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

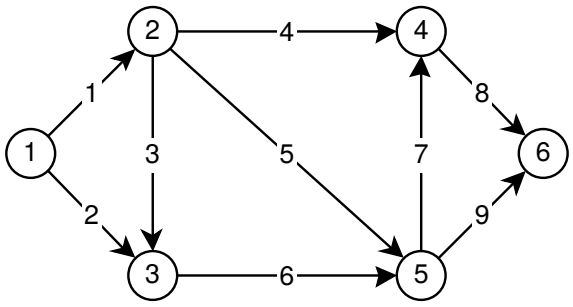
Macierz incydencji dla grafu $G = (V, E)$ jest to macierz o wymiarach $|V| \times |E|$, gdzie każda komórka jest zdefiniowana następująco:

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

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

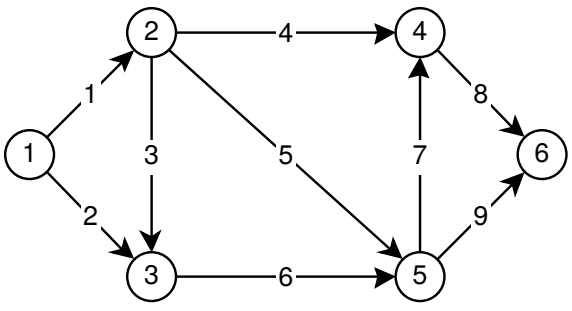
Oprócz macierzowej reprezentacji typu wierzchołek-krawędź możliwe jest także reprezentowanie struktury grafu za pomocą macierzy typu wierzchołek-wierzchołek. **Macierz sąsiedztwa** dla grafu $G = (V, E)$ jest to macierz o wymiarach $|V| \times |V|$, gdzie wartość każdej komórki jest zdefiniowana następująco:

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

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 ($v_k^{ID} = k$).

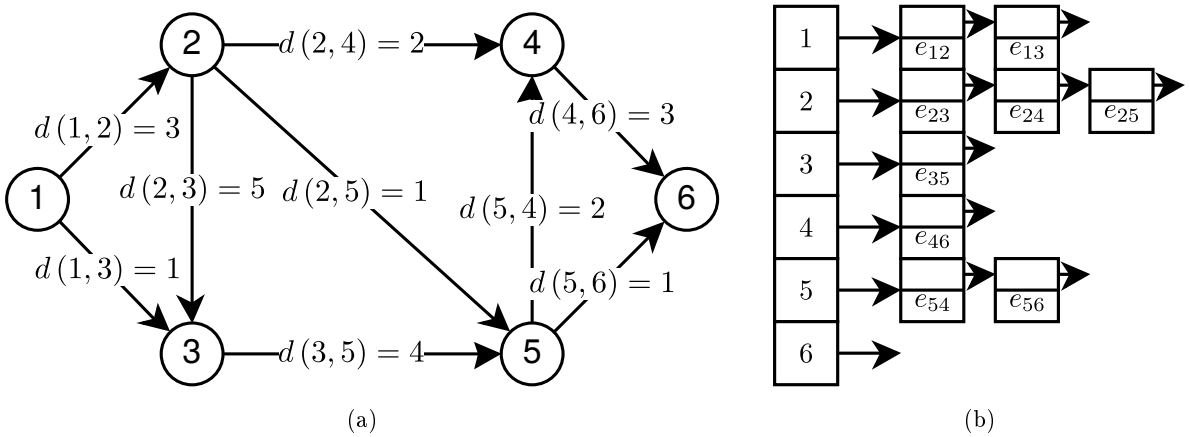
Zatem ostatecznie otrzymamy:

- 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.

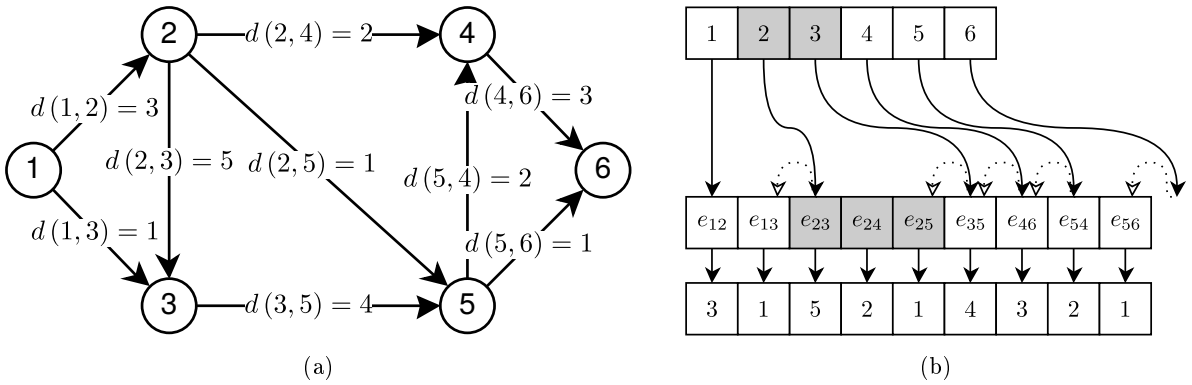
Pęki wyjściowe

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 $|E|$.

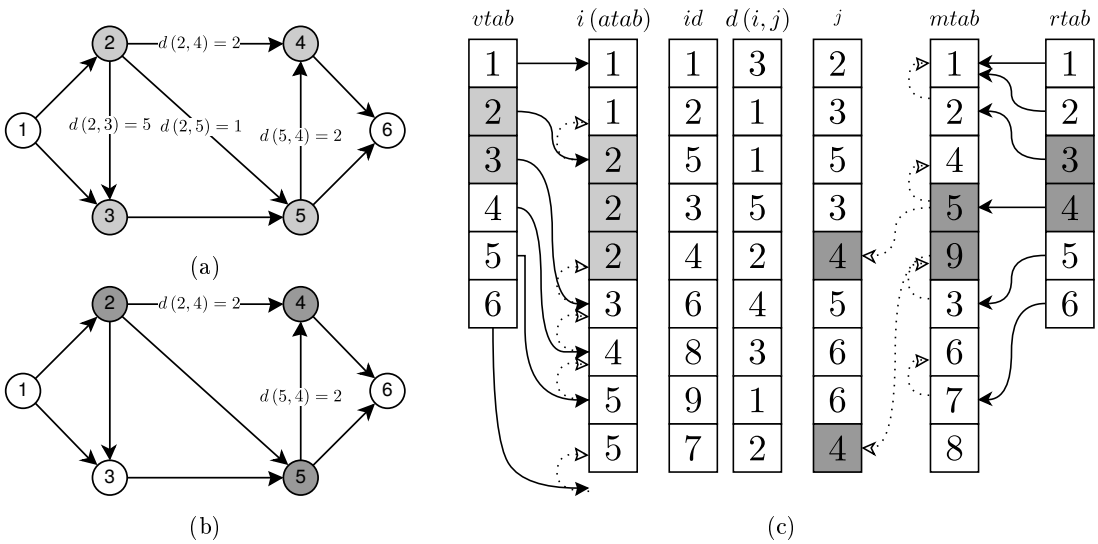
Druga z nich - *mtab* - jest tablicą wejściową dla metody, opisanej w sekcji "Pęki wejściowe". Przechowuje ona w swoich kolejnych elementach indeksy tablicy *atab* w taki sposób, aby ciąg $atab[mtab[1]], atab[mtab[2]], \ldots, atab[mtab[|E|]]$ tworzył ciąg identyfikatorów łuków, które są posortowane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

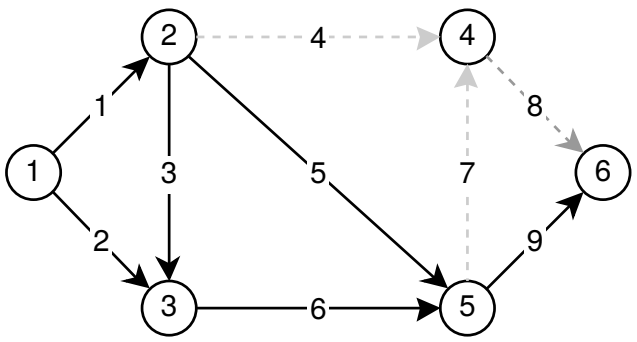
	Macierze		Listy	Pęki
	Incydencji	Sąsiedztwa	Sąsiedztwa	Wejścia-wyjścia
Dodawanie węzła	$O(\|V\| \cdot \|E\|)$	$O(1)$	$O(1)$	$O(\|V\| + \|E\|)$
Dodanie nowego węzła	$O(\|V\| \cdot \|E\|)$	$O\left(\|V\|^2\right)$	$O(1)$	$O(\|V\|)$
Przysłonięcie węzła	$O(\|V\| \cdot \|E\|)$	$O(\|V\|)$	$O(\|E\|)$	$O(\|V\|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

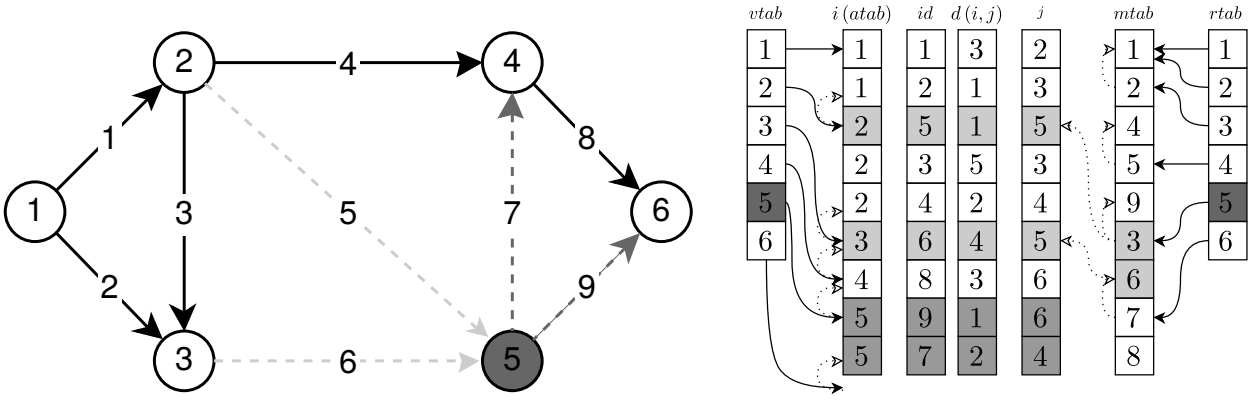
	1	2	3	4	5	6
1	0	1	2	0	0	0
2	0	0	3	4	5	0
3	0	0	0	0	6	0
4	0	0	0	0	0	8
5	0	0	0	7	0	9
6	0	0	0	0	0	0

(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.

