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

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 $i \in \{1, \ldots, |V|\}$ będziemy od tej pory oznaczać dowolny wierzchołek w grafie G (w odniesieniu do rzeczywistego ruchu drogowego może to być skrzyżowanie, dowolny punkt drogi na wysokości 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 łuku [1] 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 wpłynąć 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

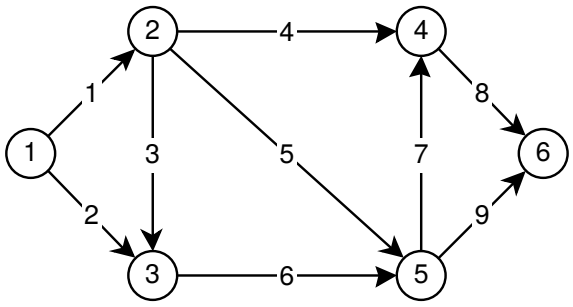
Macierz incydencji dla grafu $G = (V, E)$ jest to macierz o wymiarach $|V| \times |E|$, gdzie każda komórka jest zdefiniowana następująco:

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

gdzie:

$$\begin{aligned} i &\in \{1, \ldots, |V|\}, \\ j &\in \{1, \ldots, |E|\}. \end{aligned} \tag{1.2}$$

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



(a)

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

(b)

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 następników 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 następników węzła v_i, obejmuje koszt wyszukania wszystkich łuków, prowadzących do tych wierzchołków oraz odnalezienie ich identyfikatorów - co dla każdego odnalezionego łuku 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 $O\left(|A\left(i\right)| \cdot n\right)$ dla wyszukiwania wszystkich następników węzła v_i (wraz z wyszukiwaniem łuków $O\left(m + |A\left(i\right)| \cdot n\right)$).

1.2.2 Macierz sąsiedztwa

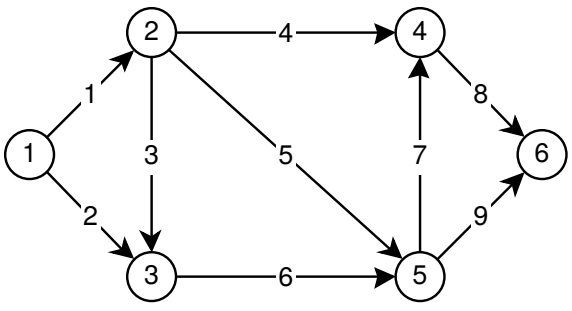
Oprócz macierzowej reprezentacji typu wierzchołek-krawędź możliwe jest także reprezentowanie struktury grafu za pomocą macierzy typu wierzchołek-wierzchołek. **Macierz sąsiedztwa** dla grafu $G = (V, E)$ jest to macierz o wymiarach $|V| \times |V|$, gdzie wartość każdej komórki jest zdefiniowana następująco:

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

O ile, w przypadku takiego przedstawienia grafu, jesteśmy w stanie uzyskać informacje o wszystkich bezpośrednich następnikach 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 będziemy szukać właśnie w krawędziach między wierzchołkami. Załóżmy, 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:

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

gdzie przez $d\left(i, j\right)$ 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 $d\left(i, j\right)$ w danych komórkach b_{ij} będzie wstawić numer identyfikatora danej krawędzi. Zapewni nam to dodatkowo jednoznaczność w przypadku chęci identyfikacji krawędzi (zwróćmy uwagę, że oba łuki, wchodzące do węzła v_4 mają ten sam koszt $d\left(2,4\right) = d\left(5,4\right) = 2$, co uniemożliwia nam ich rozróżnienie).



(a)

$d\left(1,2\right)$=3
$d\left(1,3\right)$=1
$d\left(2,3\right)$=5
$d\left(2,4\right)$=2
$d\left(2,5\right)$=1
$d\left(3,5\right)$=4
$d\left(4,6\right)$=3
$d\left(5,4\right)$=2
$d\left(5,6\right)$=1

(b)

	1	2	3	4	5	6
1	0	3	1	0	0	0
2	0	0	5	2	1	0
3	0	0	0	0	4	0
4	0	0	0	0	0	3
5	0	0	0	2	0	1
6	0	0	0	0	0	0

(c)

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 $d\left(v_p^{ID}, v_k^{ID}\right) = c$ 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 ($v_k^{ID} = k$).

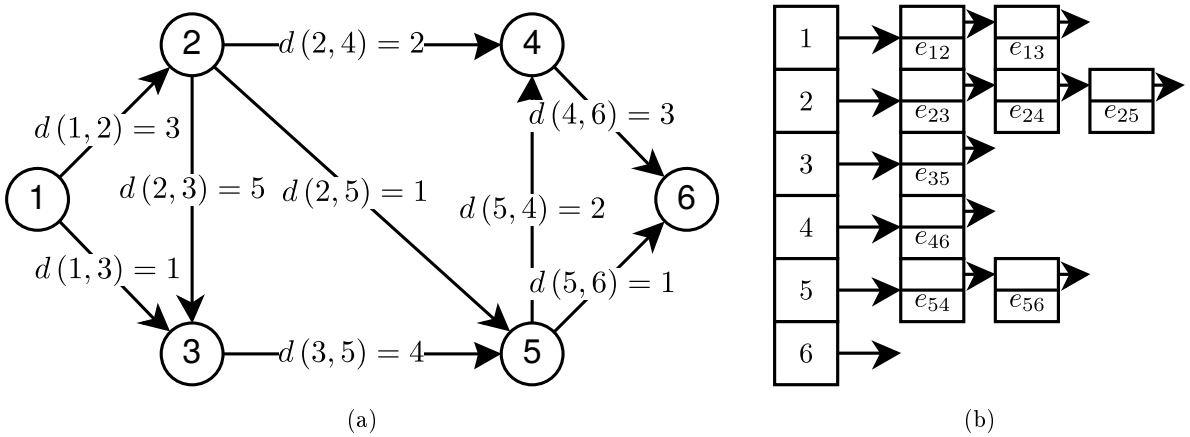
Zatem ostatecznie otrzymamy:

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

- koszt wyszukiwania wszystkich następników 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 następników 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. Słowem - połączymy poprzednio rozdzielane operacje identyfikacji łuków i węzłów w jedną. Da to nam w najgorszym przypadku liniowy czas dostępu do dowolnego wierzchołka, będącego bezpośrednim następnikiem badanego elementu (gdy będziemy musieli przejść przez całą listę $A(i)$).



(a) (b)

Rysunek 1.3: **Lista sąsiedztwa (a)** Graf 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 $v_i : i \in \{1, \ldots, 6\}$ 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 rozwiązanie z użyciem tzw. pęków. Tutaj także dla każdego węzła tworzyć będziemy łistę"węzłów, które są jego bezpośrednimi następnikami 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.

