Bit Algo
START

Algorytmy najkrótszych ścieżek w grafach

Reprezentacja grafów ważonych

- listy adjacencji: trzymamy dodatkowy atrybut oprócz numeru sąsiada (np. trzymamy 2-elementowe krotki)
- macierz adjacencji: jeżeli jest krawędź, to trzymamy jej wagę, a nie koniecznie 1; jeżeli krawędzi nie ma, a chcielibyśmy mieć w grafie krawędzie o wadze 0, to można trzymać None'y
- inne reprezentacje (np. zbiory adjacencji): podobnie jak w listach adjacencji, trzeba po prostu razem z sąsiadem trzymać wagę krawędzi do niego

Najkrótsze ścieżki w grafach

- **ścieżka** - zbiór krawędzi pomiędzy wierzchołkami
- **podścieżka** - ścieżka zawarta w innej ścieżce
- **waga ścieżki** - suma wag krawędzi na ścieżce
- **najkrótsza ścieżka (ścieżka optymalna)** - ścieżka o najmniejszej wadze
- **negatywny cykl (cykl o ujemnej wadze)** - ścieżka będąca cyklem, której suma wag jest ujemna; często wymaganiem jest, żeby takiego nie było

Optimal substructure najkrótszych ścieżek

- **najkrótsze ścieżki mają własność optimal substructure**
- “podścieżki ścieżek optymalnych są optymalne”
- na tej własności opierają się algorytmy, o których będziemy mówić, bo pozwala ona na działanie algorytmów dynamicznych i zachłannych

Rodzaje algorytmów najkrótszych ścieżek

- **1-do-1:** interesuje nas tylko optymalna droga z punktu A do punktu B, np. w Google Maps trasa pomiędzy miastami
- **1-do-wiele:** interesują nas optymalne drogi ze źródła do wszystkich pozostałych wierzchołków w grafie, np. w routingu pakietów, kiedy chcemy jak najszybciej kontaktować się z innymi komputerami w sieci
- **wiele-do-wiele:** interesują nas wszystkie możliwe ścieżki 1-do-1 w grafie, np. *nie udało mi się znaleźć zastosowań, to też pewnie o czymś świadczy*

Przygotowanie na ćwiczenia

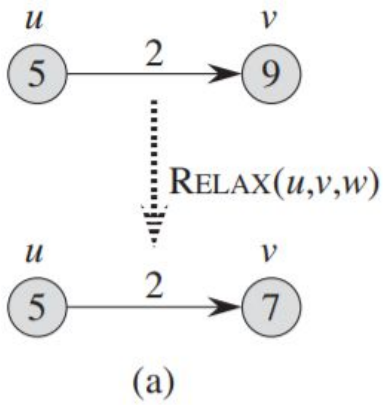
- zaimplementuj algorytmy, o których będziemy mówić; najłatwiej pewnie w postaci list adjacencji, ale bądź też gotowy/-a na implementację na macierzy adjacencji
- zmodyfikuj algorytmy tak, aby dało się dostać nie tylko odległość najkrótszej ścieżki między wierzchołkami v i u, ale także samą ścieżkę (numery wierzchołków na tej ścieżce)

Algorytm Dijkstry (1-do-wiele)

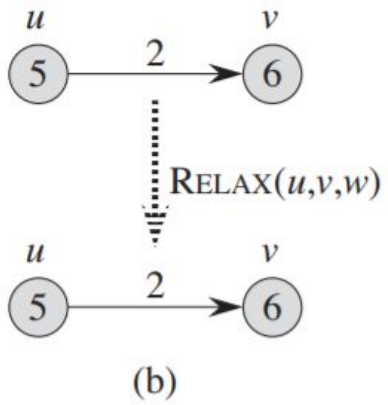
- **problem:** znaleźć najkrótsze ścieżki z wierzchołka źródłowego do wszystkich pozostałych wierzchołków
- **wejście:** graf ważony skierowany **o nieujemnych wagach**
- **idea:** algorytm zachłanny, eksplorujemy graf od źródła i zawsze bierzemy najlepszego sąsiada, czyli tego, do którego jest najbliżej (ścieżka do niego jest optymalna), a potem aktualizujemy odległości od niego do jego sąsiadów (jeżeli ścieżka przez niego jest lepsza od poprzedniej)
- **relaksacja** - proces zmniejszania wartości; można skojarzyć z rozluźnianiem naciągniętej sprężyny, bo oznacza zmniejszenie niepotrzebnie dużej wagi ścieżki do nowej, mniejszej, “luźniejszej”
- nie będziemy dowodzić greedy choice property - opiera się na sformułowaniu optimal substructure

