Ścieżki w grafach.
Problem najkrótszych ścieżek – część 1.

Algorytmy i struktury danych

Wykład 20.

3 czerwca 2024 r.

Ścieżki w grafach – wprowadzenie

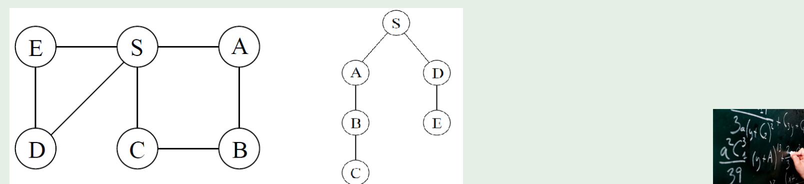
Model i dane

- graf prosty $G = (V, E)$, skierowany lub nieskierowany
- $|V| = n$ wierzchołków $\quad |E| = m = \mathrm{O}\left(n^2\right)$ krawędzi
- $s \in V$ – wierzchołek początkowy

DFS – przypomnienie

- DFS eksploruje wszystkie wierzchołki $v \in V$ osiągalne z s
  - do każdego z nich znajduje pewną ścieżkę od s
  - ścieżki te „zawarte są” w drzewie DFS
- DFS nie wyznacza najkrótszych (liczba krawędzi) ścieżek

Przykład [DPV06]



Problem najkrótszych ścieżek – wprowadzenie

Długości krawędzi

Graf ważony $G = (V, E)$ – każdej krawędzi $(u, v) \in E$ przypisana jest waga (wartość, koszt, długość) $c(u, v) = c_{uv} \in \mathbb{R}$.

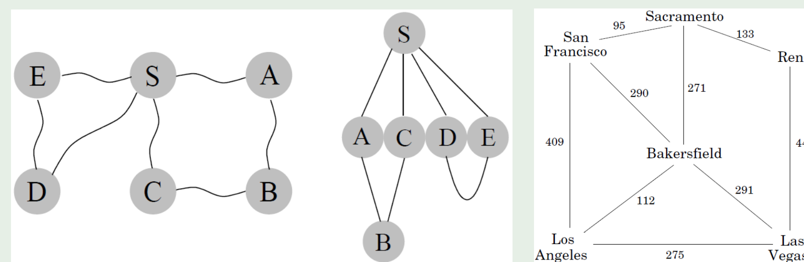
- wagi mogą reprezentować np. długości krawędzi
- w podstawowym wariancie $c(u, v) = 1$ dla każdej krawędzi (u, v)

Podstawowe pojęcia

- trasa w grafie G – ciąg wierzchołków $(v_1, v_2, \ldots, v_k)$ taki, że $(v_i, v_{i+1}) \in E$ dla $i \in \{1, \ldots, k-1\}$
- ścieżka w grafie G – trasa bez powtórzeń wierzchołków
- długość trasy (ścieżki) $(v_1, v_2, \ldots, v_k)$ to $\sum_{i=1}^{k-1} c(v_i, v_{i+1})$
  - jeśli $c(u, v) = 1$, $(u, v) \in E$, to długość ścieżki = liczba krawędzi
- odległość $d(v_i, v_j)$ między wierzchołkami v_i i v_j określamy jako długość najkrótszej ścieżki od v_i do v_j w grafie G

Problem najkrótszych ścieżek – wprowadzenie

Przykłady [DPV06]



Problem Shortest Path – wariant podstawowy

Dany jest graf $G = (V, E)$, długości krawędzi $\left(c(u, v)\right)_{(u,v) \in E}$ oraz wierzchołek startowy $s \in V$.

Dla każdego wierzchołka $v \in V$ osiągalnego z s wyznacz długość $dist(v) = d(s, v)$ najkrótszej ścieżki od s do v w grafie G.