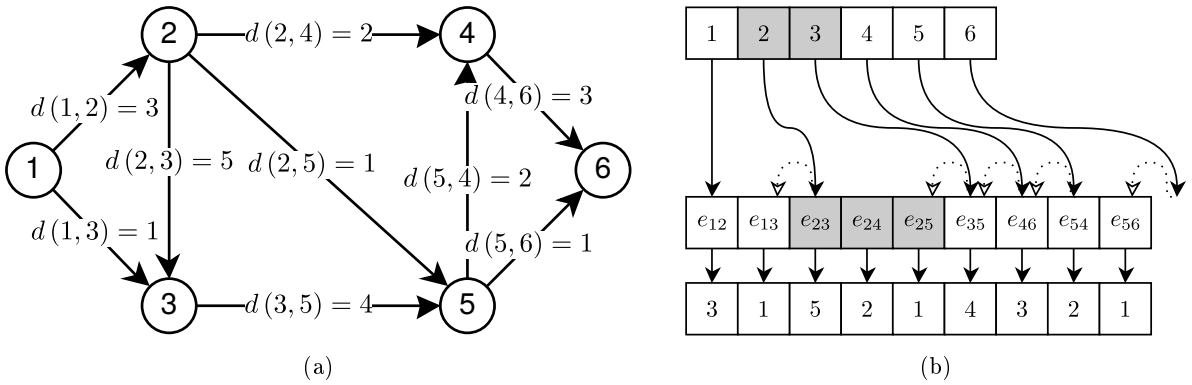
Pęki wyjściowe

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,

[2] Mówimy, 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 pierwszej tablicy (rys. 1.4).



