Algorytmy i struktury danych - Podstawowe algorytmy grafowe

Marcin Żurowski

22 maja 2025

Plan zajęć

Algorytmy grafowe

Przeszukiwanie grafu:

- Przeszukiwanie grafu polega na systematycznym przechodzeniu jego krawędzi w celu odwiedzenia wszystkich wierzchołków
- Celem przeszukiwania jest zazwyczaj zdobycie pewnych informacji o strukturze grafu
- Przeszukiwany graf może być skierowany lub nieskierowany
- Wyróżniamy dwie podstawowe metody przeszukiwania grafu

Algorytmy grafowe

Przeszukiwanie grafu:

- Przeszukiwanie grafu polega na systematycznym przechodzeniu jego krawędzi w celu odwiedzenia wszystkich wierzchołków
- Celem przeszukiwania jest zazwyczaj zdobycie pewnych informacji o strukturze grafu
- Przeszukiwany graf może być skierowany lub nieskierowany
- Wyróżniamy dwie podstawowe metody przeszukiwania grafu:

Algorytmy grafowe

Przeszukiwanie grafu:

- Przeszukiwanie grafu polega na systematycznym przechodzeniu jego krawędzi w celu odwiedzenia wszystkich wierzchołków
- Celem przeszukiwania jest zazwyczaj zdobycie pewnych informacji o strukturze grafu
- Przeszukiwany graf może być skierowany lub nieskierowany
- Wyróżniamy dwie podstawowe metody przeszukiwania grafu:
  - przeszukiwanie wszerz (breadth-first search, BFS)
  - przeszukiwanie w głąb (depth-first search, DFS)

Algorytmy grafowe

Przeszukiwanie grafu:

- Przeszukiwanie grafu polega na systematycznym przechodzeniu jego krawędzi w celu odwiedzenia wszystkich wierzchołków
- Celem przeszukiwania jest zazwyczaj zdobycie pewnych informacji o strukturze grafu
- Przeszukiwany graf może być skierowany lub nieskierowany
- Wyróżniamy dwie podstawowe metody przeszukiwania grafu:
  - przeszukiwanie wszerz (breadth-first search, BFS)
  - przeszukiwanie w głąb (depth-first search, DFS)

Algorytmy grafowe

Przeszukiwanie grafu:

- Przeszukiwanie grafu polega na systematycznym przechodzeniu jego krawędzi w celu odwiedzenia wszystkich wierzchołków
- Celem przeszukiwania jest zazwyczaj zdobycie pewnych informacji o strukturze grafu
- Przeszukiwany graf może być skierowany lub nieskierowany
- Wyróżniamy dwie podstawowe metody przeszukiwania grafu:
  - przeszukiwanie wszerz (breadth-first search, BFS)
  - przeszukiwanie w głąb (depth-first search, DFS)

Algorytmy grafowe

Przeszukiwanie grafu:

- Przeszukiwanie grafu polega na systematycznym przechodzeniu jego krawędzi w celu odwiedzenia wszystkich wierzchołków
- Celem przeszukiwania jest zazwyczaj zdobycie pewnych informacji o strukturze grafu
- Przeszukiwany graf może być skierowany lub nieskierowany
- Wyróżniamy dwie podstawowe metody przeszukiwania grafu:
  - przeszukiwanie wszerz (breadth-first search, BFS)
  - przeszukiwanie w głąb (depth-first search, DFS)

Algorytmy grafowe

Przeszukiwanie grafu:

- Przeszukiwanie grafu polega na systematycznym przechodzeniu jego krawędzi w celu odwiedzenia wszystkich wierzchołków
- Celem przeszukiwania jest zazwyczaj zdobycie pewnych informacji o strukturze grafu
- Przeszukiwany graf może być skierowany lub nieskierowany
- Wyróżniamy dwie podstawowe metody przeszukiwania grafu:
  - przeszukiwanie wszerz (breadth-first search, BFS)
  - przeszukiwanie w głąb (depth-first search, DFS)

Algorytmy grafowe

Przeszukiwanie grafu wszerz (BFS):

- Dany jest graf $G = (V, E)$ i wyróżniony wierzchołek początkowy s, nazywany źródłem
- Badamy krawędzie grafu G, aby znaleźć wszystkie wierzchołki osiągalne z s
- Wierzchołki w odległości k od źródła s zostają odwiedzone wcześniej niż wierzchołki w odległości $k + 1$. Zatem granica między wierzchołkami odwiedzonymi i nieodwiedzonymi jest przekraczana jednocześnie na całej jej szerokości – stąd nazwa algorytmu
- Podczas przeszukiwania można obliczyć odległość od źródła s do wszystkich osiągalnych wierzchołków (czyli znaleźć najkrótsze ścieżki łączące s z tymi wierzchołkami)