Problem najkrótszych ścieżek – wprowadzenie

Warianty problemu SHORTEST PATH

1. $G = (V, E)$ jest DAGiem
2. G jest dowolny, $c(u, v) = 1$ dla wszystkich $(u, v) \in E$
3. dane są $s \in V$ oraz $c(u, v) \geqslant 0 \rightarrow$ znajdź najkrótsze ścieżki od s do wszystkich $v \in V$ (SINGLE-SOURCE SHORTEST PATH)
4. jak w 3, ale $c(u, v)$ są dowolne (możliwe ujemne długości)
5. znajdź najkrótsze ścieżki między każdą parą wierzchołków (ALL-PAIRS SHORTEST PATH)
6. różne uogólnienia
   - np. dodatkowe parametry – najkrótsza ścieżka, dla której średni czas przejazdu nie przekracza podanego limitu, itp.

- Problemy najkrótszych ścieżek mają szereg różnych zastosowań – patrz np. rozdział 4.2 w [AMO14].
- Na kursie AiSD omawiamy problemy 1–4 (podstawy).
- Bardziej szczegółowo temat ten omawiany jest na kursie wybieralnym *Algorytmy optymalizacji dyskretnej*.

Krawędzie o długości 1 – algorytm BFS

Algorytm BFS (*breadth-first search*)

- przeglądanie grafu „warstwa po warstwie"
  - zaczynamy od s ($dist = 0$)
  - następnie przeglądamy po kolei jego sąsiadów ($dist = 1$)
  - następnie przeglądamy jeszcze nieodwiedzonych sąsiadów tych sąsiadów ($dist = 2$) itd.
- w iteracji d przeglądamy węzły odległe o d od węzła s
- w iteracji $d + 1$ odwiedzamy węzły odległe o $d + 1$ od s, tj. sąsiadów węzłów przeglądanych w iteracji d, którzy nie zostali wcześniej odwiedzeni
- wykorzystujemy kolejkę FIFO Q
  - początkowo w Q jest wierzchołek s
  - w każdym kroku ściągamy wierzchołek v z początku Q i dodajemy na jej koniec wszystkich jeszcze nieodwiedzonych sąsiadów v
  - kończymy, gdy kolejka Q jest pusta
- dla każdego węzła v pamiętamy jego odległość $dist(v)$ od s oraz poprzednika $pred(v)$ w drzewie BFS
  - początkowo $dist(s) = 0$ oraz $dist(v) = \infty$ dla $v \neq s$

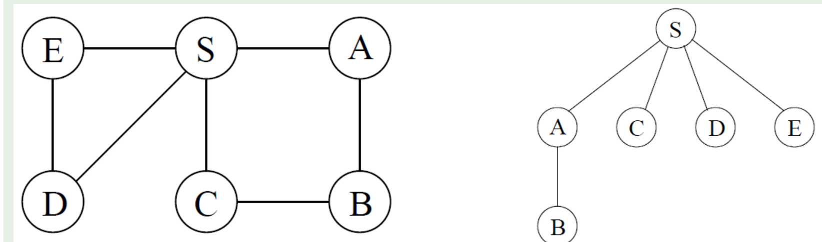
Algorytm BFS

- **Input:** graf $G = (V, E)$, $s \in V$ – wierzchołek początkowy
- **Output:**
  - $dist(v)$, $v \in V$ – długości najkrótszych ścieżek od s; $dist(v) = \infty$ gdy v nie jest osiągalny z s
  - $pred(v)$, $v \in V$ – poprzednicy w drzewie BFS

```
procedure BFS(G = (V, E), s)
1:  for all v ∈ V do  dist(v) ← ∞   pred(v) ← null
2:  dist(s) ← 0
3:  Q ← [s]
4:  while Q is not empty do
5:      u ← pull(Q)                  ▷ wybór przeglądanego wierzchołka
6:      for all (u, v) ∈ E do
7:          if dist(v) = ∞ then      ▷ v nie był wcześniej odwiedzony
8:              push(Q, v)           ▷ dodaj v na koniec kolejki
9:              dist(v) ← dist(u) + 1    ▷ ustaw odległość v od s
10:             pred(v) ← u          ▷ ustaw poprzednika v w drzewie BFS
```


Przykład

Przykład [DPV06]



Algorytm BFS

Poprawność

Dla każdego $d \geqslant 0$ istnieje iteracja, po której:

1. wszystkie wierzchołki $v \in V$ w odległości $\leqslant d$ od s mają prawidłowo ustawione odległości $dist(v)$
2. wszystkie pozostałe wierzchołki mają odległości $dist$ równe ∞
3. kolejka Q zawiera tylko wszystkie wierzchołki odległe o d od s

Własność tą można pokazać prostą indukcją.

