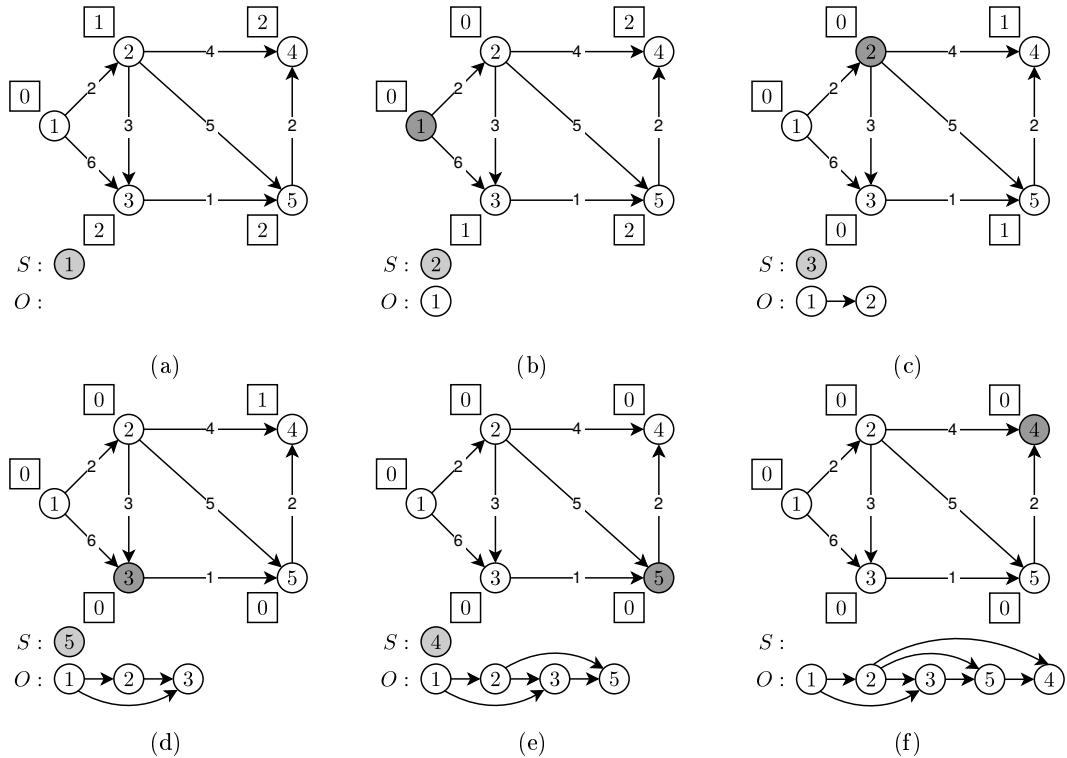
```

Input: Graf  $G = (V, E)$ .
Result: Lista  $O$  z posortowanymi topologicznie wierzchołkami.

1 begin
2    $S \leftarrow$  zbiór wszystkich wierzchołków, do których nie prowadzą żadne krawędzie
3    $O \leftarrow \emptyset$ 
4   while  $S$  nie jest pusta do
5     Przepnij wierzchołek  $v_i$  z  $S$  na koniec listy  $O$ 
6     foreach  $e_{ij} : v_i \xrightarrow{1} v_j$  do
7       Usuń krawędź  $e_{ij}$  z grafu  $G$ 
8       if do  $v_j$  nie wchodzą już żadne krawędzie then
9         Wstaw wierzchołek  $v_j$  do  $S$ 
10    if w grafie  $G$  nadal są wierzchołki then
11      return NULL
12    else
13      return  $O$ 
```

Algorytm działa niemal identycznie jak w przeprowadzonym przez nas rozumowaniu. Za punkt wyjścia obieramy listę wierzchołków, do których nie prowadzą żadne krawędzie, a następnie usuwamy wszystkie te, które wychodzą od wierzchołków, które już odwiedziliśmy. W momencie, gdy do jakiegoś węzła przestają prowadzić krawędzie w grafie, dodajemy go do listy węzłów, które sekwencyjnie odwiedzamy. Jest to sytuacja równoważna ze spełnieniem wszystkich „wymagań”, by dany wierzchołek móc odwiedzić. To co może okazać się problematyczne to zdobycie informacji na temat ilości krawędzi, wchodzących do każdego pojedynczego wierzchołka, gdyż - jak pamiętamy - zdecydowaliśmy się na przedstawienie struktury połączeń w grafie za pomocą list sąsiedztwa, które nam takiej wiedzy nie dają. Jest on jednak bardzo prosty do rozwiązania, gdyż aby takie informacje zdobyć musimy przejść po wszystkich krawędziach w grafie, gromadząc w osobnej tablicy $deg[1 \dots |V|]$ informację o liczbie łuków, wchodzących do poszczególnych węzłów o identyfikatorach z zakresu od 1 do $|V|$ (przy czym nie interesuje nas nic poza ich liczbnością). Czas, jaki potrzebujemy na jednorazowe przejście po wszystkich krawędziach grafu G jest oczywiście liniowa i wynosi $O(|E|)$. Następnie ta tablica posłuży nam do symulowania takich wydarzeń jak: usunięcie krawędzi z grafu, sprawdzenie, czy do danego wierzchołka przestały prowadzić jakiekolwiek łuki. Nie usuwając krawędzi z grafu zaraz po przejściu przez nie, a jedynie symulując ich usunięcia narażamy się na sytuację, w których algorytm zacznie nieprzerwanie krążyć pomiędzy węzłami w grafie, które tworzą cykl - sprawdzenie, czy taki występuje następuje dopiero pod sam koniec algorytmu w wierszach 9-12. Możemy sobie jednak z tym problemem poradzić równie łatwo, co z wyznaczaniem ilości węzłów wchodzących do grafu. Jedyne, co musimy zauważyć to fakt, że z każdym powtórzeniem instrukcji 3-8 dodajemy do zbioru S kolejny wierzchołek grafu. W przypadku natrafienia na cykl i nieusuwania krawędzi każde wejście do węzła v_i po ścieżce, należącej do cyklu, spowoduje, że zmniejszymy wartość licznika $deg[i]$ o jeden (symulując tym samym usunięcie krawędzi). Jeżeli okaże się, że po pierwszym przejściu taką ścieżką $deg[i] = 0$ (co odpowiada braku krawędzi wchodzących do v_i) dla każdego węzła, należącego do cyklu to wpadniemy z nieskończoną pętlą, dodając do zbioru S kolejne węzły, a następnie te same węzły usuwając w kroku 4. Jednym z wielu pomysłów na rozwiązanie tego problemu jest wprowadzenie

ograniczenia na ilość elementów listy O - dla poprawnie wykonanego algorytmu będzie ona zawsze dłuższa niż $|V|$, podczas gdy dla omawianego przypadku złego zachowania się algorytmu bardzo szybko liczba tych elementów przekroczy ich oczekiwana ilość.



Rysunek 2.1: **Działanie algorytmu Khana** (a) Sytuacja początkowa algorytmu. Na liście S znajdują się wszystkie węzły, do których - przed rozpoczęciem działania algorytmu - nie wchodziły żadne krawędzie. W kwadratach przy każdym z węzłów znajduje się liczba takich krawędzi $e \in E$, które wchodzą do danego wierzchołka - reprezentują one elementy tablicy $deg[i]$, gdzie i to identyfikator węzła, przy którym znajduje się dany element. (b) Algorytm wybiera z listy S jedyną możliwą element i usuwa z grafu wszystkie krawędzie, wychodzące z wybranego węzła, dodając jednocześnie do listy O każdy węzeł, do którego, w wyniku usunięcia tych krawędzi, nie prowadzi już żaden łuk. Usuwanie połączenia $v_i \xrightarrow{1} v_j$ symbolizujemy zmniejszeniem wartości elementu $deg[j]$. (c) Usunięcie krawędzi wychodzących z następnego, pobranego z listy S , elementu spowodowało dodanie do listy S węzła v_3 . Badany element v_2 - podobnie jak w poprzednim przypadku - po wyciągnięciu z listy S przepinamy do listy O . (d) Dodanie do listy S węzła v_5 przy usuwaniu wszystkich krawędzi $e \in \{e_{ij} : i = 3\}$ i przepięcie badanego elementu v_3 na listę O . (e) Wykonanie kolejnej pętli algorytmu i dodanie skanowanego elementu v_5 do listy wynikowej. (f) Ostatni krok algorytmu. Zauważmy, że w grafie nie ma już żadnych łuków (wszystkie elementy z tablicy deg zostały wyzerowane), więc algorytm zakończył się poprawnie, a badana sortowana sieć nie miała cykli.

Czas działania takiej metody jest oczywiście liniowy, zależny od ilości zarówno węzłów jak i krawędzi w grafie. Zależnie od tego, czy w węzłach przechowujemy informację o liczbie wchodzących do nich krawędzi, te pierwsze będziemy musieli przejrzeć w czasie $O(|V|)$ w poszukiwaniu takich węzłów, które takowych krawędzi nie mają, zaś jeżeli takich informacji nie mamy - wówczas będziemy musieli je sami wygenerować, skanując wszystkie krawędzie w grafie (co zajmie nam $O(|E|)$ czasu). Bez względu na wcześniejszy wykonany krok, właściwa część algorytmu polega na usunięciu wszystkich krawędzi z grafu (gdyż taki chcemy uzyskać rezultat dla prawidłowej sieci) w czasie $O(|E|)$.

2.1.2 Przeszukiwanie w głąb

Alternatywnym sposobem na topologiczne uporządkowanie grafów w sieci jest wykorzystanie właściwości, posiadanych przez prosty algorytm przeszukiwania w głąb, w skrócie DFS (ang. *Depth-First Search*). Aby posortować topologicznie wszystkie węzły w grafie $G = (V, E)$ wykorzystujemy fakt, że wspomniany algorytm oznacza dany węzeł V jako przetworzony dopiero w momencie, gdy wszystkie węzły, do których jest w stanie dojść z badanego węzła, są oznaczone. Innymi słowy nie jest możliwa sytuacja, by w grafie został oznaczony węzeł, którego wszystkie dzieci (a także jego dalsi potomkowie) nie zostały oznaczone. Zapisując kolejność takich operacji (oznaczania węzłów jako przetworzone), a następnie odtwarzając ją w kolejności odwrotnej uzyskujemy listę z poprawnie posortowanymi topologicznie węzłami (odwrotna sytuacja do przedstawionej wcześniej ma następującą interpretację: żaden węzeł v_i nie zostanie zaznaczony, gdy istnieje jakikolwiek węzeł v_j , który jest jeszcze nie zaznaczony, a który posiada krawędź $v_j \xrightarrow{1} v_i$).

Algorithm 7: BFS-TOPOLOGICAL-SORT (G)

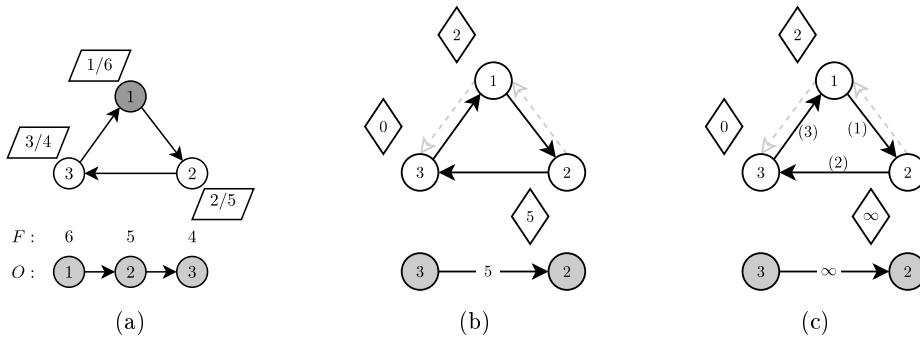
Input: Graf $G = (V, E)$.

Result: Lista O z posortowanymi topologicznie wierzchołkami.

```

1 begin
2   Wykonaj DFS dla grafu wejściowego  $G$ 
3   Wstaw na początek listy  $O$  każdy wierzchołek  $V$ , kiedy ten tylko zostanie oznaczony jako
      przetworzony.
4   return  $O$ 
```

W algorytmie od razu niejawnie dokonujemy odwrócenia elementów, które znajdują się na liście O poprzez wstawianie każdego wierzchołka na początek tej listy, nie na jej koniec. Algorytm oczywiście działa w czasie liniowym, podobnie jak sam DFS ($\Theta(|V| + |E|)$). Wspomnieliśmy na początku rozdziału, poświęconemu sortowaniom topologicznym grafów, że drugi z omawianych algorytmów posiada nad pierwszym tą przewagę, że nie przerywa pracy nawet w momencie napotkania cyklu. Nie jest to do końca prawda, gdyż zachowanie się algorytmu DFS głównie zależy od intencji jego autora, lecz możemy napisać go w taki sposób, aby przeglądając graf wgłęb, po natrafieniu na już odwiedzony wierzchołek kontynuował swoją pracę (to jest albo wycofał się z aktualnie badanego wierzchołka, zaznaczając go jako przetworzony, albo - w przypadku, gdy pozostałe krawędzie badanego węzła prowadzą do jeszcze nieodwiedzonych węzłów - kontynuował przeszukiwanie wgłęb). Innymi słowy - możemy go zmusić by ignorował cykle, występujące w badanym grafie.



Rysunek 2.2: **Przykład złego działania DFS dla grafu z cyklem** (a) Graf po wykonaniu algorytmu DFS. W rombach znajdują się charakterystyczne dla grafu wartości w postaci x/y , gdzie x oznacza czas odwiedzenia danego węzła, zaś y to czas jego przetworzenia (po przetworzeniu wszystkich jego dzieci i ich potomków oraz wycofaniu się z niego). Lista O zawiera „poprawnie” uporządkowane węzły, w kolejności malejącej względem czasu przetwarzania węzłów. Algorytm rozpoczyna pracę od pierwszego węzła w grafie. (b) Poprawnie wykonany algorytm wyszukiwania najkrótszej ścieżki $v_3 \xrightarrow{1} v_2$. $c(3,2) = \delta(3,2) = 5$. (c) Błędne rozwiązanie dla algorytmu opartego o listę O z rysunku pierwszego. Cyfry w nawiasach oznaczają kolejność wykonywania relaksacji, wynikającej z uporządkowania węzłów na liście O .

Niestety - tak wygenerowany porządek nie nadaje się do wykorzystania w algorytmie wyszukiwania najkrótszych ścieżek, który działałby w czasie liniowym. Choć początkowo może się wydawać, że ignorowanie cykli nie szkodzi, a wręcz jest po naszej myśli (algorytm DFS, napotykając ścieżkę zamkającą cykl, nie zdecyduje się na pójście tą ścieżką, podobnie jak relaksacja takiej ścieżki nigdy nie przyniesie żadnego rezultatu - innymi słowy jest zbędna), lecz prosty przykład wystarcza, by algorytm wyszukiwania najkrótszych ścieżek, który opiera się o sortowanie topologiczne, zwrócił nam niepoprawne wyniki.

2.1.3 Sortowanie topologiczne

Jak widzimy z powyższych przykładów, możliwości wykorzystania sortowania topologicznego w algorytmach wyszukiwania najkrótszych ścieżek są ograniczone jedynie do małej klasy grafów - stanowczo za małej, jeżeli chodzi o grafy, reprezentujące rzeczywiste sieci drogowe. Istnieją jednak pewne problemy (mniej trywialne od pieczenia ciasta, czy kolejności nakładania ubrań), w których taki algorytm się przydaje i jest chętnie stosowany głównie ze względu na swoją szybkość działania - liniową, proporcjonalnie do ilości krawędzi w grafie. Implementacja algorytmu sprowadza się do wykonania relaksacji dla każdej krawędzi, wychodzących z węzłów posortowanych topologicznie, w której to kolejności powinniśmy postępować.

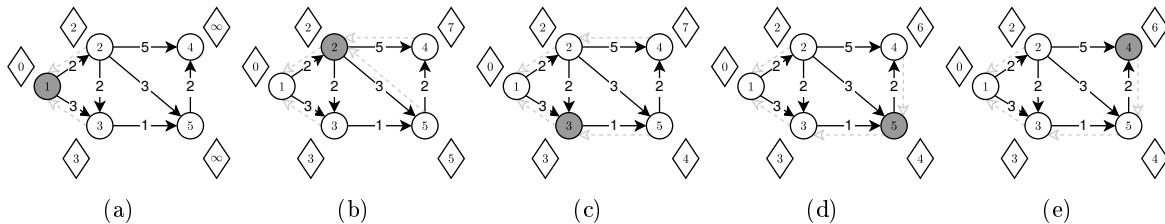
Algorithm 8: TOPOLOGICAL-SHORTEST-PATH (G)

```

1 begin
2   foreach  $v_i$  w porządku topologicznym do
3     foreach  $e_{ij} : v_i \xrightarrow{1} v_j$  do
4       RELAX ( $v_i, v_j$ )
    
```

Aby udowodnić poprawność działania takiego algorytmu, odwołamy się do dwóch, wcześniej przedstawionych (w rozdziale 1.3.3), lematów: mówiącym o optymalnej podstrukturze grafu (1.3.2) oraz o własności zbieżności dla najkrótszych ścieżek (1.3.6).

Dowód. Założymy indukcyjnie, że algorytm przeskanował już wierzchołki $v_i : i \in \{1, 2, \dots, k\}$ i dla każdego z nich ich waga jest optymalna ($d(i) = \delta(s, v_i)$). W oczywisty sposób pierwszy krok indukcyjny jest spełniony: dla $k = 1$ naszym jedynym wierzchołkiem, który został obsłużony jest wierzchołek początkowy - źródło, którego $d(s) = \delta(s, s) = 0$. Przyjrzyjmy się teraz sytuacji, w której algorytm bada węzeł $k + 1$ 'y (a nie konkretnie v_{k+1}). Niech najkrótszą ścieżką do tego węzła będzie $P = \langle v_1, v_2, \dots, v_h, k + 1 \rangle$. Z lematu 1.3.2 (o optymalnej podstrukturze) wiemy, że każda podścieżka ścieżki P jest najkrótszą ścieżką, w szczególności jest nią ścieżka $P' = \langle v_1, v_2, \dots, v_h \rangle$. Z faktu, że wszystkie wierzchołki w grafie są posortowane topologicznie oraz, że krawędź $v_h \xrightarrow{1} k + 1 \in E$ (co nam gwarantuje istnienie ścieżki P) wynika, że wierzchołek v_h jest węzłem, dla którego $h \in \{1, 2, \dots, k\}$ w związku z czym, na mocy założenia indukcyjnego, wartość $v_h.d = \delta(s, v_h)$. Zgodnie z lematem 1.3.6, jeżeli w dowolnym momencie przed relaksacją krawędzi $v_h \xrightarrow{1} k + 1$ wartość $v_h.d = \delta(s, v_h)$ to po relaksacji tej krawędzi już zawsze $(k + 1).d = \delta(s, v_h)$ (jest optymalna). Z faktu istnienia takiej krawędzi oraz z uporządkowania topologicznego wszystkich węzłów w grafie wiemy, że waga, trzymana przez wierzchołek v_h zawsze będzie optymalna przed przystąpieniem do przechodzenia po krawędzi $v_h \xrightarrow{1} k + 1$, co kończy dowód. ♦



Rysunek 2.3: Przykład dla ustalonego porządku topologicznego węzłów: v_1, v_2, v_3, v_5, v_4 .

2.2 Generyczny algorytm Dijkstry

Pokazaliśmy w poprzednich rozdziałach, że kolejność przetwarzania węzłów może mieć ogromny wpływ na szybkość działania algorytmu - od przeglądania wierzchołków w kolejności narzuconej nam przez ich uporządkowanie w strukturach grafu (w czasie $O(|V| \cdot |E|)$), aż do badania wierzchołków w porządku topologicznym ($O(|V| + |E|)$). Oba te algorytmy miały zasadnicze wady: albo działały w czasie dużo poniżej naszych oczekiwani, wykonując zatrważającą ilość niepotrzebnych operacji, albo nie nadawały się do użytku w sieciach, które my chcemy badać (z nieujemnymi cyklami). W tym rozdziale przedstawimy kolejny sposób przeglądania wierzchołków grafu $G = (V, E)$, pokażemy generyczny algorytm na nim oparty, a także udowodnimy jego poprawność. Bazując na kontrprzykładzie, wykażemy także, że nie działa on dla grafów, które zawierają krawędzie o ujemnych wagach (a co za tym idzie dla grafów z ujemnymi cyklami). Algorytm, opracowany przez holenderskiego informatyka Edsgera Dijkstrę, okaże się podstawą do powstania szeregu jego modyfikacji, którym niejednokrotnie zawdzięcza asymptotycznie szybsze czasy działania, a które omówimy w tym rozdziale, skupiając się na ich własnościach.

2.2.1 Algorytm Dijkstry

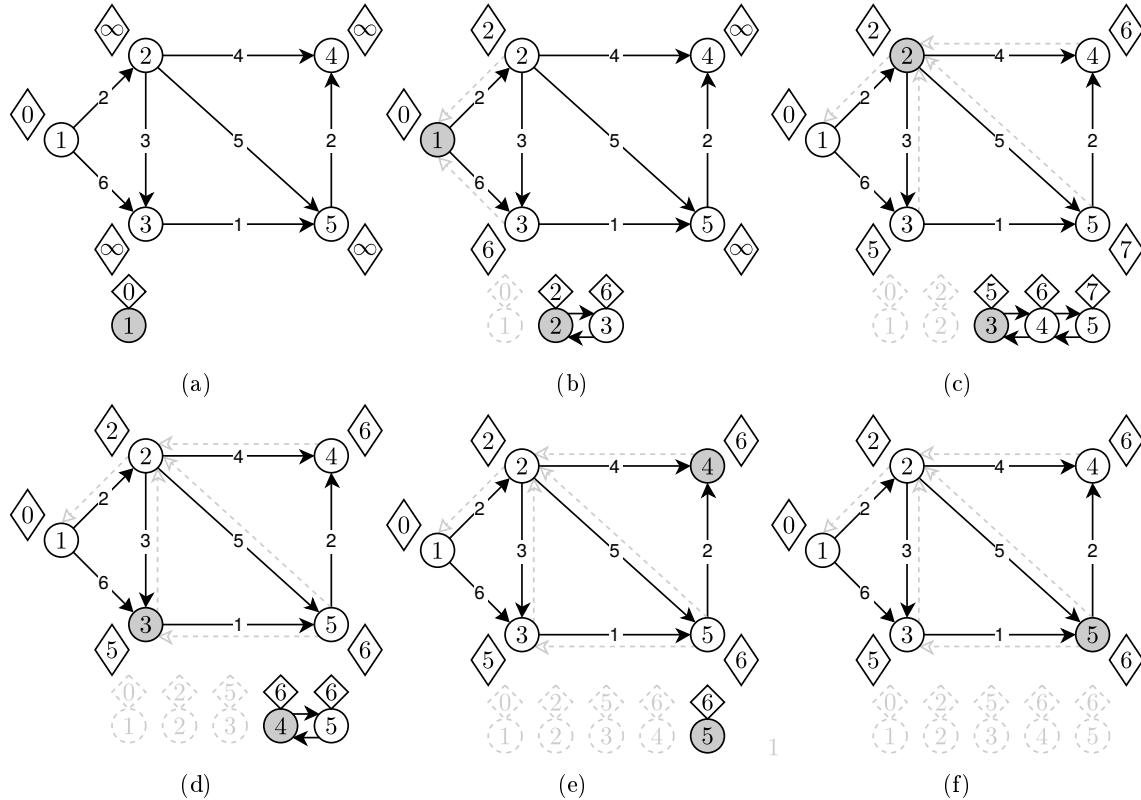
Sama idea algorytmu jest bardzo podobna do poprzedniej tj. zakłada wykonywanie relaksacji dla wszystkich krawędzi aktualnie badanych wierzchołków, których to kolejność jest w specyficzny sposób ustalana. W przypadku algorytmu opartego na sortowaniu topologicznym grafu był to właśnie ten porządek, który zapewniało nam sortowanie. W Przypadku algorytmu Dijkstry mamy regułę, która mówi, że wierzchołki są skanowane w niemalejącej kolejności ich etykiet (atrybutów d - odległości węzła od źródła s), co wiąże się z wykorzystaniem w naszym algorytmie **kolejek priorytetowych**, które zapewniają nam właśnie taki porządek. Jak się później okaże - omawiane modyfikacje algorytmu Dijkstry różnią się głównie jej implementacją.