Algorytmy grafowe

Przeszukiwanie grafu wszerz (BFS):

- Dany jest graf $G = (V, E)$ i wyróżniony wierzchołek początkowy s, nazywany źródłem
- Badamy krawędzie grafu G, aby znaleźć wszystkie wierzchołki osiągalne z s
- Wierzchołki w odległości k od źródła s zostają odwiedzone wcześniej niż wierzchołki w odległości $k + 1$. Zatem granica między wierzchołkami odwiedzonymi i nieodwiedzonymi jest przekraczana jednocześnie na całej jej szerokości – stąd nazwa algorytmu
- Podczas przeszukiwania możemy obliczyć odległości od źródła s do wszystkich osiągalnych wierzchołków (czyli długości najkrótszych ścieżek łączących s z tymi wierzchołkami)

Algorytmy grafowe

Przeszukiwanie grafu wszerz (BFS):

- Dany jest graf $G = (V, E)$ i wyróżniony wierzchołek początkowy s, nazywany źródłem
- Badamy krawędzie grafu G, aby znaleźć wszystkie wierzchołki osiągalne z s
- Wierzchołki w odległości k od źródła s zostają odwiedzone wcześniej niż wierzchołki w odległości $k + 1$. Zatem granica między wierzchołkami odwiedzonymi i nieodwiedzonymi jest przekraczana jednocześnie na całej jej szerokości – stąd nazwa algorytmu
- Podczas przeszukiwania możemy obliczyć odległości od źródła s do wszystkich osiągalnych wierzchołków (czyli długości najkrótszych ścieżek łączących s z tymi wierzchołkami)

Algorytmy grafowe

Przeszukiwanie grafu wszerz (BFS):

- Dany jest graf $G = (V, E)$ i wyróżniony wierzchołek początkowy s, nazywany źródłem
- Badamy krawędzie grafu G, aby znaleźć wszystkie wierzchołki osiągalne z s
- Wierzchołki w odległości k od źródła s zostają odwiedzone wcześniej niż wierzchołki w odległości $k + 1$. Zatem granica między wierzchołkami odwiedzonymi i nieodwiedzonymi jest przekraczana jednocześnie na całej jej szerokości – stąd nazwa algorytmu
- Podczas przeszukiwania możemy obliczyć odległości od źródła s do wszystkich osiągalnych wierzchołków (czyli długości najkrótszych ścieżek łączących s z tymi wierzchołkami)

Algorytmy grafowe

Przeszukiwanie grafu wszerz (BFS):

- Dany jest graf $G = (V, E)$ i wyróżniony wierzchołek początkowy s, nazywany źródłem
- Badamy krawędzie grafu G, aby znaleźć wszystkie wierzchołki osiągalne z s
- Wierzchołki w odległości k od źródła s zostają odwiedzone wcześniej niż wierzchołki w odległości $k + 1$. Zatem granica między wierzchołkami odwiedzonymi i nieodwiedzonymi jest przekraczana jednocześnie na całej jej szerokości – stąd nazwa algorytmu
- Podczas przeszukiwania możemy obliczyć odległości od źródła s do wszystkich osiągalnych wierzchołków (czyli długości najkrótszych ścieżek łączących s z tymi wierzchołkami)

Algorytmy grafowe

- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - `u.d` - odległość wierzchołka u od źródła s
- Jeśli nie znaleziono (jeszcze) ścieżki z s do wierzchołka u, to `u.prev = NIL` i `u.d` $= \infty$. Dla wierzchołka s mamy `s.prev = NIL` i `s.d` $= 0$

Algorytmy grafowe

- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - `u.d` - odległość wierzchołka u od źródła s
- Jeśli nie znaleziono (jeszcze) ścieżki z s do wierzchołka u, to `u.prev = NIL` i `u.d` $= \infty$. Dla wierzchołka s mamy `s.prev = NIL` i `s.d` $= 0$

Algorytmy grafowe

- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - `u.d` - odległość wierzchołka u od źródła s
- Jeśli nie znaleziono (jeszcze) ścieżki z s do wierzchołka u, to `u.prev = NIL` i `u.d` $= \infty$. Dla wierzchołka s mamy `s.prev = NIL` i s.d = 0