Czas działania

BFS ma złożoność liniową względem rozmiaru grafu, tj. $\mathrm{O}(|V| + |E|)$

- inicjalizacja: $\mathrm{O}(|V|)$
- każdy wierzchołek osiągalny z s jest raz wkładany do kolejki
    - $|V|$ operacji pull i $|V|$ operacji push o koszcie $\mathrm{O}(1)$
- wewnętrzna pętla **for** przegląda każdą krawędź dokładnie raz (graf skierowany) lub dwa razy (graf nieskierowany)
    - stała liczba operacji o koszcie $\mathrm{O}(1)$ na każdą krawędź $\rightarrow \mathrm{O}(|E|)$

Eksploracja grafu – BFS vs DFS

Generyczny algorytm eksploracji grafu spójnego

procedure SEARCH($G = (V, E)$, s)

```
1:  for all v ∈ V do  visited(v) ← false   pred(v) ← null
2:  pred(s) ← s                        ▷ techniczna rzecz „dla wygody”
3:  S ← [s]
4:  while S is not empty do
5:      u ← head(S)                    ▷ weź pierwszy element z S, ale go nie usuwaj
6:      if not visited(u) then  visited(u) ← true      ▷ 1. wejście do u
7:      if exists (u, v) ∈ E s.t. pred(v) = null then
8:          push(S, v)                 ▷ v – jakiś nieodkryty sąsiad u
9:          pred(v) ← u
10:     else                           ▷ odkryto wszystkich sąsiadów u – usuń u z S
11:         delete(S, u)
```

Implementacje algorytmu SEARCH

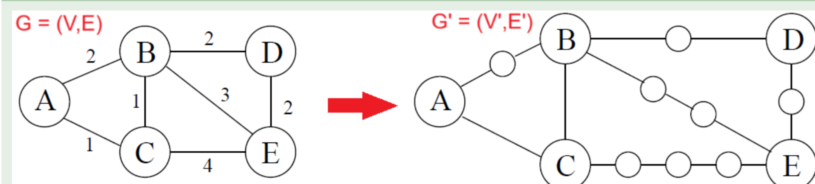
- Jeśli S jest kolejką (FIFO), otrzymujemy algorytm BFS.
- Jeśli S jest stosem (kolejka LIFO), otrzymujemy algorytm DFS.

Nieujemne długości krawędzi

Pierwsze podejście (naiwne) – adaptacja BFS

- BFS zakłada, że wszystkie krawędzie mają długość jednostkową
- jeśli długości $c(u, v) \geqslant 0$ w grafie G są dowolne, możemy wprowadzić „sztuczne wierzchołki”
    - dzielimy długie krawędzie na fragmenty dł. 1 – tworzymy graf G'
    - $c(u, v) - 1$ nowych wierzchołków dla krawędzi (u, v)
    - każda krawędź w grafie G' ma długość 1
- wszystkie wierzchołki z grafu G są w G' i odległości między nimi są takie jak w G
- do wyznaczenia odległości w G' możemy użyć algorytmu BFS

Przykład [DPV06]



Nieujemne długości krawędzi

Pierwsze podejście (naiwne) – złożoność

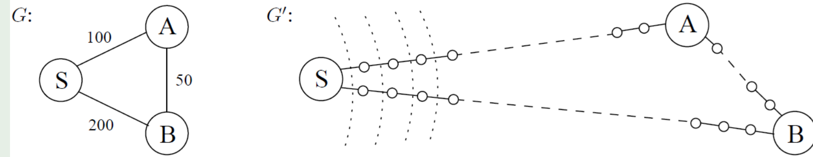
- jeśli $c(u, v) \gg 0$, to mamy bardzo dużo wierzchołków i podejście naiwne jest nieefektywne
- w $G' = (V', E')$ wyznaczamy odległości do wierzchołków, które są nieistotne
    - nie istniały w oryginalnym grafie $G = (V, E)$
- złożoność algorytmu BFS w G' to $\mathrm{O}(|V'| + |E'|)$
- w $G' = (V', E')$ liczba wierzchołków zależy od wartości $c(u, v)$
    - $|V'| = |V| + \sum_{e \in E} (c(u, v) - 1)$

Wniosek: złożoność algorytmu „naiwnego” zależy *liniowo od wartości* długości krawędzi, czyli *wykładniczo od długości ich reprezentacji* (rozmiaru danych wejściowych).