Aby wprowadzić nieco formalizmu przyjmiemy, że mamy dwa zbiory rozłączne: S , który na początku działania algorytmu jest pusty - do niego trafiać będą przetworzone już wierzchołki - oraz \bar{S} , w którym na początku przechowywane są wszystkie węzły. Jak już wspomnieliśmy, algorytm ma za zadanie sekwencyjne pobierać ze zbioru \bar{S} takie wierzchołki v_i , by $v_i.d = \min \{v_j.d : v_j \in \bar{S}\}$, przenosić je do zbioru wierzchołków S oraz wykonywać relaksacje dla każdej krawędzi, która wychodzi z tego wierzchołka. Oczywiście - tak jak w każdym poprzednim algorytmie - na początku inicjalizujemy graf metodą INIT-GRAF (G, s). Pseudokod generycznego algorytmu Dijkstry wygląda tak, jak przedstawiono poniżej.

Algorithm 9: GENERIC-DIJKSTRA (G, s)

```
1 begin
2    $S \leftarrow \emptyset$ 
3    $\bar{S} \leftarrow \{v : v \in V\}$ 
4   while  $\bar{S}$  nie jest pusty do
5      $v \leftarrow v_i : v_i.d = \min \{v_j.d : v_j \in \bar{S}\}$ 
6      $S \leftarrow S \cup \{v\}$ 
7      $\bar{S} \leftarrow \bar{S} - \{v\}$ 
8     foreach  $e_{ij} : v_i \xrightarrow[1]{} v_j$  do
9       RELAX( $v_i, v_j$ )
```

gdzie w prawdziwej implementacji linijki 5 – 7 zwykle zastępuje się operacją EXTRACT-MIN (Q), gdzie Q to nasza kolejka priorytetowa. Zbioru S zaś w ogóle się nie uwzględnia, gdyż służy on tylko do celów przeprowadzenia dowodów poprawności tego algorytmu i wykazania, że jest on poprawnie skonstruowany (niezmiennikiem pętli 4 – 9 w tym przypadku będzie $Q = V - S$). Odnajdywaniem najmniejszego elementu (w sensie dystansu wierzchołka do źródła) i usuwaniem go ze zbioru \bar{S} zajmuje się, wymieniona wyżej, operacja (wtedy $\bar{S} = Q$). Na przykładzie 2.4 jako kolejkę priorytetową wykorzystano podwójnie wiązaną listę (która oczywiście nie jest najfortunniejszym wyborem), której czas, potrzebny na wyciągnięcie z niej najmniejszego elementu, jest zależny od ilości tych elementów na liście (w najgorszym przypadku $|V|$).



Rysunek 2.4: **Działanie algorytmu Dijkstry** (a) Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAF ze źródłem $v_s.id = 1$. (b) Z listy dwukierunkowej została wyciągnięty najmniejszy element i została wykonana operacja RELAX dla krawędzi: e_{12} i e_{13} . Odpowiednio węzły v_2 i v_3 zostały wstawione do kolejki. (c) Najmniejszym elementem na liście był węzeł v_2 . Został usunięty z kolejki, algorytm wykonał relaksację krawędzi e_{23} , e_{24} i e_{25} . Etykieta węzła v_3 uległa zmniejszeniu. (d-f) Kroki analogiczne jak poprzednie.

2.2.2 Złożoność obliczeniowa

Chcąc analizować złożoność czasową algorytmu widzimy, że jego główna pętla 4 – 9 wykonuje się dokładnie $|V|$ razy, za każdym razem usuwając z kolejki priorytetowej dokładnie jeden węzeł, gdzie skanowane węzły nie powtarzają się¹. Następnie, wewnętrz pętli, wyszukiwany jest najmniejszy element w kolejce - czas tej operacji jest naturalnie zależny od wybranego przez nas sposobu jej implementacji i na pożytek naszych rozważań niech zajmuje czas $O(\text{EXTRACT-MIN}(Q))$. Takich operacji podczas działania algorytmu wykonamy $|V|$. Dla każdego węzła, wyjętego z kolejki, dla wszystkich luków, wychodzących z danych węzłów, wykonywana jest operacja relaksacji - łatwo zauważać, że podczas całej procedury metoda RELAX zostanie wywołana dokładnie $|E|$ razy, podczas której może być wymagane zmniejszenie atrybutu d któregoś z węzłów - koszt takiej operacji jest znowu zależny od implementacji kolejki priorytetowej (wraz ze zmniejszeniem się klucza może zajść konieczność przemieszczenia węzła bliżej głowy kolejki) i dla naszej analizy niech wyniesie $O(\text{DECREASE-KEY}(Q, v, k))$, gdzie k to nowy klucz (nowa odległość od źródła $s - v.d$) węzła v . Pozostaje nam jeszcze metoda INSERT(Q, v), która wstawia nam elementy do kolejki. Czas jej działania jest również zależny od wybranej implementacji kolejki Q , zaś miejsce jej wywołania zależne jest już od woli programisty; może on przed rozpoczęciem algorytmu wstawić wszystkie wierzchołki grafu do kolejki (wiersz 3) bądź też wstawiać je na bieżąco w chwili, gdy algorytm potrafi już do nich dojść (wtedy podczas relaksacji podejmowana jest decyzja, czy wstawić nowy element do kolejki, czy taki już w kolejce istnieje i należy tylko zmniejszyć jego klucz i zadbać o zachowanie prawidłowego porządku w strukturze danych). Bez względu na

¹Ilość wykonywanych pętli możemy ograniczyć, kończąc algorytm, gdy z kolejki zostanie wyjęty taki węzeł v , że $v.d = \infty$. Jak wiemy, relaksacja żadnej z krawędzi, wychodzących z takiego węzła nie zmieni nam sytuacji w grafie, a z własności kolejki priorytetowej wiemy, że pozostałe elementy u , które w niej pozostały, spełniają $u.d \geq v.d = \delta(s, v) = \infty$.

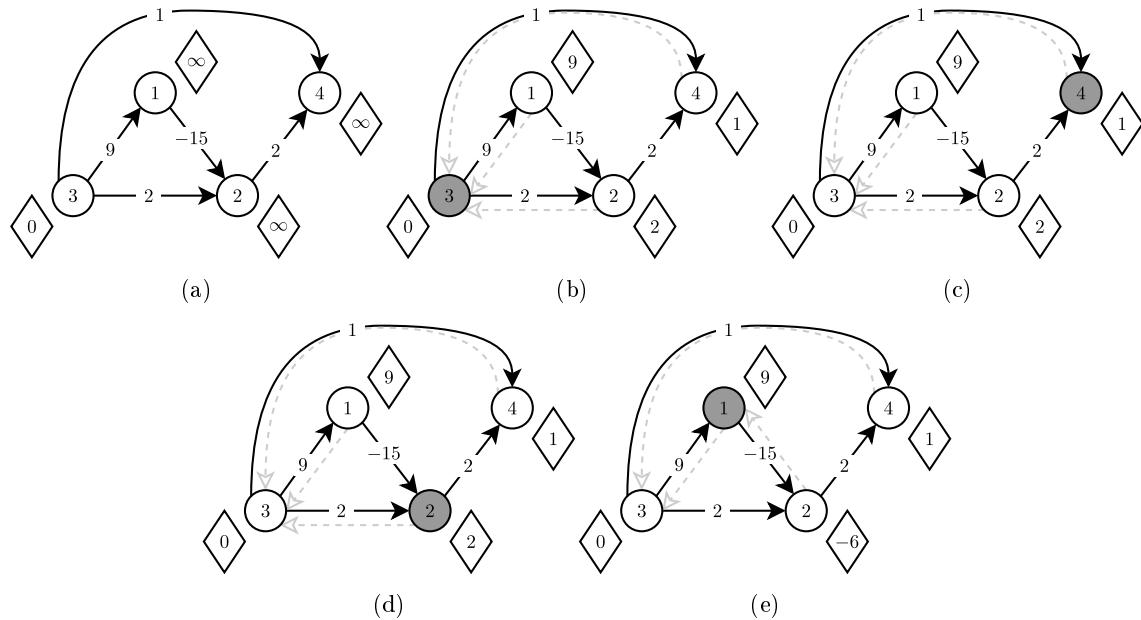
preferowane rozwiązanie ilość takich operacji wyniesie dokładnie $|V|$, jako że każdy wierzchołek wstawimy do kolejki (i wyjmujemy go z niej) tylko raz. Reasumując - złożoność algorytmu Dijkstry w uogólnionym przypadku jest ograniczona z góry przez:

$$O(|E| \cdot O(\text{DECREASE-KEY}(Q, v, k)) + |V| \cdot [O(\text{INSERT}(Q, v)) + O(\text{EXTRACT-MIN}(Q))]) \quad (2.1)$$

Wąskim gardłem algorytmu nazywamy taki jego element składowy, który przesądza o złożoności obliczeniowej całego algorytmu, niejednokrotnie go zwalniając. W tym przypadku nie ma wątpliwości, że takim elementem w algorytmie Dijkstry jest zastosowana struktura, odpowiedzialna za wykonywanie tych trzech operacji.

2.2.3 Ujemne koszty krawędzi

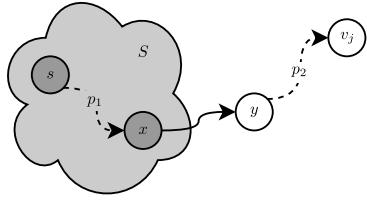
Nim udowodnimy poprawność algorytmu Dijkstry przeanalizujmy jeszcze prosty przykład, w którym dopuścimy wystąpienie krawędzi o ujemnym koszcie i pokażmy, że dla takiego grafu nasz algorytm zwróci błędny wynik.



Rysunek 2.5: Działanie algorytmu Dijkstry w grafie z ujemnymi kosztami krawędzi

Jak powiedzieliśmy, algorytm Dijkstry analizuje wierzchołki grafu $G = (V, E)$ w ścisłe określonym porządku tj. bada zawsze taki węzeł v , którego $v.d$ jest najmniejsze spośród wszystkich pozostałych, jeszcze nie przeanalizowanych węzłów. Na kolejnych rysunkach zaznaczono kolejność przeglądania węzłów jaka wynika z tej własności (za węzeł startowy przyjmując v_3). Widzimy, że algorytm zwrócił błędną ścieżkę dla pary węzłów: v_3 i v_4 (poprawną, najkrótszą ścieżką $v_3 \xrightarrow{*} v_4$ jest ścieżka $P = \langle v_3, v_1, v_2, v_4 \rangle$) ze względu na wystąpienie w grafie krawędzi o ujemnym koszcie.

2.2.4 Poporawność działania



Rysunek 2.6: **Dowód poprawności algorytmu Dijkstry**

Tuż przed wstawieniem wierzchołka v_j do S ten ostatni nie jest pusty. Przerywanymi strzałkami zaznaczono ścieżki p_1 oraz p_2 , które mogą mieć dowolnie dużą ilość składowych (w skrajnym przypadku $s = x$ i/lub $y = v_j$). Dodatkowo $x \neq y$.

Dowód. Założymy indukcyjnie, że dla każdego węzła v_i w momencie dodawania go do zbioru wierzchołków przetworzonych S zachodzi $v_i.d = \delta(s, v_i)$. Pierwszy krok indukcyjny jest oczywisty, gdyż na samym początku zbior S jest pusty i założenie jest prawdziwe. Pierwszym wierzchołkiem, który jest dodawany do tego zbioru jest wierzchołek s , będący źródłem, którego w oczywisty sposób $s.d = 0 = \delta(s, s)$ (w grafie dla algorytmu Dijkstry założyliśmy brak krawędzi o ujemnych długościach). Przyjmijmy nie w prost, że istnieje w grafie taki wierzchołek v_j , dla którego $v_j.d \neq \delta(s, v_j)$ w trakcie jego dodawania do zbioru S i będzie on pierwszym takim wierzchołkiem, jaki będziemy chcieli do tego zbioru dodać. Wiemy, że $v_j \neq s$ oraz, że do takiego wierzchołka na pewno istnieje najkrótsza ścieżka ze źródła s (gdyby tak nie było to odpowiednio wtedy $v_j.d = s.d = 0 = \delta(s, s) = \delta(s, v_j)$ lub $v_j.d = \delta(s, v_j) = \infty$ - z własności braku ścieżki). Bezpośrednio z poprzednich spostrzeżeń wynika, że w momencie dodawania wierzchołka v_j do zbioru S ten jest niepusty i zawiera co najmniej jeden element - źródło. Niech istniejąca ścieżka z s do v_j nazywa się P oraz rozważmy sytuację taką, jaką widać na rysunku 2.6. Rozbiliśmy na nim ścieżkę P na dwie składowe: p_1 i p_2 , gdzie $v_s \xrightarrow{p_1} x \xrightarrow{1} y \xrightarrow{p_2} v_j$ oraz pierwsza z nich składa się tylko z węzłów należących do zbioru S , zaś druga - tylko z węzłów poz tym zbiorem. Dodatkowo węzeł y jest pierwszym na ścieżce P , który jest poza tym zbiorem. Pokażemy teraz, że w momencie dodawania wierzchołka v_j ($v_j.d \neq \delta(s, v_j)$) do zbioru S zachodzi $y.d = \delta(s, y)$. Aby udowodnić ten fakt, wystarczy zauważyć, że skoro wierzchołek v_j był pierwszym takim, dla którego zachodzi $v_j.d \neq \delta(s, v_j)$, to wstawiając do zbioru S wierzchołek x na pewno $v_x.d = \delta(s, x)$, a ze zbieżności (lemat 1.3.6) mamy, że zachodzi również $y.d = \delta(s, y)$ (w trakcie dodawania wierzchołka x do zbioru S zajdzie relaksacja krawędzi $x \xrightarrow{1} y$, gdzie wcześniej $v_x.d = \delta(s, x)$).

Ponieważ na naszym rysunku wierzchołek y występuje na ścieżce P wcześniej od wierzchołka v_j i każda krawędź w grafie ma koszt nieujemny to $\delta(s, y) \leq \delta(s, v_j)$, co prowadzi do szeregu nierówności:

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, v_j) \\ &\leq v_j.d \quad (\text{z lematu 1.3.5 o górnym ograniczeniu}) \end{aligned} \tag{2.2}$$

Wiemy jednak, że z własności algorytmu Dijkstry zawsze wybieramy wierzchołek spoza zbioru S o jak najmniejszej wartości atrybutu d , a skoro oba wierzchołki (y i v_j) nie należą do zbioru S w chwili wyboru wierzchołka v_j mamy zagwarantowane, że $v_j.d = \min\{v.d : v \notin S\}$, w szczególności $v_j.d \leq y.d$. Dodając to ostatnie równanie do szeregu poprzednich nierówności otrzymujemy:

$$y.d = \delta(s, y) = \delta(s, v_j) = v_j.d \tag{2.3}$$

Widzimy, że $v_j.d = \delta(s, v_j)$, co jest sprzeczne z naszym założeniem (dodanie do zbioru S pierwszego wierzchołka v_j o własności $v_j.d = \delta(s, v_j)$). Rozumowanie jest identyczne w przypadku, gdyby na ścieżkach p_1 i/lub p_2 znajdowała się dowolna liczba węzłów, spełniających nasze założenia. Pokazaliśmy zatem, że dla każdego wierzchołka $v \in V$ w momencie jego dodawania do zbioru S zachodzi $v.d = \delta(s, v)$. Algorytm kończy działanie, gdy w kolejce Q nie ma już żadnych wierzchołków (wszystkie zostały dodane do zbioru S), tak więc w momencie, gdy każdy wierzchołek spełnia $v.d = \delta(s, v)$, co kończy dowód. ♦

2.3 Podstawowe struktury danych

Jak pokazaliśmy wcześniej - efektywność algorytmu Dijkstry w głównej mierze zależy od efektywności implementacji struktury, od której będziemy wymagać wykonywania trzech, podstawowych operacji: $\text{INSERT}(Q, v)$, $\text{EXTRACT-MIN}(Q)$ i $\text{DECREASE-KEY}(Q, v, k)$. Wyspecjalizowanymi strukturami do ich wykonywania są kolejki priorytetowe, choć - jak mogliśmy się już przekonać - inne struktury, takie jak tablice, listy jednokierunkowe czy podwójnie wiązane (ang. *double-linked lists*), także umożliwiają nam poprawną konstrukcję algorytmu Dijkstry. Korzystając z nich musimy jednak płacić cenę za ich wysoką nieefektywność i tak dla listy dwukierunkowej (wykorzystanej przy omawianiu algorytmu) wyszukanie najmniejszego elementu kosztuje nas proporcjonalnie do ilości wierzchołków, znajdujących się na niej w trakcie wyszukiwania. Jeżeli spojrzymy na złożoność algorytmu Dijkstry (2.2.2) natychmiastowo uzyskamy górne ograniczenie na poziomie $O(|V|^2)$, co niewiele oddala nas złożoności tak prostego algorytmu, jakim jest algorytm Bellmana-Fora.

Mówiąc o różnych wcieleniach algorytmu Dijkstry nie sposób jest więc poruszyć tematu podstawowych struktur danych, jakie możemy wykorzystać do implementacji różnych kolejek priorytetowych, ich właściwościach i czasach działania podstawowych operacji, których wykonywanie dane struktury umożliwiają - w szczególności $\text{INSERT}(Q, v)$, $\text{EXTRACT-MIN}(Q)$ i $\text{DECREASE-KEY}(Q, v, k)$. W niniejszym rozdziale omówimy takie struktury jak: kopce binarne (w uogólnionym spojrzeniu na kopce r -arne), kopce Fibonacciego, kolejki z przepelnieniem oraz szereg innych pomysłów, opartych o kontenery, zwane dalej kubelkami (ang. *buckets*).

2.4 Struktury oparte na kopcach

Jedną ze struktur, przystosowanych do operacji charakterystycznych dla kolejki priorytetowej jest kopiec (ang. *heap*). Jego najogólniejszą własnością jest to, że operacja, zwracająca najmniejszy (lub największy) element, który znajduje się w kopcu, działa w czasie stałym i polega na odwołaniu się do szczytu takiego kopca. Kopce to szczególne przypadki drzew, gdzie pomiędzy rodzicem, a potomkami zwykle jest ustalona stała relacja (w przypadku, który nas interesuje - kopiec typu *min* - klucze, przechowywane przez potomków węzła v powinny być zawsze większe od klucza rodzica: $v.d \leq v_i : v_i.\Pi = v$). Przedstawimy dwa, powszechnie znane rodzaje kopców: zwykły kopiec, do którego implementacji wykorzystamy tablicę, oraz drugi - Fibonacciego, który - jak się okaże - pomimo swojej teoretycznej przewagi w szybkości wykonywania poszczególnych operacji, w praktyce często działa wolniej od - dużo prostszego w implementacji - wspomnianej wcześniej wersji.

2.4.1 Kopiec R-arny

Kopien R -arny jest uogólnieniem kopca binarnego - podczas gdy dla tego drugiego każdy rodzic może posiadać do 2 potomków, w pierwszym przypadku takich węzłów rodzic może mieć od 0 do R , co zauważalnie zmniejsza wysokość takiego kopca kosztem jego szerokości. Poniższa tabela przedstawia koszty poszczególnych operacji dla kopca binarnego i R -arnego:

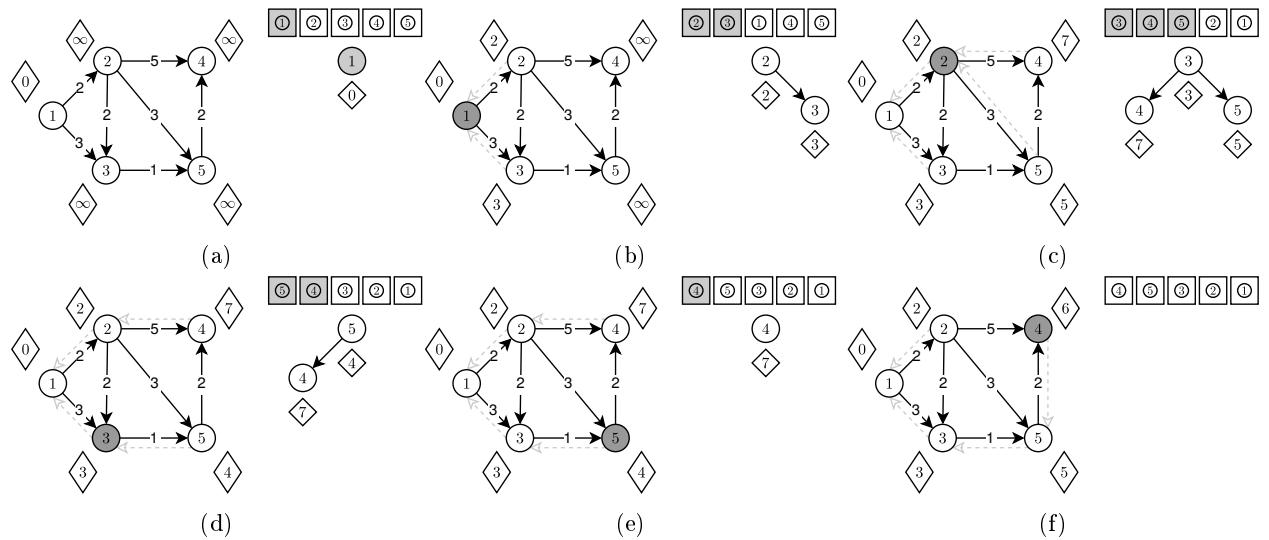
Operacja	Kopiec	
	binarny	R -arny
$\text{INSERT}(Q, v)$	$O(\log(n))$	$O(\log_R(n))$
$\text{EXTRACT-MIN}(Q)$	$O(\log(n))$	$O(R \cdot \log_R(n))$
$\text{DECREASE-KEY}(Q, v, k)$	$O(\log(n))$	$O(\log_R(n))$

Implementacja

Algorytm wyszukiwania najkrótszych ścieżek oparty na strukturze D -arnego kopca jest jedynym algorymem, który nie wymaga od nas tworzenia dodatkowych struktur. Jak dobrze wiemy, jedną z właściwości kopców jest ich zdolność do pracy w miejscu tj. nie wykorzystywania dodatkowej pamięci podczas działania. Pod pojęciem naszego grafu $G = (V, E)$ kryje się tablica $tab[1 \cdots |V|]$, przechowująca wierzchołki, indeksowane ich identyfikatorami ($tab[i] = v_i$) oraz listy sąsiedztwa, przyporządkowane do każdego z takich węzłów.

Aby skorzystać z właściwości kopca, będziemy chcieli zbudować jego strukturę bezpośrednio na wspomnianej tablicy węzłów tj. kopiec o k elementach będziemy chcieli przedstawić jako tablica $\text{vertices}[1 \dots k]$. W takiej sytuacji, jeżeli dowolny wierzchołek v znajduje się na pozycji $i \leq k$ w tablicy tab to znaczy, że w którymś momencie został on wstawiony do naszej kolejki priorytetowej i nie opuści jej, dopóki nie stanie się on najmniejszy spośród tych k elementów. Aby jednak nie stracić informacji o pierwotnym rozmieszczeniu wierzchołków w tablicy tab , będziemy chcieli wprowadzić pomocniczą tablicę $\text{heapIDArray}[1 \dots |V|]$, której to wartości będą odzwierciedlać faktyczne rozmieszczenie wierzchołków w tab po modyfikacjach ($\text{tab}[\text{heapIDArray}[i]] = v_i$), jakich dopuści się na niej nasza kolejka priorytetowa (indeksami tablicy tab pierwotnie były identyfikatory wierzchołków, a ich nie możemy zmieniać).