Relaksacja odległości

- “rozluźniamy” zbyt duże odległości - są one “napięte” i po znalezieniu krótszej ścieżki możemy zmniejszyć je do lepszej, mniejszej wartości
- z tej funkcji pomocniczej korzysta dużo algorytmów



(a)



(b)

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

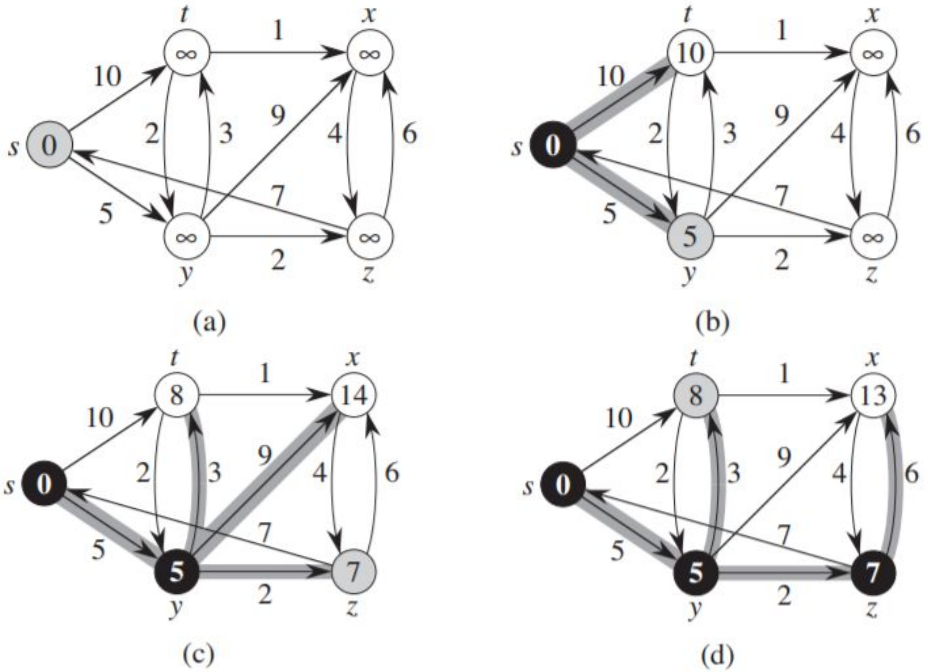
Algorytm Dijkstry

- inicjalizacja: wszystkie wierzchołki poza źródłem mają odległość nieskończoność, źródło 0
- S to “zakończone” wierzchołki, do których znamy najkrótszą ścieżkę
- Q to kolejka minimum wag ścieżek do wierzchołków; zakładamy, że każdy wierzchołek V ma zapisany atrybut d najkrótszej ścieżki do niego i z tego korzysta kolejka
- póki mamy wierzchołki, to wyciągamy je i relaksujemy odległości do sąsiadów

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

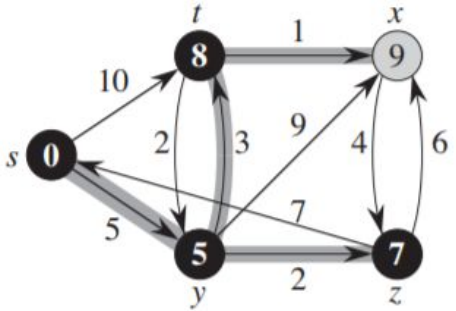
Algorytm Dijkstry - przykład



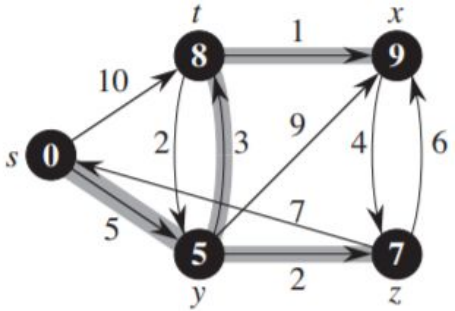

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Algorytm Dijkstry - przykład