Rysunek 1.4: **Pęki wyjściowe (a)** Graf 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 kolei 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 przeglądniemy 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 przeglądniemy 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 ... |E|]
1  begin
2      aIdx ⟵ 1
3      vtab[1 ... |V|] ⟵ aIdx
4      for vIdx ∈ 1 ... |V| do                    /* Dla każdego węzła w vtab */
5          for każdy łuk e, prowadząca z v_vIdx do wierzchołka ∈ A(vIdx) do
6              atab[aIdx] = e
7              aIdx ⟵ aIdx + 1
8          vtab[vIdx + 1] ⟵ aIdx
```

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 początek. W obu przypadkach kolejność występowania krawędzi, wychodzących/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 następników/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 następników, 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:

- $vtab[1 \ldots |V|]$ - przechowującą informacje o łukach dla węzłów o identyfikatorach od 1 do $|V|$,
- $atab[1 \ldots |E|]$ - przechowującą krawędzie o identyfikatorach od 1 do $|E|$, posortowane rosnąco według identyfikatorów wierzchołków początkowych, gdzie dla każdego węzła v_k wszystkie identyfikatory łuków wychodzących 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:

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

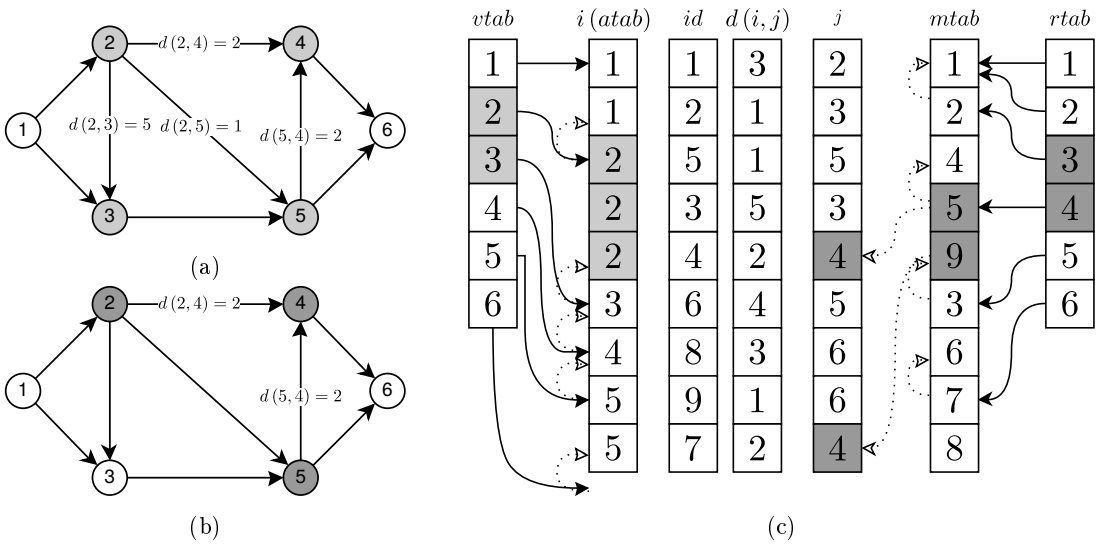
Druga z nich - *mtab* - jest tablicą wejściową dla metody, opisanej w sekcji "Pęki wejściowe". Przechowuje ona w swoich kolejnych elementach indeksy tablicy *atab* w taki sposób, aby ciąg $atab[mtab[1]], atab[mtab[2]], \ldots, atab[mtab[|E|]]$ tworzył ciąg identyfikatorów łuków, które są posortowane 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 $O(|V| \ldots |E|)$ lub $O(|V| \ldots |V|)$, 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

[3] graf, 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 następnikami węzła v_2. **(b)** Ten sam graf skierowany z oznaczonymi poprzednikami węzła v_2. **(c)** Graf przedstawiony w formie tablic.

odpowiednie kodowanie, możemy wpłynąć 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 przeciwne.

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

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

Następnie zamieniamy każdą macierz na ciągi binarne długości $|V| \ldots |E|$ 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 $O\left(|V|^2\right)$, 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		Listy	Pęki
	Incydencji	Sąsiedztwa	Sąsiedztwa	Wejścia-wyjścia
Potrzebna pamięć	$O(\|V\| \cdot \|E\|)$	$O\left(\|V\|^2\right)$	$O(\|V\| + \|E\|)$	$O(\|V\| + \|E\|)$
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 rozdzieliliśmy operacje dodawania i usuwania elementów grafu, wyróżniając takie, które mają na celu tylko żakryćóbecność danego elementu, bądź z powrotem przywrócić jego widoczność oraz na takie, 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 "usunąćź 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 następników 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.

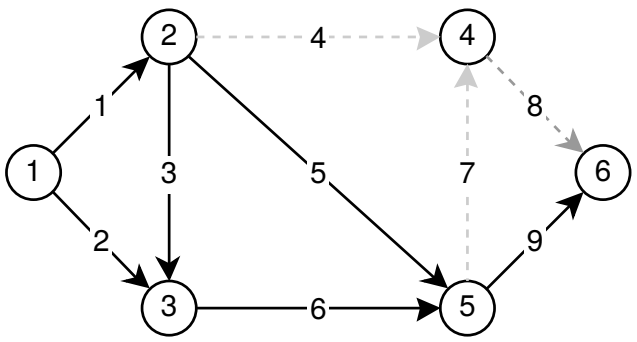
	Macierze		Listy	Pęki
	Incydencji	Sąsiedztwa	Sąsiedztwa	Wejścia-wyjścia
Dodawanie węzła	$O(\|V\| \cdot \|E\|)$	$O(1)$	$O(1)$	$O(\|V\| + \|E\|)$
Dodanie nowego węzła	$O(\|V\| \cdot \|E\|)$	$O\left(\|V\|^2\right)$	$O(1)$	$O(\|V\|)$
Przysłonięcie węzła	$O(\|V\| \cdot \|E\|)$	$O(\|V\|)$	$O(\|E\|)$	$O(\|V\|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

Poza takimi operacjami jak: dodawanie, usuwanie krawędzi grafu, wyznaczanie stopnia wierzchołka, jego następników 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 - podstawowym pomysłem 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 górne ograniczenie na złożoność tej operacji na poziomie $O(|V|)$.

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

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



(a)

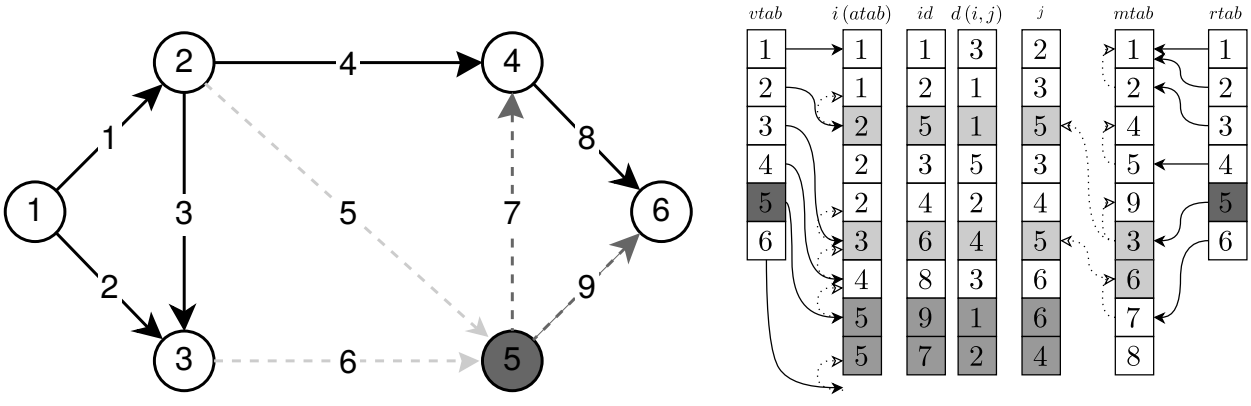
	1	2	3	4	5	6
1	0	1	2	0	0	0
2	0	0	3	4	5	0
3	0	0	0	0	6	0
4	0	0	0	0	0	8
5	0	0	0	7	0	9
6	0	0	0	0	0	0

(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.

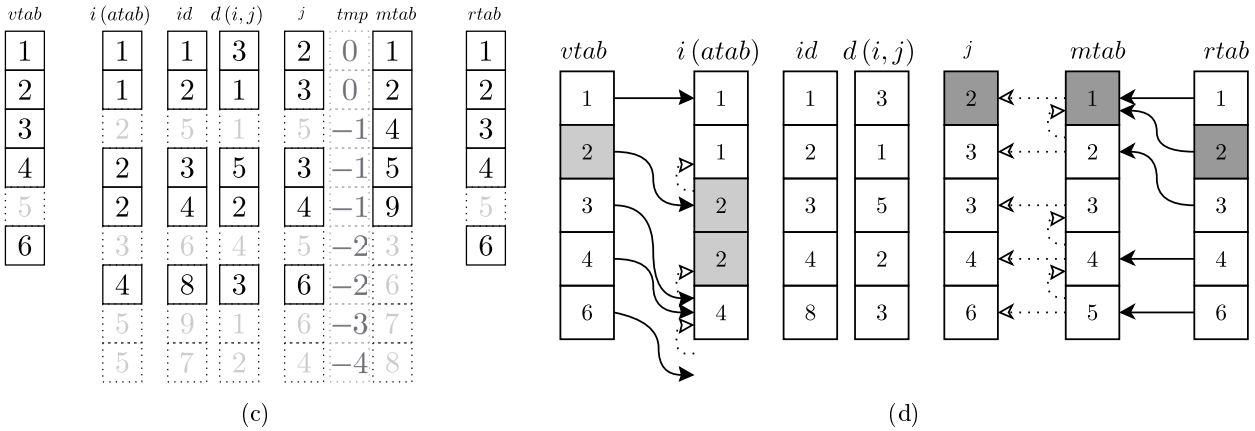
W przypadku pozostałych dwóch struktur znowu nie odnotujemy żadnej zmiany - aby usunąć dany węzeł z list sąsiedztwa będziemy musieli odnaleźć wszystkie łuki, które prowadzą do usuwanego węzła, a na koniec usunąć całą listę jego sąsiedztwa, wraz z łukami, które się na niej znajdują. W najgorszym przypadku 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 $O\left(A(i) + A^R(i)\right)$ [6], zmiany ich rozmiarów ($O(|E|)$), usunięcia danego węzła z dwóch pozostałych tablic, indeksowanych od $1 \ldots |V|$ (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) (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 $O\left(A^R(5)\right)$ dla przychodzących. Węzeł, który należy usunąć wyznaczamy w czasie $O(1)$.