Zasada działania takiego kopca nie różni się niczym od zastosowania takiej samej struktury do posortowania n liczb, gdzie rozmiar kopca monotonicznie rośnie w trakcie jego budowania, a następnie maleje w czasie działania takiego algorytmu. W naszym przypadku liczba jego elementów może się zwiększać jak i zmniejszać w dowolnej kolejności - jeżeli się zwiększa to ostatni element w części tablicy należącej do powiększonego kopca zamieniamy z elementem, który chcemy faktycznie do niego wstawić, a następnie „wywycham” go ku górze (analogiczna procedura jest wykonywana podczas powiększania kopca w czasie jego budowania dla algorytmu sortowania), zaś jeżeli rozmiar kopca maleje to zachowanie, w porównaniu z algorytmem sortującym, jest identyczne.



Rysunek 2.7: **Działanie algorytmu Dijkstry w oparciu o kopiec R-arny** **(a)** Sytuacja po zainicjowaniu grafu G przez INIT-GRAH ze źródłem v_1 . W tablicy na szaro zaznaczone są elementy należące do kopca, przedstawione poniżej. Niech k oznacza rozmiar kopca, tablica tab jest indeksowana od 1. **(b)** Z kopca zostaje usunięty węzeł v_1 ($k = 0$). W wyniku relaksacji na kopiec zostaje przeniesiony węzeł v_2 ($k = 1$ i $\text{tab}[k] = v_2$), zaś na stare miejsce wstawionego węzła zostaje przeniesiony węzeł v_1 . Analogicznie na koniec kopca wstawiany jest v_3 ($k = 2$, $\text{tab}[k] = v_3$) w wyniku czego $\text{tab}[3] = v_1$. **(c)** Z kopca zostaje pobrany węzeł v_2 , a na szczyt stosu zostaje przeniesiony ostatni element w kopcu (v_3). W wyniku relaksacji krawędzi, wychodzących w pobranego węzła, do kopca zostają dodane węzły: v_4 i v_5 . Tablica tab zmienia się odpowiednio: $\{3\} [2] [1] [4] [5] \rightarrow \{3\} \{4\} [1] [2] [5] \rightarrow \{3\} \{4\} \{5\} [2] [1]$, gdzie w klamrach „{}” zostały zaznaczone węzły, znajdujące się w kopcu. Żadna operacja nie narusza własności kopca. **(d)** Wybrano kolejny węzeł ze szczytu stosu: v_3 , zamieniając go z ostatnim elementem w kopcu i zmniejszając jego rozmiar ($k = 2$). Zachowana jest własność kopca. W wyniku relaksacji zostają zmienione atrybuty: $v_5.\Pi = v_3$ i $v_5.d = 4$. **(e-f)** Z kopca zostaje zabrany jego najmniejszy element: v_5 i wykonywana jest operacja RELAX dla krawędzi z niego wychodzących. Na szczyt kopca zostaje wstawiony jego ostatni element ($k = 1$). Własność kopca jest zachowana.

Złożoność obliczeniowa

Uzupełniając wzór 2.2.2 na ogólną złożoność generycznego algorytmu Dijkstry z wykorzystaniem kolejek priorytetowych dla kopców R -arnych natychmiast otrzymujemy $O(m \cdot \log_d(n) + n \cdot [\log_d(n) + d \cdot \log_d(n)]) = O(m \cdot \log_d(n) + n \cdot d \cdot \log_d(n))$ (dla przejrzystości zapisu przyjęliśmy $n = |V|$ i $m = |E|$). Dla kopca binarnego mamy: $O(m \cdot \log(n) + n \cdot \log(n)) = O(m \cdot \log(n))$ dla $m \geq n$ (stałe wyeliminowaliśmy). Przypomnijmy sobie, że naiwna implementacja algorytmu Dijkstry miała złożoność $O(m + n^2) = O(n^2)$. Łatwo zauważać, że dla bardzo gęstych grafów (gdzie $m = \Omega(n^2)$) nasz nowy algorytm asymptotycznie staje się wolniejszy nawet od wspomnianej, naiwnej implementacji, jednak sytuacja zmienia się na korzyść kopców, gdy ilość krawędzi w grafie jest z góry ograniczona przez $O\left(\frac{n^2}{\log(n)}\right)$ (wtedy $O(m \cdot \log(n)) \leq O\left(\frac{n^2}{\log(n)} \cdot \log(n)\right) = O(n^2)$).

Z kolei dla kopków, których arność jest większa ($d \geq 2$) mamy: $O(m \cdot \log_d(n) + n \cdot d \cdot \log_d(n))$. Z tego bezpośrednio wynika, że optymalną wartością parametru d jest $\max\{2, \lceil \frac{m}{n} \rceil\}$, dla którego zrównują nam się obie strony sumy, otrzymanej wcześniej, złożoności ($n \cdot \frac{m}{n} \cdot \log_d(n) = m \cdot \log_d(n)$). Otrzymaliśmy złożoność algorytmu Dijkstry, opartego o kopce R -arne, który znów w zależności od sieci, dla której go zastosujemy, będzie porównywalny albo do podstawowej, naiwnej implementacji tego samego algorytmu (dla sieci gęstych, gdzie $m = \Omega(n^2)$ mamy: $O(m \cdot \log_d(n)) = O\left(n^2 \cdot \log_{\frac{n^2}{n}}(n)\right) = O(n^2 \cdot \log_n(n)) = O(n^2)$), albo do opartego na kopcu binarnym (w przypadku, gdy sieć jest bardzo rzadka). Dla tej drugiej możliwości otrzymujemy natychmiastowo złożoność $O(n \cdot \log(n))$ dla $m = n$.

Co więcej, jeżeli założymy $m = \Omega(n^{1+\epsilon})$ dla $\epsilon > 0$ i $d = \lceil \frac{m}{n} \rceil > 2$ to będziemy mogli wyprowadzić następujący ciąg równości:

$$\begin{aligned} O(m \cdot \log_d(n)) &= O\left(m \cdot \frac{\log(n)}{\log(d)}\right) \quad (\text{zamiana podstawy logarytmu}) \\ &= O\left(m \cdot \frac{\log(n)}{\log(n^\epsilon)}\right) \quad \left(d = \lceil \frac{m}{n} \rceil = \frac{n^{1+\epsilon}}{n}\right) \\ &= O\left(m \cdot \frac{\log(n)}{\epsilon \cdot \log(n)}\right) \quad (\log_a(b^c) = c \cdot \log_a(b)) \\ &= O\left(\frac{m}{\epsilon}\right) \\ &= O(m) \quad \left(\frac{1}{\epsilon} \text{ jest stałą.}\right) \end{aligned} \tag{2.4}$$

Jeśli $\epsilon = 1$ to $m = \Omega(n^2)$, a ten wariant analizowaliśmy już wcześniej. Widzimy więc, że w zależności od gęstości grafu te same algorytmy mogą zachowywać się zupełnie inaczej, a co za tym idzie - nie jesteśmy w stanie wskazać jednej implementacji algorytmu wyszukiwania najkrótszych ścieżek, która działałaby równie szybko (w porównaniu do reszty algorytmów) dla każdej z możliwych sieci.

Drzewa

Strukturami bardzo podobnymi do kopców są drzewa K -arne - należy wręcz powiedzieć, że kopce są pełnymi drzewami (ang. *complete tree*), podczas gdy struktura zwykłego drzewa jest mniej rygorystyczna. Pełnym drzewem R -arnym (jakim jest kopiec tej samej arności) nazywamy takie drzewo, na poziomach którego, poza ostatnim, wszystkie węzły mają dokładnie R potomków. W przypadku drzew K -arnych każdy węzeł może mieć co najwyżej K węzłów potomnych, co nie musi wcale oznaczać, że stworzone tak drzewo, jest drzewem pełnym. Obie struktury da się zaimplementować przy wykorzystaniu zwykłych tablic choć, w przypadku drzew, pomiędzy kolejnymi elementami takie tablice mogą pojawić się miejsca puste, gdy któryś z węzłów drzewa ma mniej niż K potomków. Inne są także założenia samych struktur: każdy węzeł w drzewie K -arnym posiada tyleż potomków w ścisłe zdefiniowanym porządku (niemalejącym lub nierosnącym), zaś reguły, odnoszące się do kopców nic o takim porządku nie mówią - jedyna własność, która musi zostać spełniona dla węzła to przewyższanie jego priorytetem wszystkich swoich potomków (bądź posiadanie najmniejszego priorytetu pośród nich w przypadku kopca typu *min*). Bezpośrednią konsekwencją tych własności są różne zastosowania wymienionych struktur danych:

Operacja	Drzewo		Kopcic	
	binarne ²	K -arne	binarny	R -arny
INSERT (Q, v)	$O(\log(n)) / O(1)$	$O(K \cdot \log_K(n))$	$O(\log(n))$	$O(\log_R(n))$
EXTRACT-MIN (Q)	$O(\log(n)) / O(n)$	$O(\log_K(n))$	$O(\log(n))$	$O(R \cdot \log_R(n))$
DECREASE-KEY (Q, v, k)	$O(\log(n)) / O(1)$	$O(K \cdot \log_K(n))$	$O(\log(n))$	$O(\log_R(n))$
SEARCH (Q, k)	$O(\log(n)) / O(n)$	$O(K \cdot \log_K(n))$	$O(n)$	$O(n)$

strukturę drzew stosuje się dla problemów, gdzie nacisk jest kładziony na wyszukiwanie elementów po ich właściwościach, zaś wybieranie minimum jest sprawą drugorzędną. Odwrotna sytuacja występuje w przypadku kopców, które w żaden sposób nie wspierają operacji wyszukiwania, sprowadzając ją do przeszukania całej tablicy reprezentującej kopiec. Innymi słowy: drzewa K -arne nie są przystosowane do pełnienia roli kolejki priorytetowej. Przywołując wzór na ogólną złożoność algorytmu Dijkstry (2.2.2):

$$O(m \cdot O(DK(Q, v, k)) + n \cdot [O(I(Q, v)) + O(EM(Q))]) \quad (2.5)$$

i porównując czasy wykonywanych operacji dojdziemy do następujących złożoności:

- $O((n+m) \cdot \log(n))$ dla zbalansowanych drzew przeszukiwań binarnych,
- $O(m+n^2) = O(n^2)$ dla niezbalansowanych drzew przeszukiwań binarnych,
- $O((n+m) \cdot K \cdot \log_K(n))$ dla zbalansowanych drzew K -arnych,

gdzie złożoności algorytmu wyszukiwania najkrótszych ścieżek w oparciu o kopce policzyliśmy w poprzednim podrozdziale i wynosiły one: $O((n+m) \cdot \log(n))$ i $O(m \cdot \log_d(n))$ odpowiednio dla kopków binarnych i R -arnych. Na podstawie powyższego zestawienia możemy podejrzewać, że struktura zbalansowanych drzew binarnych jest w pewnym stopniu konkurencyjna dla kopków tej samej arności³, jednak w tej analizie nie braliśmy w ogóle pod uwagę stałych czynników, jakie pojawiają się podczas wykonywania wszystkich, wyżej przeanalizowanych, operacji, a które przemawiają na niekorzyść zbalansowanych drzew przeszukiwań - te struktury (takie jak Drzewo Czerwono-Czarne czy Adelsona-Velskiego-Landisa) są znacznie bardziej złożone przez co wymagają nie tylko więcej pamięci na przechowywanie danych, ale też wykazują się mniejszą efektywnością niż prostsze struktury o tych samych, asymptotycznych czasach działania. Jak się przekonamy w następnym rozdziale, prawidłowość ta dotyczy również kopków Fibonacciego, które, pomimo lepszych wyników teoretycznych, nie sprawdzą się jako kolejka priorytetowa dla algorytmu Dijkstry właśnie ze względu na możliwość zastąpienia tej struktury przez dużo prostsze i mniej skomplikowane rozwiązania.

2.4.2 Kopiec Fibonacciego

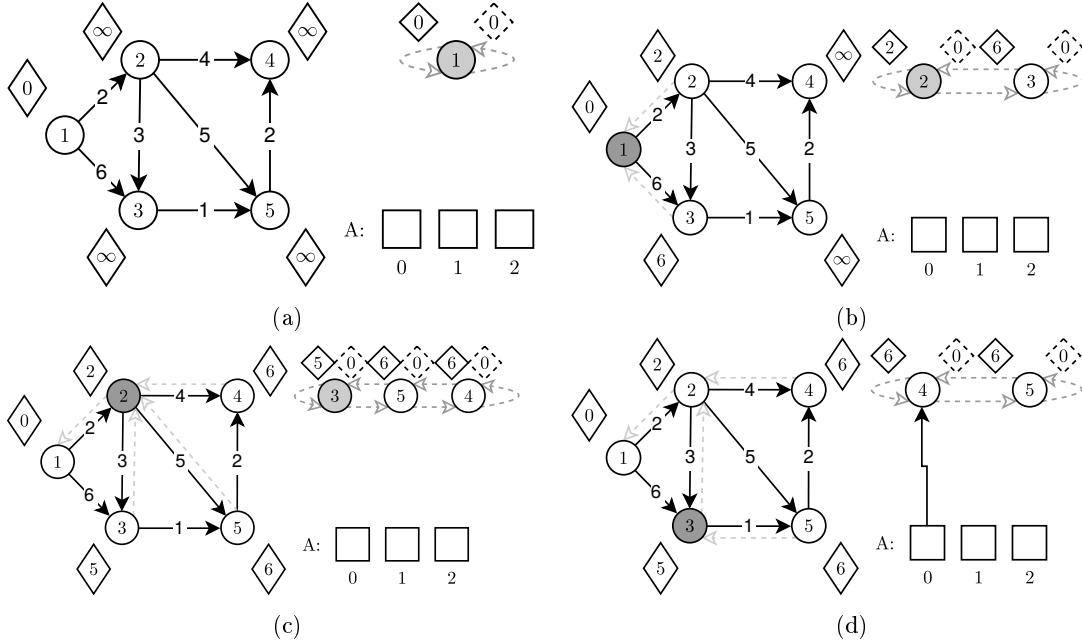
Przedstawiona w tym rozdziale implementacja algorytmu Dijkstry jako kolejkę priorytetową będzie wykorzystywać jedną z bardziej złożonych struktur danych, jakie będziemy omawiać - kopce Fibonacciego. Zaletą jej wykorzystania okaże się amortyzacyjnie lepszy czas wykonywania dla dwóch, podstawowych operacji, wykorzystywanych podczas działania naszego algorytmu - INSERT (Q, v) i DECREASE-KEY (Q, v, k).

Dodatkowo, aby jeszcze przyśpieszyć działanie podstawowej wersji implementacji kopca Fibonacciego, możemy dostosować ją do właściwego środowiska, w którym to oparta na kopcu kolejka priorytetowa będzie wykorzystywana. Pierwszą rzeczą, jaką możemy zauważyć to sposób zmiany ilości elementów, które znajdują się na kopcu - w odróżnieniu od algorytmu wyszukiwania najkrótszych ścieżek dla danego grafu $G = (V, E)$, gdzie ilość węzłów jest z góry znana, dla ogólnego przypadku nie jesteśmy w stanie nic powiedzieć o maksymalnej ilości elementów, jakie znajdą się na kopcu. Konsekwencją tej niewiedzy jest konieczność rezerwowania dodatkowej pamięci dla pomocniczych tablic za każdym razem, gdy wykonujemy operację EXTRACT-MIN (Q). Choć rozmiar takich tablic jest z góry znany i wynosi on $\lfloor \log_{\Phi}(n) \rfloor$ to bez znajomości maksymalnej wartości parametru n nie jesteśmy w stanie tego faktu w jakikolwiek sposób wykorzystać. Inaczej jest w przypadku,

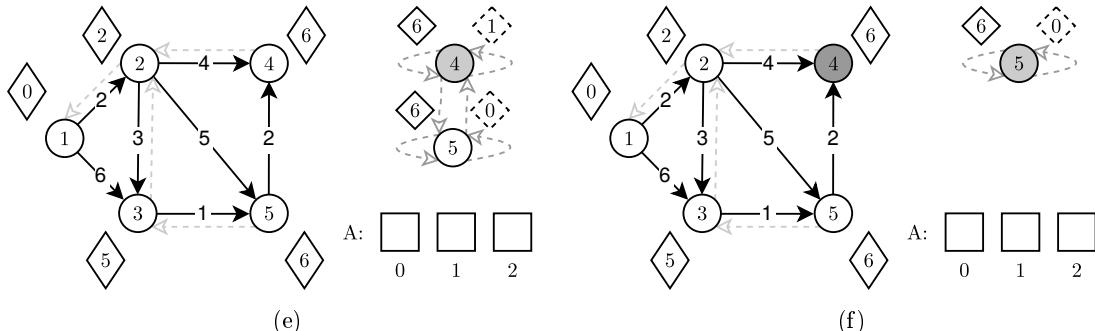
²Przedstawiono czasy dla odpowiednio: drzew zbalansowanych (takich jak RBT, AVL) i drzew niezbalansowanych, dla których pesymistyczna wysokość wynosi $O(n)$.

³Jeżeli byśmy chcieli uzyskać dla zbalansowanych drzew K -arnych takie samo oszacowanie na asymptotyczną złożoność obliczeniową, musielibyśmy przyjąć $d = \frac{m}{m+n}$ (wtedy $O((n+m) \cdot K \cdot \log_K(n)) = O(m \cdot \log_K(n))$), lecz nie możemy mieć struktury, której współczynnik rozgałęzienia (d) jest mniejszy od dwóch (przypadek drzewa binarnego)

gdy mamy dany graf G , którego liczba wierzchołków wynosi dokładnie $|V|$, co przekłada się na maksymalną liczbę elementów, jakie jednocześnie mogą znaleźć się na kopcu Fibonacciego - wówczas z każdą operacją EXTRACT-MIN (Q) korzystamy z tej samej tablicy pomocniczej $A[1 \dots \lfloor \log_{\Phi}(n) \rfloor]$, którą „czyścimy” pod sam koniec procedury, budując - zgodnie z podstawowym algorytmem - nową listę korzeni kopca Fibonacciego (iterując po całej tablicy i dodając, trzymane w niej drzewa ukorzenione, do głównej listy, zaś elementy samej tablicy zerując). Inną, bardziej oczywistą modyfikacją, jest wykorzystanie faktu, iż każda lista, która znajduje się wewnętrznej strukturze kopca, jest cykliczną listą dwukierunkową co, znów w przypadku wykonywania procedury EXTRACT-MIN (Q), pozwoli nam zaoszczędzić trochę czasu podczas pierwszych kroków tego algorytmu (zamiast usuwać najmniejszy element v z listy korzeni i iteracyjnie przepinać potomków tego węzła do wspomnianej listy, możemy „rozerwać” obie listy w wybranym przez nas punkcie, a następnie połączyć w czasie $O(1)$, zaś usuwanie powiązań między potomkami usuwanego węzła tymczasowo zignorować - podstawowa wersja algorytmu podczas przepinania węzłów u takich, że $u.\Pi = v$ ustawa te parametry na wartość **NULL**. Podkreślić należy słowo: „tymczasowo”, gdyż ta czynność zostanie wykonana w momencie rekonstrukcji kopca Fibonacciego z drzew, przechowywanych w tablicy $A[1 \dots \lfloor \log_{\Phi}(n) \rfloor]$ - wiedząc, że każdy jej element przechowuje wskaźnik do przyszłego węzła na liście korzeni będziemy dla każdego elementu z tablicy A dodatkowo niszczyć wskazanie tego węzła na jego rodzica, którego nie zniszczyliśmy wcześniej.).



Rysunek 2.8: **Działanie algorytmu Dijkstry w oparciu o kopiec Fibonacciego** (a) Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAF ze źródłem $v_s.id = 1$. Przerwanymi strzałkami odwzorowane są relacje między węzłami na kopcu Fibonacciego. W rombach od lewej dla każdego węzła v kolejno przedstawione są jego atrybuty: $v.d$ i $v.deg$. Szarym kolorem zaznaczono węzeł, który aktualnie jest minimalnym węzłem w kopcu. (b) Usunięto z kopca jego najmniejszy węzeł: v_1 . W wyniku relaksacji do listy korzeni kopca Fibonacciego zostały dodane węzły v_2 - ustawiony jako węzeł minimalny w momencie, gdy był on jedynym elementem na kopcu - oraz v_3 . (c) Wykonano relaksację dla następnego, wyjętego z kopca, węzła v_2 . W kopcu pozostały tylko jeden wierzchołek v_3 , który stał się tym samym jego najmniejszym elementem. W wyniku relaksacji do kopca zostały dodane nowe elementy: v_5 oraz v_4 . (d) Usunięto najmniejszy węzeł z kopca: v_3 . Po tej operacji na kopcu zostało więcej niż jeden element, więc wykonujemy operację CONSOLIDATE (Q). Przeglądamy kolejno węzły v na liście korzeni i, jeżeli $A[v.deg] = \mathbf{NULL}$ (zaczynamy od węzła v_4 , $v_4.deg = 0$) to zapamiętujemy w $A[v.deg]$ korzeń drzewa v . Skanujemy kolejny węzeł na liście korzeni: v_5 .



Rysunek 2.8: (e) W przypadku, gdy dla skanowanego elementu (v_5) zachodzi warunek przeciwny ($A[v.deg] \neq \text{NULL}$) tworzymy nowe drzewo typu *min*, łącząc to o korzeniu w aktualnie badanym wierzchołku z drzewem, na którego korzeń wskazuje element $A[v.deg]$. Korzeniem nowego drzewa zostaje węzeł o mniejszym kluczcu, zaś drzewo, mające korzeń w drugim węźle, staje się potomkiem tego pierwszego. W tym przypadku oba korzenie drzew (v_4 i v_5) mają te same priorytety ($v_4.d = v_5.d$) - korzeniem nowego drzewa staje się wcześniej badany węzeł v_4 , a jego stopień zostaje zwiększyony o 1. Korzeń nowo powstałego drzewa jest zapamiętywany w $A[v_4.deg]$ (jeśli przed tym krokiem zachodzi $A[v.deg] \neq \text{NULL}$, gdzie $v.deg$ to stopień korzenia nowego drzewa, to operacja d-e powtarza się). (f) Z kopca usuwany jest kolejny wierzchołek o najmniejszej wartości atrybutu d . Brak jest dla niego krawędzi wychodzących. Na szczytce kopca zostaje ostatni węzeł, który staje się jego najmniejszym elementem. Następny krok opróżnia zawartość kopca, wykonuje relaksację krawędzi, wychodzących z węzła v_5 i kończy algorytm.

Złożoność obliczeniowa

Przyjrzyjmy się na początek złożonościom poszczególnych operacji, wykorzystywanych w algorytmie Dijkstry, dla - wcześniej omawianych - kopków R -arnych oraz dla nowo przedstawionej struktury.

Operacja	Kopiec		
	(koszt pesymistyczny)	(koszt zamortyzowany)	
	binarny	R -arny	Fibonacciego
INSERT (Q, v)	$O(\log(n))$	$O(\log_R(n))$	$O(1)$
EXTRACT-MIN (Q)	$O(\log(n))$	$O(R \cdot \log_R(n))$	$O(\log(n))$
DECREASE-KEY (Q, v, k)	$O(\log(n))$	$O(\log_R(n))$	$O(1)$

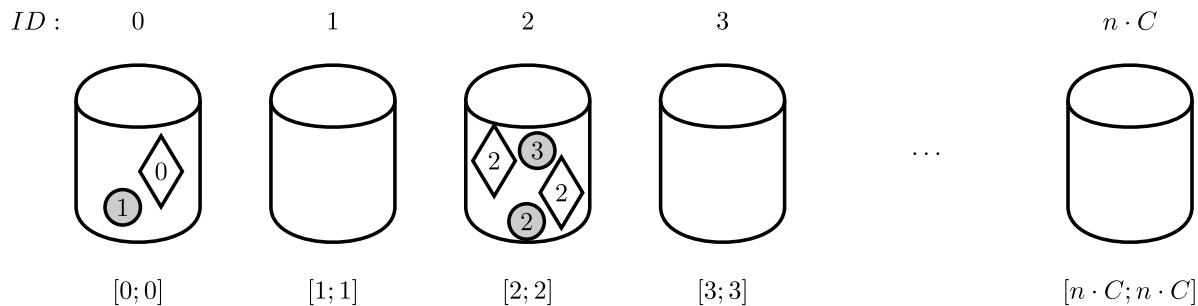
Przytaczając raz jeszcze ogólny wzór na złożoność obliczeniową generycznego algorytmu Dijkstry (2.4.1) natychmiast otrzymamy następującą złożoność: $O(m \cdot O(\text{DK}(Q, v, k)) + n \cdot [O(\text{I}(Q, v)) + O(\text{EM}(Q))]) = O(m \cdot O(1) + n \cdot [O(1) + O(\log(n))]) = O(m + n \cdot \log(n))$.