(e)



(f)

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Algorytm Dijkstry - złożoność

- każdy wierzchołek dokładnie raz jest wyjęty z kolejki i odwiedzany; dla kolejki minimum na kopcu jest to O(log(V))
- odwiedzając wierzchołek sprawdzamy wszystkie jego krawędzie i ew. relaksujemy - O(E)
- każdy wierzchołek wyjmujemy z kolejki i odwiedzamy jego sąsiadów, więc mnożymy złożoności
- **złożoność dla list adjacencji:** O(E * log(V))
- **złożoność dla macierzy adjacencji:** O(V * log(V))

Algorytm Bellmana-Forda (1-do-wiele)

- **problem:** znaleźć najkrótsze ścieżki z wierzchołka źródłowego do wszystkich pozostałych wierzchołków
- **wejście:** graf ważony skierowany **o dowolnych wagach**
- **idea:** k-ta relaksacja znajduje najkrótsze ścieżki o długości do k włącznie (dowód - Cormen); najdłuższa możliwa ścieżka niebędąca cyklem (w sensie liczby wierzchołków) ma długość V - 1, więc tyle razy trzeba zrelaksować wszystkie krawędzie; w ten sposób każdy wierzchołek w każdej z V - 1 iteracji “dowiaduje się” o najkrótszych ścieżkach długości o 1 większej niż poprzednio znane
- **wykrywa negatywne cykle** - jeżeli w grafie jest cykl o sumie ujemnej, to te relaksacje to ujawnią i na koniec trzeba sprawdzić, czy taki cykl jest; jeżeli tak, to zwracamy informację o tym, a **nie** najkrótsze ścieżki; w Cormenie zwracany jest False

Algorytm Bellmana-Forda, złożoność

- relaksujemy V - 1 razy, za każdym razem “wydłużając” znane najkrótsze ścieżki o 1 krawędź
- na koniec sprawdzamy, czy jest cykl ujemny; jeżeli jest, to aktualna krawędź musi móc być “zmniejszona” przez uwzględnienie kolejnej krawędzi, czyli da się “skrócić” najkrótszą ścieżkę (oczywiście sprzeczność)
- **złożoność:** O(V * E)

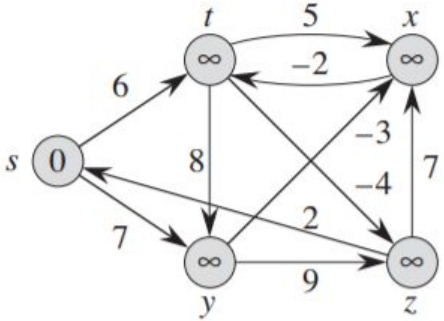
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

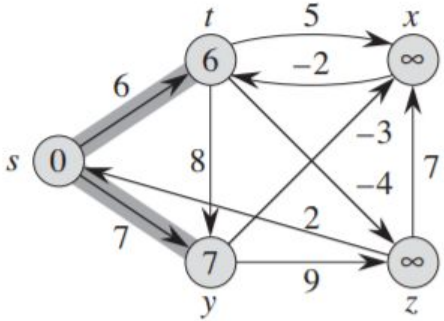
Algorytm Bellmana-Forda - przykład

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

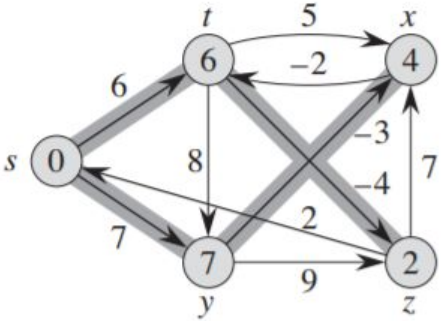
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```