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



Rysunek 1.7: **(c)** Tablice z usuniętymi powiązaniami - po usunięciu chcianych elementów odtworzymy je w czasie liniowym. Rekonstrukcja związków, zachodzących między tablicami *vtab*, a *atab* wymagać będzie przejścia przez ich wszystkie elementy, wyjąwszy krawędzie, wychodzące z ostatniego węzła, gdyż jego lista następników 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 $j\,[mtab\,[1]], \ldots, j\,[mtab\,[|E|]]$ 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 ułatwią 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 vk, 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 oznaczali przez $d\,(v, u, \ldots)$, 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 $P = \langle v_0, v_1, \ldots, v_k \rangle$, że suma kosztów c_{ij} ścieżek jest najmniejsza:

$$\sum_{e_{ij} \in P'} c_{ij} = minimum : e_{ij} \in P^{'} \Leftrightarrow v_i, v_j \in P \ \wedge \ v_i \rightsquigarrow v_j = e_{ij} \ni E. \tag{1.7}$$

gdzie przez $v_i \rightsquigarrow v_j$ będziemy oznaczać pojedynczą ścieżkę z węzła v_i do węzła v_j. Wprowadzimy także zapis $v_i \overset{k}{\rightsquigarrow} v_k$, przez który będziemy rozumieć drogę złożoną z k ścieżek, prowadzących od punktu v_i do v_k (użyty symbol "$*$" 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, rozwiązujących 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 $P = \langle v_0, v_1, \ldots, v_{i-1}, v_i, \ldots v_k \rangle$),
- **waga najkrótszej ścieżki do węzła (**$d(i)$**)**, który dla węzła v_i będzie przyjmował zawsze wartość najmniejszego 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 $v_S \overset{*}{\rightsquigarrow} v_i$, aby być tą najkrótszą musi mieć łączny koszt nie większy niż $d(i)$, czyli mniejszy lub równy k). Ponad to 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:

$$d(i) = \begin{cases} \min \left\{ c(s,i) : v_s \overset{*}{\rightsquigarrow} v_i \right\} & \text{jeśli } \exists v_s \overset{*}{\rightsquigarrow} v_i \\ \infty & \text{w przeciwnym przypadku} \end{cases} \tag{1.8}$$

gdzie:

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

Do wszystkich atrybutów węzłów będziemy odwoływać się w dalszej części albo umieszczając jego nazwę w górnym indeksie, jak to robiliśmy do tej pory (np. vi^{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 rodzica [7] węzła v_i).

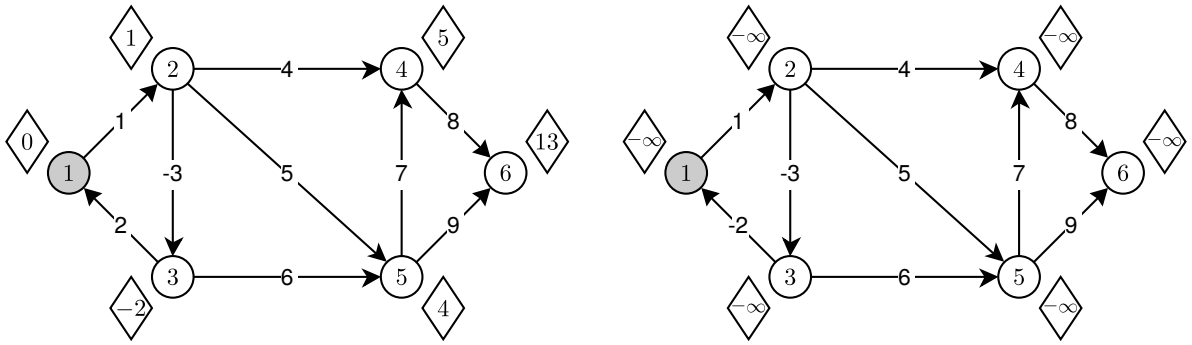
Aby efektywnie móc rozwiązywać problem najkrótszych ścieżek musimy przyjąć kilka założeń, które wynikają zarówno ze specyfikacji samego problemu, z przyjętego modelu (rzeczywistych sieci drogowych) oraz

[7] W 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.

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 $e \in E$ - pozostaniemy 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 słuszne z siecią drogową jako modelem, który sobie obraliśmy, lecz także bezpośrednio z niego wypływa 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 ($c(i,i) < 0$).



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 pojawieniem się cyklu o ujemnej wadze ($v_1 \rightsquigarrow v_2 \rightsquigarrow v_3 \rightsquigarrow v_1 \rightsquigarrow \ldots$) wszystkie koszty w wierzchołkach stają się nieskończenie 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 $-\infty + n = -\infty$). Oczywistym 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 uważać wszystkie, które zawierają cykl także o koszcie dodatnim (z analogicznego powodu). Załóżmy, że istnieje najkrótsza ścieżka $P = \langle v_0, v_1, \ldots, v_k \rangle$ taka, że $c(0,k) = D$ i zawiera ona cykl dowolnej, niezerowej długości (powiedzmy $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$, gdzie $v_i = v_j$ oraz $c(i,j) \neq 0$). Nasza ścieżka zatem wygląda tak: $P = \langle v_0, v_1, \ldots, v_i, v_{i+1}, \ldots, v_j, \ldots, v_k \rangle$ (bez straty ogólności załóżmy, 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: $P^{'} = \langle v_0, v_1, \ldots, v_i, v_{j+1}, \ldots, v_k \rangle$ o tym samym węźle początkowym i końcowym [8], co poprzednia ścieżka (a więc "tą samą" ścieżkę), tyle że o zmienionym koszcie. Jeżeli usunęliśmy 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-GRAPH (G, s)