Algorytmy grafowe

- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - `u.d` - odległość wierzchołka u od źródła s
- Jeśli nie znaleziono (jeszcze) ścieżki z s do wierzchołka u, to `u.prev = NIL` i `u.d` $= \infty$. Dla wierzchołka s mamy `s.prev = NIL` i s.d = 0

Algorytmy grafowe

- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - `u.d` - odległość wierzchołka u od źródła s
- Jeśli nie znaleziono (jeszcze) ścieżki z s do wierzchołka u, to `u.prev = NIL` i `u.d` $= \infty$. Dla wierzchołka s mamy `s.prev = NIL` i s.d = 0

Algorytmy grafowe

- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - `u.d` - odległość wierzchołka u od źródła s
- Jeśli nie znaleziono (jeszcze) ścieżki z s do wierzchołka u, to `u.prev = NIL` i `u.d` $= \infty$. Dla wierzchołka s mamy `s.prev = NIL` i s.d = 0

Algorytmy grafowe

- Pierwszym odwiedzanym wierzchołkiem jest źródło s
- W momencie, kiedy odwiedzamy dany wierzchołek, wstawiamy go do kolejki Q (FIFO) przechowującej wierzchołki, których sąsiadów należy przejrzeć
- W głównym kroku algorytmu pobieramy element u z kolejki Q i przechodzimy jego listę sąsiedztwa. Jeśli dany sąsiad v nie został jeszcze odwiedzony, to odwiedzamy go. Ustawiamy przy tym $v.prev = u$ i $v.d = u.d + 1$ oraz wstawiamy v do kolejki Q
- Algorytm kończy się, kiedy kolejka Q jest pusta

Algorytmy grafowe

- Pierwszym odwiedzanym wierzchołkiem jest źródło s
- W momencie, kiedy odwiedzamy dany wierzchołek, wstawiamy go do kolejki Q (FIFO) przechowującej wierzchołki, których sąsiadów należy przejrzeć
- W głównym kroku algorytmu pobieramy element u z kolejki Q i przechodzimy jego listę sąsiedztwa. Jeśli dany sąsiad v nie został jeszcze odwiedzony, to odwiedzamy go. Ustawiamy przy tym $v.prev = u$ i $v.d = u.d + 1$ oraz wstawiamy v do kolejki Q
- Algorytm kończy się, kiedy kolejka Q jest pusta

Algorytmy grafowe

- Pierwszym odwiedzanym wierzchołkiem jest źródło s
- W momencie, kiedy odwiedzamy dany wierzchołek, wstawiamy go do kolejki Q (FIFO) przechowującej wierzchołki, których sąsiadów należy przejrzeć
- W głównym kroku algorytmu pobieramy element u z kolejki Q i przechodzimy jego listę sąsiedztwa. Jeśli dany sąsiad v nie został jeszcze odwiedzony, to odwiedzamy go. Ustawiamy przy tym $v.prev = u$ i $v.d = u.d + 1$ oraz wstawiamy v do kolejki Q
- Algorytm kończy się, kiedy kolejka Q jest pusta

Algorytmy grafowe

- Pierwszym odwiedzanym wierzchołkiem jest źródło s
- W momencie, kiedy odwiedzamy dany wierzchołek, wstawiamy go do kolejki Q (FIFO) przechowującej wierzchołki, których sąsiadów należy przejrzeć
- W głównym kroku algorytmu pobieramy element u z kolejki Q i przechodzimy jego listę sąsiedztwa. Jeśli dany sąsiad v nie został jeszcze odwiedzony, to odwiedzamy go. Ustawiamy przy tym $v.prev = u$ i $v.d = u.d + 1$ oraz wstawiamy v do kolejki Q
- Algorytm kończy się, kiedy kolejka Q jest pusta

Algorytmy grafowe

```
procedure BFS(G, s)
 for each u ∈ V
  u.visited = false
  u.d = ∞
  u.prev = NIL
 Q = ϕ
 s.visited = true
 s.d = 0
 ENQUEUE(Q,s)
 while not QUEUE-EMPTY(Q)
  u = DEQUEUE(Q)
  for each v ∈ A[u]
   if v.visited = false
    v.visited = true
    v.d = u.d + 1
    v.prev = u
    ENQUEUE(Q,v)
```