(a)



(b)

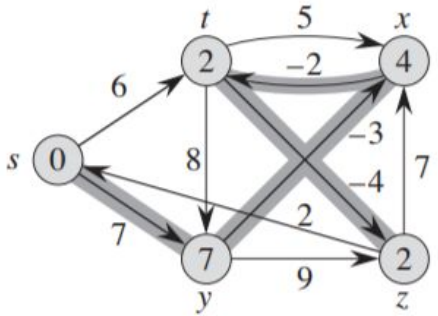


(c)

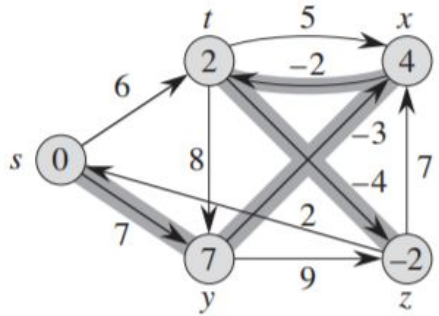
Algorytm Bellmana-Forda - przykład

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```



(d)



(e)

Najkrótsze ścieżki między każdą parą wierzchołków

Opisane wcześniej algorytmy Dijkstry i Bellmana-Forda, pozwalają znaleźć najkrótsze ścieżki do wszystkich wierzchołków z jednego źródła. Chcąc znaleźć najkrótsze ścieżki między każdą parą wierzchołków, należałoby wykonać te algorytmy z każdego wierzchołka jako źródła. Dla Bellmana-Forda daje to złożoność $O(V^2 * E)$, dla algorytmu Dijkstry daje $O((V^2 + V* E) * \log(V))$, co jest dobrą złożonością dla grafów rzadkich, mających wagi dodatnie.

Najkrótsze ścieżki między każdą parą wierzchołków - heurystyka rekurencyjna

Podstawą do stworzenia algorytmu wyznaczającego wagi i najkrótsze ścieżki między każdą parą wierzchołków może być następujące twierdzenie:

Jeżeli w grafie ważonym najkrótsza ścieżka od i do j wiedzie przez k, to fragmenty od i do k i k do j, stanowią najkrótsze ścieżki między tymi parami wierzchołków.

Dowód nie wprost: gdyby tak nie było, to moglibyśmy poprawić naszą ścieżkę od i do j, puszczając od i do k najkrótszą ścieżką od i do k. A to oznaczałoby, że nie wzięliśmy najkrótszej ścieżki od i do j.

Najkrótsze ścieżki między każdą parą wierzchołków - funkcja dynamiczna

Udowodnione poprzednio twierdzenie można wykorzystać do utworzenia rekurencyjnej funkcji, która podaje wagę najkrótszej ścieżki między parą i oraz j:

f(i, j) = min [k in V] (f(i, k) + f(k,j))

ma ona następujące warunki brzegowe:

i = j -> 0

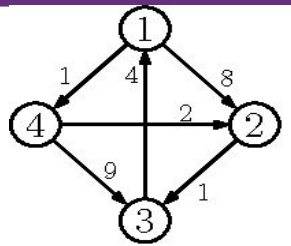
(i,j) in E -> w(i,j)

_ -> +infinity

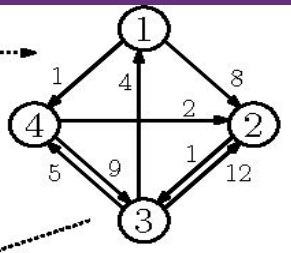
Najkrótsze ścieżki między każdą parą wierzchołków - algorytm Floyda - Warshalla

```
Floyd - Warshall(V, E):
dp = array(V, V)
for i = 0 to V-1:
        for j = 0 to V-1:
                if i == j:
                        dp[i][j] = 0
                elif (i,j) in E:
                        dp[i][j] = w(i,j)
                else:
                        dp[i][j] = +inf
for k  = 0 toV-1:
        for i = 0 to V -1:
                for j = 0 to V-1:
                        dp[i][j] = min(dp[i][j], dp[i][k]+dp[k][j])
return dp
```