2.5 Struktury oparte na kubełkach

Analizując wszystkie algorytmy, które do tej pory omawialiśmy, możemy dojść do wniosku, że każdy kolejny do poprawnego działania wykorzystywał coraz to bardziej skomplikowaną strukturę danych (od algorytmu Bellmana-Forda, który nie korzystał z żadnych dodatkowych struktur, przez proste uporządkowanie wierzchołków w grafie dla algorytmów, opartych na sortowaniu topologicznym, wykorzystanie prostych struktur danych w charakterze kolejek priorytetowych takich jak listy i tablice, aż po te bardziej złożone). W tych ostatnich wykorzystywaliśmy szeroki wachlarz struktur danych ogólnego przeznaczenia, a które wykazywały własności odpowiednie kolejkom priorytetowym. W tym rozdziale przyjrzymy się rodzinie modyfikacji algorytmów Dijkstry, opartych na **kubełkach** (ang. *buckets*) oraz strukturach z nich zbudowanych. Przestaniamy także opierać naszą analizę złożoności obliczeniowej o trzy, podstawowe operacje $\text{INSERT}(Q, v)$, $\text{EXTRACT-MIN}(Q)$ i $\text{DECREASE-KEY}(Q, v, k)$ na rzecz bardziej indywidualnej analizy każdego z przedstawionych rozwiązań. O ile w poprzednich podejściach do generycznego algorytmu Dijkstry i jego modyfikacji większość czasu poświęciliśmy tylko na zamianach jednych struktur kolejek priorytetowych na drugie, co pozwalało nam na taką analizę, to w przypadku niżej omawianych implementacji będziemy niejednokrotnie chcieli zmienić podstawowy szkielet algorytmu, jaki przedstawiliśmy jakiś czas temu (pseudokod 9 w podrozdziale 2.2.1).

2.5.1 Pierwsze podejście

Pierwszą, naiwną próbą zrezygnowania z tradycyjnych kolejek priorytetowych będzie stworzenie tablicy o liczbie elementów, odpowiadającej wszystkim możliwym kluczom wierzchołków, jakie mógł wygenerować algorytm Dijkstry w momencie wstawiania danego węzła do jednej z wybranych implementacji kolejki priorytetowej. Mówiąc prościej, nasza tablica będzie mieć taką liczbę elementów, by jej ostatni indeks odpowiadał odległości najdłuższej z możliwych ścieżek dla danego grafu $G = (V, E)$. Zdefiniujmy $C = \max\{c_{ij} : e_{ij} \in E_{\text{right}}\}$ jako największy koszt ścieżki, występującej w grafie G . Wiedząc, że najdłuższa ścieżka bez cykli (w sensie ilości składowych) ma co najwyżej $|V| - 1$ krawędzi, możemy oszacować z góry liczbę potrzebnych elementów tablicy przez $(n - 1) \cdot C + 1 \leq n \cdot C + 1$ (ostatni szacunek robimy tylko dla własnej wygody, gdyż taka sytuacja, gdzie na ścieżce, składającej się z $|V| - 1$ krawędzi, gdzie koszt każdej z nich będzie maksymalny i wynosił C , najprawdopodobniej się nie zdarzy, a rzeczywista koszt najdłuższej takiej ścieżki będzie z reguły dużo mniejszy).



Rysunek 2.9: **Struktura, oparta na kubełkach.** Pod każdym kubełkiem zaprezentowany jest indeks elementu tablicy, w jakim się znajduje, oraz jego zakres. Do kubełka należą te węzły v , których $v.d$ znajduje się w, podanych dla kubełka, zakresach.

Samo zaimplementowanie tablicy, indeksowanej od 0 do $n \cdot C$ oczywiście nie zagwarantuje nam jeszcze poprawności działania algorytmu, polegającego na przeszukiwaniu w kolejności rosnących indeksów tablicy w celu znalezienia węzła v o najmniejszej wartości etykiety d i zwróceniu go jako wynik operacji $\text{EXTRACT-MIN}(Q)$ takiej naiwnej struktury. Musimy rozważyć także sytuację, w której dwa lub więcej węzłów w trakcie działania algorytmu okazuje się być równoodległymi od źródła. Stąd pojawia się pojęcie **kubełków**, które będziemy traktować jako kontenery, najczęściej zaimplementowane jako listy dwukie-

runkowe, z możliwymi operacjami IS-EMPTY (B), EXTRACT-HEAD (B), EXTRACT-TAIL (B) czy EXTRACT-MIN (B), przechowującymi wierzchołki grafu o zadanych właściwościach, które różnić się będą w zależności od algorytmu. W tym przypadku będziemy mieli tablicę składającą się z $n \cdot C + 1$ kubełków, gdzie każdy z nich będzie mógł zawierać tylko takie węzły v , których $v.d = k$, gdzie k to numer indeksu w naszej tablicy (rysunek 2.9).

Algorytm po kolej skanuje kubelki, poczynając od tego o najniższym indeksie, wykonując relaksację dla napotkanych węzłów. W naszym grafie nie ma krawędzi o ujemnym koszcie (pokazaliśmy, że dla takiego przypadku algorytm Dijkstry nie działa), więc wykonując operację relaksacji dla dowolnego węzła v_i w kubełku o indeksie k ($v_i.d = k$), dla wszystkich krawędzi z niego wychodzących $e : v_i \xrightarrow{1} v_j$, mamy pewność, że po jej wykonaniu dla każdego węzła v_j zachodzi $v_i.d \leq v_j.d$. To zaś gwarantuje nam, że analizując kubelki w takiej kolejności, jaką podaliśmy, nie pominiemy żadnego z węzłów, należących do grafu G (jeżeli tylko będą osiągalne ze źródła).

Algorytm działa w czasie $O(m + n \cdot C)$:

- W najgorszym przypadku musimy przeskanować $O(n \cdot C)$ kubełków,
- Operację relaksacji w najgorszym możliwym przypadku wykonamy za każdym razem, każdorazowo powodując przepięcie węzła u z jednego kubełka do drugiego (co na liście dwukierunkowej zrobimy w czasie stałym).

2.5.2 Z przepełnieniem

Jak nietrudno zauważyc, poprzednie podejście wykorzystywało ogromnie dużo pamięci na dodatek nie zapewniając, że algorytm będzie działał dostatecznie szybko - stała C , którą ustalaliśmy na początku algorytmu może być dowolnie duża, w szczególności przekraczać n - otrzymalibyśmy wtedy algorytm nie tyle co o dużej złożoności pamięciowej, ale także czasowej. W tym podrozdziale postaramy się zaradzić pierwszemu z problemów, ograniczając liczbę kubełków z $n \cdot C$ do $a + 1$, gdzie za a przyjmiemy dowolną liczbę mniejszą od $C + 1$. Dodatkowo wprowadzimy specjalny kubełek, do którego będą trafiać wszystkie takie węzły, których dystans do źródła będzie większy, niż zakres ostatniego z a kubełków. Innymi słowy ograniczymy ich liczbę z pierwszego algorytmu, pozostawiając a pierwszych kubełków, zaś wszystkie następne zastępując jednym, o potencjalnie nieskończonym zakresie $[A + a; \infty]$, gdzie początkowo $A = 0$. $a + 1$ 'y kubełek będziemy nazywać przepełnieniem (ang. *overflow bucket*).

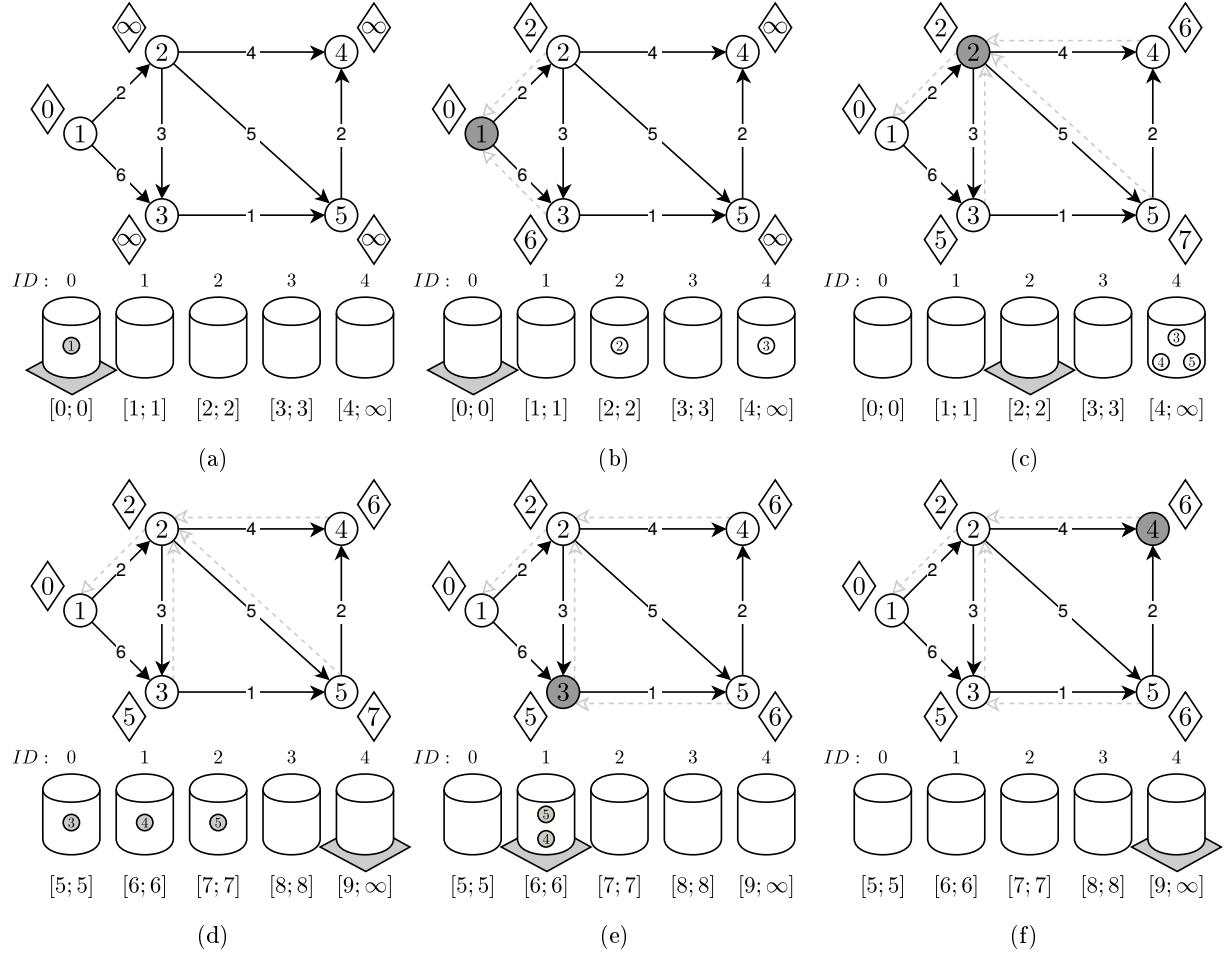
Algorithm 10: DKM (G, s)

```

1 begin
2   INIT-GRAPH ( $G, s$ )
3   foreach  $i \in \{0 \dots a - 1\}$  do
4      $B[i] \leftarrow$  pusty kubełek o zakresie  $[i; i]$ 
5    $B[a] \leftarrow$  Overflow bag
6   while Wszystkie kubełki nie są puste do
7     for  $i = 0$  to  $a - 1$  do
8       foreach  $v_j \in B[i]$  do
9         Usuń  $v_j$  z  $B[i]$ 
10        foreach  $e_{jk} : v_j \xrightarrow{1} v_k$  do
11           $\text{RELAX}(v_j, v_k)$  /* Jeśli  $v_k.d$  się zmieniło to przenieś  $v_k$  do odpowiedniego kubelka. */
12         $minDist \leftarrow \min \{v_j.d : v_j \in B[a]\}$ 
13        foreach  $i \in \{0 \dots a - 1\}$  do
14           $B[i] \leftarrow [minDist + i; minDist + i]$ 
15        foreach  $v_j \in B[a]$  do
16          Przenieś  $v_j$  do odpowiedniego kubełka.

```

Algorytm rozpoczyna pracę od skanowania wszystkich kubełków, poczynając od tego, o najmniejszym zakresie. Dla kolejno napotkanych wierzchołków algorytm usuwa je, po czym wykonuje relaksacje dla wszystkich krawędzi z nich wychodzących, odpowiednio aktualizując tablicę z kubełkami. W przypadku dojścia do ostatniego kubełka o zakresie $[A + a; \infty]$ algorytm wyszukuje w nim najmniejszy (w sensie odległości od źródła) wierzchołek v i aktualizuje A , które jako wartość przyjmuje $v.d$. Następnie algorytm aktualizuje zakresy wszystkich pierwszych a kubełków, których całkowity zakres wynosi $[A(i); A(i) + a - 1]$ w i -tej iteracji algorytmu. Po przeniesieniu wszystkich wierzchołków do prawidłowych kubełków po zmianie ich zakresów algorytm rozpoczyna kolejną iterację, wracając do pierwszego kubełka i działa, dopóki wszystkie nie zostaną opróżnione.



Rysunek 2.10: **Działanie algorytmu DKM (alg. Dijkstry opartego na kubełkach z przepełnieniem)**
(a) Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAH ze źródłem $v_s.id = 1$ i $a = 4$. Badany kubełek jest na rysunku podświetlony zaciemionym rombem. **(b)** Z pierwszego kubełka zostało wyciągnięty jedyny wierzchołek. W wyniku relaksacji krawędzi z niego wychodzących do odpowiednich kubełków zostały dodane nowe węzły ($v_2.d = 2$ i $v_2.d \in B[2].range$ oraz $v_3.d = 6$ i $B[a-1].range < v_3.d$). **(c)** Kubełek $B[1]$ był pusty. Algorytm wykonał operację RELAX dla następnego, najmniejszego węzła: v_2 i usunął go z jego kubełka. W jej wyniku zostały zrelaksowane krawędzie: e_{24} , e_{23} , i e_{25} . Dla wszystkich węzłów v , wskazywanych przez te krawędzie, ich $v.d \in B[a].range$. **(d)** Kubełek $B[3]$ był pusty i algorytm dotarł do ostatniego kubełka. Najmniejsza wartość $v.d$ dla $v \in B[a]$ to 5. Algorytm zaktualizował zakresy wszystkich kubełków i przepiął wszystkie węzły z badanego kubełka na ich odpowiednie miejsca. **(e)** Algorytm ponownie zaczął skanować kubełki od $B[0]$ i usunął z niego węzeł v_3 . W wyniku relaksacji węzeł v_5 został przepięty do węzła, odpowiadającego zakresem jego nowej odległości od źródła. **(f)** Po usunięciu z kubełka $B[1]$ węzłów v_4 i v_5 (w dowolnej kolejności jako, że $v_4.d = v_5.d$) wszystkie kubełki są puste.

Złożoność obliczeniowa

Algorytm działa w czasie zależnym od parametru $a < C + 1$. Analizując zachowanie się algorytmu w najgorszym, możliwym przypadku mamy kolejno:

- $O\left(\frac{n \cdot C}{a}\right)$ - tyle razy zostaną przeskanowane wszystkie kubelki, gdzie w każdej iteracji i ich łączny zakres to $[A(i); A(i) + a - 1]$, nie licząc ostatniego węzła. Przed rozpoczęciem następnej iteracji algorytm zmienia zakresy wszystkich kubelków, wyszukując w ostatnim z nich węzeł v taki, że $v.d = \min\{v_i.d : v_i \in B[a]\}$, gdzie $v.d$ jest na pewno większe od prawej strony zakresu przedostatniego kubelka w tablicy (wynika to z samej własności zakresów kubelków, które są rozłączne). Niech zakres kubelka z przepełnieniem podczas i -tej iteracji wynosi $[A(i) + a; \infty]$. W najgorszym przypadku $v.d = A(i) + a$ - wtedy podczas zmiany zakresów kubelków od $B[0]$ do $B[a - 1]$ lewa strona nowego zakresu pierwszego kubelka jest o 1 większa od starego zakresu $B[a - 1]$. Dla tak zmienianych zakresów mamy zapewnione, że podczas $\frac{n \cdot C}{a}$ iteracji każda wartość od 0 do $n \cdot C - 1$ będzie występowała jako zakres danego kubelka, nie będącego przepełnieniem, dokładnie jeden raz. W najgorszym przypadku w grafie będzie istnieć węzeł, którego odległość od źródła wynosi $(n - 1) \cdot C$, więc algorytm będzie musiał wykonać wszystkie $O\left(\frac{n \cdot C}{a}\right)$ iteracji, nim do niego dotrze i usunie go z kubelka.
- W najgorszym, możliwym przypadku w każdej kolejnej, i -tej iteracji w ostatnim kubelku znajduje się $k(i)$ węzłów tak, że $\bigcup_{i=1}^{\frac{n \cdot C}{a}} |k(i)| = n - 1$ (na przestrzeni działania całego algorytmu każdy z $|V| - 1$ wierzchołków, poza źródłem, znajdzie się w tym kubelku dokładnie jeden raz). Łączny koszt wyszukiwania takiego węzła, że $v.d = \min\{v_i.d : v_i \in B[a]\}$ we wszystkich iteracjach możemy zatem oszacować z góry przez: $O(n)$.
- $O(n \cdot C)$ - w takim czasie zostaną przeskanowane wszystkie kubelki, wyłączając kubelek ostatni, jeżeli algorytm wykona $O\left(\frac{n \cdot C}{a}\right)$ iteracji (za każdym razem skanując $a + 1$ kubelków).
- Pod koniec każdej iteracji algorytm wykonuje przeniesienia wszystkich węzłów z ostatniego kubelka - w najgorszym, możliwym przypadku takich wierzchołków będzie $\sum_{i=1}^{\frac{n \cdot C}{a}} |v_j : v_j \in B[a]| = O(n)$.
- Poza przenoszeniem węzłów pod koniec każdej iteracji, algorytm wykonuje także m relaksacji podczas skanowania pierwszych $a - 1$ kubelków w trakcie trwania wszystkich rund.
- Przeniesienie węzła z kubelka $B[i]$ do $B[j]$ jest wykonywane w czasie stałym.

W najgorszym, możliwym przypadku algorytm wykona łącznie $O(m + n)$ relaksacji oraz przeglądnie $O(n \cdot C + \frac{n \cdot C}{a})$ kubelków, co daje nam łączny czas działania: $O(m + n \cdot C + \frac{n \cdot C}{a})$

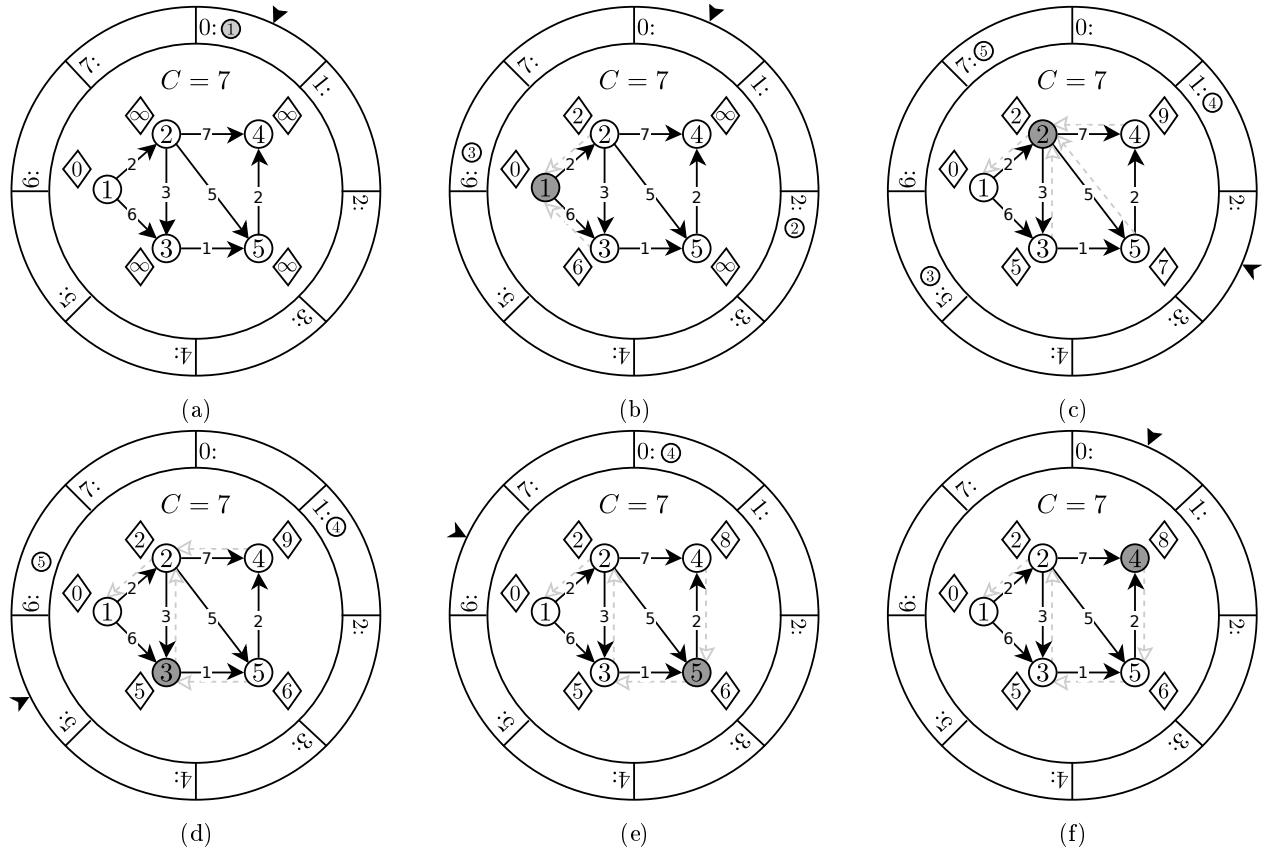
2.5.3 Dial

Zwrócić uwagę, że dla $a = C$ poprzedni algorytm zachowuje się niemal identycznie, co opisany jako pierwszy (z $n \cdot C$ kubelkami), z tą różnicą, że działa on w rundach, gdzie maksymalnie może być ich $O\left(\frac{n \cdot C}{a}\right)$. Pokazaliśmy, że dla tak dobranego parametru a zakresy kubelków we wszystkich rundach tworzą partycję⁴ taką, że $\bigcup_{i=1}^{\frac{n \cdot C}{a}} \bigcup_{j=0}^{a-1} [A(i) + j; A(i) + j] = \bigcup_{i=1}^{\frac{n \cdot C}{a}} [A(i); A(i) + a - 1] = [0; n \cdot C - 1]$, co sprowadza się do skanowania wszystkich kubelków o zadanych zakresach z najnowszej wersji algorytmu. Dopiero dla parametru $a < C$ możemy zacząć liczyć na to, że z każdą iteracją pewna liczba kubelków zostanie pominięta.