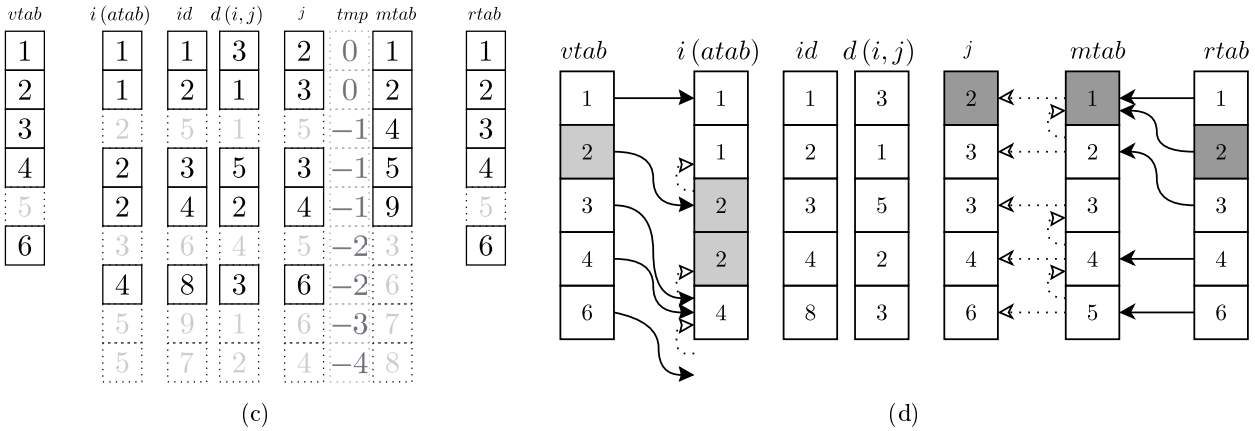
W przypadku pozostałych dwóch struktur znowu nie odnotujemy żadnej zmiany - aby usunąć dany węzeł z list sąsiedztwa będziemy musieli odnaleźć wszystkie łuki, które prowadzą do usuwanego węzła, a na koniec usunąć całą listę jego sąsiedztwa, wraz z łukami, które się na niej znajdują. W najgorszym przypadku 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 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 $v_S \overset{*}{\rightsquigarrow} v_i$, 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:

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

gdzie:

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

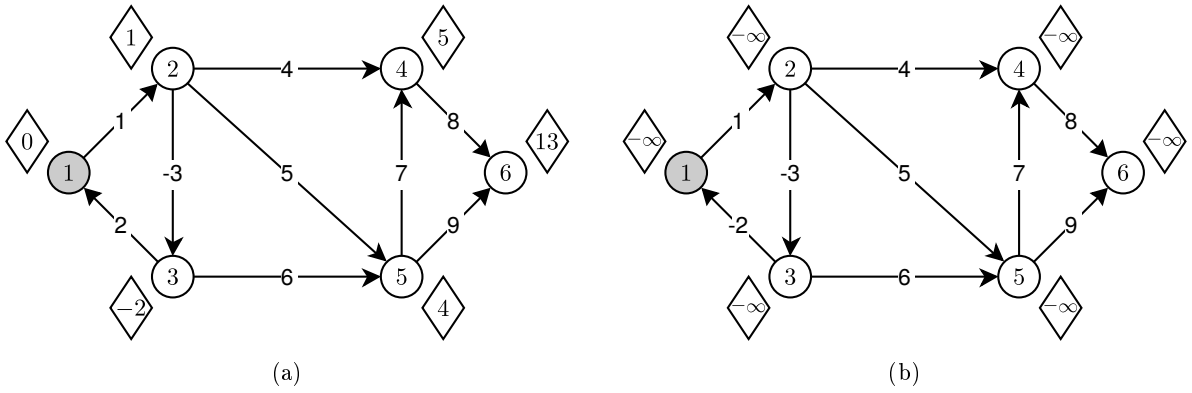
Do wszystkich atrybutów węzłów będziemy odwoływać się w dalszej części albo umieszczając jego nazwę w górnym indeksie, jak to robiliśmy do tej pory (np. vi^{ID} oznaczał identyfikator węzła i), albo pisząc ją za operatorem kropki w przypadku, gdy górny indeks będzie potrzebny nam do czego innego (np. zapis $v_i^{(k)}.\Pi$ będzie oznaczał k'tego rodzica [7] węzła v_i).

[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.

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 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 ($c(i,i) < 0$, 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 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, znanemu kosztowi ścieżki, prowadzącej ze źródła do danego węzła , co formalnie możemy w skróconej formie wyrazić:

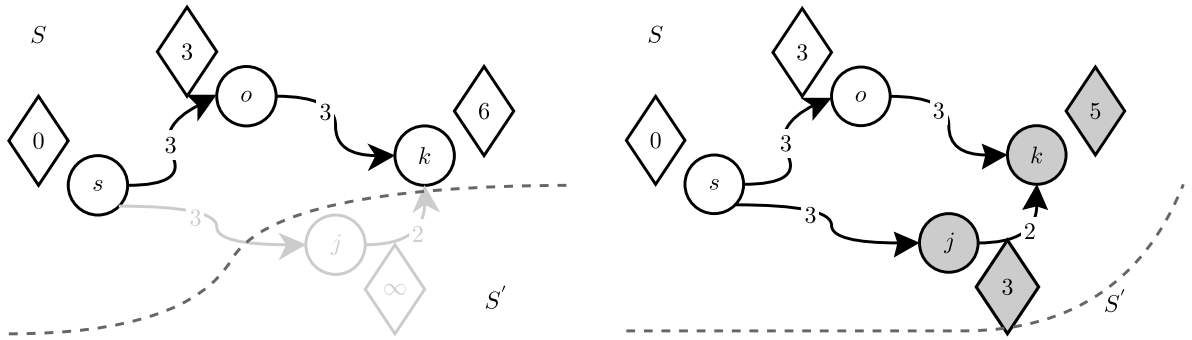
[8] Nie usuwamy pierwszego węzła cyklu, więc jeżeli cykl znajdowałby się na początku ścieżki to w oczywisty sposób węzeł v_p na ścieżce $v_p \overset{*}{\rightsquigarrow} v_k$ nie uległby zmianie. Analogicznie w przypadku, gdyby cykl znajdował się na końcu tj. $P = \langle v_0, v_1, \ldots, v_i, v_{i+1},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

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

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 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. 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

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

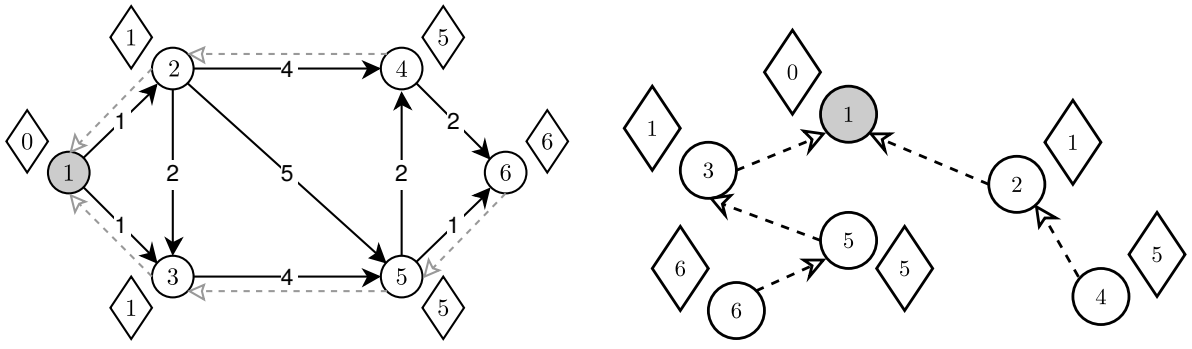
na rodzica d oraz wartość $d(i)$ dla wierzchołka, który tego wymaga. Metoda relaksacji przyjmuje trzy parametry, z którego dwa są wierzchołkami, a trzeci funkcją wagową krawędzi, którą zdefiniowaliśmy w podrozdziale 1.3.

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 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 czy też zastosować rekursję).

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 $\delta(v_k, v_j) \leqslant \delta(v_k, v_i) + c_{ij}$.*

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 $P = \langle v_p, v_{p+1}, \ldots, v_k \rangle$ to dla każdych i, j takich, że $p \leqslant i \leqslant j \leqslant k$ podścieżka $P_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ 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 $P = \langle v_p, v_{p+1}, \ldots, v_k \rangle$ z v_p do v_i, na której znajduje się cykl o ujemnej wadze $P = \langle v_i, v_{i+1}, \ldots, v_j \rangle$, gdzie $p \leqslant i \leqslant j \leqslant k$ to zawsze $v_k.d = \delta(p, k) = -\infty$*

Lemat 1.3.5 (Własność górnego ograniczenia) *Dla każdego wierzchołka $v_i \in V$ zachodzi $v_i.d \geqslant \delta(s, v_i)$, 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 $P = \langle v_p, v_{p+1}, \ldots, v_{k-1}, v_k \rangle$ 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 $P = \langle v_p, v_{p+1}, \ldots, v_k \rangle$ 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 równych rzeczywistym wagom najkrótszych ścieżek $v_s \overset{*}{\rightsquigarrow} v_i$ - 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.