Takie algorytmy nazywamy algorytmami pseudowielomianowymi.

Ulepszenie algorytmu naiwnego

Przykład [DPV06]



Jak ulepszyć algorytm „naiwny”? – Intuicje

- dopóki BFS w G' nie dotrze do któregoś z wierzchołków z G, „nic ciekawego się nie dzieje”
- w każdej iteracji BFS w G' dla każdego wierzchołka v w G mamy pewne ograniczenie górne $T(v)$ na „czas dotarcia” do v
  - początkowo $T(s) = 0$, $T(v) = \infty$ dla $v \neq s$
  - w kolejnym kroku mamy $T(v) = c(s, v)$ dla wszystkich sąsiadów s (ich potencjalnym poprzednikiem jest s)
  - są to ograniczenia górne, bo w trakcie przechodzenia grafu możemy znajdować „skróty”
- ...

Algorytm Dijkstry – wprowadzenie

Jak ulepszyć algorytm „naiwny”? – Intuicje

- [...] pierwsza „ważna rzecz” dzieje się po dotarciu do $u \neq s$ o najmniejszym estymowanym czasie dotarcia $T(u)$
- ponieważ długości krawędzi $c(\bullet, \bullet) \geqslant 0$, do u nie możemy dotrzeć wcześniej $\rightarrow$ $dist(u) = T(u)$, $pred(u) = s$ i najkrótsza ścieżka od s do u jest już wyznaczona
- z u mogą wychodzić krawędzie do pewnych v t. że:
  - $T(v) = \infty \rightarrow$ dla v możemy ustawić estymowany czas dotarcia na $T(v) = dist(u) + c(u, v)$ oraz potencjalnego poprzednika na u
  - $T(v) < \infty \rightarrow$ sprawdzamy, czy nie znaleźliśmy krótszej ścieżki od s do v. Jeśli $T(v) > dist(u) + c(u, v)$, to szybciej dotrzemy do v idąc przez u, więc uaktualniamy (zmniejszamy) estymowany czas dojścia do v i jego potencjalnego poprzednika
- kolejna „ważna rzecz” dzieje się w momencie, gdy dotrzemy do wierzchołka u' o najmniejszym estymowanym czasie dotarcia $T(u')$, do którego najkrótsza ścieżka nie jest jeszcze ustalona
  - powtarzamy procedurę, aż wyznaczymy najkrótsze ścieżki do wszystkich wierzchołków osiągalnych z s

Algorytm Dijkstry – wprowadzenie

Idea algorytmu Dijkstry

- algorytm iteracyjnie oblicza $dist(v)$ dla $v \in V$
- w każdym kroku dzieli (niejawnie) V na zbiory S oraz $\overline{S} = V \setminus S$
  - S – wierzchołki z wyznaczonymi najkrótszymi ścieżkami od s (*permanentne*, optymalne wartości $dist(\bullet)$)
  - $\overline{S}$ – wierzchołki z pewnym ograniczeniem górnym na długość najkrótszej ścieżki od s (*tymczasowe* wartości $dist(\bullet)$)
- permanentne wartości $dist(v)$ ustalane są w kolejności według rosnącej odległości węzłów od s

Algorytm Dijkstry w każdym kroku wybiera $u \in \overline{S}$ z najmniejszą tymczasową wartością $dist(u)$ (zaczynając od s), oznacza $dist(u)$ jako optymalne („przenosi u z $\overline{S}$ do S”) i wykonuje distance update.

Kluczowa obserwacja

Dla $u \in \overline{S}$ o najmniejszej tymczasowej wartości $dist(u)$, ta wartość jest optymalna i może zostać uznana za permanentną (docelową).