Wybór takiego ograniczenia dla a ma swoje uzasadnienie i wnioski, z niego wynikające, zostały wykorzystane do usprawnienia następnego algorytmu, jakim jest algorytm wymyślony przez Roberta B. Diala, nazwany od nazwiska jego pomysłodawcy, którego pierwotna implementacja w najgorszym przypadku także wymagała użycia $n \cdot C$ kubelków. W wyniku modernizacji tego algorytmu, ograniczono liczbę potrzebnych kubelków do $C + 1$, powołując się na fakt, że w żadnym momencie algorytmu po wykonaniu relaksacji krawędzi, prowadzącej z węzła v_i do v_j odległości od źródła tych dwóch węzłów są w relacji: $v_i.d \leq v_j.d \leq v_i.d + C$, gdzie $C = \max\{c_{ij} : e_{ij} \in E\}$. Wynika z tego, że w momencie skanowania k -tego kubelka spośród $C + 1$ kubelków i zakładając, że następnym kubelkiem po $B[C]$ jest $B[0]$ wszystkie węzły v_j , których odległość od źródła $v_j.d$ ulegnie zmianie w wyniku relaksacji krawędzi, wychodzących z węzła v_i (gdzie $v_i.d = k$) zostaną

⁴Rodzina niepustych, rozłącznych podzbiorów danego zbioru dająca w sumie cały zbiór.

przemieszczone do kubełków o indeksach od k do $k + C$. Z założenia, że kubełki ułożyliśmy na okręgu tak, by po ostatnim z nich następował pierwszy, to indeks $k + C$ w rzeczywistości odpowiada kubełkowi o indeksie $(k + C) \bmod C + 1 = k - 1$ w związku z czym dla $C + 1$ kubełków nigdy nie zajdzie sytuacja, by w jednym kubełku znajdowały się węzły v i u , dla których $v.d \neq u.d$ (co oczywiście mogłoby się wydarzyć, gdybyśmy wzięli $c < C + 1$ kubełków).



Rysunek 2.11: **Działanie algorytmu Dial z $C + 1$ kubełkami** (a) Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez **INIT-GRAF** ze źródłem $v_s.id = 1$. Mamy $C + 1$ kubełków, indeksowanych od 0 do C . Grot strzałki wskazuje obecnie badany kubełek. (b) Z aktualnie badanego kubełka pobierany jest węzeł i wykonywana jest relaksacja jego krawędzi wychodzących. W jej wyniku do kubełków $B[2]$ i $B[6]$ odpowiednio wstawione są węzły v_2 ($v_2.d = 2$) i v_3 ($v_3.d = 6$). Następny kubełek ($B[1]$) jest pusty. (c) Algorytm przeskanował następny, niepusty kubełek $B[2]$, w wyniku czego dokonał relaksacji krawędzi e_{23} , e_{25} oraz e_{24} . Węzły na końcach tych krawędzi zmniejszyły swoją odległość od źródła v_1 i zostały przeniesione do odpowiednich kubełków ($v_4.d = 9$ więc został przeniesiony do kubełka o indeksie: $9 \bmod C + 1 = 1$). (d) Następny, niepusty kubełek to $B[5]$, zawierający węzeł v_3 ($v_3.d = 5$), dla którego, w wyniku relaksacji jego wychodzących krawędzi, przepięty zostaje węzeł v_5 - jego odległość od węzła zmniejsza się z $v_5.d = 7$ do $v_5.d = 6$. (e) Analogicznie jak w poprzednim kroku, w wyniku operacji **RELAX**, wykonanej dla krawędzi e_{54} , przepinany jest wierzchołek v_4 (z kubełka $B[9 \bmod C + 1]$ do $B[8 \bmod C + 1]$). (f) Algorytm skanuje następny, niepusty kubełek, wykonuje potrzebne operacje relaksacji, po czym kończy działanie, jeżeli podczas całego „obrotu” nie napotkał żadnego, niepstego kubełka.

Uwagi do algorytmu

Jak w każdym poprzednim przypadku, gdy zakres kubełków był jednostkowy (to znaczy $B[i].range = [k; k]$), nie ważna jest kolejność wyciągania wierzchołków z takiego kubełka. Co więcej, warto zauważać, że dla pewnych sytuacji bardziej opłacalną operację jest obliczanie nowych indeksów kubełków, do których

mają trafić wierzchołki, w czasie rzeczywistym, niż modyfikować już istniejące zakresy (w tym przypadku wykonywaliśmy operację $\text{mod } C + 1$, zamiast modyfikować zakresy wszystkich kubelków - przy omawianiu algorytmu opartego na kubelkach z przepełnieniem taki indeks mogliśmy obliczyć, odejmując od $v.d$, dla dowolnego v , wyrażenie $A(i)$). Analizując obydwa podejścia mamy następującą sytuację:

	Liczba iteracji	Złożoność obliczeniowa pojedynczego wywołania
Zmiana zakresu kubelków	$O(n)$	$\Theta(C + 1)$
Modulowanie indeksów	$\Theta(m)$	$\Theta(1)$

W pierwszym przypadku liczba modyfikacji zakresów każdego z $C + 1$ kubelków jest ściśle zależna od liczby „okrążeń”, jakie musi wykonać algorytm Dijkstry, by przeskanować wszystkie wierzchołki w grafie. Dla drugiego przypadku liczba operacji obliczania nowych indeksów wierzchołków jest ściśle ograniczona do liczby wykonywanych relaksacji na przestrzeni działania całego algorytmu. Zastosowanie pierwszego wariantu dodatkowo wymaga od nas stworzenia osobnej struktury, która umożliwiałaby przechowywanie zakresów każdego z $C + 1$ kubelków (gdź, w przypadku algorytmu Dial, podstawą do obliczenia zakresów kubelków są ich indeksy w tablicy $B[0 \dots C]$).

Na sam koniec możemy zauważać, że warunek, który musi zajść, by algorytm Dial zakończył działanie (to jest w momencie, gdy wszystkie kubelki są puste) wymaga od nas wykonania $O(n \cdot C)$ operacji (w najgorszym przypadku), podczas gdy możemy zastąpić je przez sprawdzanie, czy w momencie wykonywania relaksacji węzła, którego odległość od źródła się zmienia, zostanie przeniesiony do kubelka o mniejszym indeksie niż ten, który aktualnie skanujemy. Jeżeli taka sytuacja nie będzie miała miejsca dla żadnego ze skanowanych $C + 1$ kubelków to znaczy, że wszystkie kubelki są puste i po wykonaniu skanowania ostatniego kubelka algorytm powinien zakończyć pracę. Jednak, tak jak w poprzednim przypadku, jest to tylko niewielka modyfikacja, zamieniająca złożoność jednej ze składowych operacji algorytmu Dijkstry z $O(n \cdot C)$ na $O(m)$ i na odwrót.

Złożoność obliczeniowa

Algorytm Dijkstry działa w czasie $O(m + n \cdot C)$ - algorytm wykona co najwyżej m operacji relaksacji, każdą w czasie $O(1)$, i będzie musiał przeskanować, w najgorszym przypadku, $O(n \cdot C)$ kubelków.

2.5.4 Aproksymacja zakresu

W poprzednich algorytmach skupialiśmy się na redukcji ilości wykorzystywanych kubelków, w celu oszczędzenia jak największej części pamięci operacyjnej i przyspieszeniu obliczeń, zachowując jednocześnie jednostkowe zakresy kubelków, co w najgorszych przypadkach zmuszało nas do wykonania $O(n \cdot C)$ operacji bez względu na to, jak przeglądaliśmy stworzone przez nas kubelki, bez względu na ich strukturę. Przedstawiony poniżej algorytm jest pierwszym algorytmem typu korekcyjnego (ang. *Label Correcting Algorithm*, oparty na kubelkach). Algorytmy tego typu różnią się od tej pory omawianych (*LSA* - ang. *Label Setting Algorithm*) tym, że nie posiadają własności, zapewniającej nas o zajściu warunku $v.d = \delta(s, v)$ tuż po wykonaniu relaksacji dla wszystkich krawędzi, wychodzących z tego wierzchołka i usunięciu go z listy wierzchołków, czekających na przetworzenie. Innymi słowy dopuszczały w nich możliwość wielokrotnego poprawiania odległości od źródła dla dowolnego węzła, bez zachowania kolejności ich poprawiania w porządku, jaki wcześniej zapewniały nam kolejki priorytetowe (algorytm Bellmana-Forda jest też takim algorytmem).

Algorithm 11: GENERIC-LABEL-CORRECTING-ALGORITHM (G, s)

```

1 begin
2   INIT-GRAF ( $G, s$ )
3   while  $\exists e_{ij} : v_i \xrightarrow{1} v_j \wedge v_j.d > v_i.d + c_{ij}$  do
4     RELAX ( $v_i, v_j$ )

```

Dowód.

Bardzo łatwo wykazać, że powyższy algorytm działa w skończonym czasie i zwraca poprawne wyniki. Wiemy, że dla każdego węzła v po inicjalizacji grafu procedurą INIT-GRAF (G, s) zachodzi $v.d \geq \delta(s, v)$,

a dodatkowo $n \cdot C \geq v.d$. Dla grafu z ujemnymi kosztami ścieżek, z którymi algorytm typu *LCA* jest w stanie sobie poradzić, możemy także dopisać $\delta(s, v) > -n \cdot C$. Dla każdego znalezionej krawędzi e_{ij} w grafie, takiego, że zachodzi $v_j.d > v_i.d + c_{ij}$ algorytm wykonuje relaksację, po której $v_j.d = v_i.d + c_{ij}$ - inaczej mówiąc podczas każdej relaksacji zmniejsza wartość $v_j.d$ co najmniej o 1. Z ograniczenia $n \cdot C \geq v_j.d \geq \delta(s, v_j) > -n \cdot C$ wynika zatem, że w najgorszym przypadku algorytm potrzebuje wykonać $2 \cdot n \cdot C$ kroków, by $v_j.d \leq v_i.d + c_{ij}$ dla każdego węzła v_i (dla $v_j.d = \delta(s, v_j)$ z lematu 1.3.5). Analogicznie wykonamy operacje relaksacji dla wszystkich n węzłów w grafie, więc po $O(n \cdot (n \cdot C))$ krokach algorytm zakończy działanie i dla wszystkich węzłów v : $v.d = \delta(s, v)$. \blacklozenge

W algorytmie, wykorzystującym aproksymacyjne kubełki (ang. *Dijkstra algorithm with implemented with approximate buckets*), zakres każdego, i -tego kubełka określany jest jako $[i \cdot b; (i+1) \cdot b - 1]$, gdzie $b < C + 1$. Algorytm łączy w sobie ideę konieczności posiadania jedynie $C + 1$ kubełków, dodatkowo jeszcze zmniejszając ich liczbę, poszerzając ich zakresy, co powoduje, że potrzebujemy ich już tylko $\lfloor \frac{C}{b} \rfloor + 1$. Wierzchołki w każdym kubełku są trzymane w kolejce *FIFO* (nie przejmujemy się kolejnością wyciągania wierzchołków w przypadku, gdy jest ich kilka w jednym kubełku), a ich skanowanie odbywa się na dokładnie tej samej zasadzie jak w algorytmie Dial.

Algorithm 12: DKA (G, s)

```

1 begin
2   INIT-GRAPH ( $G, s$ )
3    $K \leftarrow \lfloor \frac{C}{b} \rfloor + 1$ 
4   foreach  $i \in \{0 \dots K\}$  do
5      $B[i] \leftarrow$  pusty kubełek o zakresie  $[i \cdot b; (i+1) \cdot b - 1]$ 
6   while Wszystkie kubełki nie są puste do
7     for  $i = 0$  to  $K$  do
8       foreach  $v_j \in B[i]$  do
9         Usuń  $v_j$  z  $B[i]$ 
10        foreach  $e_{jk} : v_j \xrightarrow{1} v_k$  do
11          RELAX ( $v_j, v_k$ ) /* Jeśli  $v_k.d$  się zmieniło to przenieś  $v_k$  do odpowiedniego kubelka. */

```

Złożoność obliczeniowa

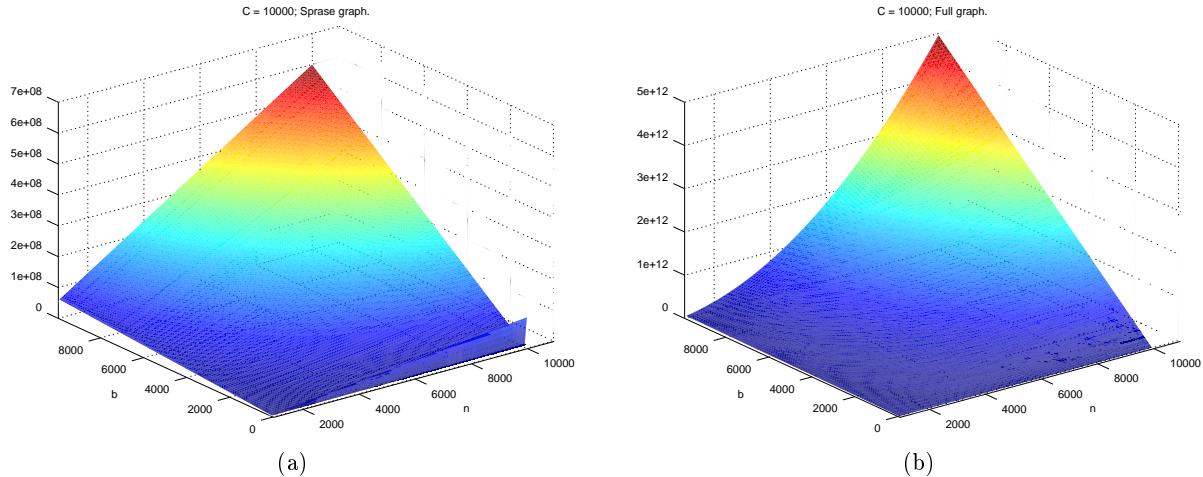
W najgorszym przypadku algorytm działa w czasie $O(m \cdot b + n \cdot (b + \frac{C}{b}))$:

- Tak jak w każdym przypadku, gdy mieliśmy do czynienia ze strukturą, opartą na kubełkach, algorytm, by skończyć działanie, musi je wszystkie przeglądać. Podobnie jak w algorytmie Dial musimy w najgorszym przypadku wykonać n przebiegów dla wszystkich kubełków w strukturze - tych mamy $O(\frac{C}{b})$, co łącznie daje nam czas, ograniczony z góry przez: $O(n \cdot \frac{C}{b})$
- Przyjrzyjmy się sytuacji, w której do kubełka k o zakresie $[i \cdot b; (i+1) \cdot b - 1]$ pierwszy raz jest wstawiany węzeł v_j taki, że $v_j.d = (i+1) \cdot b - 1$. Przyjmijmy dodatkowo, że dla każdej relaksacji krawędzi, która prowadzi do tego węzła odległość $v_j.d$ jest zmniejszana o 1 tak, że jest on b razy wstawiany do tego samego kubełka. W takim przypadku algorytm, w wyniku skanowania kubełka k , wyciągnie dany wierzchołek dokładnie b razy, gdzie za ostatnim razem wierzchołek znajduje się na samym początku kubełka ($v_j.d = i \cdot b$). W czasie skanowania kubełka k relaksacje, które są wykonywane, dotyczą tylko takich krawędzi $e : v_l \xrightarrow{1} v_j$, dla których węzły $v_l \in B[k]$, z czego wynika, że dla dowolnego takiego wierzchołka $i \cdot b \leq v_l.d \leq (i+1) \cdot b - 1$. Zatem, gdy $v_j.d = i \cdot b$ to dla dowolnej operacji $RELAX(v_l, v_j)$ będziemy mieli $v_j.d \geq v_l + c_{lj} = i \cdot b \geq i \cdot b + c_{lj}$ (dla kosztów ścieżek nieujemnych), co jest równoważne temu, że relaksacja krawędzi nigdy już nie zajdzie, a tym samym nie dodamy wierzchołka v_j ponownie do któregoś z kubełków (algorytm skanuje kubełki od lewej do prawej, więc relaksacja, którą opisaliśmy,

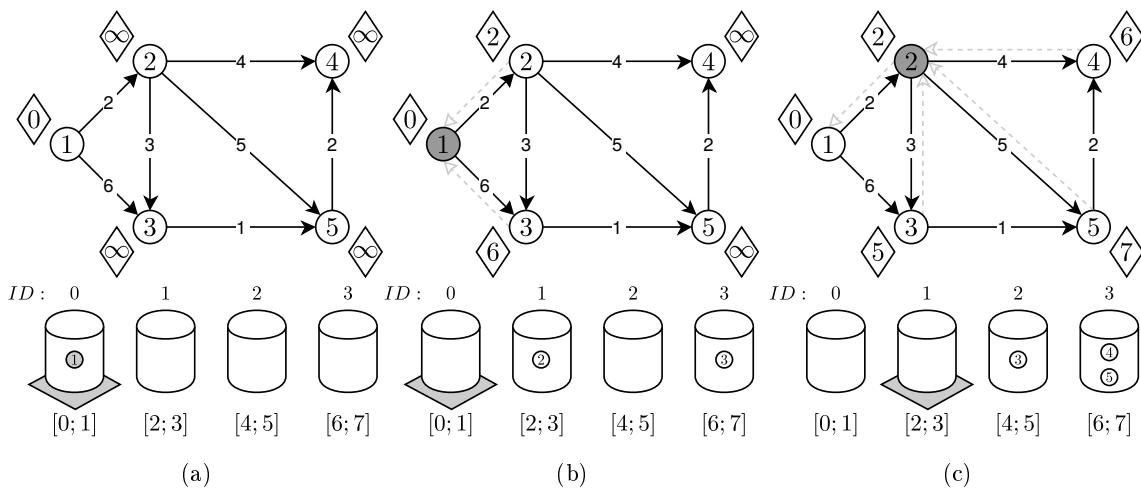
tym bardziej nie zajdzie, jeżeli pójdziemy do dalszych kubełków). W związku z tym dany wierzchołek algorytm będzie badał co najwyżej b razy. Dla n wierzchołków mamy ograniczenie $O(n \cdot b)$.

- Bezpośrednio z powyższego wniosku wynika także, że algorytm może w najgorszym przypadku wykonać $O(m \cdot b)$ relaksacji (dla każdego kubełka b razy).

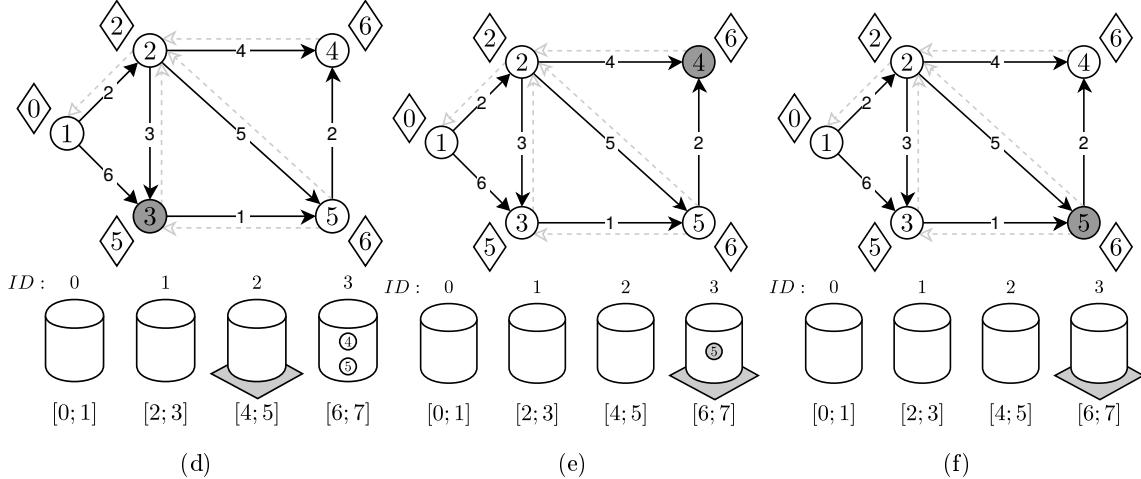
Ostatecznie daje nam to złożoność algorytmu, ograniczoną z góry przez $O(m \cdot b + n \cdot (b + \frac{C}{b}))$. Warto zwrócić uwagę, że dla $b = 1$ (czyli, gdy kubełki mają jednostkowe zakresy) otrzymujemy algorytm Dial, którego złożoność wynosiła $O(m + n \cdot C)$.



Rysunek 2.12: Wykresy zależności $O(m \cdot b + n \cdot (b + \frac{C}{b}))$ dla $C = 10000$ (a) Wykres przedstawia zależność czasu działania algorytmu od ilości węzłów i szerokości kubełków dla grafu rzadkiego ($m = n$). (b) Wykres przedstawia zależność czasu działania algorytmu od ilości węzłów i szerokości kubełków dla grafu pełnego ($m = \frac{n \cdot (n-1)}{2}$). Wyraźnie widzimy, że algorytm zamienia złożoność obliczeniową na pamięciową (i odwrotnie).



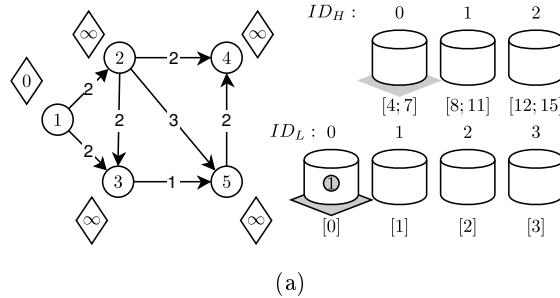
Rysunek 2.13: Działanie algorytmu Dijkstry, opartego na aproksymacji kubełków (a) Początkowa sytuacja w grafie G ($b = 2$). (b) Z pierwszego, przeskanowanego kubełka zostało wyciągnięty węzeł v_1 . W wyniku relaksacji do kubełków $B[1]$ i $B[3]$ odpowiednio zostały włożone węzły: v_2 i v_3 . (c) Po wykonaniu relaksacji $B[0]$ jest pusty - algorytm rozpoczął skanowanie kubełka $B[1]$, po czym wyciągnął z niego wierzchołek v_2 , wykonując następnie relaksacje krawędzi z niego wychodzących.



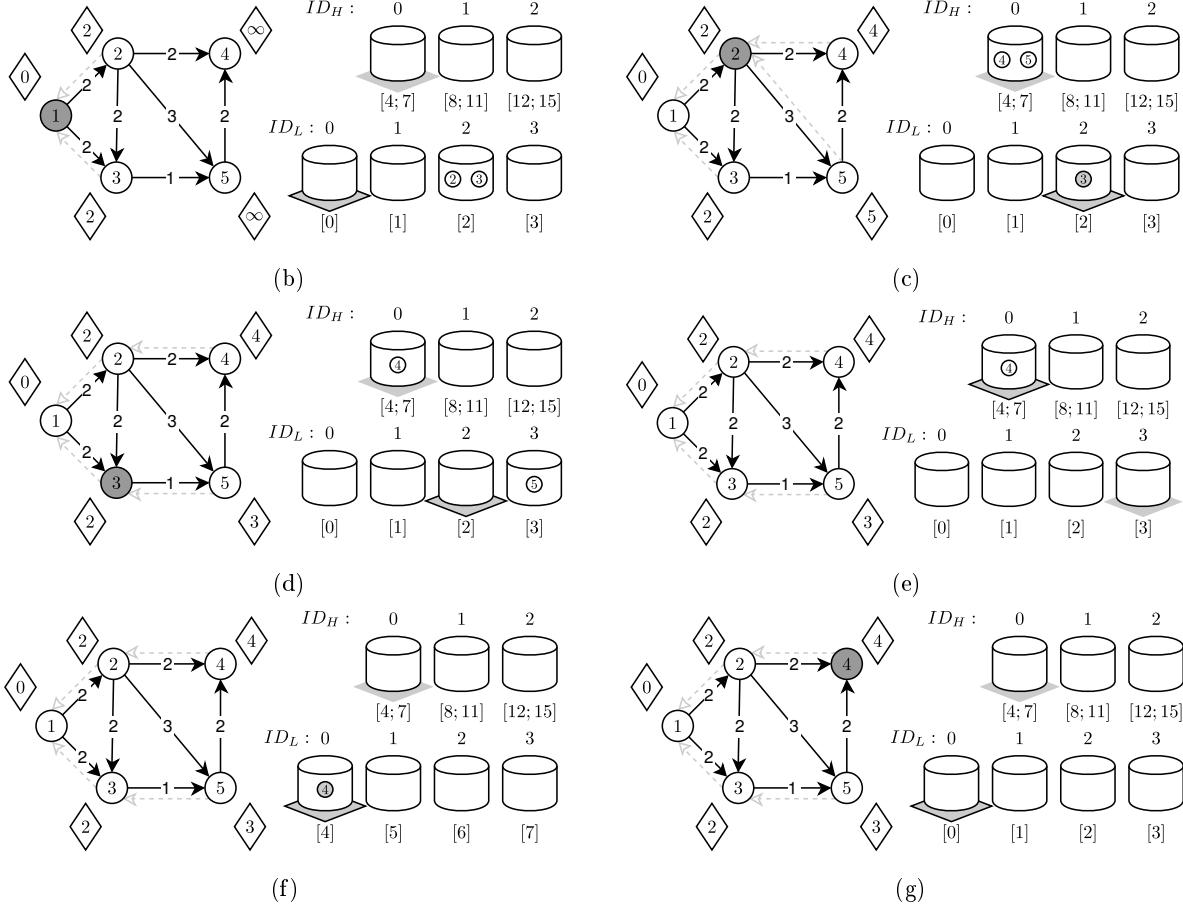
Rysunek 2.13: (d) Po wykonaniu relaksacji dla wszystkich wierzchołków $v \in B[1]$ algorytm rozpoczyna skanowanie następnego kubelka w tablicy. Dla wierzchołka $v_3 \in B[2]$ wykonana jest relaksacja krawędzi e_{35} , a następnie - dla pustego już kubelka - algorytm przechodzi do kolejnego. (e -f) Z kubelka $B[3]$ kolejno są wyciągane węzły. Algorytm przeprowadza ostatnie relaksacje i kończy działanie.