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 FALSE
8     return TRUE
```

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 instrukcje (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 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 $C = \langle v_0, v_1, \ldots, v_k \rangle$, 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 $v.d \leqslant u.d + c_{uv}$ 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 1.3.1). Dokładniej:

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

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 $\forall_{v_i \in C}\ v_i.d \leqslant v_{i-1}.d + c_{v_{i-1}v_i}$, gdzie sumując po $v_1, \cdots, v_k \in C$ otrzymujemy:

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

.

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

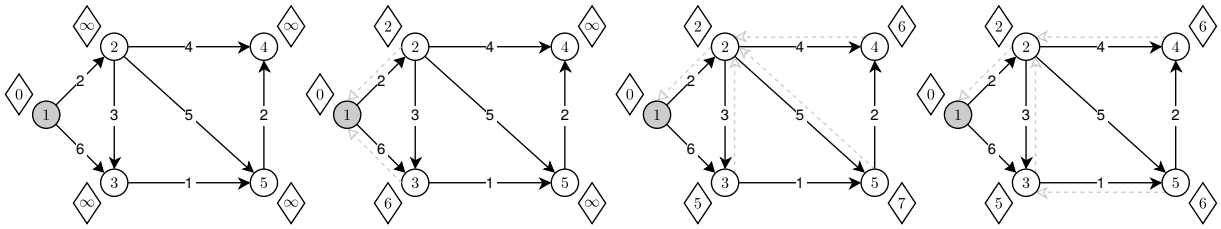
$$\sum_{i=0}^{k} v_i.d = v_0.d + \sum_{i=1}^{k-1} v_i.d + v_k.d = \sum_{i=1}^{k} v_{i-1}.d + v_k.d = v_0.d + \sum_{i=1}^{k} v_i.d \tag{1.13}$$

zaś z obu wyprowadzeń w ostateczności otrzymujemy:

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

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 pierwowzoru. 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 $(1, 3)$ 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.

Usprawnienie algorytmu

Pierwsze co możemy zauważyć to fakt, że jeśli uporządkujemy wszystkie krawędzie e_{ij} względem pierwszego indeksu (czyli tak, aby kolejność przeglądania łuków w wierszach 3-4 była podyktowana przeglądanymi 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 1.3.3 wiemy, że jeśli w grafie istnieje najkrótsza ścieżka $P = \langle v_p, v_{p+1}, \ldots, v_k \rangle$ 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 $i = 1, 2, \cdots |V| - 1$ 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, przynajmniej 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.

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** narzuci 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 okaże 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 rozpoczęcie ich spisywania od tych, do których nie prowadzą żadne krawędzie - powiedzmy, że takie wierzchołki

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:

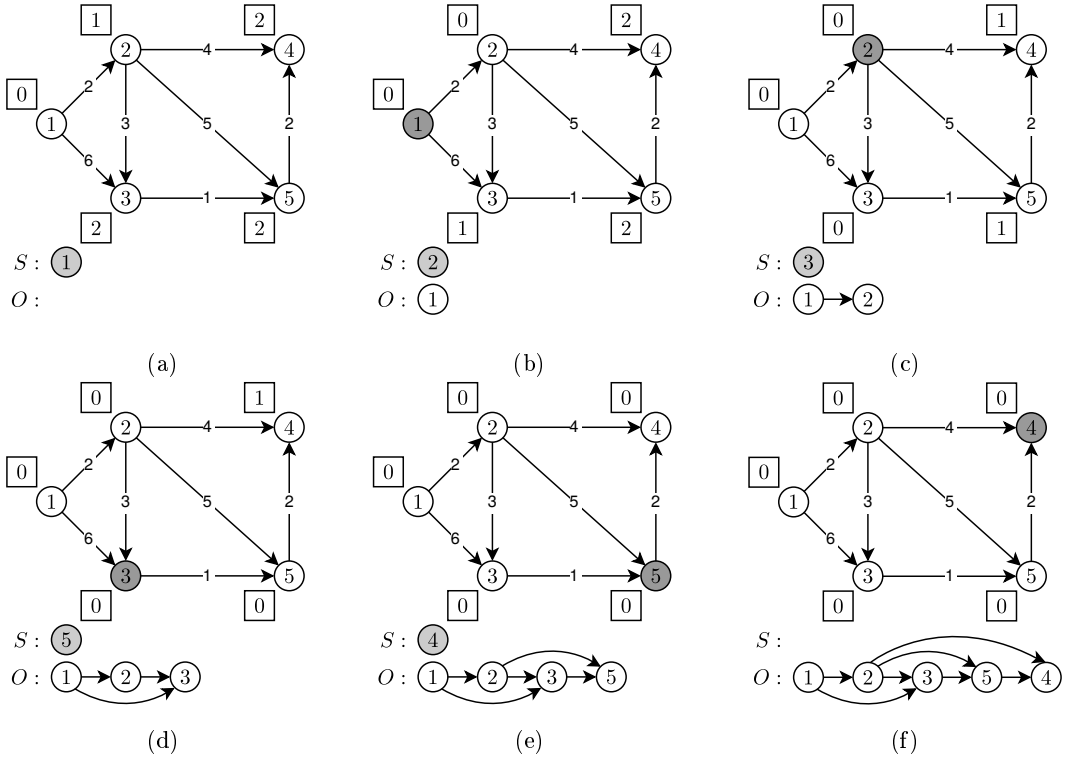
Algorithm 6: KHAN-TOPOLOGICAL-SORT (G)

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

```
1  begin
2      S ⟵ zbiór wszystkich wierzchołków, do których nie prowadzą żadne krawędzie
3      O ⟵ ∅
4      while S nie jest pusta do
5          Przepnij wierzchołek v_i z S na koniec listy O
6          foreach e_ij : v_i ⇝(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 $deg\,[1 \cdots |V|]$ informację o liczbie łuków, wchodzących do poszczególnych węzłów o identyfikatorach z zakresu od 1 do $|V|$ (przy czym nie interesuje nas nic poza ich liczebnością). 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 sytuacje, 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 pętlę, 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 długości $|V|$, podczas gdy dla omawianego przypadku złego zachowania się algorytmu bardzo szybko liczba tych elementów przekroczy ich oczekiwaną 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 jedyny możliwy 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 $v_i \overset{1}{\rightsquigarrow} v_j$ 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 wcześniej 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

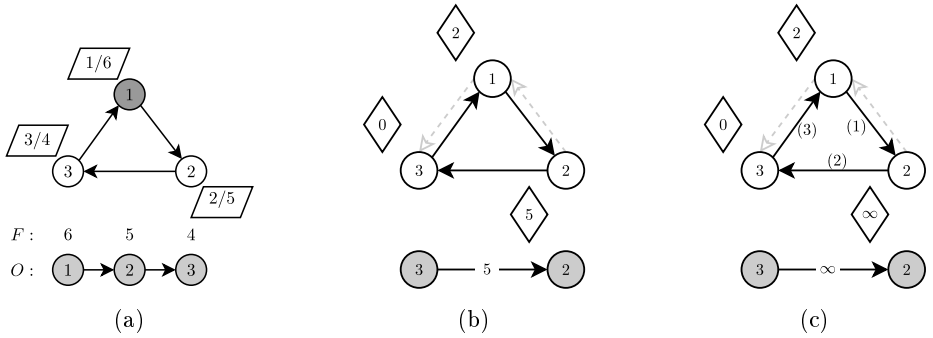
Alternatywnym sposobem na topologiczne uporządkowanie grafów w sieci jest wykorzystanie właściwości, posiadanych przez prosty algorytm przeszukiwania w głąb, w skrócie DFS (ang. *Depth-First Search*). Aby posortować topologicznie wszystkie węzły w gafie $G = (V, E)$ wykorzystujemy fakt, że wspomniany 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 interpretacje: żaden węzeł v_i nie zostanie zaznaczony, gdy istnieje jakikolwiek węzeł v_j, który jest jeszcze nie zaznaczony, a który posiada krawędź $v_j \overset{1}{\rightsquigarrow} v_i$).

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 tę przewagę, że nie przerywa pracy nawet w momencie napotkania cyklu. Nie jest to do końca prawdą, gdyż zachowanie się algorytmu DFS głównie zależy od intencji jego autora, lecz możemy napisać go w taki sposób, aby przeglądając graf wgłąb, 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 wgłąb). Innymi słowy - możemy go zmusić by ignorował cykle, występujące w badanym grafie.



(a) (b) (c)

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 $v_3 \overset{1}{\rightsquigarrow} v_2$. $c(3,2) = \delta(3,2) = 5$. **(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 zamykającą 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, zwracał nam niepoprawne wyniki.

2.1.3 Sortowanie topologiczne

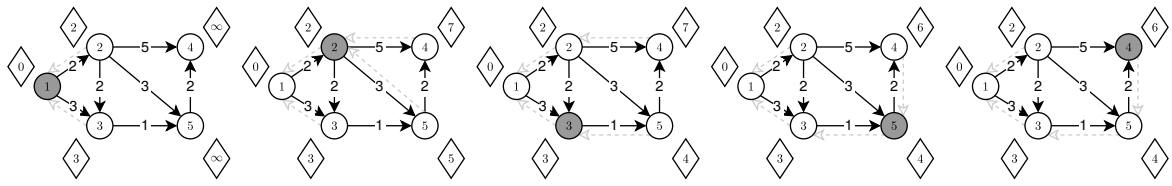
Jak widzimy z powyższych przykładów, możliwości wykorzystania sortowania topologicznego w algorytmach wyszukiwania najkrótszych ścieżek są ograniczone jedynie do małej klasy grafów - stanowczo za małej, jeżeli chodzi o grafy, reprezentujące rzeczywiste sieci drogowe. Istnieją jednak pewne problemy (mniej 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 ⇝(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 (1.3.3) oraz o własności zbieżności dla najkrótszych ścieżek (1.3.3).

Dowód. Załóżmy indukcyjnie, że algorytm przeskanował już wierzchołki $v_i : i \in \{1, 2, \cdots, k\}$ 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 $P = \langle v_1, v_2, \ldots, v_h, k+1 \rangle$. Z lematu 1.3.3 (o optymalnej podstrukturze) wiemy, każda podścieżka ścieżki P jest najkrótszą ścieżką, w szczególności jest nią ścieżka $P' = \langle v_1, v_2, \ldots, v_h \rangle$. Z faktu, że wszystkie wierzchołki w grafie są posortowane topologicznie oraz, że krawędź $v_h \overset{1}{\rightsquigarrow} k+1 \in E$ (co nam gwarantuje istnienie ścieżki P) wynika, że wierzchołek v_h jest węzłem, dla którego $h \in \{1, 2, \cdots, k\}$ w związku z czym, na mocy założenia indukcyjnego, wartość $v_h.d = \delta(s, v_h)$. Zgodnie z lematem 1.3.3, jeżeli w dowolnym momencie przed relaksacją krawędzi $v_h \overset{1}{\rightsquigarrow} k+1$ 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 $v_h \overset{1}{\rightsquigarrow} k+1$, 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 oczekiwań, 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