Algorytm Dijkstry – implementacja generyczna

- **Input:**
  - graf $G = (V, E)$ z wagami $c(u, v) \geqslant 0$ dla $(u, v) \in E$
  - $s \in V$ – wierzchołek początkowy
- **Output:**
  - $dist(v)$, $v \in V$ – długości najkrótszych ścieżek od s
  - $pred(v)$, $v \in V$ – poprzednicy w drzewie najkrótszych ścieżek

1: $S \leftarrow \emptyset$, $\overline{S} \leftarrow V$
2: **for all** $v \in V$ **do** $dist(v) \leftarrow \infty$, $pred(v) \leftarrow null$
3: $dist(s) \leftarrow 0$
4: **while** $\overline{S} \neq \emptyset$ **do**
5: select $u \in \overline{S}$ with smallest $dist(\bullet)$ ▷ *node selection*
6: $S \leftarrow S \cup \{u\}$, $\overline{S} \leftarrow \overline{S} \setminus \{u\}$
7: **for all** $(u, v) \in E$ **do**
8: **if** $dist(v) > dist(u) + c(u, v)$ **then** ▷ *distance update*
9: $dist(v) \leftarrow dist(u) + c(u, v)$
10: $pred(v) \leftarrow u$

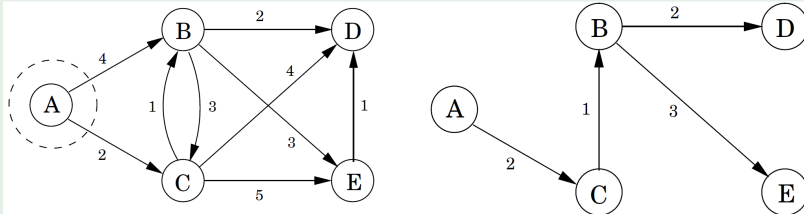
Algorytm Dijkstry – Przykład

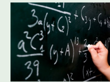
Przykład

Pojedyncza iteracja (dopóki $\overline{S} \neq \emptyset$)

- wybierz $u \in \overline{S}$ z najmniejszą tymczasową wartością *dist*(u)
- przenieś u z $\overline{S}$ do S
- wykonaj operację *distance update*:

$$(\forall (u, v) \in E)(dist(v) \leftarrow \min\{dist(v), dist(u) + c(u, v)\})$$





Algorytm Dijkstry – obserwacje

Najważniejsze obserwacje

- algorytm Dijkstry zatrzymuje się, gdy wyznaczy najkrótsze ścieżki do wszystkich wierzchołków osiągalnych z s (wszystkie wartości *dist*(v) są optymalne)
- w kolejnych iteracjach wyznaczane jest (niejawnie) drzewo o korzeniu w s złożone z wierzchołków $v \in V$, dla których $dist(v) \neq \infty$ (permanentne i tymczasowe)
- na koniec będzie to drzewo najkrótszych ścieżek
- to drzewo możemy odtworzyć na podstawie wskaźników *pred*(v)

Kluczowe operacje

- *node selection* – wybór wierzchołka $u \in \overline{S}$ t. że $dist(u) = \min\{dist(v) : v \in \overline{S}\}$
- *distance update* – dla wybranego wierzchołka u przeglądamy wszystkie krawędzie $(u, v) \in E$ i wyznaczamy $dist(v) = \min\{dist(v), dist(u) + c(u, v)\}$

Generyczny algorytm Dijkstry – złożoność

Czas działania ($|V| = n$, $|E| = m$)

Główne operacje: *node selection* oraz *distance update*

- node selection: $O(n^2)$
  - łącznie n operacji
  - czas pojedynczego wyboru ograniczony z góry przez $|\overline{S}| \leqslant n$
- distance update: $O(m)$
  - dla każdego $u \in V$ robimy *update* dla każdej krawędzi (u, v)
  - jedna operacja o koszcie $O(1)$ dla m krawędzi

Razem: $O(n^2 + m)$

Uwagi i optymalizacje

- dla grafów gęstych $m = \Theta(n^2)$ asymptotycznie lepiej się nie da
  - musimy przejrzeć każdą krawędź $\rightarrow$ składnik $O(m)$ zostanie
- da się poprawić złożoność dla grafów rzadkich z $m = o(n^2)$
- miejsce do optymalizacji: operacja *node selection*
  - albo poprawiamy worst-case complexity do $o(n^2)$
  - albo znacząco redukujemy czas dla „praktycznych instancji”