Algorytmy grafowe

- W najgorszym przypadku kolejka Q musi mieć możliwość przechowania $|V| - 1$ elementów (w przypadku tablicowej implementacji kolejki potrzebna jest tablica długości $|V|$)
- W efekcie przeszukiwania otrzymujemy drzewo przeszukiwania wszerz, zawierające wszystkie wierzchołki osiągalne z s. Krawędziami tego drzewa są pary wierzchołków postaci $(v.prev, v)$. Dla każdego osiągalnego z s wierzchołka v ścieżka od s do v w tym drzewie odpowiada najkrótszej ścieżce z s do v w grafie G
- Algorytm odwiedza tylko wierzchołki osiągalne z s. Aby przejrzeć cały graf, należy po wykonaniu funkcji BFS sprawdzić, czy pozostały jakieś nieodwiedzone wierzchołki. Jeśli tak, to wybieramy jeden z nich jako źródło i po raz kolejny przeszukujemy graf (oczywiście nie zmieniamy przy tym wartości atrybutów odwiedzonych wcześniej wierzchołków). Powtarzamy, aż nie będzie już nieodwiedzonych wierzchołków
- Złożoność algorytmu: $O(|V| + |E|)$

Algorytmy grafowe

- W najgorszym przypadku kolejka Q musi mieć możliwość przechowania $|V| - 1$ elementów (w przypadku tablicowej implementacji kolejki potrzebna jest tablica długości $|V|$)
- W efekcie przeszukiwania otrzymujemy drzewo przeszukiwania wszerz, zawierające wszystkie wierzchołki osiągalne z s. Krawędziami tego drzewa są pary wierzchołków postaci $(v.prev, v)$. Dla każdego osiągalnego z s wierzchołka v ścieżka od s do v w tym drzewie odpowiada najkrótszej ścieżce z s do v w grafie G
- Algorytm odwiedza tylko wierzchołki osiągalne z s. Aby przejrzeć cały graf, należy po wykonaniu funkcji BFS sprawdzić, czy pozostały jakieś nieodwiedzone wierzchołki. Jeśli tak, to wybieramy jeden z nich jako źródło i po raz kolejny przeszukujemy graf (oczywiście nie zmieniamy przy tym wartości atrybutów odwiedzonych wcześniej wierzchołków). Powtarzamy, aż nie będzie już nieodwiedzonych wierzchołków
- Złożoność algorytmu: $O(|V| + |E|)$

Algorytmy grafowe

- W najgorszym przypadku kolejka Q musi mieć możliwość przechowania $|V| - 1$ elementów (w przypadku tablicowej implementacji kolejki potrzebna jest tablica długości $|V|$)
- W efekcie przeszukiwania otrzymujemy drzewo przeszukiwania wszerz, zawierające wszystkie wierzchołki osiągalne z s. Krawędziami tego drzewa są pary wierzchołków postaci $(v.prev, v)$. Dla każdego osiągalnego z s wierzchołka v ścieżka od s do v w tym drzewie odpowiada najkrótszej ścieżce z s do v w grafie G
- Algorytm odwiedza tylko wierzchołki osiągalne z s. Aby przejrzeć cały graf, należy po wykonaniu funkcji BFS sprawdzić, czy pozostały jakieś nieodwiedzone wierzchołki. Jeśli tak, to wybieramy jeden z nich jako źródło i po raz kolejny przeszukujemy graf (oczywiście nie zmieniamy przy tym wartości atrybutów odwiedzonych wcześniej wierzchołków). Powtarzamy, aż nie będzie już nieodwiedzonych wierzchołków
- Złożoność algorytmu: $O(|V| + |E|)$

Algorytmy grafowe

- W najgorszym przypadku kolejka Q musi mieć możliwość przechowania $|V| - 1$ elementów (w przypadku tablicowej implementacji kolejki potrzebna jest tablica długości $|V|$)
- W efekcie przeszukiwania otrzymujemy drzewo przeszukiwania wszerz, zawierające wszystkie wierzchołki osiągalne z s. Krawędziami tego drzewa są pary wierzchołków postaci $(v.prev, v)$. Dla każdego osiągalnego z s wierzchołka v ścieżka od s do v w tym drzewie odpowiada najkrótszej ścieżce z s do v w grafie G
- Algorytm odwiedza tylko wierzchołki osiągalne z s. Aby przejrzeć cały graf, należy po wykonaniu funkcji BFS sprawdzić, czy pozostały jakieś nieodwiedzone wierzchołki. Jeśli tak, to wybieramy jeden z nich jako źródło i po raz kolejny przeszukujemy graf (oczywiście nie zmieniamy przy tym wartości atrybutów odwiedzonych wcześniej wierzchołków). Powtarzamy, aż nie będzie już nieodwiedzonych wierzchołków
- Złożoność algorytmu: $O(|V| + |E|)$