```
1 begin
2     for vIdx ∈ 1...|V| do                    /* Dla każdego węzła w vtab */
3         vtab[vIdx].d ⟵ ∞
4         vtab[vIdx].Π ⟵ null
5     vtab[s].d ⟵ 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 kosztowi ścieżki, prowadzącej ze źródła do danego węzła, co formalnie możemy w skróconej formie wyrazić:

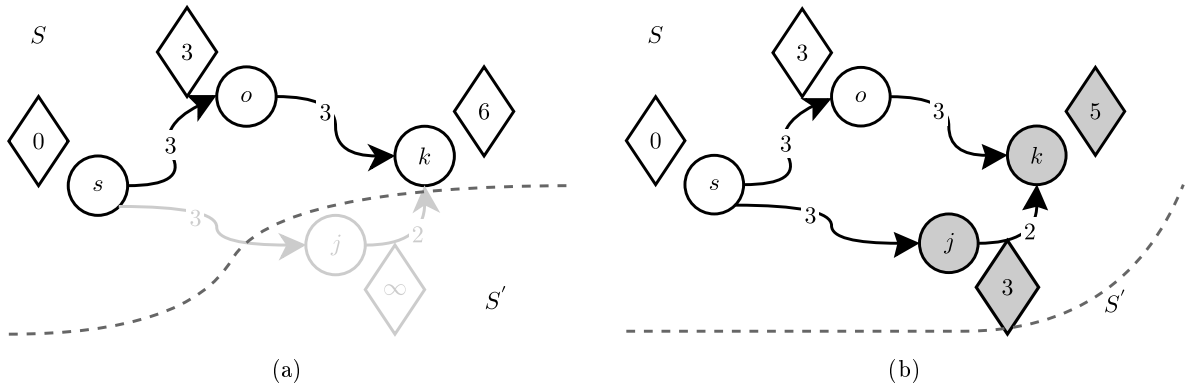
$$d(i) = \sum_{e_{jk} \in P} c_{jk} \rightarrow min \ : \ P = \left\langle e_{Sj^{(1)}}, e_{j^{(1)}j^{(2)}}, \ldots, e_{j^{(m)}i} \right\rangle \tag{1.10}$$

[8] Nie usuwamy pierwszego węzła cyklu, więc jeżeli cykl znajdowałby się na początku ścieżki to w oczywisty sposób węzeł v_p na ścieżce $v_p \overset{*}{\rightsquigarrow} v_k$ nie uległby zmianie. Analogicznie w przypadku, gdyby cykl znajdował się na końcu tj. $P = \langle v_0, v_1, \ldots, v_i, v_{i+1}, \ldots, v_j \rangle$ i $v_j = v_k$, gdzie usunięcie cyklu spowodowałoby powstanie ścieżki $P^{'} = \langle v_0, v_1, \ldots, v_i \rangle$. 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 $v_S \overset{m+1}{\rightsquigarrow} v_i$.