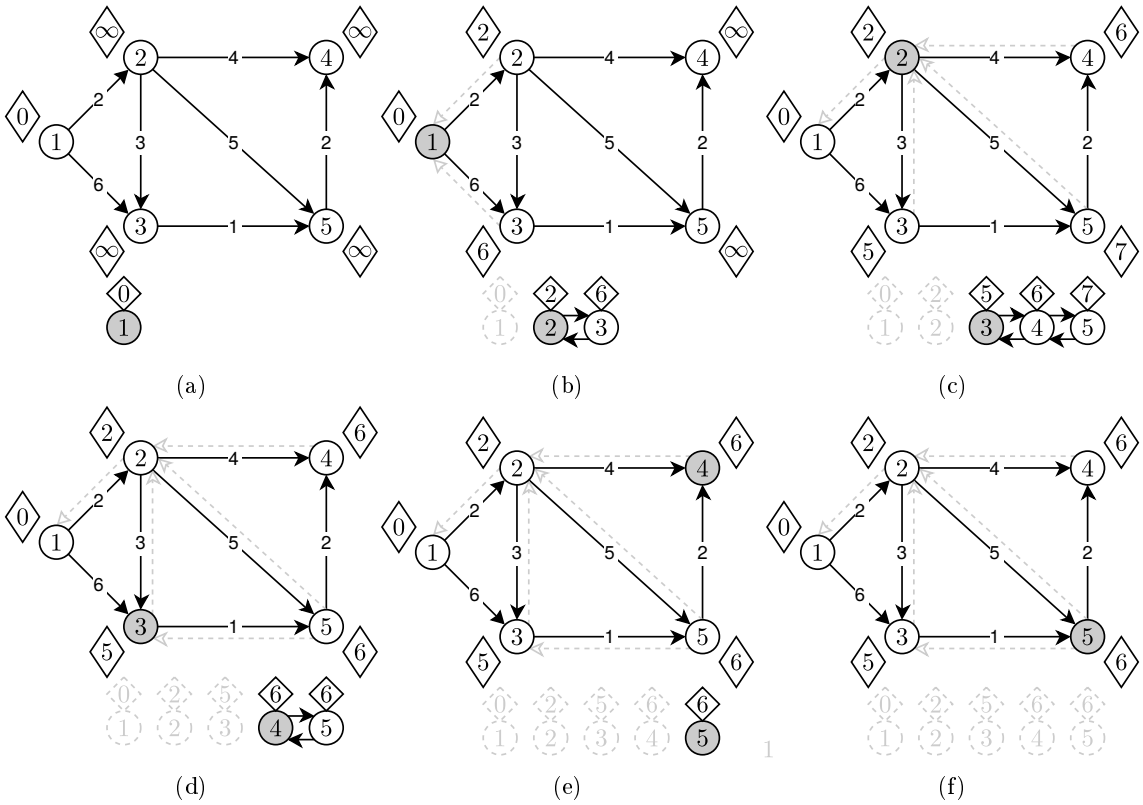
Sama idea algorytmu jest bardzo podobna do poprzedniej tj. zakłada wykonywanie relaksacji dla wszystkich krawędzi aktualnie badanych wierzchołków, których to kolejność jest w specyficzny sposób ustalana. W przypadku algorytmu opartego na sortowaniu topologicznym grafu był to właśnie ten porządek, który 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 $\overline{s}$, w którym na początku przechowywane są wszystkie węzły. Jak już wspomnieliśmy, algorytm ma za zadanie sekwencyjne pobierać ze zbioru $\overline{S}$ takie wierzchołki v_i, by $v_i.d = \min\{v_j.d : v_j \in \overline{S}\}$, 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-GRAPH (G, s). Pseudokod generycznego algorytmu Dijkstry wygląda tak, jak przedstawiono poniżej.