Algorytm Dijkstry – implementacje kopcowe

Optymalizacja operacji node selection

- w każdej iteracji wybieramy wierzchołek u z najmniejszą tymczasową wartością *dist*(u)
- **pomysł:** użyj kolejki priorytetowej (typu min)
  - kluczem jest tymczasowa wartość *dist*($\bullet$)
  - w kolejce przechowujemy jedynie wierzchołki z $\overline{S}$
  - nie musimy przechowywać wierzchołków $v \in \overline{S}$ z $dist(v) = \infty$
- efektywne implementacje kolejek priorytetowych – kopce

Kolejka priorytetowa

- każdy element posiada wartość liczbową – klucz (priorytet)
- używamy kolejki typu min
- wspierane operacje („minimalny interfejs” kolejki priorytetowej):
  - create-queue()
  - find-min(Q)
  - extract-min(Q)
  - insert(Q, i, k)
  - decrease-key(Q, i, k)

Algorytm Dijkstry – implementacje kopcowe

procedure Heap-Dijkstra($G = (V, E)$, c, s)

```
 1: H ← create-heap()                                  ▷ klucze – wartości dist(•)
 2: for all v ∈ V do   dist(v) ← ∞,   pred(v) ← null
 3: dist(s) ← 0
 4: insert(H, s, dist(s))
 5: while H is not empty do
 6:     u ← extract-min(H)                             ▷ node selection
 7:     for all (u, v) ∈ E do
 8:         if dist(v) > dist(u) + c(u, v) then         ▷ distance update
 9:             prev_dv ← dist(v)
10:             dist(v) ← dist(u) + c(u, v)
11:             pred(v) ← u
12:             if prev_dv = ∞ then
13:                 insert(H, v, dist(v))
14:             else
15:                 decrease-key(H, v, dist(v))
```


Złożoność algorytmu Dijkstry

Implementacja kopcowa – złożoność ($|V| = n$, $|E| = m$)

- każdy wierzchołek osiągalny z s wkładamy do kopca i wyciągamy z niego dokładnie jeden raz
- mamy więc po $\leqslant n$ operacji extract-min oraz insert
- każdą krawędź ze spójnej składowej s przeglądamy jeden raz – $\mathrm{O}(m)$ operacji decrease-key
- złożoność zależy od użytej implementacji kopca

Kopiec binarny ($|V| = n$, $|E| = m$)

- insert, extract-min i decrease-key: $\mathrm{O}(\log n)$
- BinaryHeap-Dijkstra: $\mathrm{O}((m + n) \log n)$
  - $\mathrm{O}(m \log n)$ gdy graf jest spójny
  - dla $m = \Theta(n^2)$ mamy $\mathrm{O}(n^2 \log n)$ – gorzej niż wersja generyczna (zwykła tablica zamiast kopca)
  - dla $m = \Theta(n)$ mamy $\mathrm{O}(n \log n)$ – dużo lepiej

Złożoność algorytmu Dijkstry

Kopiec d-arny

- insert i decrease-key: $\mathrm{O}(\log_d n)$, extract-min: $\mathrm{O}(d \log_d n)$
- d-aryHeap-Dijkstra: $\mathrm{O}(m \log_d n + n\, d \log_d n)$
- Jakie d jest asymptotycznie optymalne?
  - $m \log_d n = n\, d \log_d n \iff d = \max\{2, \lceil m/n \rceil\}$
  - dostajemy wówczas złożoność $\mathrm{O}(m \log_d n)$
  - gdy $m = \Omega(n^{1+\epsilon})$ dla pewnego $\epsilon > 0$, to $d = \lceil m/n \rceil = \Omega(n^{\epsilon})$ i mamy złożoność $\mathrm{O}(m)$ (algorytm asymptotycznie optymalny)

Kopiec Fibonacciego

- extract-min: $\mathrm{O}(\log n)$
- pozostałe: $\mathrm{O}(1)$ (koszt zamortyzowany)
- FibonacciHeap-Dijkstra: $\mathrm{O}(m + n \log n)$
- jeden z asymptotycznie najszybszych silnie wielomianowych algorytmów dla problemu najkrótszej ścieżki z wagami $\geqslant 0$
- kopiec Fibonacciego to skomplikowana struktura – wymaga złożonej implementacji (w praktyce duży koszt)

Algorytm Dijkstry – uwagi i poprawność

Inne implementacje algorytmu Dijkstry

Istnieją implementacje algorytmu Dijkstry, które wykorzystują nieco inne podejście. Ich czas działania zależy od kosztów $c(u, v)$, ale dla pewnych rodzin grafów w praktyce okazują się być wydajne.