2.5.5 Kubełki wielopoziomowe

Kolejnym algorytmem, któremu poświęcimy naszą uwagę, będzie algorytm oparty na kubełkach wielopoziomowych (w naszym przypadku będą to d poziomy). Dotychczasowe algorytmy, które były oparte o tą strukturę danych, wykorzystywały model jednopoziomowy, w różny sposób podchodząc do jego implementacji. W tym algorytmie, zamiast jednego poziomu kubełków, który wcześniej zwykliśmy oznaczać przez $B[0 \dots k]$, będziemy wykorzystywać oznaczenia $B_L[0 \dots k_L]$ i $B_U[0 \dots k_U]$ odpowiednio dla kubełków niskiego (ang. *Low-level buckets*) oraz wysokiego poziomu (ang. *High-level buckets*). Ideą algorytmu jest stworzenie $\lfloor \frac{n \cdot C}{d} \rfloor$ kubełków wysokiego poziomu, gdzie każdy i -ty taki kubełek ma zakres $[i \cdot d; (i + 1) \cdot d - 1]$ oraz d kubełków niskiego poziomu o zakresach jednostkowych - tym samym suma ich zakresów wynosi dokładnie $[L \cdot d; (L + 1) \cdot d - 1]$ (początkowo $L = 0$). W czasie pracy algorytm nieprzerwanie skanuje kubełki o zakresach jednostkowych od lewej do prawej. Gdy wszystkie wierzchołki z kubełków niskiego poziomu zostaną usunięte, algorytm wyszukuje pierwszy, niepusty kubełek $k \in B_U$, odpowiednio zwiększa L , a następnie przekonwertowuje wszystkie wierzchołki $v \in k$ do odpowiednich kubełków niskiego poziomu, rozpoczynając ich skanowanie od początku. Algorytm wykonuje pracę, dopóki na którejś z list znajdują się nieprzetworzone wierzchołki. Algorytm powstał jako złączenie dwóch poprzednich: w oparciu o kubełki z przepełnieniem oraz algorytmem DKA (szczególnie do tego ostatniego widzimy podobieństwo z sposobie implementacji kubełków wysokiego poziomu).



Rysunek 2.14: **Działanie algorytmu DKD** (a) Sytuacja po zainicjowaniu grafu G ($d = 4$, $n \cdot C = 15$). Do pierwszego kubelka w B_L wstawiony zostaje wierzchołek, będący źródłem.



Rysunek 2.14: (b) Algorytm rozpoczyna skanowanie kubelków niskiego poziomu (aktualny postęp algorytmu dla obu poziomów jest zaznaczony za pomocą rombów pod odpowiednimi kubelkami). Algorytm wykonuje relaksację dla wszystkich krawędzi $e : v_1 \xrightarrow{1} v_i$, w wyniku której do kubelka $K_{\mathbb{L}}[2]$ wstawione zostają węzły v_2 oraz v_3 ($v_2.d = v_3.d = 2$). (c) Algorytm zatrzymuje się na pierwszym, niepustym kubelku, wyjmując z niego dowolny element (na rysunku v_2) oraz przeprowadzając relaksacje jego wychodzących krawędzi (e_{23} , e_{24} i e_{25}). W ich wyniku węzły v_4 oraz v_5 stały się osiągalne ze źródła i zostały wstawione do kubelka $K_{\mathbb{U}}[0]$ ($v_4.d, v_5.d \in K_{\mathbb{U}}[1].range$). (d) W wyniku relaksacji krawędzi, wychodzących z aktualnie badanego węzła v_3 , dla wierzchołka v_5 została zmniejszona odległość od źródła tak, że $v_5.d \leq K_{\mathbb{L}}[d-1].range$. Wierzchołek został z powrotem przepięty do tablicy kubelków niskiego poziomu, do kubelka $K_{\mathbb{L}}[3]$. (e) Z ostatniego, niepustego kubelka został wyjęty węzeł v_5 . Po przeprowadzeniu operacji RELAX wszystkie kubelki $k \in K_{\mathbb{L}}$ są puste. Algorytm kontynuuje skanowanie kubelków wysokiego poziomu, zatrzymując się na pierwszym niepustym. (f) Następuje zmiana zakresów kubelków dolnego poziomu (tak, że zachodzi $[K_{\mathbb{L}}[0].range; K_{\mathbb{L}}[d-1].range] = K_{\mathbb{U}}[0].range$), a następnie wszystkie wierzchołki z kubelka $K_{\mathbb{U}}[0]$ zostają przeniesione do odpowiednich kubelków w tablicy $K_{\mathbb{L}}$. Po przeniesieniu wszystkich węzłów, algorytm rozpoczyna na nowo skanowanie kubelków $K_{\mathbb{L}}[0 \cdots d-1]$. (g) Z kubelka $K_{\mathbb{L}}[0]$ zostaje wyjęty ostatni wierzchołek. Po wykonaniu relaksacji dla krawędzi, wychodzących z v_4 (brak jest takich), wszystkie kubelki są puste. Algorytm, doszedłszy do końca $B_{\mathbb{U}}$, kończy pracę.

Złożoność obliczeniowa - kubełki 2-poziomowe

Analizując złożoność obliczeniową algorytmu Dijkstry, opartego o kubełki wielopoziomowe, podzielimy naszą analizę na dwie części: w pierwszej z nich pokażemy złożoność algorytmu, którego przykład przedstawiono na rysunku 2.14, zaś następnie uogólnimy algorytm na k kubełków.

Algorytm, oparty na kubełkach dwupoziomowych składa się z trzech faz, którymi są: skanowanie górnej części kubełków, przenoszenie wierzchołków do tablicy $B_{\mathbb{L}}[0 \dots d - 1]$ oraz wykonywanie skanowania kubełków niskiego poziomu w celu wykonania relaksacji dla krawędzi, wychodzących z wierzchołków, które tam się znajdują. Kolejno mamy zatem:

- $O(\frac{n \cdot C}{d})$ - jest to czas potrzebny na przeskanowanie kubełków wysokiego poziomu, których jest $\lfloor \frac{n \cdot C}{d} \rfloor$. W najgorszym przypadku będziemy musieli przeskanować je wszystkie.
- Bez względu na to, ile kubełków wysokiego poziomu przeskanowaliśmy, co najwyżej $|V|$ wierzchołków zostanie przeniesionych z $B_{\mathbb{U}}[0 \dots \lfloor \frac{n \cdot C}{d} \rfloor - 1]$ do $B_{\mathbb{L}}[0 \dots d - 1]$, a zatem w najgorszym, możliwym przypadku będziemy zmuszeni do wykonania $|V|$ skanowań kubełków niskiego poziomu, za każdym razem skanując ich d , gdzie na skanowanie jednego kubełka przypada $O(1)$. Zatem łączny kosz tej operacji to $O(n \cdot d)$.
- W omawianym powyżej przypadku algorytm będzie musiał $n = |V|$ razy wykonać zmiany zakresów każdego z $o \cdot d - 1$ kubełków. Wykorzystując fakt, że zakres każdego z kubełków jest opisany w zależności od jednej zmiennej L ($B_{\mathbb{L}}[i].range = [L \cdot d + i; (L + 1) \cdot d + i]$), gdzie L oznacza najmniejszą wartość zakresu aktualnie badanego kubełka wysokiego poziomu, jesteśmy w stanie przeprowadzić taką zmianę w czasie stałym (obliczenia związane z wstawianiem wierzchołka do odpowiedniego kubełka zostaną wykonane każdorazowo w czasie $O(1)$ i są zależne od liczby przemieszczanych z $B_{\mathbb{U}}$ do $B_{\mathbb{L}}$ wierzchołków - maksymalnie n).
- Ostatnią fazą algorytmu, jaką wyodrębniliśmy, była faza relaksacji wierzchołków, a tych jest co najwyżej m - każdy wierzchołek, będący w kubełkach niskiego poziomu, skanujemy dokładnie raz.

Podsumowując powyższe rozważania: algorytm Dijkstry, zbudowany w oparciu o strukturę kubełków dwupoziomowych, działa w czasie nieprzekraczającym $O(\frac{n \cdot C}{d} + n \cdot d + m)$. Tak jak w przypadku, gdy analizowaliśmy ten sam algorytm, oparty na kopach R -arnych, tak samo tutaj zoptymalizowany czas działania algorytmu otrzymamy, gdy doprowadzimy największe wyrażenia, od których zależy ten czas, do równości, czyli do: $\frac{n \cdot C}{d} = n \cdot d$, z którego natychmiastowo wynika $d = \sqrt{C}$.

Warto też zwrócić uwagę na to, że naiwna implementacja, którą zajmowaliśmy się na samym początku to tak naprawdę algorytmy jednopoziomowe, dla których poniższa analiza jest jak najbardziej prawidłowa ($O(\frac{n \cdot C}{1} + n \cdot 1 + m) = O(m + n \cdot C)$). Nasuwa się pytanie, czy takiej samej analizy nie możemy przeprowadzić dla algorytmu Dial, który także jest implementacją jednopoziomową (jego kubełki mają identyczne własności co $B_{\mathbb{L}}$). Odpowiedź oczywiście jest pozytywna.

Niech maksymalna liczba kubełków wysokiego poziomu wynosi $\lfloor \frac{C+1}{d} \rfloor$ i niech dodatkowo $d = \sqrt{C+1}$ ($C+1$ kubełków wymagała implementacja algorytmu Dial). Zauważmy, że w takiej sytuacji mamy tyle samo kubełków na każdym poziomie, a każdy z $\sqrt{C+1}$ ma ten sam zakres, gdzie dla i -tego kubełka wynosi on $[i \cdot d; (i+1) \cdot d - 1]$. Łącznie daje to nam rozpiętość kubełków: $[0; (\sqrt{C+1} + 1) \cdot \sqrt{C+1} - 1] = [0; (C+1) + \sqrt{C+1} - 1]$ (dla przypadku algorytmu Dial wymagany zakres kubełków to $[0; C]$). Wykonując teraz analizę okaże się, że otrzymamy takie samo oszacowanie na złożoność obliczeniową, co dla przypadku, gdy $|B_{\mathbb{U}}| = \lfloor \frac{n \cdot C}{d} \rfloor$, gdzie $d = \sqrt{C}$ (a zatem $O(\frac{n \cdot C}{d} + n \cdot d + m) = O(n \cdot \sqrt{C} + m)$).

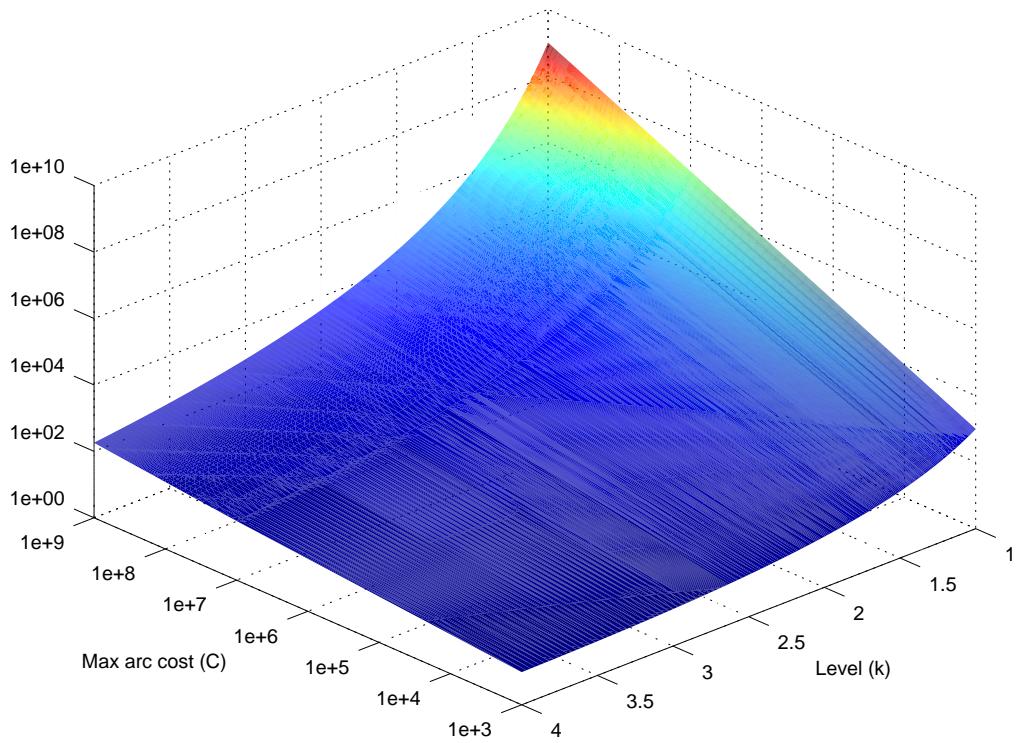
Podstawową naszą obserwacją będzie fakt, że algorytm do pracy wymaga $2 \cdot \sqrt{C+1}$ kubełków. Jak przekonaliśmy się podczas poprzedniej analizy, każdy z kubełków może zostać zbadany co najwyżej raz i taką samą liczbę razy (n) może być wymagane wykonanie skanowania dla wszystkich kubełków niskiego poziomu ($\sqrt{C+1}$). Dodatkowo może być wymagane przeskanowanie każdego kubełka należącego do $K_{\mathbb{U}}$ w celu zakończenia pracy algorytmu ($\sqrt{C+1}$). Daje nam to łącznie złożoność, ograniczoną przez $O(m + n \cdot (1 + \sqrt{C}))$ (relaksacji jak zawsze potrzebujemy wykonać najwyżej m), co daje nam $O(n \cdot \sqrt{C} + m)$.

Złożoność obliczeniowa - kubelki k-poziomowe

Po pokazaniu powyższej prawidłowości dla $d = \sqrt{C + 1}$, analiza k -poziomowej struktury okazuje się być prosta:

Załóżmy, że mamy k poziomów kubelków, gdzie przez B_0 będziemy oznaczać poziom najniższy, zaś przez B_{k-1} - najwyższy. Każdy z poziomów ma $d = \sqrt[k]{C+1}$ kubelków, ponumerowanych od 0 do $d-1$. Zakresy każdego z kubelków rosną analogicznie jak w przypadku dla struktury dwupoziomowej (dla najniższego stopnia mamy $\sqrt{C+1}$ kubelków o zakresach jednostkowych, kubelki, należące do B_1 posiadają zakresy, obejmujące $\sqrt[k]{C+1}$ wartości, zaś na ostatnim, k ’tym poziomie każdy kubelek i ma zakres, do którego wpadają wartości z przedziału $[i \cdot \sqrt[k]{C+1}; (i+1) \cdot \sqrt[k]{C+1} - 1]$) [1, 295].

Analiza problemu dla k -poziomów jest analogiczna jak w przypadku poprzednim, co od razu daje nam złożoność algorytmu $O(m + n \cdot (k + \sqrt[k]{C}))$ i wykorzystuje $\Theta(k \cdot \sqrt[k]{C})$ kubelków (w najgorszym przypadku teraz, nim wierzchołki trafiają do najniższego poziomu, muszą przejść kolejno przez wszystkie poziomy, co może oznaczać konieczność przepięcia każdego z n wierzchołków d razy, zaś skanowanie najniższego poziomu oraz sama relaksacja krawędzi odbywa się analogicznie jak w przypadku dla $k = 2$).



Rysunek 2.15

Rysunek 2.16: Wykresy zależności $O(m + n \cdot (k + \sqrt[k]{C}))$ w zależności od $C \in \{10 \cdot 10^3, \dots, 10 \cdot 10^{11}\}$ (na osi X) i $k \in \{2, 3, 4, 5, 6\}$ (na osi Y). Tak jak w przypadku algorytmu, opierającego się na kubelkach aproksymacyjnych, widzimy, że algorytm zamienia złożoność obliczeniową na pamięciową i odwrotnie.

2.5.6 Kopce pozycyjne

Kopcami pozycyjnymi (ang. *Radix Heap*) nazywamy jednopoziomową strukturę kubełków, których zakresy definiowane są w następujący sposób:

$$B[i].range = \begin{cases} [i; i] & \text{gdy } i = 0 \\ [2^{i-1}; 2^i - 1] & \text{gdy } i > 0 \end{cases} \quad (2.6)$$

Od razu widzimy, że w ten sposób ograniczamy liczbę kubełków do $\lceil \log_2(L) \rceil + 1$, gdzie L to największy zakres kubełków, co w zasadniczy sposób przełoży się na szybkość działania algorytmu (zawsze musielibyśmy skanować wszystkie kubełki - w najgorszym przypadku n razy).

Celowo użyliśmy tutaj zmiennej L , gdyż - tak jak w poprzednich przypadkach - możemy skorzystać z własności, która pozwoli nam ograniczyć liczbę kubełków z $n \cdot C$ do zaledwie $C + 1$, co pozwoli nam na skonstruowanie jeszcze szybszego algorytmu. Najpierw jednak zajmiemy się omówieniem jego podstawowej wersji - z $n \cdot C$ kubełkami.

Pierwsze podejście

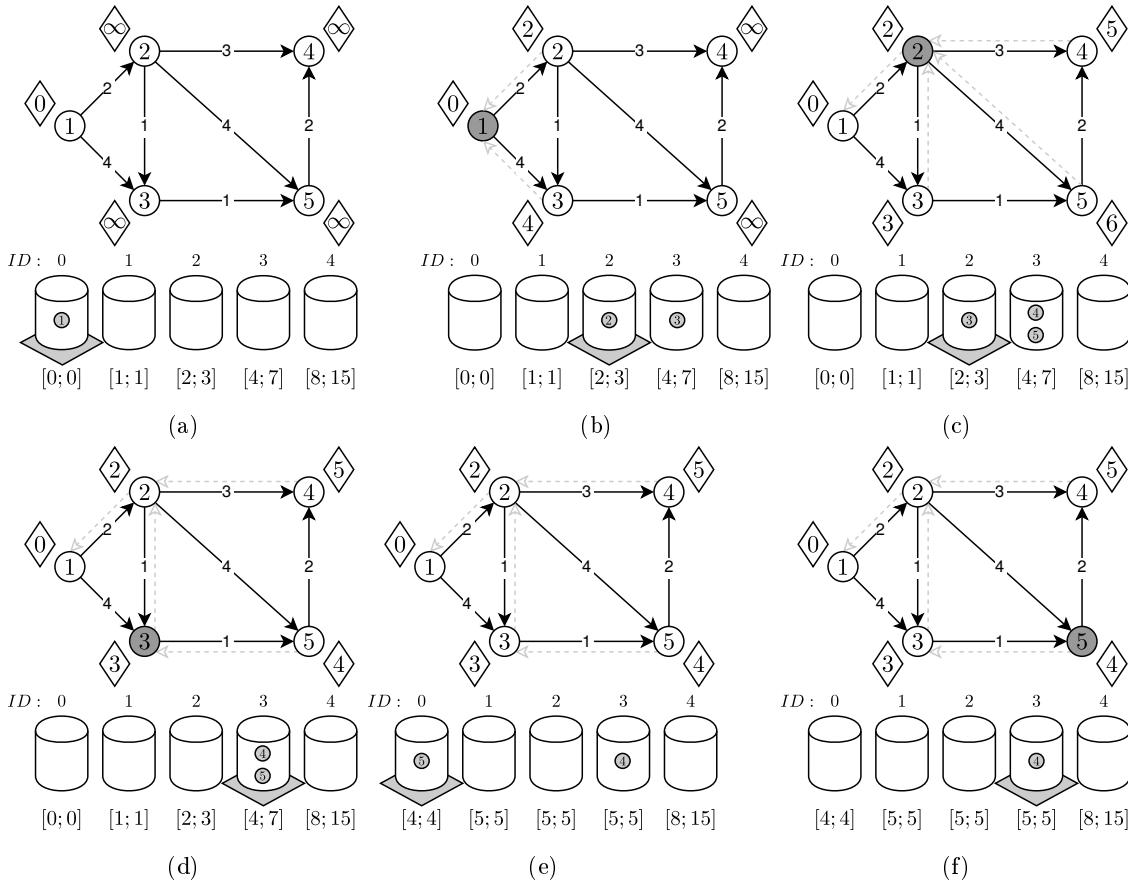
Mamy $\lceil \log_2(n) + \log_2(C) \rceil + 1$ kubełków, którego każdy zakres został zdefiniowany jest tak jak przedstawiono powyżej. Algorytm dąży to tego, by wykonywać relaksacje krawędzi wierzchołków, których odległość od źródła w danej chwili jest najmniejsza (wśród węzłów, które nie zostały jeszcze przetworzone). Aby zapewnić takie właściwości, będziemy dążyć do tego, by w każdym momencie działania algorytmu pierwszy kubełek $B[0]$ zachował swój jednostkowy zakres, najmniejszy spośród wszystkich pozostałych. Z niego na początku będziemy pobierać kolejno wierzchołki do przetworzenia, skanując kubełki w kolejności ich rosnących indeksów. Jako, że dopuszczałyśmy niejednostkowe zakresy (tak samo jak przy kubełkach aproksymacyjnych), a nie chcemy dla każdego z nich wyszukiwać najmniejszego (w sensie odległości od źródła) wierzchołka v (co natychmiast da nam górne ograniczenie na złożoność algorytmu $O(n^2)$ w przypadku, gdy wszystkie wierzchołki znajdują się w jednym kubełku), będziemy rozróżniać w naszym algorytmie trzy sytuacje:

- 1 Algorytm, skanując kubełki od lewej do prawej, napotyka na kubełek i o zakresie jednostkowym $[k; k]$ (taka sytuacja zawsze zachodzi dla pierwszych dwóch kubełków). W takiej sytuacji algorytm wykonuje relaksacje krawędzi, wychodzących ze wszystkich wierzchołków, znajdujących się w kubełku jako, że wszystkie są równoodległe od źródła ($\forall (v_i, v_j) \in K[i] : K[i].range = [k; k] \Rightarrow v_i.d = v_j.d$), usuwa przetworzone wierzchołki z kubełka, a na końcu przechodzi do następnego.
- 2 Wykonując skanowanie od lewej do prawej strony, algorytm napotyka na kubełek i o szerszym zakresie $([2^{i-1}; 2^i - 1] : i > 1)$. W takim wypadku jeśli:
 - w kubełku znajduje się tylko jeden wierzchołek to algorytm wykonuje relaksacje dla wszystkich krawędzi, które z niego wychodzą, usuwając wierzchołek z kubełka lub
 - w kubełku znajduje się więcej niż jeden wierzchołek algorytm zmienia zakresy wszystkich kubełków o indeksie mniejszym od i tak, aby ich sumaryczny zakres wynosił $[v_{min}.d; 2^i - 1]$, gdzie $v_{min}.d = \min\{v \in B[i]\}$ (z faktu, że nigdy nie będziemy przetwarzając wierzchołków o mniejszym $v_{min}.d$, wiemy, że wszystkie kubełki o indeksie mniejszym od i nigdy nie będą już używane, więc wolno nam je wykorzystać). Po wykonaniu tej operacji w kubełku $B[0]$ (o nowym zakresie $[v_{min}.d; v_{min}.d]$) znajdą się wszystkie wierzchołki o takiej samej odległości od źródła jak $v_{min}.d$. Algorytm rozpocznie wtedy skanowanie kubełków od początku, powracając do kroku 1.