Algorithm 9: GENERIC-DIJKSTRA (G, s)

```
1 begin
2     S ⟵ ∅
3     S̄ ⟵ {v : v ∈ V}
4     while S̄ nie jest pusty do
5         v ⟵ v_i : v_i.d = min{v_j.d : v_j ∈ S̄}
6         S ⟵ S ∪ {v}
7         S̄ ⟵ S̄ − {v}
8         foreach e_ij : v_i ⇝(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 $\overline{S}$ zajmuje się, wymieniona wyżej, operacja (wtedy $\overline{S} = Q$). 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-GRAH ze źródłem $v_s.id = 1$. **(b)** Z listy dwukierunkowej został 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.

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 się [1]. Następnie, wewnątrz 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 $O\left(\text{EXTRACT-MIN}\left(Q\right)\right)$. Takich operacji podczas działania algorytmu wykonamy $|V|$. Dla każdego węzła, wyjętego z kolejki, dla wszystkich łuków, wychodzących z danych węzłów, wykonywana jest operacja relaksacji - łatwo zauważyć, 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 zależna od implementacji kolejki priorytetowej (wraz ze zmniejszeniem się klucza może zajść konieczność przemieszczenia węzła bliżej głowy kolejki) i dla naszej analizy niech wyniesie $O\left(\text{DECREASE-KEY}\left(Q, v, k\right)\right)$, 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

[1] Ilość 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 zostały, spełniają $u.d \geqslant v.d = \delta\left(s, v\right) = \infty$.

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

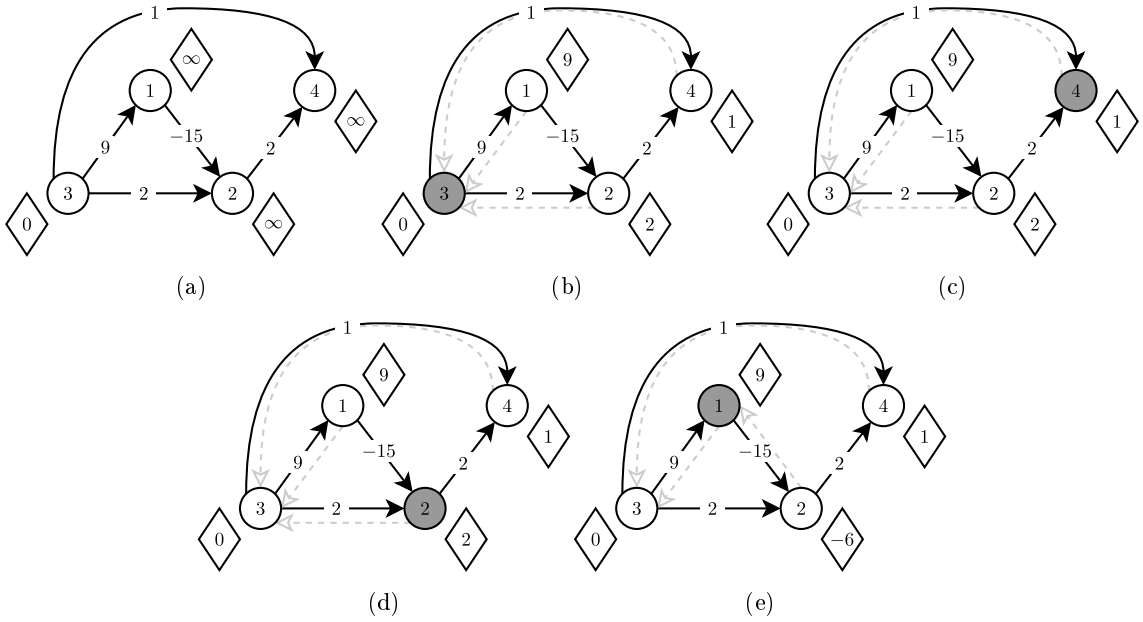
$$O\left(|E| \cdot O\left(\text{DECREASE-KEY}\left(Q, v, k\right)\right) + |V| \cdot \left[O\left(\text{INSERT}\left(Q, v\right)\right) + O\left(\text{EXTRACT-MIN}\left(Q\right)\right)\right]\right) \quad (2.1)$$

.

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.

Ujemne koszty krawędzi

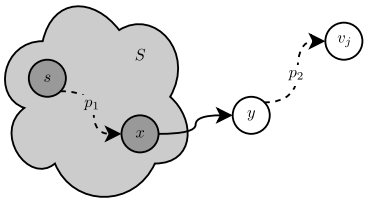
Nim udowodnimy poprawność algorytmu Dijkstry przeanalizujmy jeszcze prosty przykład, w którym dopuścimy wystąpienie krawędzi o ujemnym koszcie i pokażmy, że dla takiego grafu nasz algorytm zwróci błędny wynik.



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

Jak powiedzieliśmy, algorytm Dijkstry analizuje wierzchołki grafu $G = (V, E)$ w ściśle 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 $v_3 \overset{*}{\rightsquigarrow} v_4$ jest ścieżka $P = \langle v_3, v_1, v_2, v4 \rangle$) ze względu na wystąpienie w grafie krawędzi o ujemnym koszcie.

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. Załóżmy 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 zbiór 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 $v_s \overset{p_1}{\rightsquigarrow} x \overset{1}{\rightsquigarrow} y \overset{p_2}{\rightsquigarrow} v_j$ 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 1.3.3) mamy, że zachodzi również $y.d = \delta(s, y)$ (w trakcie dodawania wierzchołka x do zbioru S zajdzie relaksacja krawędzi $x \overset{1}{\rightsquigarrow} y$, 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 $\delta(s, y) \leqslant \delta(s, v_j)$, co prowadzi do szeregu nierówności:

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leqslant \delta(s, v_j) \\ &\leqslant v_j.d \ \text{(z lematu 1.3.3 o górnym ograniczeniu)} \end{aligned} \tag{2.2}$$

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 $v_j.d \leqslant y.d$. 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: INSERT (Q, v), EXTRACT-MIN (Q) i 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 (2.2.1) natychmiastowo uzyskamy górne ograniczenie na poziomie $O\left(|V|^2\right)$, 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 INSERT (Q, v), EXTRACT-MIN (Q) i 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 przepełnieniem oraz szereg innych pomysłów, opartych o kontenery, zwane dalej kubełkami (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: $v.d \leqslant v_i : v_i.\Pi = v$). 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 tabelka przedstawia koszty poszczególnych operacji dla kopca binarnego i R-arnego:

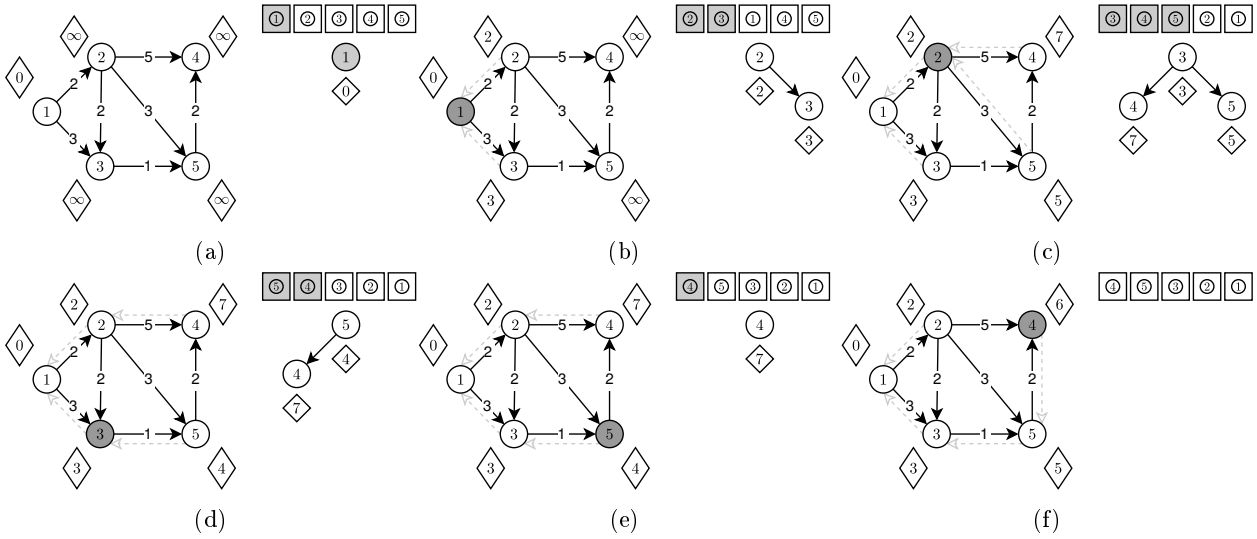
	Kopiec	
Operacja	binarny	R-arny
INSERT (Q, v)	$O(\log(n))$	$O(\log_R(n))$
EXTRACT-MIN (Q)	$O(\log(n))$	$O(R \cdot \log_R(n))$
DECREASE-KEY (Q, v, k)	$O(\log(n))$	$O(\log_R(n))$

Implementacja