- algorytm Diala
- algorytm RADIX HEAP

Będą one omówione na kursie *Algorytmy optymalizacji dyskretnej*.

Poprawność algorytmu Dijkstry (formalnie)

Hipoteza indukcyjna (indukcja po $|S|$):

1. dla $v \in S$ wartości $dist(v)$ są optymalne (permanentne)
2. dla $v \in \overline{S}$ wartości $dist(v)$ są długościami najkrótszych ścieżek od s do v, których wszystkie węzły pośrednie są z S (przechodzą tylko przez węzły z optymalnymi wartościami $dist(v)$)

Hipoteza indukcyjna jest oczywista dla $S = \emptyset$.

Poprawność algorytmu Dijkstry

★ Szkic dowodu – część 1.

- w każdej iteracji do S dodajemy jeden wierzchołek $u \in \overline{S}$ t. że $dist(u) = \min\{dist(v) \colon v \in \overline{S}\}$
- z zał. ind. wiemy, że $dist(u)$ jest długością najkrótszej ścieżki P_0 od s do u przechodzącej jedynie po wierzchołkach z S
- rozważmy dowolną ścieżkę P' od s do u zawierającą wierzchołki pośrednie z $\overline{S}$
- niech w – pierwszy wierzchołek z $\overline{S}$ na P'
    - w rozdziela P' na P'_1 i P'_2
    - P'_1 zawiera tylko wierzchołki pośrednie z S
- z zał. ind. i sposobu wyboru u mamy
$$length(P'_1) \geqslant dist(w) \geqslant dist(u) = \min\{dist(v) \colon v \in \overline{S}\}$$
($dist(w)$ to dł. najkrótszej ścieżki (po wierzch. z S) od s do w).
- $length(P'_2) \geqslant 0$, zatem $length(P') \geqslant dist(u) = length(P)$
- **Wniosek:** $dist(u)$ to długość najkrótszej ścieżki od s do u w grafie $G = (V, E)$

Poprawność algorytmu Dijkstry

★ Szkic dowodu – część 2.

- po ustaleniu optymalnej wartości $dist(u)$ i przeniesieniu u z $\overline{S}$ do S, długości najkrótszych ścieżek po wierzchołkach z S dla pozostałych wierzchołków z $\overline{S}$ mogły zmaleć
    - ścieżki mogą teraz przechodzić dodatkowo przez u
- po oznaczeniu $dist(u)$ jako *permanentna*, algorytm wykonuje operację *distance update* dla wszystkich $v \in V$ t. że $(u, v) \in E$
- po tych update'ach ścieżka $s = v_0 - v_1 - \cdots - v_h = v$ wyznaczona przez indeksy $pred(\bullet)$ spełnia $dist(v_{i+1}) = dist(v_i) + c(v_i, v_{i+1})$, $0 \leqslant i < h$
    - ta ścieżka po drodze przechodzi tylko po wierzchołkach z $S \cup \{u\}$
- można pokazać, że jeśli spełnione są powyższe równości, to wartości $dist(v)$ dla $v \in \overline{S} \setminus \{u\}$ są długościami najkrótszych ścieżek (przechodzących po węzłach z $S \cup \{u\}$)

Literatura

Wykład 20. został w większości przygotowany na podstawie książki

[DPV06] S. Dasgupta, Ch. Papadimitriou, U. Vazirani, *Algorithms*, 1st edition, McGraw-Hill Education, 2006 (fragmenty rozdziału 4)

Literatura dodatkowa

[AMO14] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Pearson Education, 2014 (fragmenty rozdziału 4)

[CLRS22] T.H. Cormen, Ch.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, 4th edition, MIT Press, 2022 (fragmenty rozdziałów 20 i 22)