W czasie działania wszystkich algorytmów wyszukiwania najkrótszych ścieżek będziemy chcieli zachować tą własność, by w momencie zakończenia ich działania otrzymywać poprawne wyniki. Aby to było możliwe, musimy wprowadzić kolejną operację, którą będziemy nazywać operacją relaksacji krawędzi. Przyjrzyjmy 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 $i \in \left\{ i' : v_{i'} \in S \right\}$. **(b)** Dodanie do zbioru S wierzchołka v_j spowodowało powstanie nowej ścieżki $v_s \overset{*}{\rightsquigarrow} v_k$, 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 wyszarzono 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 vS - 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 $v_o = v_k^{\Pi}$ nie należy już do zbioru węzłów, leżących na najkrótszej ścieżce $v_S \overset{*}{\rightsquigarrow} v_k$. Łatwo zauważyć, 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                                   /* Oznaczenia węzłów zachowane z rysunku 1.9 */
2     if k.d > j.d + d(j,k) then          /* d(j,k,...) - funkcja wagi */
3         k.d ⟵ j.d + d(u,k)
4         k.Π ⟵ j
```

Algorytm relaksacji wierzchołków sprawdza, czy nowo dodany wierzchołek zaburza, interesującą nas, własność 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

[10] O tym, że tak jest w istcie przekonamy się w następnym rozdziale.

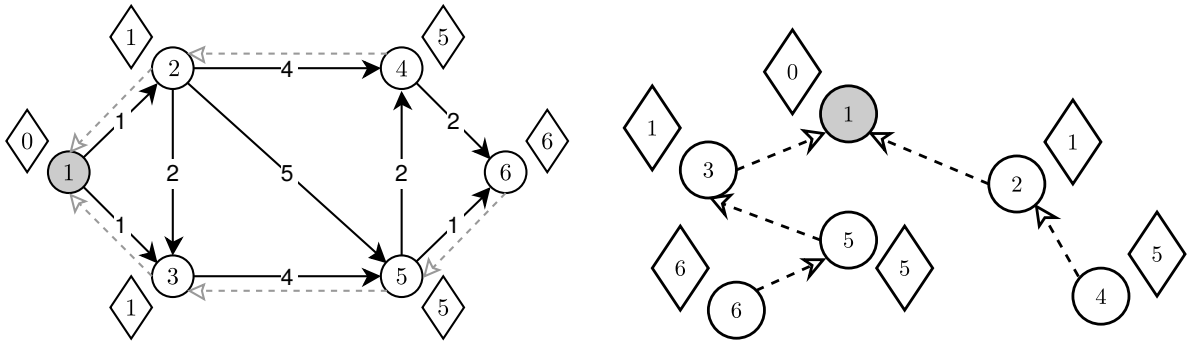
ścieżki nie zawsze okazuje się wystarczająca. Przy omawianiu dwóch poprzednich algorytmów na równi z operowaniem na atrybutach węzłów $d(i)$ pozwalaliśmy sobie zmieniać także atrybut, odpowiadający wskazaniu na rodzica danego węzła (Π), doprowadzając zawsze do sytuacji, w której rodzicem danego węzła, 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 rozpoczynała. Innymi słowy dbaliśmy o to, by dla każdej najkrótszej ścieżki $P = \langle v_0, v_1, \ldots, vk \rangle$ dla każdego $v_i : i \in \{1, \ldots, k\}$ zachodziła prawidłowość: $v_i.\Pi = v_{i-1}$ (dla $i = 0$ w oczywisty sposób $v_0.\Pi = null$), a zatem nasz algorytm, pozwalający nam na wypisanie po kolei węzłów na danej, najkrótszej ścieżce $v_0 \overset{k}{\rightsquigarrow} v_k$, 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.Π ≠ null do
3         Wypisz v
4         v ⟵ v.Π
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 $v.\Pi = null$. **(b)** Poddrzewo grafu, zawierające najkrótsze ścieżki od źródła do wszystkich pozostałych węzłów w grafie G, w raz 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, zastosować rekursję)

1.3.3 Właściwości najkrótszych ścieżek

TODO

Lemat 1.3.1 (Własność optymalnej podstruktury) *TODO*

Lemat 1.3.2 (Własność braku ścieżki) *TODO*

Lemat 1.3.3 (Własność górnego ograniczenia) *TODO Zawsze v.d >= d(v)*

Lemat 1.3.4 (Własność zbieżności) *Jeśli s do d jest optymalne przed relaksacją to s–>d–>w po relaksacji (d,w), s—>w też optymalne*

Lemat 1.3.5 (Własność relaksacji dla ścieżki) *Jeśli p=(vp....vk) to najkrótsza ścieżka to po |P| relaksacjach vk.d = min*

Lemat 1.3.6 (Własność podgrafu poprzedników) *TODO*

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 w poprzednim rozdziale

TODO

Algorithm 5: BELLMAN-FORD (G, s)

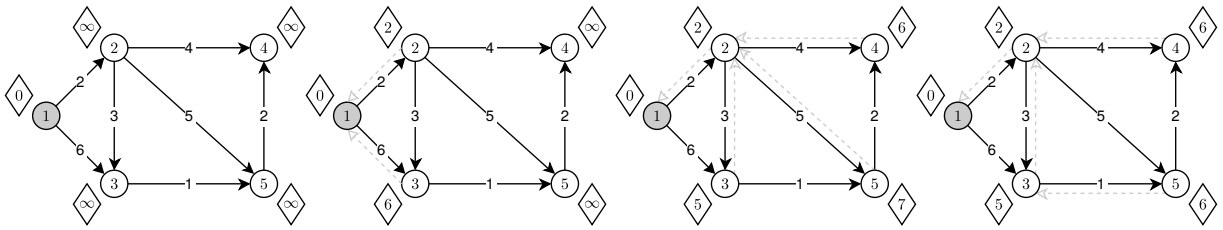
```
1 begin
2     for i = 1 to |V| − 1 do
3         forall the (u, v) ∈ E do
4             RELAX (u, v)
5     forall the (u, v) ∈ E do
6         if v.d > u.d + c_uv then
7             return TRUE
```

można też sprawdzać, czy brak cykli ujemnych.

Oczywiście sprawdzanie czy wyskoczyliśmy poza nieskończoność wymaga dosłownego porównania no i można to przyspieszyć, nie skanujących tych krawędzi, które wychodzą z węzłów, które mają d = infty. Mamy w końcu listę sąsiadów, więc przeglądnięcie wszystkich krawędzi jest równoważne z przeglądnięciem wszystkich A(i).

Można to usprawnić poprzez: sprawdzanie, czy była jakaś zmiana w czasie iteracji, jak nie to przerwać. Można też nie sprawdzać A(i) jeśli vi ma d = infty.

TODO



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 $(1, 3)$ i dla tych węzłów (v_2 i v_3) zostały zaktualizowani ich poprzednicy (zaznaczeni szarymi strzałkami) oraz etykiety d. Dla pozostałych algorytm nie wprowadził żadnych zmian w trakcie iterowania po wszystkich $A(i) : i \in \{1, \ldots, 5\}$. **(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.

Struktury danych

2.1 Złożoność obliczeniowa

2.1.1 Analiza asymptotyczna

2.1.2 Analiza amortyzacyjna

2.2 Generyczny algorytm Dijkstry

TODO

Algorithm 6: DIJKSTRA (G, s)