Algorytm wyszukiwania najkrótszych ścieżek oparty na strukturze D-arnego kopca jest jedynym algorytmem, 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 $vertices\,[1 \cdots k]$. W takiej sytuacji, jeżeli dowolny wierzchołek v znajduje się na pozycji $i <= k$ 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 $heapIDArray\,[1 \cdots |V|]$, której to wartości będą odzwierciedlać faktyczne rozmieszczenie wierzchołków w tab po modyfikacjach ($tab\,[heapIDArray\,[i]] = v_i$), 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 "wypychamy"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, przedstawionego 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 $tab\,[k] = v_2$), zaś na stare miejsce wstawionego węzła zostaje przeniesiony węzeł v_1. Analogicznie na koniec kopca wstawiany jest v_3 ($k = 2$, $tab\,[k] = v_3$) w wyniku czego $tab\,[3] = v_1$. **(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 2.2.1 na ogólną złożoność generycznego algorytmu Dijkstry z wykorzystaniem kolejek priorytetowych dla kopców R-arnych natychmiast otrzymujemy $O\left(m \cdot \log_d(n) + n \cdot [\log_d(n) + d \cdot \log_d(n)]\right) = O\left(m \cdot \log_d(n) + n \cdot d \cdot \log_d(n)\right)$ (dla przejrzystości zapisu przyjęliśmy $n = |V|$ i $m = |E|$). Dla kopca binarnego mamy: $O\left(m \cdot \log(n) + n \cdot \log(n)\right) = O\left(m \cdot \log(n)\right)$ dla $m \geqslant n$ (stałe wyeliminowaliśmy). Przypomnijmy sobie, że naiwna implementacja algorytmu Dijkstry miała złożoność $O\left(m + n^2\right) = O\left(n^2\right)$. Łatwo zauważyć, że dla bardzo gęstych grafów (gdzie $m = \Omega\left(n^2\right)$) 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 $O\left(m \cdot \log(n)\right) \leqslant O\left(\frac{n^2}{\log(n)} \cdot \log(n)\right) = O\left(n^2\right)$).

Z kolei dla kopców, których arność jest większa ($d \geqslant 2$) mamy: $O\left(m \cdot \log_d(n) + n \cdot d \cdot \log_d(n)\right)$. Z tego bezpośrednio wynika, że optymalną wartością parametru d jest $\max\left\{2, \left\lceil \frac{m}{n} \right\rceil\right\}$, 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 $m = \Omega\left(n^2\right)$ mamy: $O\left(m \cdot \log_d(n)\right) = O\left(n^2 \cdot \log_{\frac{n^2}{n}}(n)\right) = O\left(n^2 \cdot \log_n(n)\right) = O\left(n^2\right)$), albo do opartego na kopcu binarnym (w przypadku, gdy sieć jest bardzo rzadka). Dla tej drugiej możliwości otrzymujemy natychmiastowo złożoność $O\left(n \cdot \log(n)\right)$ dla $m = (n)$.

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

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

Jeśli $\epsilon = 1$ to $m = \Omega\left(n^2\right)$, 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 tablicy 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 ściśle 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:

	Drzewo		Kopciec	
Operacja	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 (2.2.1):

$$O(m \cdot O(\mathrm{DK}(Q, v, k)) + n \cdot [O(\mathrm{I}(Q, v)) + O(\mathrm{EM}(Q))]) \tag{2.5}$$

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 kopców binarnych i R-arnych. Na podstawie powyższego zestawienia możemy podejrzewać, że struktura zbalansowanych drzew binarnych jest w pewnym stopniu konkurencyjna dla kopców tej samej arności [3], 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 kopców 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

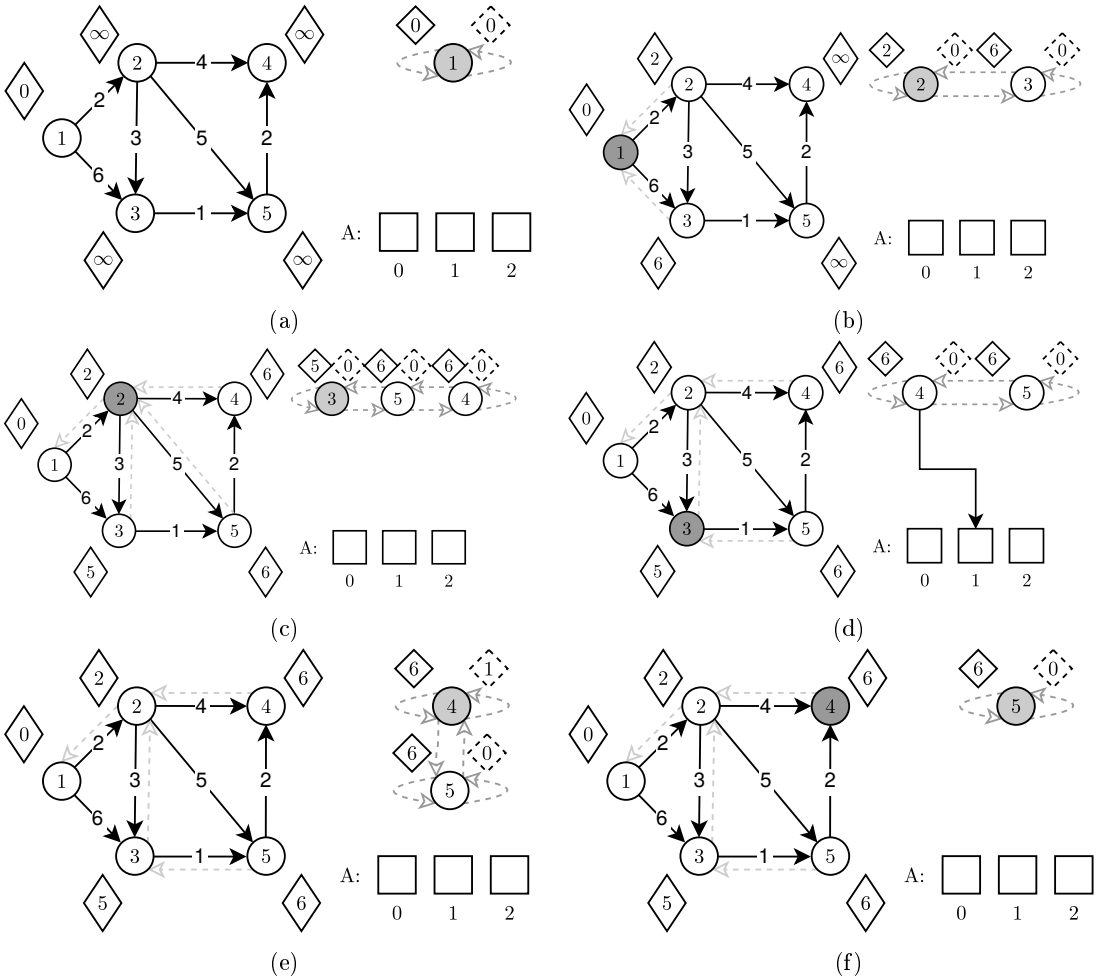
Przedstawiona w tym rozdziale implementacja algorytmu Dijkstry jako kolejkę priorytetową będzie wykorzystywać jedną z bardziej złożonych struktur danych, jakie będziemy omawiać - kopce Fibonacciego. Zaletą jej wykorzystania okaże się amortyzacyjnie lepszy czas wykonywania dla dwóch, podstawowych operacji, 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,

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

[3] Jeżeli 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 $A\left[1 \cdots \lfloor \log_{\Phi}(n) \rfloor\right]$, którą ćzyścimy"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 w 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 żozerwaćóbie 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 $u.\Pi = v$ ustawia te parametry na wartość ***NULL***. Podkreślić należy słowo: "tymczasowo" , gdyż ta czynność zostanie wykonana w momencie rekonstrukcji kopca Fibonacciego z drzew, przechowywanych w tablicy $A\left[1 \cdots \lfloor \log_{\Phi}(n) \rfloor\right]$ - 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) (b) (c) (d) (e) (f)**