Uwagi do algorytmu

W czasie wykonywania zmian zakresów należy podkreślić, że nie możemy dopuścić, by którykolwiek ze zmienionych zakresów $B[j].range : j < i$, gdzie i to aktualnie skanowany kubełek, przekroczył prawą stronę zakresu badanego kubełka, stąd wartość $U = 2^i - 1$ będzie największą wartością, jaką możemy przypisać nowym zakresom (czy to z ich lewej, czy prawej strony). Podczas takiej zmiany szerokość zakresów poszczególnych kubełków nie ulega zmianie. Możemy zatem opisać nowe zakresy każdego z j kubełków jako

$B[j].range = [v_{min}.d + 2^{j-1}; \min\{U, v_{min}.d + 2^j - 1\}]$, wyłączając z tego oczywiście pierwszy kubełek, którego zakres zdefiniujemy jako $[v_{min}.d; v_{min}.d]$ (gdzie $v_{min}.d = \min\{v \in B[i]\}$). Aby usprawnić przeliczanie zakresów kubełków, możemy skorzystać z wcześniejszego faktu i na równi z tablicą $B[0 \dots \lceil \log_2(n \cdot C) \rceil]$ stworzyć tablicę $S[0 \dots \lceil \log_2(n \cdot C) \rceil]$, w której będziemy przechowywać stałe szerokości wszystkich kubełków. W związku z tym, że w algorytmie, opartym o kopce pozycyjne, zakresy kubełków zmieniają się dynamicznie w czasie trwania algorytmu, nie możemy w prosty sposób określić do którego kubełka należy przenieść dany wierzchołek (czy to w wyniku relaksacji prowadzącej do niego krawędzi, czy też zmiany zakresów). Możemy jednak usprawnić ten proces, zauważając, że ilekroć dany wierzchołek v będzie przemieszczał się między kubełkami, zostanie od wstawiony do kubełka o indeksie mniejszym, niż tej, w którym się znajdował (wynika to z własności relaksacji, o sposobie konstrukcji nowych zakresów już nie wspominamy). Zauważając, że każdy kolejny kubełek posiada dwukrotnie szerszy zakres, możemy za każdym razem rozpoczynać proces szukania nowego kubełka dla danego wierzchołka v (przyjmijmy, że znajduje się w kubełku k) od kubełka o indeksie k , dla którego mamy największą szansę, że wspomniany wierzchołek trafi (lub w nim pozostanie).



Rysunek 2.17: **Działanie algorytmu Dijkstry z wykorzystaniem kopków pozycyjnych** (a) Sytuacja po zainicjowaniu grafu G ($n = 5$; $C = 4$; $\lceil \log_2(n \cdot C) \rceil + 1 = 5$) (b) W wyniku relaksacji krawędzi, wychodzących z węzła v_1 , wstawiono do kubełków wierzchołki: v_2 oraz v_3 . (c) Z pierwszego, niepstego kubełka zostaje wyjęty węzeł v_2 . Po relaksacji jego krawędzi zostają do kubełków dodane wierzchołki v_4 i v_5 , a węzeł v_3 zostaje przeniesiony do aktualnie skanowanego kubełka. (d) Następuje relaksacja wszystkich krawędzi, prowadzących do sąsiadów wierzchołka v_3 . Wierzchołek zostaje usunięty, a algorytm idzie do następnego kubełka. (e) W wyniku napotkania kubełka i o niejednostkowym zakresie, w którym znajdująły się więcej niż 1 wierzchołek, algorytm wyszukał najmniejszy węzeł, należący do tego kubełka ($v_{min}.d = 4$), po czym zmienił zakres wszystkich kubełków $B[0 \dots i - 1]$, rozpoczynając skanowanie od początku. (f) Algorytm wykonał relaksacje, wyciągając z kubełka $B[0]$ węzeł v_5 . Po wyszukiwaniu następnego, niepstego kubełka i usunięciu ostatniego wierzchołka, algorytm skończył działanie.

Złożoność obliczeniowa

Algorytm działa w czasie $O(m + n \cdot \log(n \cdot C))$. Na tą złożoność składają się następujące elementy, które możemy wyróżnić podczas działania algorytmu:

- Aby zakończyć działanie, algorytm musi - w najgorszym przypadku - przeskanować wszystkie kubelki w tablicy $B[0 \dots \lceil \log_2(n \cdot C) \rceil]$, w czasie $O(1)$ dla każdego z nich. Daje nam to łącznie czas na poziomie $O(K)$, gdzie za K przyjmijmy liczbę kubelków ($K = \lceil \log_2(n \cdot C) \rceil + 1$).
- W przypadku napotkania takiej sytuacji, w której algorytm rozpocznie zmienianie zakresów kubelków, będzie musiał w najgorszym przypadku wykonać K takich zmian (każda zmiana zakresu jest wykonywana w stałym czasie).
- W każdym takim przypadku, nim algorytm zacznie zmieniać zakresy kubelków, będzie musiał najpierw odnaleźć v_{min} w aktualnie skanowanym, i 'tym kubelku. Czas, jaki jest potrzebny by odnaleźć taki element, jest proporcjonalny do ilości wierzchołków w kubelku i (założymy, że jest ich d). Dla najgorszego przypadku będzie to n . Dodatkowo wiemy, że po znalezieniu takiego wierzchołka i zmienieniu zakresów dla wszystkich kubelków, każdy węzeł z kubelka i będzie musiał zostać przeniesiony do któregoś z pozostałych, którego indeks jest mniejszy od i . Jak już wcześniej wspominaliśmy, dla każdego takiego węzła konieczne jest znalezienie jego nowego kubelka, zaś pesymistyczny czas tej operacji to $O(K)$. Z tego wynika, że czas, potrzebny do przemieszczenia wszystkich wierzchołków z kubelka i , wynosi $O(d \cdot K)$, co pozwala nam zaniedbać czas, potrzebny do wyszukania v_{min} .
- Algorytm wykonuje m relaksacji i każdy wierzchołek aktualizowany jest tylko raz, co daje nam łącznie $O(m + n \cdot K)$ czasu, jaki jest potrzebny na relaksację m krawędzi oraz przeniesienie maksymalnie n węzłów (każde wykonane w czasie $O(K)$).

Podsumowując, daje nam to następujące czasy wykonywanych operacji:

- czas potrzebny na procedurę zmiany zakresów i przeniesienie wszystkich wierzchołków: $O(K)$ (razem $O(n \cdot K)$),
- czas na znalezienie najmniejszego wierzchołka w kubelku i : $O(d)$, gdzie $d = |v \in B[i]|$ (razem $O(d \cdot K)$, zdominowane przed $O(n \cdot K)$),
- czas wykonywania wszystkich relaksacji w grafie: $O(m)$,
- czas aktualizacji wszystkich wierzchołków: $O(n \cdot K)$ (dla pojedynczego wierzchołka $O(K)$)

Sumując wszystko razem, otrzymujemy, podaną na wstępie, złożoność: $O(m + n \cdot K)$, gdzie, po zastąpieniu K , otrzymujemy $O(m + n \cdot \log(n \cdot C))$.

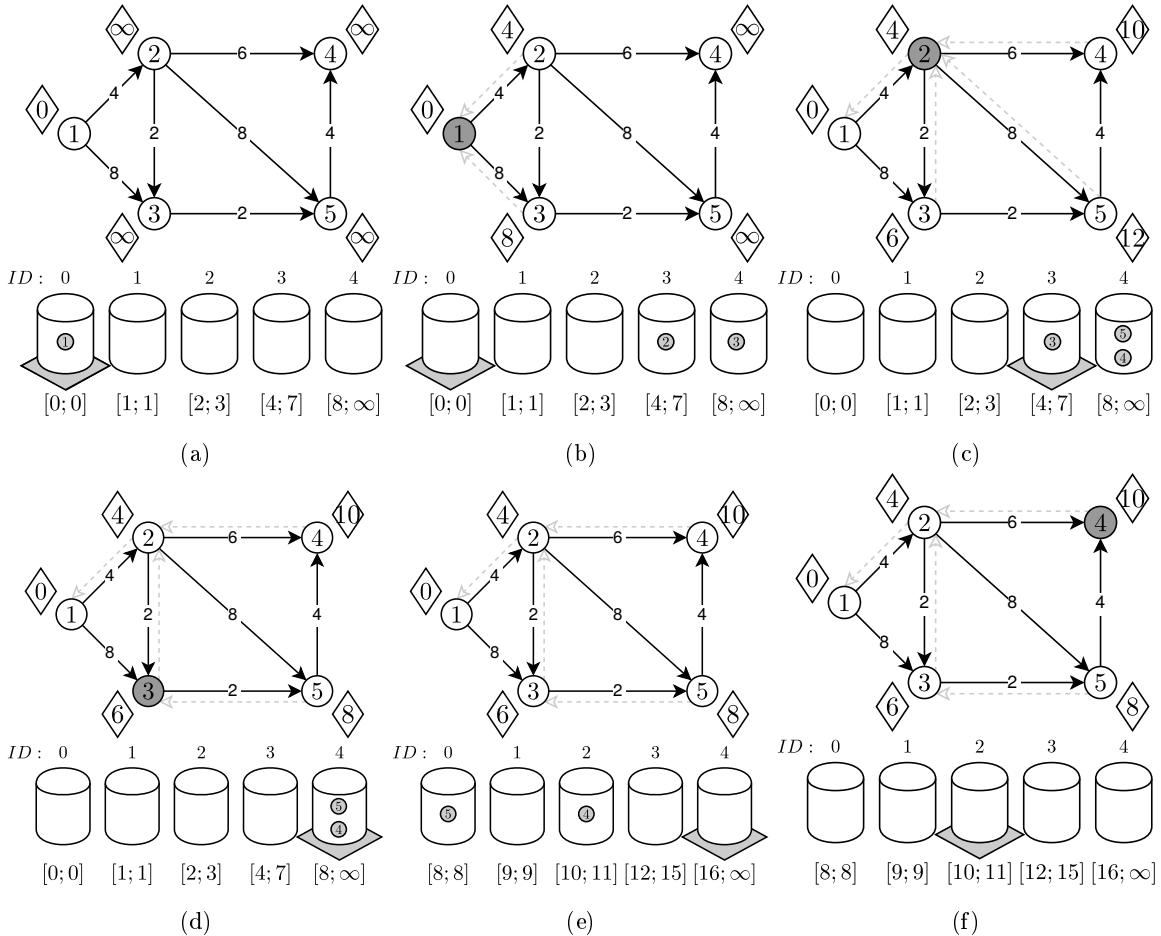
Drugie podejście

Chcemy ograniczyć liczbę potrzebnych nam kubelków do $\lceil \log_2(C) \rceil + 1$. By osiągnąć nasz cel wykorzystamy pomysł, który omówiliśmy, podczas opisywania implementacji algorytmu *DKM* (opartego na kubelkach z przepełnieniem). Przypomnijmy, że zgodnie ze wspomnianą implementacją mieliśmy do dyspozycji $a < C + 1$ kubelków, gdzie $a + 1$ 'y miał tą własność, że jego teoretyczny zakres wynosił $[k; \infty]$, a pozostałe zakresy kubelków były jednostkowe. W przypadku kopów pozycyjnych zdefiniujemy je następująco:

$$B[i].range = \begin{cases} [i; i] & \text{gdy } i = 0 \\ [2^{i-1}; 2^i - 1] & \text{gdy } i \in \{1, \dots, C\} \\ [2^{i-1}; \infty] & \text{gdy } i = C + 1 \end{cases} \quad (2.7)$$

Algorytm wykonywania wszystkich operacji pozostaje ten sam - w przypadku napotkania przez algorytm kubelka $B[C]$ traktuje go on jako zwykły kubelek o niejednostkowym zakresie z - najprawdopodobniej - wielema wierzchołkami w środku. Dla tego przypadku opisywaliśmy już kroki, jakie podejmie algorytm z tą tylko różnicą, że nie znamy górnego zakresu skanowanego kubelka (gdyż jest on teoretycznie nieskończony). W taka sytuacji zamiast zmieniać zakresy wszystkich kubelków (poza aktualnie skanowanym, ostatnim w tablicy

$K[0 \dots C]$) według wzoru: $B[j].range = [v_{min}.d + 2^{j-1}; \min\{U, v_{min}.d + 2^j - 1\}]$ (patrz 2.5.6, będziemy korzystać z nieznacznie zmodyfikowanej jego wersji, bez górnego ograniczenia (którego nie ma - wynosi ∞): $B[j].range = [v_{min}.d + 2^{j-1}; v_{min}.d + 2^j - 1]$). Dodatkowo, po przesunięciu zakresów wszystkich kubelków, będziemy musieli zaktualizować dolną granicę aktualnie badanego kubelka $B[C]$ tak, aby jego zakres był o 1 większy, od maksymalnie ustalonego (tak, aby zakresy nie nachodziły na siebie, a do przepełnienia trafiały tylko te wierzchołki v , których $v.d$ jest za duże, by trafić do któregoś z pozostałych kubelków). Zwrócmy uwagę, że zastosowanie w naszej implementacji kubelków z przepełnieniem nie ogranicza nas tylko do $C + 1$ kubelków - możemy ich zastosować dowolną ilość taką, że $a < C + 1$. Poniżej przedstawiamy przykład, dla którego mamy mniej kubelków niż C przy nadal poprawnie działającym algorytmie.



Rysunek 2.18: **Działanie algorytmu Dijkstry z wykorzystaniem kopków pozycyjnych** (a) Sytuacja po zainicjowaniu grafu G ($C = 8, a = 4$). (b) Sytuacja po zdjęciu węzła v_1 z kubelka i wykonaniu relaksacji. (c) Po relaksacji wszystkich krawędzi, wychodzących z wierzchołka v_2 , który został wyjęty z następnego, niepustego kubelka, węzeł v_3 zmniejszył swoją odległość od źródła na tyle, by został przepięty z ostatniego kubelka do tego, który aktualnie jest badany. (d) Algorytm usunął z kubelka $B[4]$ ostatni wierzchołek oraz wykonał relaksacje jego krawędzi. Następnym, niepustym kubelkiem jest kubelek z nieograniczonym zakresem. (e) Algorytm wyszukuje w kubelku $B[a+1]$ najmniejszy wierzchołek v_{min} , a następnie zmienia zakresy wszystkich wcześniejszych kubelków. Zakres pierwszego kubelka z definicji jest ustawiany na $[v_{min}.d; v_{min}.d]$, zaś dolna granica zakresu aktualnie badanego kubelka jest ustawiana na $B[a].range.max + 1$. Z kubelka $B[5]$ przepinane są wszystkie wierzchołki v , których $v.d \notin B[5]$. (f) Po przepięciu wszystkich węzłów algorytm rozpoczyna skanowanie kubelków od pierwszego z nich. Po usunięciu jeszcze dwóch wierzchołków wszystkie kubelki są puste, a algorytm kończy pracę.

Uwagi do algorytmu

Tak samo jak wykorzystaliśmy ideę algorytmu *DKM*, implementując kopce pozycyjne z przepełnieniem, podobnie możemy skonstruować algorytm w oparciu o strukturę dwupoziomową, zastosowaną w algorytmie *DKD* (ang. *Dijkstra with Double-Level Buckets*). W tym przypadku, dla $\lceil \log_2(C) \rceil + 1$ kubełków naszym celem jest stworzenie dwóch identycznych, jednopoziomowych struktur o tyłu właśnie kubełkach, które na przemianie będzie wykorzystywał nasz algorytm. Jak pamiętamy, w wyniku relaksacji dowolnej krawędzi, dla dowolnego wierzchołka w grafie nigdy nie otrzymamy takiego węzła, którego odległość od źródła, w porównaniu z tą, którą ma skanowany węzeł, byłaby większa od tej ostatniej o więcej niż C . Tym samym każdy wierzchołek, który wstawialiśmy do kubełków czy go przesuwaliśmy między nimi, nie „okrągał” aktualnie badanego kubełka (patrz 2.11). Jak wyraźnie widzieliśmy w algorytmie Dial, który wykorzystywał jednopoziomową strukturę z kubełkami, często zdarzały się w nim sytuacje, w których wstawialiśmy wierzchołki do kubełków, będących „za” aktualnie skanowanym kubełkiem (to jest jeżeli skanowaliśmy kubełek o indeksie $i > 0$ to wierzchołek był wstawiany do kubełka o indeksie $0 \leq j < i$). W implementacji algorytmu opartego na kopcach pozycyjnych naszą kluczową obserwacją było to, że przy skanowaniu k' tego kubełka wszystkie wcześniejsze (o indeksach $0 \leq k' < k$) nie były już potrzebne i mogliśmy zmienić ich zakresy. Aby uniknąć zdarzenia, polegającego na wstawieniu wierzchołka do jednego z tych kubełków, rozmieszczałyśmy kubełki na dwóch poziomach, gdzie łączny zakres każdego z nich wynosi C . Teraz, jeżeli wydarzy się opisywana przez nas sytuacja, zamiast wstawać węzeł do kubełka, znajdującego się „za” aktualnie badanym kubełkiem, będziemy taki wierzchołek wstawać do drugiego poziomu kubełków (traktując go jako naturalne przedłużenie pierwszego, lecz zachowując dla niego zakresy takie same jak gdyby był on osobną strukturą). Wreszcie - jeżeli skończymy proces skanowania pierwszego poziomu kubełków (który, poza omawianą sytuacją, działa identycznie jak dla omawianych wcześniej algorytmów, opartych o kopce pozycyjne) - to następnym naszym krokiem będzie „podmienienie” poziomów i rozpoczęcie wykonywania tej samej procedury skanowania dla drugiego poziomu. Pierwszy poziom kubełków wraca wtedy do stanu początkowego (z tą różnicą, że teraz najmniejszy zakres powinien być większy o $2 \cdot C$ tak, aby stanowił naturalną kontynuację aktualnie skanowanego poziomu) i cały proces zaczyna się od początku. Czas działania takiego algorytmu jest asymptotycznie taki sam, jak złożoność obliczeniowa poprzedniego podejścia, którą udowodnimy w następnym podrozdziale.

Złożoność obliczeniowa

Analiza złożoności obliczeniowej dla wersji algorytmu, wykorzystującego kopce pozycyjne z ograniczoną ilością kubełków jest dokładnie taka sama, jak dla wersji, którą analizowaliśmy poprzednio. Należy jednak się zastanowić nad możliwością dalszego zmniejszania ilości kubełków, jako że złożoność algorytmu, który przedstawiliśmy, wynosi $O(m + n \cdot K)$, gdzie K to liczba kubełków. Dla wersji, w której mamy $\lceil \log_2(C) \rceil + 1$ kubełków jest to oczywiście $O(m + n \cdot \log(C))$. Co się stanie, jeżeli weźmiemy $K < \lceil \log_2(C) \rceil + 1$?

Przypomnijmy sobie, że dla algorytmu, który wykorzystywał kubełki z przepełnieniem, wyznaczyliśmy największą możliwą ilość iteracji, jakie algorytm musi wykonać, by zakończyć działanie (przez iterację rozumiemy przeskanowanie wszystkich kubełków $B[0 \dots K-1]$). Wykazałyśmy, że w najgorszym możliwym przypadku takich iteracji jest $O(\frac{n \cdot C}{a})$, gdzie a było mniejsze od $C+1$. Zauważmy, że pierwszym elementem, którego złożoność analizowaliśmy przy badaniu algorytmu, opartego o kopce pozycyjne, była operacja zmiany zakresów, gdzie:

- czas potrzebny na procedurę zmiany zakresów i przeniesienie wszystkich wierzchołków: $O(K)$ (razem $O(n \cdot K)$).

Dla $\lceil \log_2(C) \rceil + 1$ kubełków mieliśmy pewność, że podczas każdej iteracji algorytmu zostanie z nich usunięty co najmniej jeden węzeł (dla węzłów v i u różnica $v.d$ i $u.d$ nie mogła być większa niż C). Dla mniejszej ilości kubełków taka własność nie zachodzi i stąd - zamiast otrzymać dobrze oszacowanie $O(n \cdot K)$ - otrzymujemy $O(o \cdot K)$, gdzie $o > n$ w związku z czym rośnie nam także złożoność obliczeniowa całego algorytmu (dokładnie $o = O(\frac{n \cdot C}{2^K})$, gdzie dla $K = \lceil \log_2(C) \rceil + 1$ mamy $o = O(n)$ - szacowanie wynika z tego, że w najgorszym możliwym przypadku w każdej następnej iteracji zakresy kubełków będą dokładnie o 2^K większe - żaden zakres nie zostanie pominięty, a suma zakresów kubełków ze wszystkich iteracji będzie tworzyła partycję zakresu $[0 \dots n \cdot C]$).

Inne algorytmy

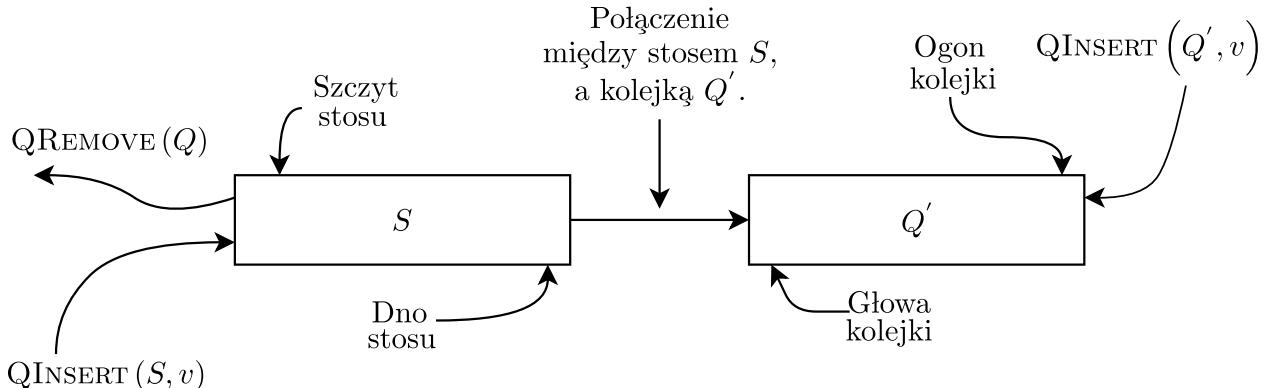
W poprzednim rozdziale skupiliśmy się głównie na algorytmie Dijkstry i jego różnych implementacjach, wprowadzając też po drodze algorytmy, które stanowiły dla nas tło do dalszych rozważań (algorytm sortowania topologicznego, czy Bellmana-Forda). Rozpoczęliśmy od algorytmów o kwadratowej złożoności obliczeniowej ($O(n^2)$), kończąc na implementacji kopców pozycyjnych, dla których czas działania w najgorszym przypadku wynosił $O(m + n \cdot \log(C))$. W tym rozdziale będziemy chcieli zaprezentować kilka mniej znanych algorytmów, rozwiązujących problem najkrótszych ścieżek - między innymi algorytmy zaproponowane przez U.Pape'a (algorytm *PAP*) oraz Stefana Pallottino (modyfikacja poprzedniego algorytmu). Na końcu przedstawimy algorytm progowy, który jest algorymem typu *Label-Correcting*.

3.1 Kombinacje struktur

Do konstrukcji pierwszych dwóch algorytmów wykorzystamy połączenia dwóch różnych typów abstrakcyjnych struktur danych, którymi będą kolejki: *FIFO* (ang. *First In First Out*) oraz *LIFO* (ang. *Last In First Out*). W implementacji pierwszej z nich oryginalnie był wykorzystany stos, zaś drugą będziemy reprezentować przez zwykłą listę podwójnie dowiązaną.