```
1 begin
2     for i = 1 to |V| − 1 do
3         forall the (u, v) ∈ E do
4             RELAX (u, v)
5     forall the (u, v) ∈ E do
6         if v.d > u.d + c_uv then
7             return TRUE
```

TODO

2.3 Podstawowe struktury danych

2.4 Struktury oparte na kopcach

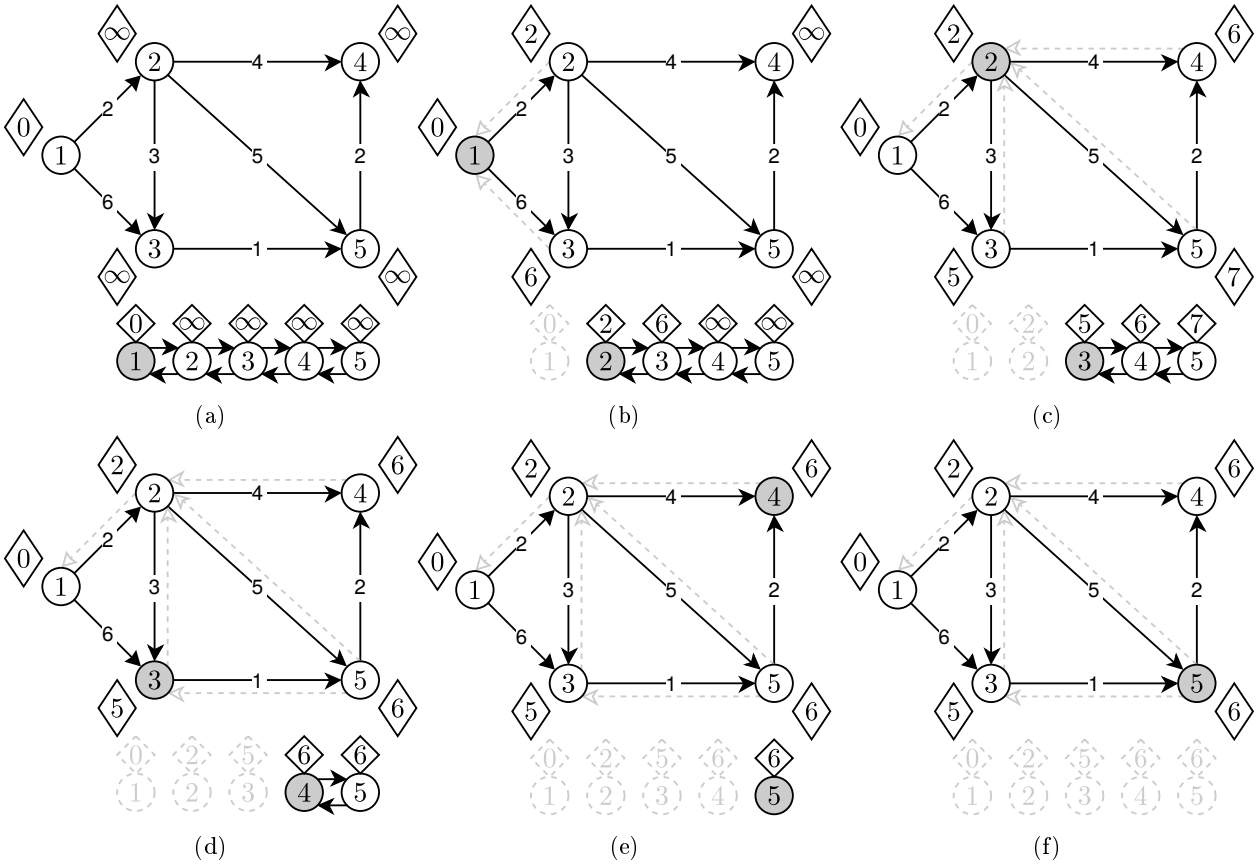
2.4.1 Kopiec R-arny

2.4.2 Drzewo k-arne

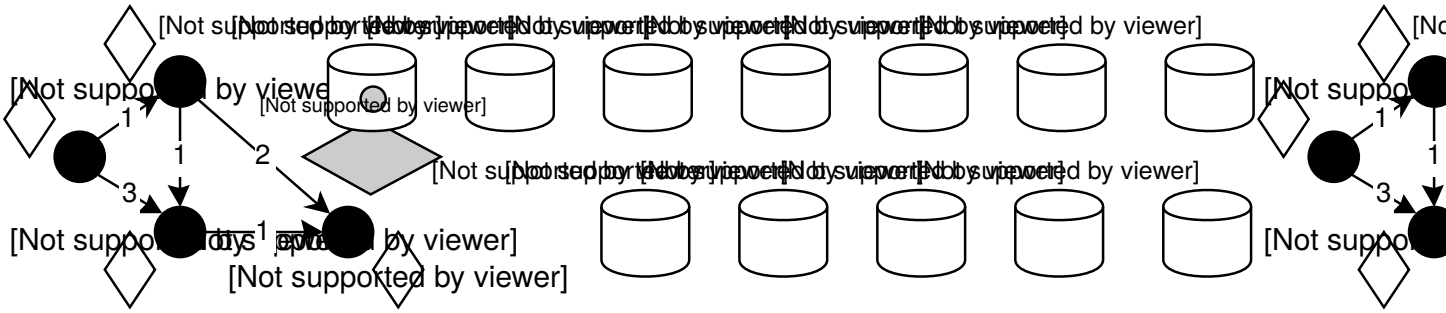
2.4.3 Kopiec Fibonacciego

2.5 Struktury oparte na kubełkach

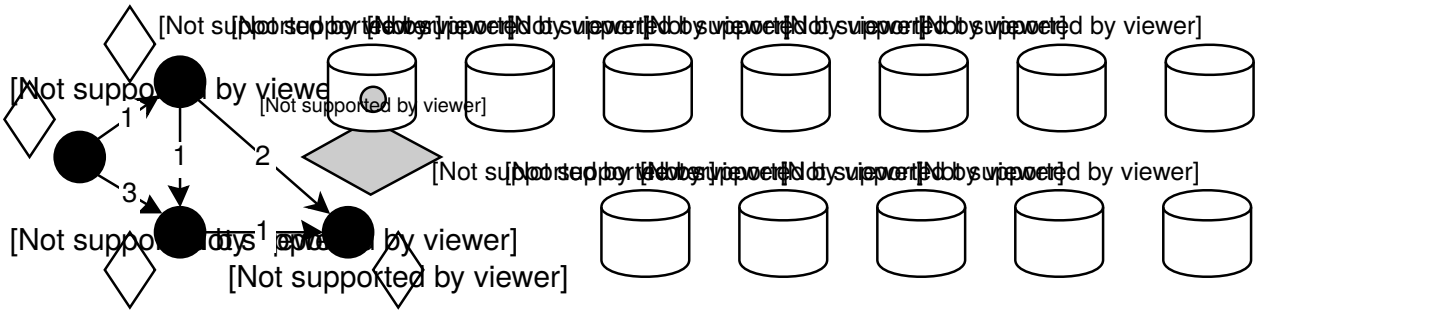
nC+1 kubełków, nic ciekawego



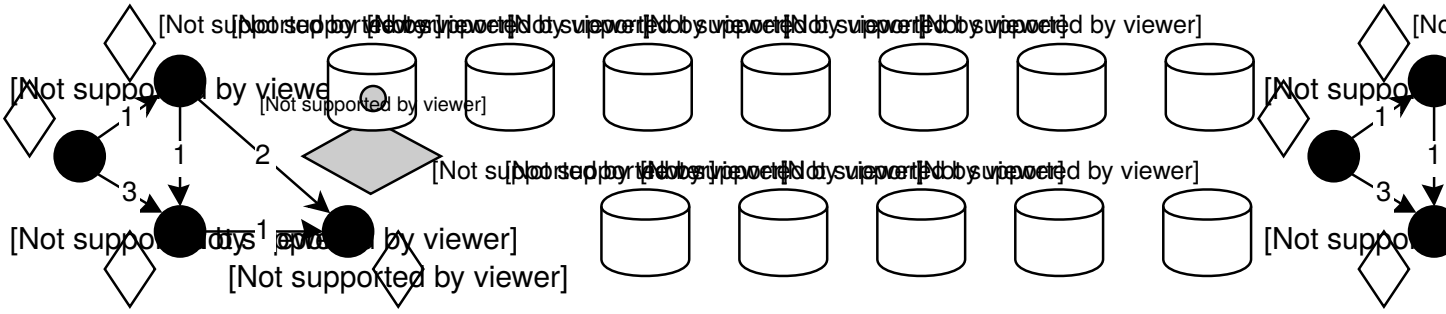
Rysunek 2.1: **Działanie algorytmu Bellmana-Forda (a)** Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAPH ze źródłem $v_s.id = 1$. **(b)** Warunek $v.d > u.d + c_{uv}$ dla krawędzi (u, v) spełniony jest tylko dla krawędzi: $(1, 2)$ i $(1, 3)$ i dla tych węzłów (v_2 i v_3) zostały zaktualizowani ich poprzednicy (zaznaczeni szarymi strzałkami) oraz etykiety d. Dla pozostałych algorytm nie wprowadził żadnych zmian w trakcie iterowania po wszystkich $A\,(i) : i \in \{1, \ldots, 5\}$. **(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 po wszystkich krawędziach, jednak wprowadziliśmy modyfikację, która przerywa działanie algorytmu, jeżeli podczas pełnej iteracji nie nastąpi w grafie G żadna zmiana.



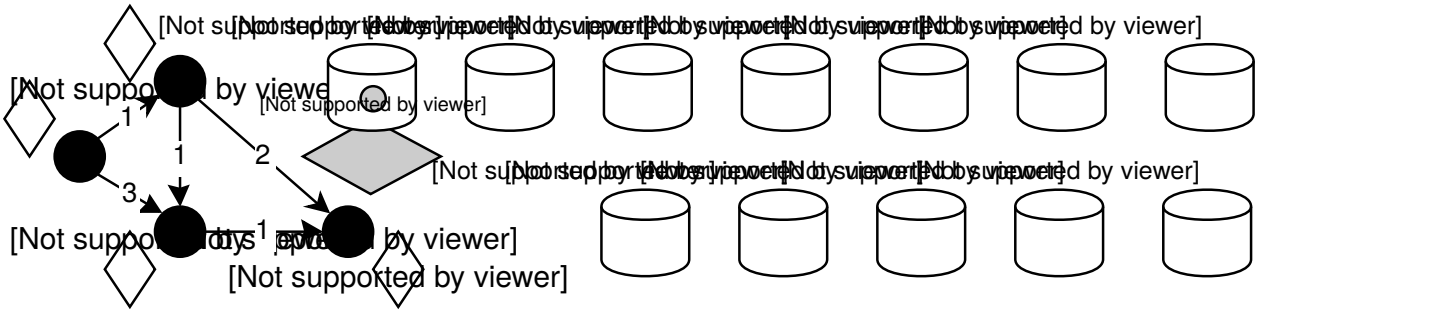
(a)



(c)



(d)



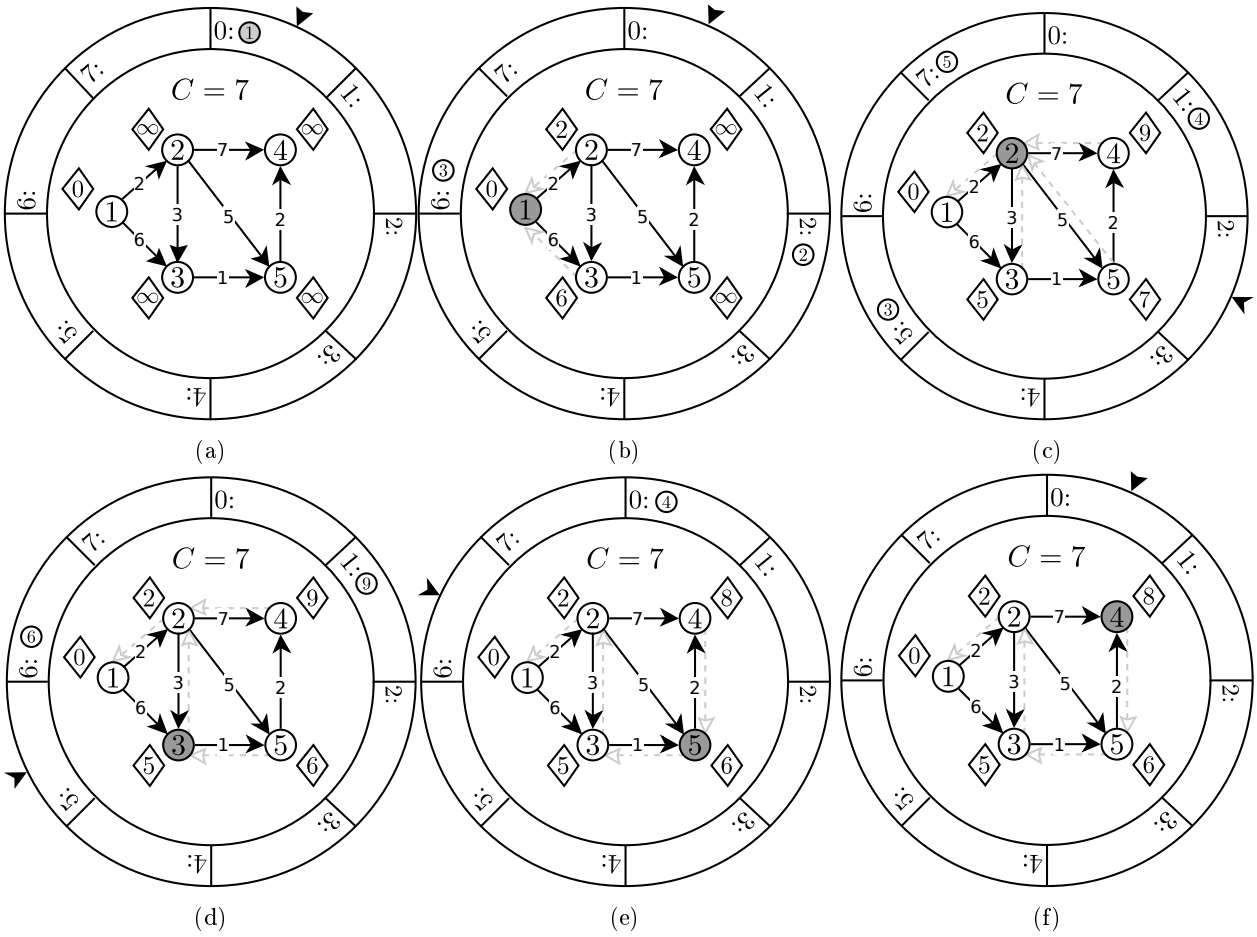
(f)

Rysunek 2.2: **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 $(1, 3)$ i dla tych węzłów (v_2 i v_3) zostały zaktualizowani ich poprzednicy (zaznaczeni szarymi strzałkami) oraz etykiety d. Dla pozostałych algorytm nie wprowadził żadnych zmian w trakcie iterowania po wszystkich $A(i) : i \in \{1, \ldots, 5\}$. **(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 po wszystkich krawędziach, jednak wprowadziliśmy modyfikację, która przerywa działanie algorytmu, jeżeli podczas pełnej iteracji nie nastąpi w grafie G żadna zmiana.

2.5.1 Z przepełnieniem

2.5.2 Dial

skanujemy wszystkie kubełki w kółko. C+1 kubełków



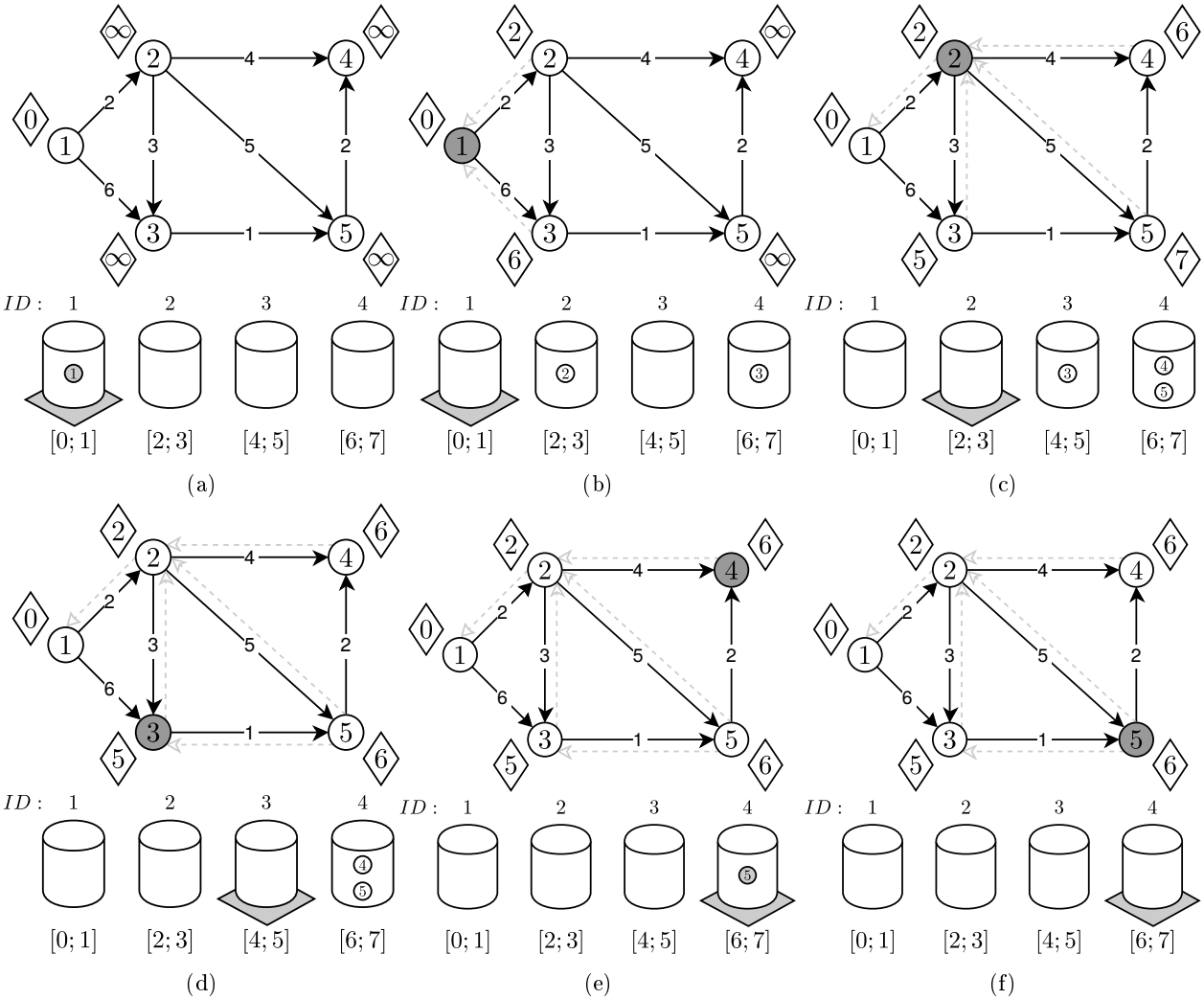