2.5 Struktury oparte na kubełkach

nC+1 kubełków, nic ciekawego
brak rysunku, bo za duży

2.5.1 Z przepełnieniem

budujemy a < C + 1 main bucketów i overflow. każdy main bucket ma szerokość 1. Ogólnie main: $[a(i), a(i) + a - 1]$. Skanujemy main buckety i wykonujemy relaksacje, dorzucając do mainów lub overflow. Gdy dojdziemy do końca maina to aktualizujemy ich szerokość $a(i) = min_overflow$ i przenosimy z overflowu do main wszystkie [a(i),a(i)+a-1] i reskanujemy mainy od 0 do a. Jeśli przeskanowaliśmy wszystkie i overflow pusty to koniec.

2.5.2 Dial

skanujemy wszystkie kubełki w kółko. C+1 kubełków

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??).

2.5.4 Kubełki wielopoziomowe

Mamy k kubełków niskiego poziomu, każdy o szerokości 1. Mamy floor(n*C/k) + 1 kubełków wysokiego poziomu, każdy szerokości [k*i,k*(i+1)-1] dla każdego i. Np. dla n=7, C=2 i k = 5 mamy 3 wysokich (0..2) i ostatni ma [10,14]. Dla ceil(nC/k) i nC=15 i k = 5 mamy też 0..2 i jest źle.

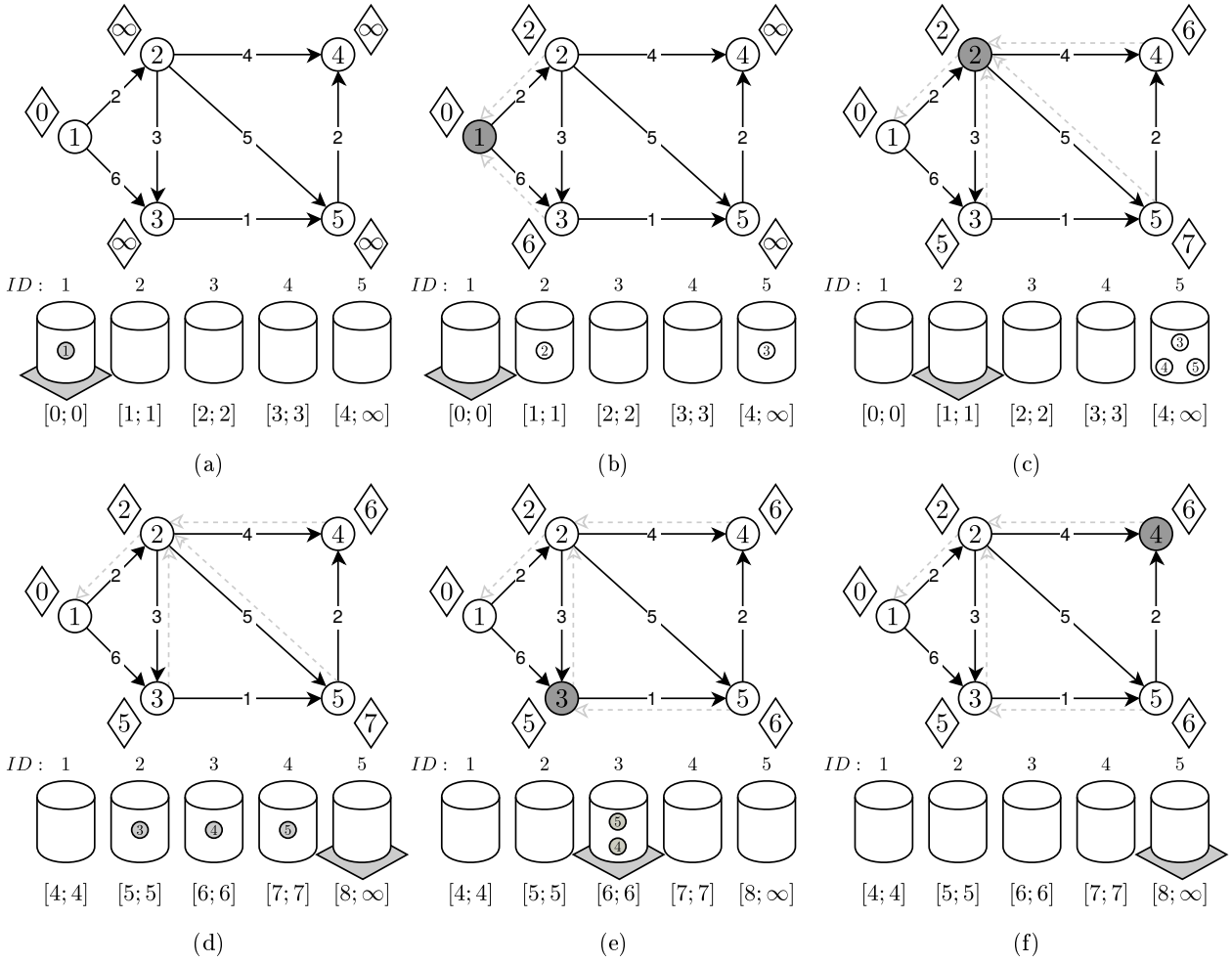
Skanujemy po kolei wysokie, przerzucając z każdego po kolei do niskiego poziomu. Skanujemy niski poziom, gdy jest już pusty, to idziemy do następnego wysokiego. Przenosimy z niego do niskich. I tak dalej.

2.5.5 Kopce pozycyjne

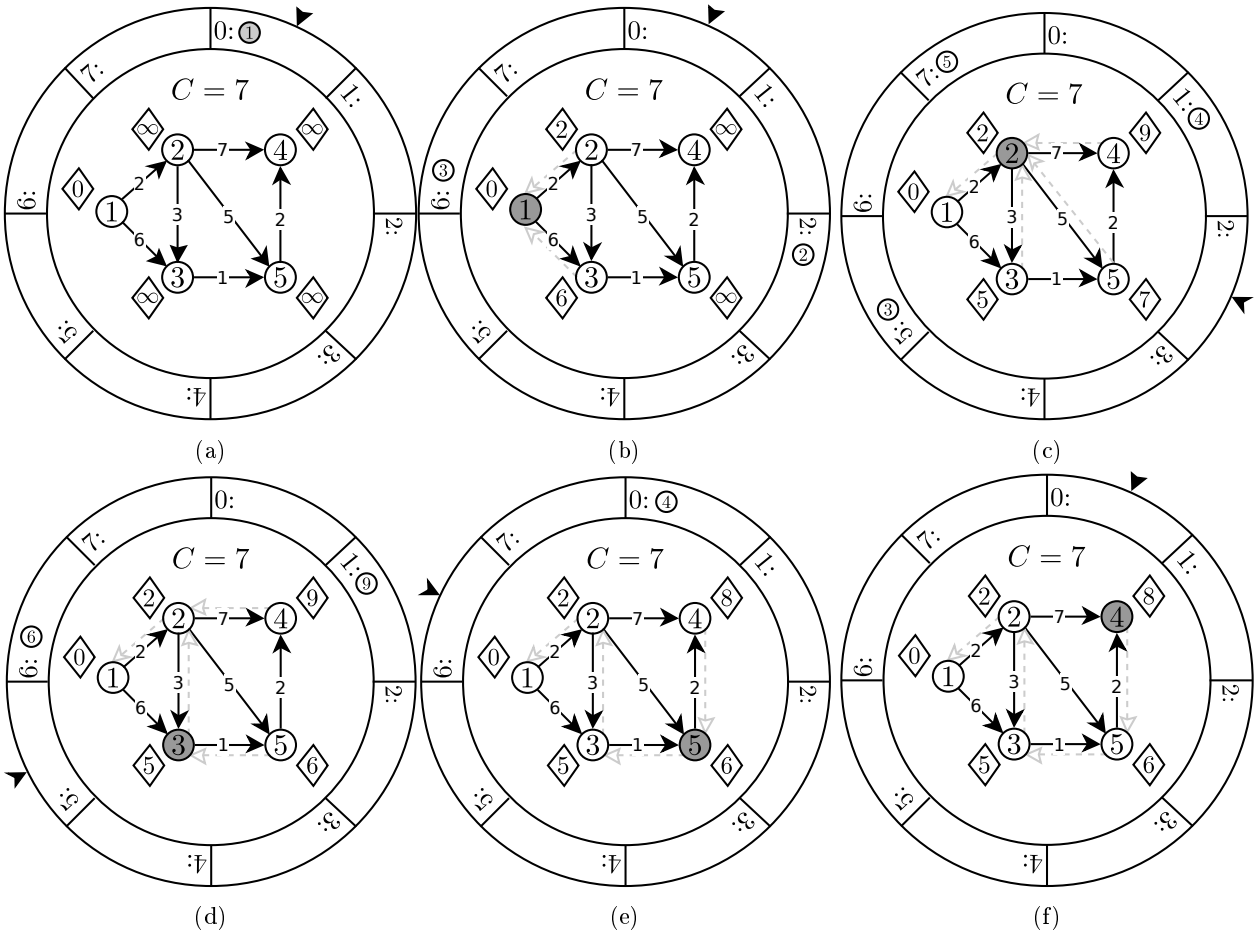
nC

Mamy n*C kubełków (floor(log2(numberOfNodes)+log2(maxCost)) + 2 dokładnie), gdzie każdy zbiera nody o dystansie: $[0] = [min + 0; min + 0][1] = [min + 2^(i - 1); min + 2^i - 1] = [min + 1; min + 1][2] = ...[n * c][...; <= n * C]$

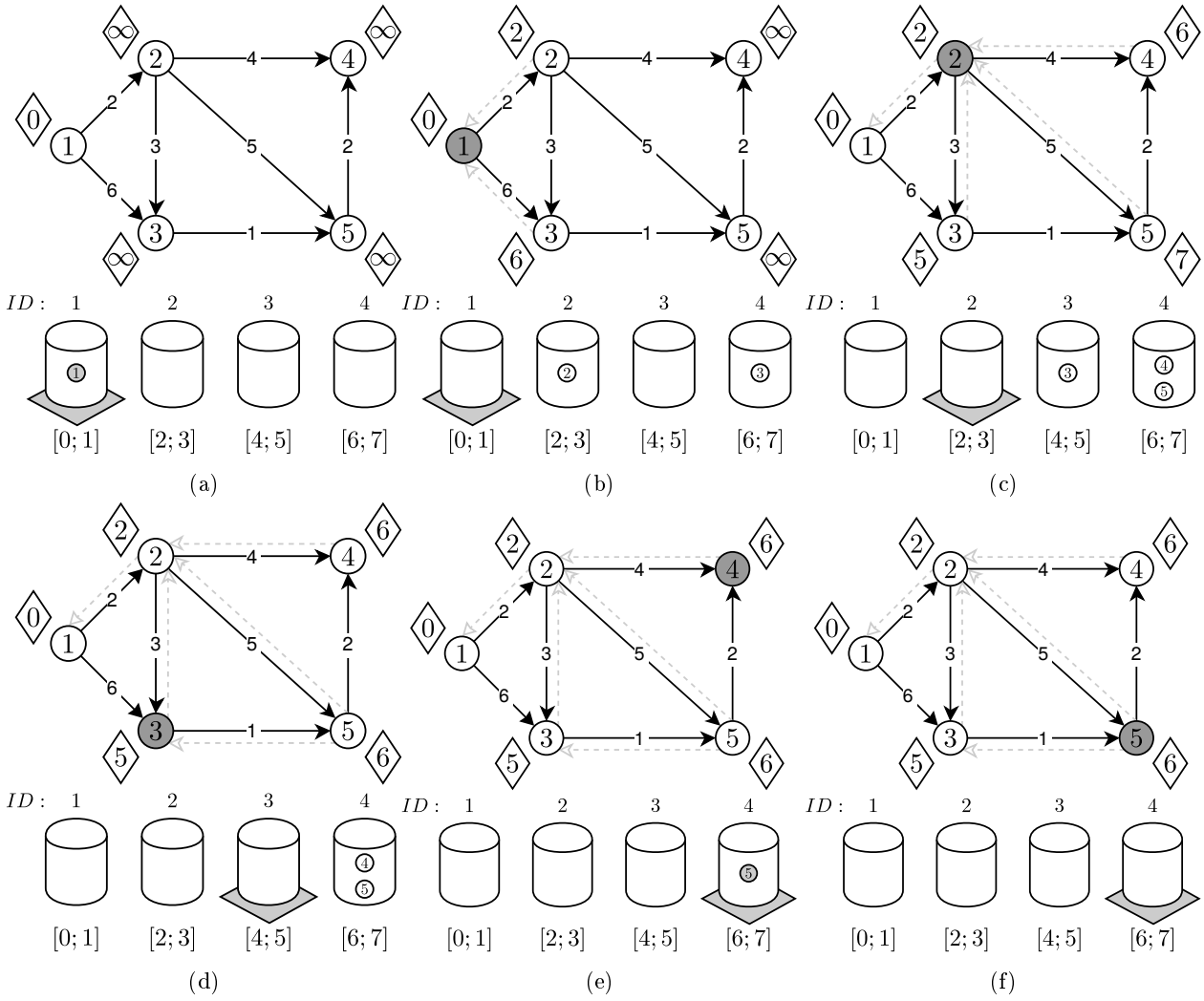
Skanujemy je po kolei i jeśli ma długość = [k;k] albo jest tylko 1 node to wyciągamy nody z kubełka i robimy normalną relaksację (przed tym wyciągamy minimalny element - jeśli [k;k] lub to wyciągamy pierwszy lepszy, jeśli nie to przyda nam się to dalej). Jeśli w buckecie jest więcej węzłów o różnych etykietach to mamy już wyciągnięte minimum - min będzie się równać etykiecie tego minimalnego węzła, a pozostałe od 0 do j (gdzie badamy j'ty kubełek) będą zmienione, zachowując swoje pojemności (tylko min się zmieni, a pojemności trzymamy w tablicy osobno). J'ty kubełek zostanie zmieniony o +1, tak, aby j-1 miał [...;u[j]-1], [u[j];u[j]], a następny miał już naturalnie u[j]+1 - czyli teraz będzie się stykać. Realocujemy potem wszystkie węzły z j'tego kubełka do niższych. Ich nowych kubełków szukamy od j do 0, gdyż 1) od wyższych jest lepiej, bo mają większe zakresy i większa szansa, że się szybko zatrzymamy, 2) możemy mieć węzeł, który nawet po realokacji zostanie w j'tym kubełku.



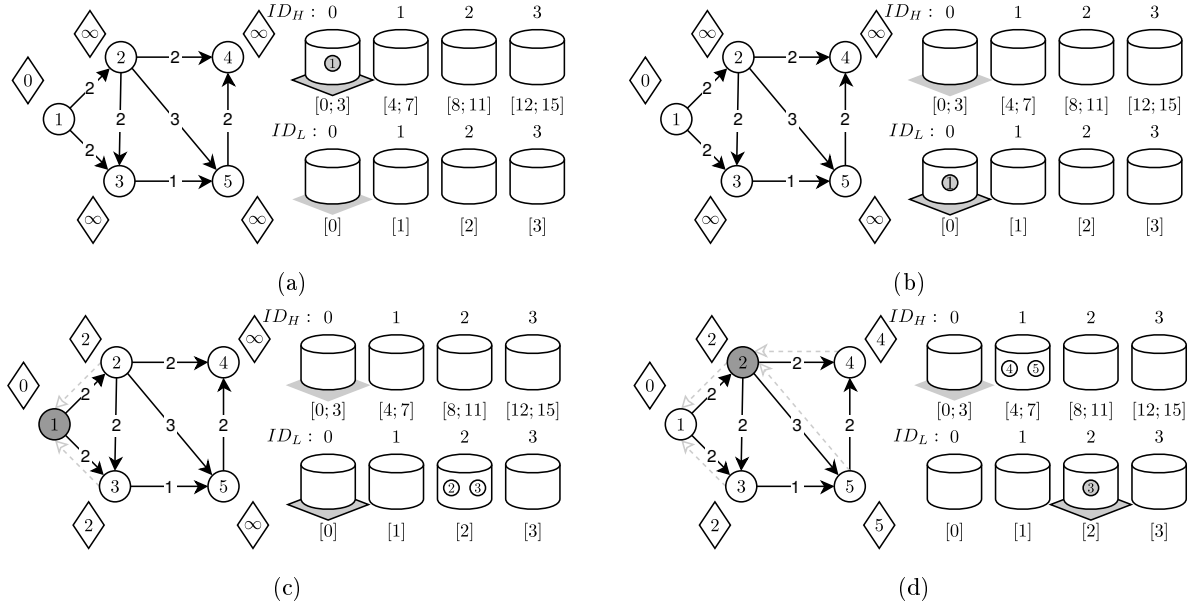
Rysunek 2.9: **Działanie algorytmu Bellmana-Forda (a)** Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAPH ze źródłem $v_s.id = 1$. **(b)** Warunek $v.d > u.d + c_{uv}$ dla krawędzi (u, v) spełniony jest tylko dla krawędzi: $(1, 2)$ i $(1, 3)$ i dla tych węzłów (v_2 i v_3) zostały zaktualizowani 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.



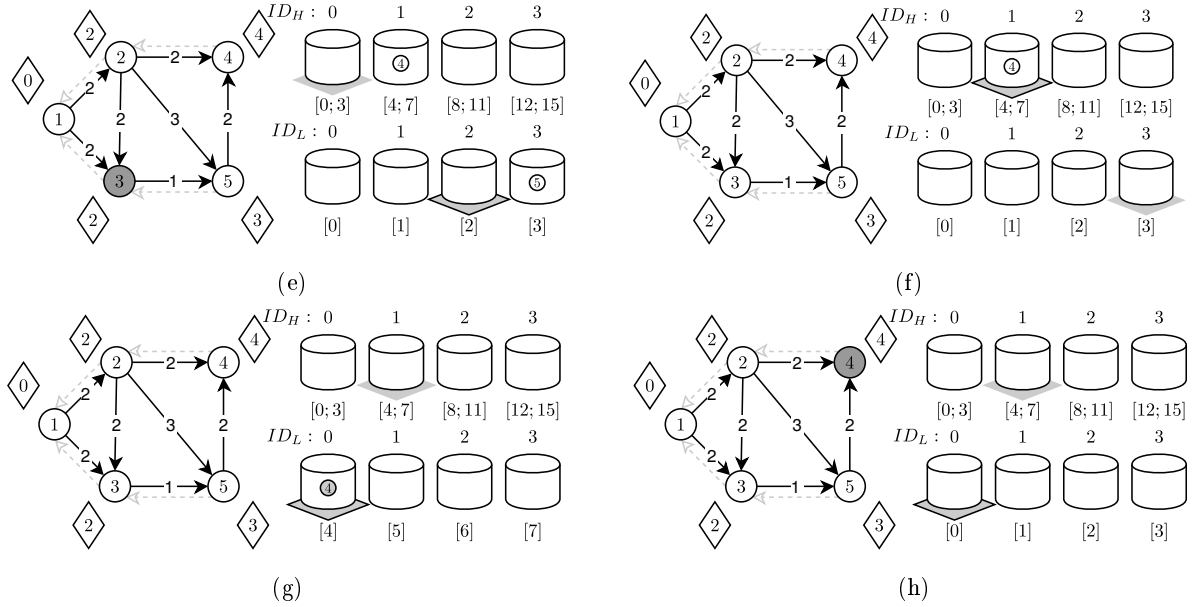
Rysunek 2.10: **Działanie algorytmu Bellmana-Forda (a)** Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAPH ze źródłem $v_s.id = 1$. **(b)** Warunek $v.d > u.d + c_{uv}$ dla krawędzi (u, v) spełniony jest tylko dla krawędzi: $(1, 2)$ i $(1, 3)$ i dla tych węzłów (v_2 i v_3) zostały zaktualizowani 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.



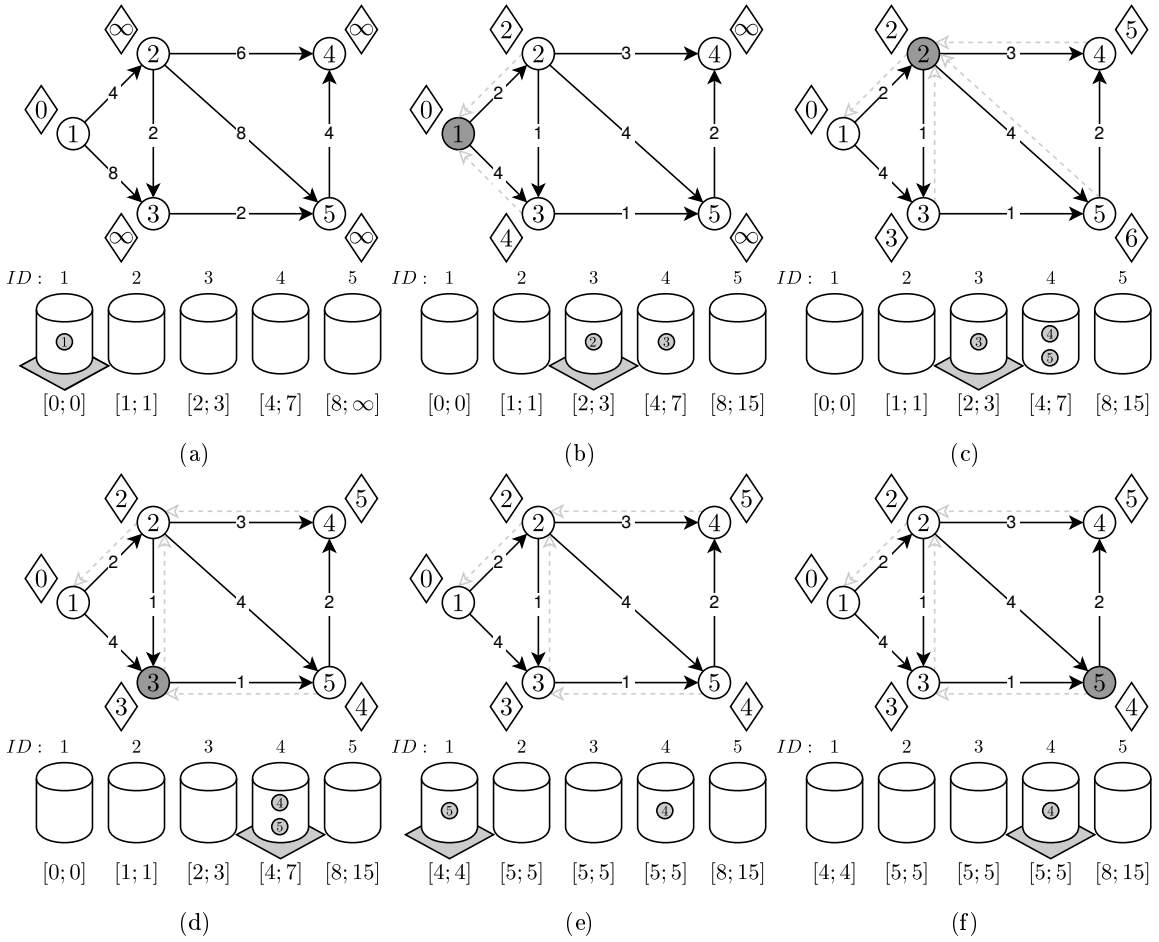
Rysunek 2.11: **Działanie algorytmu Bellmana-Forda (a)** Sytuacja po zainicjowaniu grafu $G = (V, E)$ przez INIT-GRAPH ze źródłem $v_s.id = 1$. **(b)** Warunek $v.d > u.d + c_{uv}$ dla krawędzi (u, v) spełniony jest tylko dla krawędzi: $(1, 2)$ i $(1, 3)$ i dla tych węzłów (v_2 i v_3) zostały zaktualizowani 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 po wszystkich krawędziach, jednak wprowadziliśmy modyfikację, która przerywa działanie algorytmu, jeżeli podczas pełnej iteracji nie nastąpi w grafie G żadna zmiana.



Rysunek 2.12: **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.



Rysunek 2.12: **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.