3.1.1 Algorytm przyrostowy

Pseudokod algorytmu przyrostowego (ang. *Graph Growth Algorithm*) wygląda identycznie jak ten, który przedstawiliśmy dla generycznego algorytmu Dijkstry. Różnica między tymi dwoma implementacjami wynika ze specyficznej struktury, jaka została zastosowana w tym przypadku, a której przykład znajduje się poniżej:



Rysunek 3.1: Struktura, wykorzystywana w algorytmie *PAP*

W zastosowanym rozwiązaniu wyróżniamy dwie struktury, z których pierwsza jest implementacją kolejki *FIFO* (stos), druga zaś to zwykła kolejka typu *LIFO*. Obie są połączone ze sobą, co oznacza, że za ostatnim elementem, należącym do stosu S , znajduje się pierwszy wierzchołek w kolejce Q' . Dla takiej struktury wyróżniamy następujące operacje:

- $\text{QREMOVE}(S)$ - usuwa ze szczytu stosu (lub początku kolejki Q' , gdy stos jest pusty) wierzchołek v ,
- $\text{QINSERT}(\cdot, v)$ - gdy wierzchołka v nie ma jeszcze ani na stosie S , ani w kolejce Q' , w zależności od właściwości danego wierzchołka wykonujemy jedną z możliwych operacji:
 - $\text{QINSERT}(S, v)$ - jeżeli wierzchołek, w momencie wstawiania, jest osiągalny ze źródła ($v.d < \infty$) to jest on wstawiany na szczyt stosu S ,
 - $\text{QINSERT}(Q, v)$ - w przeciwnym przypadku (gdy węzeł jest nieosiągalny - $v.d = \infty$) wierzchołek wstawiany jest na koniec kolejki Q' .

oraz, typową dla takiej struktury, operację $\text{IS-EMPTY}(Q)$, gdzie poprzez Q oznaczamy całą, powyżej opisaną strukturę (jest ona pusta, gdy zarówno w S jak i Q' nie ma wierzchołków). Dodatkowo chcemy, by pierwsza ze struktur (ta, z której ściągamy elementy) miała własności kolejki priorytetowej.

Uwagi do algorytmu

Aby efektywnie zaimplementować kolejkę priorytetową, wykorzystując do tego stos, będziemy potrzebować albo rekurencyjnie wyciągać elementy ze stosu, aż do momentu, gdy natrafimy na odpowiednie miejsce, w które włożymy nowy (bądź aktualizowany) element, albo możemy wykorzystać do tego drugi stos, przemieszczając elementy pierwszego stosu na drugi, porównując przy okazji każdy taki element z tym, który chcemy na stos wstawić (niech będzie to element v). W momencie, gdy z pierwszego stosu wyciągniemy taki element u , dla którego $u.v \geq v.d$, wstawiamy na stos wierzchołek v , a następnie z powrotem przekładamy elementy z pomocniczego stosu na ten właściwy. Pierwotna kolejność pozostałych elementów nie ulega zmianie. W przypadku modyfikacji wierzchołka v już będącego na stosie przekładamy elementy ze stosu do momentu, w którym nie wyciągniemy interesującego nas węzła. Następnie przekładamy z powrotem elementy, które dopiero co włożyliśmy na drugi stos, do momentu, w którym na szczytzie tego stosu nie znajdzie się wierzchołek u , którego $u.d \leq v.d$ (lub $u.d < v.d$ w zależności co chcemy osiągnąć). Po wystąpieniu takiej sytuacji na oryginalny stos wstawiamy zaktualizowany element, a następnie kontynuujemy przerzucanie wierzchołków z tymczasowego stosu, dopóki ten nie zostanie pusty, a na szczytzie tego pierwszego z powrotem nie znajdzie się element, który był tam uprzednio (chyba, że zaktualizowany element okaże się być teraz najmniejszym elementem na stosie). Należy jednak zwrócić uwagę, że w oryginalnej implementacji nie mamy wsparcia dla operacji aktualizacji wierzchołka w kolejce, więc własność kolejki priorytetowej dla stosu będziemy musieli całkowicie odbudowywać przy operacji $\text{QREMOVE}(S)$.

Złożoność obliczeniowa

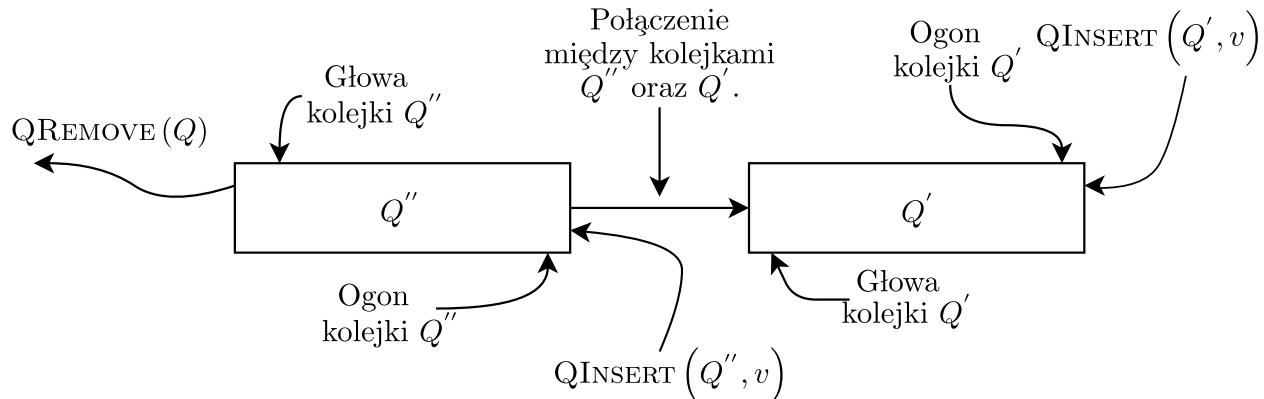
Jak w każdym przypadku, gdzie wykorzystywaliśmy kolejki priorytetowe od zwracania wierzchołków v takich, że $v.d = \min\{u \in Q\}$, także tutaj nasza analiza czasu działania algorytmu opierać się będzie na analizie kosztów, związanych z kosztami wykonywania operacji na takiej kolejce (wzór 2.4.1), a zatem:

- w każdym kroku algorytm trwale usuwa jeden wierzchołek v z kolejki, którego $v.d = \delta(s, v)$ (jak udowodniliśmy w 2.2.4).
- Wyszukanie na stosie takiego elementu v , by $v.d = \min\{u \in Q\}$ w najgorszym przypadku zajmie $O(n)$ (gdzie przy wykorzystaniu dwóch stosów sprowadzi się to do wyciągnięcia wszystkich elementów z pierwszego, zapamiętania najmniejszego podczas wstawiania ich na drugi stos, a następnie - przy operacji odwrotnej - wstawić na pierwszy stos, wyciągając kolejno elementy ze stosu tymczasowego, wszystkie wierzchołki, poza tym najmniejszym).
- Algorytm wykona skanowanie każdego wierzchołka dokładnie jeden raz, więc ilość relaksacji, jakie wykona, będzie równa m .

Da nam to złożoność algorytmu PAP na poziomie $O(m + n^2)$.

3.1.2 Algorytm przyrostowy z dwoma kolejkami

Konsekwencją chęci usprawnienia poprzedniego algorytmu jest modyfikacja jego słabego punktu - stosu, na którym wymusiliśmy własności kolejki priorytetowej. Algorytm, oparty na dwóch kolejkach (ang. *Graph Growth Algorithm with Two Queues*) został opracowany przez Stefana Pallottino jako modyfikacja pierwszego algorytmu, wprowadzająca w miejsce stosu S drugą kolejkę:

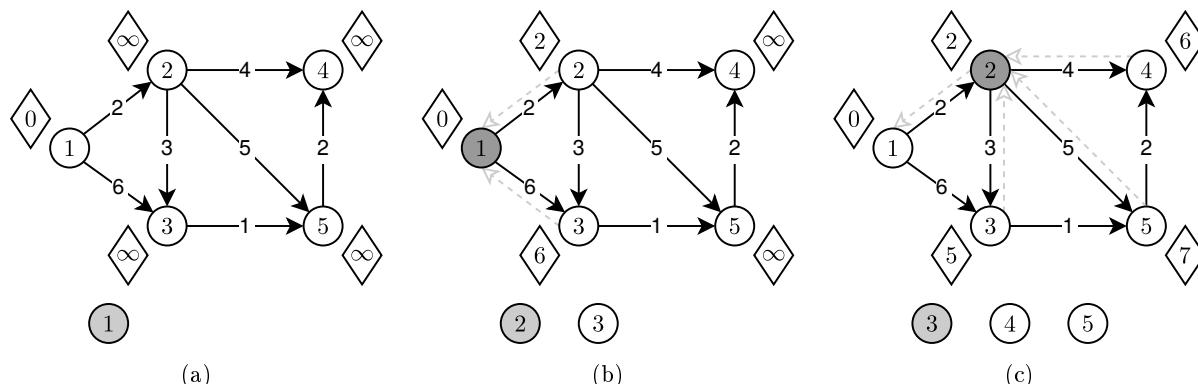


Rysunek 3.2: Struktura, wykorzystywana w algorytmie TQQ

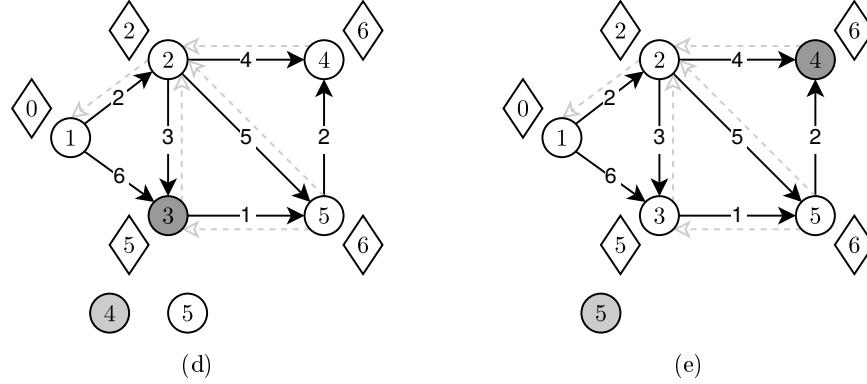
Widzimy, że jedyna modyfikacja, jaką została dokonana w oryginalnej strukturze to przedefiniowanie operacji wstawiania elementów do struktury, która teraz wygląda w następujący sposób:

- $\text{QINSERT}(\cdot, v)$ - gdy wierzchołka v nie ma w żadnej z kolejek Q' i Q'' , w zależności od właściwości danego wierzchołka wykonujemy jedną z możliwych operacji:
 - $\text{QINSERT}(S, v)$ - jeżeli wierzchołek, w momencie wstawiania, jest osiągalny ze źródła ($v.d < \infty$) to jest on wstawiany na koniec kolejki Q'' ,

Reszta operacji nie ulega zmianie, zaś analiza złożoności obliczeniowej pozostaje dokładnie taka sama (z wyjątkiem mniej kosztownej operacji wyszukiwania minimum w kolejce priorytetowej).



Rysunek 3.3: Działanie algorytmu z dwoma kolejkami (a) Algorytm rozpoczyna pracę ze źródłem w wierzchołku v_1 . (b) W wyniku relaksacji jego krawędzi wychodzących na koniec kolejki Q'' kolejno trafiają wierzchołki v_2 i v_3 . (c) Algorytm pobiera element ze stosu, wykonując dla krawędzi e_{23} , e_{24} , e_{25} operację RELAX.



Rysunek 3.3: (d-e) Algorytm kontynuuje działanie według tego samego schematu, kończąc pracę w momencie, gdy na żadnej z kolejek Q'' i Q' nie ma już elementów.

3.2 Algorytm progowy

Ostatnim algorytmem, jaki omówimy, będzie algorytm progowy (ang. *Threshold algorithm*). Pojęcie progu pojawia się w wielu dziedzinach informatyki (na przykład w grafice komputerowej), oznaczając graniczną wartość której dla konkretnego zbioru danych, będącego poniżej/powyżej wartości progowej, ma zajść konkretna operacja. Nie inaczej jest w przypadku algorytmu wyszukiwania najkrótszych ścieżek. Wartością progową tutaj, jak można się domyślić, będzie odległość wierzchołków od źródła w czasie pracy algorytmu. Wszystkie krawędzie, których odległość będzie większa od zadanej wartości progowej, będą trafiały do zbioru wierzchołków, których badaniem zajmiemy się dopiero wówczas, gdy wykonamy wszystkie możliwe operacje dla pozostałych elementów. Gdy taka sytuacja nastąpi, wartość graniczna ulegnie stosownemu zwiększeniu, tym samym odległość pewnej ilości wierzchołków od źródła stanie się mniejsza niż nowo ustalony próg. Następnie algorytm wykona te same operacje, które wykonał dla wcześniejszego zbioru, kończąc dopiero pracę, gdy podniesie wartość progową na tyle wysoko, by zbiór wierzchołków, których odległość od źródła przekracza tą wartość, był pusty, podobnie jak zbiór pozostałych wierzchołków, jeszcze nie przeskanowanych.

Algorithm 13: THR (G, s)

```

1 begin
2   INIT-GRAF ( $G, s$ )
3    $t \leftarrow$  wylicz wartość progową
4    $L \leftarrow \{s\}$                                /* Lista, zawierająca wierzchołki  $v : v.d < t$  */
5    $\bar{L} \leftarrow \emptyset$                       /* Lista, zawierająca pozostałe wierzchołki */
6   while Lista  $L$  nie pusta do
7     while Lista  $L$  nie pusta do
8       Usuń wierzchołek  $v_i$  z końca listy  $L$ 
9       foreach  $e_{ij} : v_i \xrightarrow{1} v_j$  do
10        RELAX ( $v_i, v_j$ )
11        if  $v_j \notin \{L, \bar{L}\}$  then
12          Wstaw węzeł  $v_j$  na koniec  $L$ , jeśli  $v_j.d < t$ . W przeciwnym przypadku wstaw na
13          koniec  $\bar{L}$ 
14        else if Po relaksacji  $v_j.d < t$  i  $v_j \in \bar{L}$  then
15          Przenieś węzeł  $v_j$  na koniec listy  $L$ 
16         $t \leftarrow t + \min \{v.d : v \in \bar{L}\}$ 
17        Przepnij wszystkie  $v : v.d < t$  do  $L$ 

```

Złożoność obliczeniowa

W najgorszym możliwym przypadku wartość progowa na samym początku zostanie ustawiona na tyle wysoko, by wszystkie wierzchołki trafiły do listy L (patrz 13). W takim przypadku otrzymujemy natychmiastowo złożoność $O(m \cdot n)$, gdyż nie jesteśmy w stanie nic powiedzieć o kolejności skanowania wierzchołków (za każdym razem wybieramy po prostu ostatni z listy, gdzie $|L| = n$). Założymy jednak, że mamy daną wartość progową t i niech funkcja $L(i)$ oznacza zbiór wierzchołków, jakie znajdują się na liście L po i -tym zwiększeniu wartości progowej.

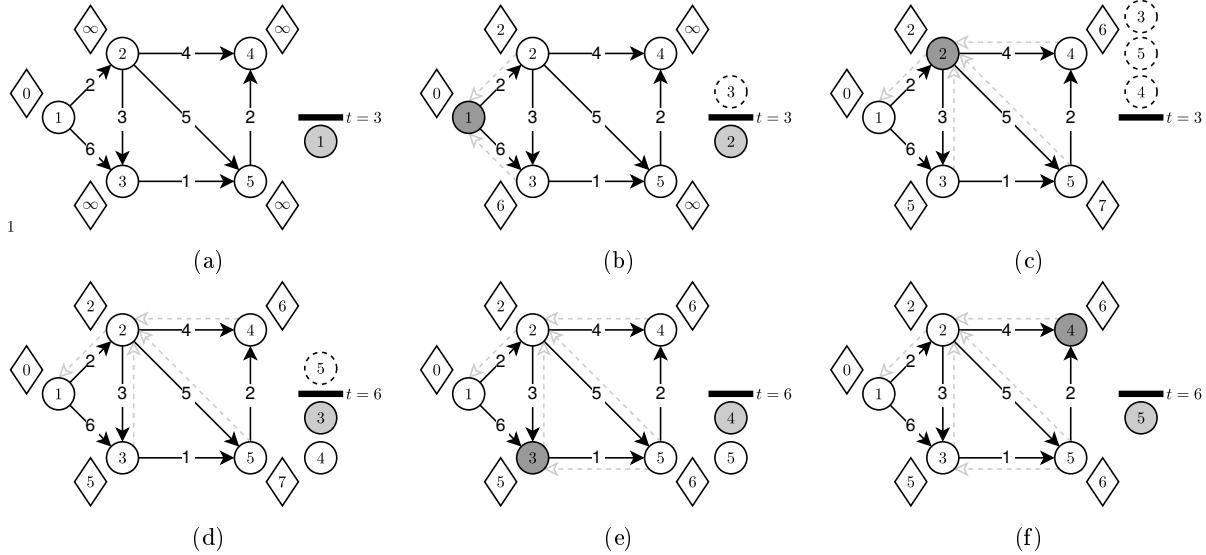
- W najgorszym, możliwym przypadku największą odległośćą węzła od źródła będzie $(n - 1) \cdot C$. Znając wartość progową t , oraz zakładając, że za każdym razem w zbiorze wierzchołków \bar{L} najmniejszy wierzchołek v_{min} to taki, dla którego $v_{min}.d = t + 1$, możemy wyznaczyć liczbę iteracji, jaką w takim przypadku będzie musiał wykonać algorytm, nim skończy działanie. Takich iteracji zostanie wykonanych co najwyżej $O\left(\frac{n \cdot C}{t}\right)$ (podobną analizę przeprowadziliśmy dla algorytmu, wykorzystującego kubelki z przepelnieniem, gdzie ostatni kubek można porównać do zbioru \bar{L}).
- W każdej i -tej iteracji do listy L w danym przypadku trafia $|L(i)|$ węzłów. Dla każdego z nich zachodzi $t_{nowe} < v.d < t_{nowe} + t$. Wiedząc, że $t_{nowe} = \min\{v.d : v \in \bar{L}\}$, możemy oszacować maksymalną ilość razy, jaką każdy wierzchołek v będzie skanowany, zanim nie zostanie usunięty z aktualnej listy L . Ponownie, wykorzystując analogię do analizy algorytmu kubekowego (tym razem DKA) zauważmy, że z każdą wykonaną relaksacją krawędzi, która prowadzi do wierzchołka v , jego odległość od źródła ulega zmniejszeniu o co najmniej 1. Zatem, jeżeli dla aktualnie badanego L znajdują się w nim wierzchołki, spełniające $t_{nowe} < v.d < t_{nowe} + t$, to po najwyżej t skanowaniach wierzchołka v zostanie on trwale usunięty z listy L (w następnych iteracjach do listy L będą trafiać wierzchołki u o $u.d > v.d$, więc ponowna relaksacja krawędzi, wchodzącej do węzła v na pewno nie zajdzie). Zatem nasz algorytm wykona maksymalnie $O(m \cdot t)$.
- W każdej iteracji usunięcie wszystkich elementów z $L(i)$ zajmuje $O(|L(i)|)$. Z faktu, że możemy wykonać maksymalnie $O(m \cdot t)$ relaksacji wynika, że podczas trwania algorytmu usuniemy co najwyżej $O(n \cdot t)$ wierzchołków.

Łącznie zatem otrzymujemy złożoność na poziomie $O(m \cdot t + n \cdot t + \frac{n \cdot C}{t}) = O(m \cdot t + n \cdot (t + \frac{C}{t}))$.

Niestety nie jest to koniec, przeprowadzanej przez nas analizy, gdyż omineliśmy analizę czasu, jaki potrzebny jest do odnalezienia minimalnych wierzchołków w \bar{L} dla każdej z $O\left(\frac{n \cdot C}{t}\right)$ iteracji. Zakładając najgorszy możliwy przypadek, natychmiast otrzymujemy górne ograniczenie $O(n \cdot \frac{n \cdot C}{t})$. Jest ono jednak bardzo niedokładne, gdyż zakłada, że z każdą iteracją musimy przeszukać $O(n)$ wierzchołków w \bar{L} , by znaleźć ten najmniejszy. Nie jest to prawda, chociażby z tego względu, że algorytm kończy działanie tylko wtedy, gdy w powyższym zbiorze nie ma już żadnych elementów. Przyjmijmy zatem bardziej optymistyczną wersję: niech w każdej z $O\left(\frac{n \cdot C}{t}\right)$ iteracji ze zbioru \bar{L} będzie usuwanych tyle samo elementów. Daje nam od razu ich liczbę: mamy $n - 1$ wierzchołków, z czego ich ilość w liniowy sposób maleje w czasie trwania wymienionej iteracji, a zatem każdorazowo usuwanych jest ich $\frac{t}{C}$. W czasie działania całego algorytmu suma liczby elementów w zbiorze \bar{L} wynosi zatem:

$$\begin{aligned} D &= \sum_{i=\frac{(n-1) \cdot C}{t}}^1 i \cdot \frac{t}{C} \approx \frac{t}{C} \cdot \sum_{i=1}^{\frac{n \cdot C}{t}} i \\ &= \frac{t}{C} \cdot \frac{C \cdot n \cdot (C \cdot n + t)}{2 \cdot t^2} \\ &= \frac{n \cdot (n \cdot C + t)}{2 \cdot t}. \end{aligned} \tag{3.1}$$

Czas wyszukiwania minimalnego wierzchołka w zbiorze \bar{L} zawsze jest proporcjonalny do ilości jego elementów, więc ostatecznie mamy: $O\left(\frac{n^2 \cdot C}{t}\right)$. Taki sam wynik otrzymaliśmy dla analizy wcześniejszego przypadku. Tym samym złożoność całego algorytmu to $O\left(m \cdot t + \frac{n^2 \cdot C}{t}\right)$.



Rysunek 3.4: **(Działanie algorytmu opartego o wartość progową** (a) Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAPH ($t = 3$). Poniżej wartości progowej znajduje się węzeł v_1 , będący źródłem. Powyżej znajduje się zawartość listy \bar{L} (obecnie pusta). (b) Sytuacja po wykonaniu relaksacji krawędzi e_{12} oraz e_{13} . Wartość $v_3.d \geq t$ - wierzchołek został wstawiony do \bar{L} . (c) Po wykonaniu relaksacji krawędzi, wychodzących z wierzchołka v_2 , na koniec kolejki ponad wartością progową kolejno zostały dodane węzły: v_5 i v_4 . Lista L jest pusta. (d) Prób zostaje zwiększony o t oraz zostały przeniesione na koniec kolejki wszystkie wierzchołki v , dla których $v.d < t$ (wierzchołki z \bar{L} do L są przepinane w kolejności od góry pierwszej listy do ogona, a każdy węzeł zostaje wstawiony na koniec drugiego zbioru). (e) Z listy L zostaje pobrany minimalny wierzchołek, a następnie algorytm wykonuje relaksację krawędzi, wychodzącej z danego wierzchołka (e_{35}), w wyniku czego węzeł v_5 zostaje przepięty na koniec listy L . (f) Po wykonaniu pozostałych relaksacji i opróżnieniu listy \bar{L} algorytm kończy działanie.

Biblioteka: Take Me Home

4.1 Opis

4.1. OPIS

Zakończenie

Instrukcja

A.1 Wymagania i instalacja

A.2 Użytkowanie

A.2.1 konfiguracja

A.2.2 API

A.2. UŻYTKOWANIE

Dowody twierdzeń

Trochę matmy

C.1 Złożoność obliczeniowa

TODO

C.1.1 Analiza asymptotyczna

TODO

C.1.2 Analiza amortyzacyjna

TODO

C.1. ZŁOŻONOŚĆ OBliczeniowa

Bibliografia

- [1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM'01*, San Diego, California, USA, August 2001.