Rysunek 2.3: **Działanie algorytmu Bellmana-Forda (a)** Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAPH ze źródłem $v_s.id = 1$. **(b)** Warunek $v.d > u.d + c_{uv}$ dla krawędzi (u, v) spełniony jest tylko dla krawędzi: $(1, 2)$ i $(1, 3)$ i dla tych węzłów (v_2 i v_3) zostały zaktualizowani ich poprzednicy (zaznaczeni szarymi strzałkami) oraz etykiety d. Dla pozostałych algorytm nie wprowadził żadnych zmian w trakcie iterowania po wszystkich $A(i) : i \in \{1, \ldots, 5\}$. **(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 po wszystkich krawędziach, jednak wprowadziliśmy modyfikację, która przerywa działanie algorytmu, jeżeli podczas pełnej iteracji nie nastąpi w grafie G żadna zmiana.

2.5.3 Aproksymacja zakresu

skanujemy po kolei wszystkie kubełki, każdy z nich ma kolejkę fifo z wierzchołkami

W najgorszym przypadku by doskanować się do najmniejszego elementu musimy przeglądnąć wszystkie C+1 kubełków i wybrać z fifo ogon. Dodawanie/przepinanie wierzchołków wymaga wrzucenia nodea na szczyt

listy fifo i zrzucenia go do pozycji, gdzie niżej jest już węzeł o <= koszcie. W najgorszym przypadku czeka nas m updatów, każdy trwający b (wsadzanie do listy, by był zachowany priorytet) - szerokość kubełka (lecz czemu zakładamy, że na liście jest max b elementów??).



Rysunek 2.4: **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 $(1, 3)$ i dla tych węzłów (v_2 i v_3) zostały zaktualizowani ich poprzednicy (zaznaczeni szarymi strzałkami) oraz etykiety d. Dla pozostałych algorytm nie wprowadził żadnych zmian w trakcie iterowania po wszystkich $A(i) : i \in \{1, \ldots, 5\}$. **(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 po wszystkich krawędziach, jednak wprowadziliśmy modyfikację, która przerywa działanie algorytmu, jeżeli podczas pełnej iteracji nie nastąpi w grafie G żadna zmiana.

2.5.4 Kubełki wielopoziomowe

2.5.5 Radix-Heap

2.6 Analiza

Inne algorytmy

3.1 Kombinacje struktur

3.1.1

3.1.2

3.2 Sortowanie topologiczne

3.3 Algorytm progowy

3.4 Analiza

Biblioteka: Take Me Home

4.1 Opis

Zakończenie

Instrukcja

A.1 Wymagania i instalacja

A.2 Użytkowanie

A.2.1 konfiguracja

A.2.2 API

Instrukcja

B.1 Wymagania i instalacja

B.2 Użytkowanie

B.2.1 konfiguracja

B.2.2 API

Instrukcja

C.1 Wymagania i instalacja

C.2 Użytkowanie

C.2.1 konfiguracja

C.2.2 API