Rysunek 2.13: **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.

C

Mamy C+1 kubełków (ceil(log2(maxCost+1)) + 2 dokładnie), gdzie każdy zbiera nody o dystansie:
$[0] = [min + 0; min + 0][1] = [min + 2^(i - 1); min + 2^i - 1] = [min + 1; min + 1][2] = ...[c][...; ...]$
a ostatni zbiera wszystko ponad. Ide jest taka, by miał n*C jako max i u[c]+1 jako minimum, lecz chcemy uniknąć problemu z poprzedniej implementacji: dla n*C dla największych testów (23947347 węzłów i 58333344 połączeń) liczba n*C była aż 51 bitowa, co nakładało konieczność użycia takowych w całym algorytmie, zwiększając potrzebną pamięć (same łuki to 26 bitów). Tutaj umowa taka, że overflow to kubełek widmo, który definiujemy inaczej (podobnie jak w alg. z faktycznym overflowem).

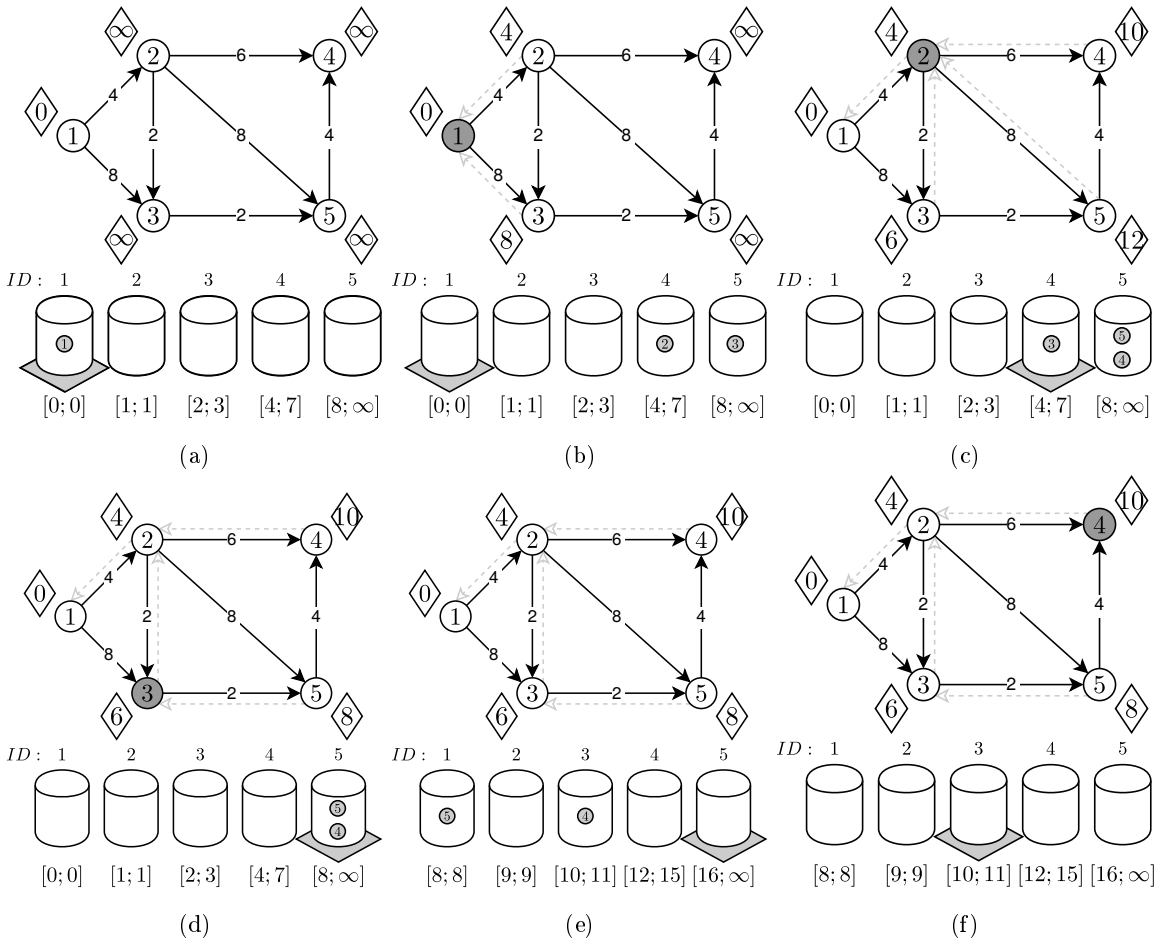
Skanujemy je po kolei i jeśli ma długość = [k;k] albo jest tylko 1 node to wyciągamy nody z kubełka i robimy normalną relaksację, JEŚLI bucket to nie jest overflow - z niego z definicji tylko robimy realokacje. (przed tym wyciągamy minimalny element - jeśli [k;k] lub to wyciągamy pierwszy lepszy, jeśli nie to przyda nam się to dalej - TU ZMIANA w odniesieniu do porzedniego - wyciągamy już konkretny element w środku if'a lub elsa, co oszczędza 1 if'a, a kolejność zbudowania warunku jest taka, by były sprawdzane warunki od najczęściej występującego, do najrzadziej - jeśli nie overflow I (nie jest i < 2 - kubełek na pewno długości 1 LUB inny kubełek długości 1 LUB gdybyśmy wyciągnęli z listy 2 kierunkowej to nie byłoby już w niej elementów)). Jeśli robimy zwykła relaksację to ZMIANA: robimy ją tylko gdy bucket toNode jest overflowem (DOMYŚLNIE KAŻDY NOWY JEST W NIM) I ALBO nie był wcześniej wykorzystywany ALBO w przeciwnym wypadku, jeśli jego nowy dystans jest mniejszy od minimum overflowu. Inaczej nie ma sensu robić update'a. Tak samo dla toNode gdy jest z niższych bucketów - także aktualizujemy końcówkę overflowu (jeśli będziemy robić for'a to jeśli node ma trafić do overflowu- czyli ma większą etykietę niż początek overflowu - to MUSIMY dociągnąć drugi "]"tak, aby nowa etykieta wstawianego noda się mieściła pomiędzy). Ta końcówka nie ma nigdzie więcej znaczenia, bo korzystamy z niej tylko tutaj. TO, że coś jest w overflow to NIE ZNACZY, że nie było używane (mogło zostać w overflow). Jeśli nie został spełniony żaden z warunków (!= NULL i dystans nie jest mniejszy od obecnego bucketa) to nie robimy fora ZMIANA, a jeśli któryś był spełniony to na pewno nowy bucket będzie mniejszy od starego lub == NULL - nie był wykorzystywany.

Jeśli w buckecie jest więcej węzłów o różnych etykietach to ZMIANA wyciągamy dopiero minimum - min będzie się równać etykiecie tego minimalnego węzła, a pozostałe od 0 do j (gdzie badamy j'ty kubełek) będą zmienione, zachowując swoje pojemności (tylko min się zmieni, a pojemności trzymamy w tablicy osobno). J'ty kubełek zostanie zmieniony na [max;max]. Realocujemy potem wszystkie węzły z j'tego kubełka do niższych. Ich nowych kubełków szukamy od j do 0, gdyż 1) od wyższych jest lepiej, bo mają większe zakresy i większa szansa, że się szybko zatrzymamy, 2) możemy mieć węzeł, który nawet po realokacji zostanie w j'tym kubełku.

JEŚLI skanowany bucket to overflow to nie ma ograniczeń min(..., maxRange) dla nowych zasięgów bucketów (gdyż overflow z definicji był nieskończony), czyli prawie tak jakbyśmy zaczynali od początku, tyle, że przesunięci o min. MaxRange ustawiamy tylko, gdy skanujemy niżej niż overflow - wtedy = ->end z bucketa, który skanujemy. Potem usawiamy J'ty [max;max] oraz ustawiamy ->begin overflowa, jeśli trzeba na o 1 większy niż poprzedzający go kubełek - jeśli zmienialiśmy rozmiar tego ostatniego.

Przy realocacji także patrzymy, czy zmiana zakresów spowoduje zmiany bucketów nodów, które są w tym j'tym buckecie i były przyczyną przeskalowania bucketów od 0 do j - zostawienie [max;max] spowoduje, że nie przesuniemy niepotrzebnie tych węzłów, które były na końcu tego kubełka. Reszte przesuwamy w dół.

2.6 Analiza



Rysunek 2.14: **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.

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

Dowody twierdzeń

Trochę matmy

C.1 Złożoność obliczeniowa

TODO

C.1.1 Analiza asymptotyczna

TODO

C.1.2 Analiza amortyzacyjna

TODO