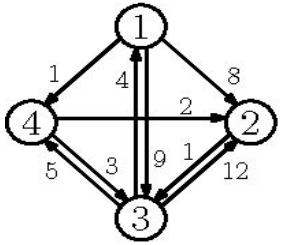
Algorytm Floyda - Warshalla



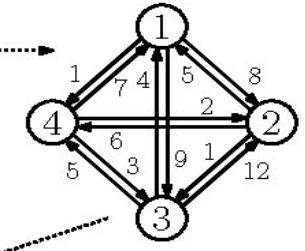
$$d^{(0)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



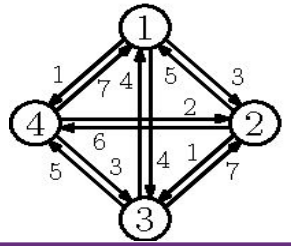
$$d^{(1)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}$$



$$d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$



$$d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

$$\text{final} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Algorytm Floyda - Warshalla - znajdowanie ścieżek

Aby znajdować faktyczne ścieżki, wystarczy oprócz dp dołożyć drugą tablicę solution[V][V], gdzie pod solution[i][j] znajduje się k, dla którego uzyskaliśmy najmniejszą wartość dp[i][j]. Intuicyjnie, jeśli najkrótsza ścieżka między i i j idzie przez k, to można najpierw wypisać najkrótszą ścieżkę między i i k, a potem najkrótszą ścieżkę między k i j. Mamy zatem:

```
printPath(solution, i, j, E):
    if (i,j) in E:
        return to_string(i)+to_string(j)
    k = solution[i][j]
    return printPath(i,k) - to_string(k) + printPath(k,j)
```

Algorytm Floyda - Warshalla - podsumowanie

Algorytm Floyda Warshalla działa dla każdego grafu, zawsze w złożoności O(V^3), co dla grafów gęstych jest lepsze niż Dixtra każdy z każdym. Jego złożoność pamięciowa wynosi zawsze O(V^2). Algorytm nie może zostać wykorzystany do obliczenia najkrótszych ścieżek między wierzchołkami z jakiegoś podzbioru wierzchołków.

Problem wymiany walut

Problem: Proszę podać algorytm, który mając tablicę przeliczników wymiany walut stwierdzi, czy istnieje seria wymian, taka, że startujemy z jednej waluty i wracamy do tej samej, ale z większą ilością pieniędzy.

Problem wymiany walut - model grafowy

Aby wykorzystać algorytm grafowy, należy zdefiniować dla problemy strukturę grafową. Wierzchołkami będą konkretne waluty, a krawędziami (skierowanymi!!!) przeliczniki między nimi. Problem przekłada się następująco na język grafowy: Mając dany graf ważony skierowany podaj algorytm, który sprawdzi, czy istnieje cykl, taki, że iloczyn wag krawędzi >1. Jeżeli tak, to dowolna liczba przejść po tym cyklu może służyć, za serię wymiany walut, która daje na końcu więcej gotówki tej samej waluty.

No ale mamy problem: Nie znamy algorytmów grafowych, operujących na iloczynach wag.

Problem wymiany walut - modyfikacja grafu

Możemy zauważyć, że:

$a*b*c*d*... > 1 \Leftrightarrow \log(a*b*c*d*...) > 0 \Leftrightarrow -\log(a*b*c*d*...) < 0 \Leftrightarrow -\log(a) + -\log(b) + -\log(c) + -\log(d) + \ldots < 0$

W takim razie zamiast przeliczników walut, na każdej krawędzi trzymamy ujemny logarytm z tego przelicznika. Teraz nasz problem sprowadza się do znalezienia ujemnego cyklu w grafie ważonym skierowanym. Na to istnieje już standardowy algorytm: algorytm Bellemana - Forda

Bit Algo
START