Algorytmy grafowe

Jeśli wykonano algorytm $BFS(G, s)$, to najkrótszą ścieżkę z s do dowolnego wierzchołka v można wypisać za pomocą następującej procedury:

```
procedure PATH(G, s, v)
 if v = s
  write(s)
 else if v.prev = NIL
  write("nie ma ścieżki z s do v")
 else
  PATH(G, s, v.prev)
  write(v)
```

Algorytmy grafowe

Jeśli wykonano algorytm $BFS(G, s)$, to najkrótszą ścieżkę z s do dowolnego wierzchołka v można wypisać za pomocą następującej procedury:

```
procedure PATH(G, s, v)
 if v = s
  write(s)
 else if v.prev = NIL
  write("nie ma ścieżki z s do v")
 else
  PATH(G, s, v.prev)
  write(v)
```

Algorytmy grafowe

Przykładowe zastosowania algorytmu BFS:

- znajdowanie najkrótszej ścieżki między dwoma wierzchołkami grafu
- znajdowanie najkrótszych ścieżek z jednego wierzchołka grafu do wszystkich pozostałych wierzchołków
- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- sprawdzanie, czy graf nieskierowany jest acykliczny
- sprawdzanie, czy graf jest dwudzielny

Algorytmy grafowe

Przykładowe zastosowania algorytmu BFS:

- znajdowanie najkrótszej ścieżki między dwoma wierzchołkami grafu
- znajdowanie najkrótszych ścieżek z jednego wierzchołka grafu do wszystkich pozostałych wierzchołków
- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- sprawdzanie, czy graf nieskierowany jest acykliczny
- sprawdzanie, czy graf jest dwudzielny

Algorytmy grafowe

Przykładowe zastosowania algorytmu BFS:

- znajdowanie najkrótszej ścieżki między dwoma wierzchołkami grafu
- znajdowanie najkrótszych ścieżek z jednego wierzchołka grafu do wszystkich pozostałych wierzchołków
- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- sprawdzanie, czy graf nieskierowany jest acykliczny
- sprawdzanie, czy graf jest dwudzielny

Algorytmy grafowe

Przykładowe zastosowania algorytmu BFS:

- znajdowanie najkrótszej ścieżki między dwoma wierzchołkami grafu
- znajdowanie najkrótszych ścieżek z jednego wierzchołka grafu do wszystkich pozostałych wierzchołków
- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- sprawdzanie, czy graf nieskierowany jest acykliczny
- sprawdzanie, czy graf jest dwudzielny

Algorytmy grafowe

Przykładowe zastosowania algorytmu BFS:

- znajdowanie najkrótszej ścieżki między dwoma wierzchołkami grafu
- znajdowanie najkrótszych ścieżek z jednego wierzchołka grafu do wszystkich pozostałych wierzchołków
- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- sprawdzanie, czy graf nieskierowany jest acykliczny
- sprawdzanie, czy graf jest dwudzielny

Algorytmy grafowe

Przykładowe zastosowania algorytmu BFS:

- znajdowanie najkrótszej ścieżki między dwoma wierzchołkami grafu
- znajdowanie najkrótszych ścieżek z jednego wierzchołka grafu do wszystkich pozostałych wierzchołków
- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- sprawdzanie, czy graf nieskierowany jest acykliczny
- sprawdzanie, czy graf jest dwudzielny

Algorytmy grafowe

Przykładowe zastosowania algorytmu BFS:

- znajdowanie najkrótszej ścieżki między dwoma wierzchołkami grafu
- znajdowanie najkrótszych ścieżek z jednego wierzchołka grafu do wszystkich pozostałych wierzchołków
- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- sprawdzanie, czy graf nieskierowany jest acykliczny
- sprawdzanie, czy graf jest dwudzielny

Algorytmy grafowe

Przeszukiwanie grafu w głąb (DFS):

- Przeszukiwanie w głąb polega na sięganiu "głębiej" w graf (do dalszych wierzchołków) tak długo, jak to możliwe – stąd nazwa algorytmu
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:

Algorytmy grafowe

Przeszukiwanie grafu w głąb (DFS):

- Przeszukiwanie w głąb polega na sięganiu "głębiej" w graf (do dalszych wierzchołków) tak długo, jak to możliwe – stąd nazwa algorytmu
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - $u.visited$ - true lub false, określający, czy wierzchołek był już odwiedzony
  - $u.prev$ - poprzednik wierzchołka u na ścieżce z wierzchołka s

Algorytmy grafowe

Przeszukiwanie grafu w głąb (DFS):

- Przeszukiwanie w głąb polega na sięganiu "głębiej" w graf (do dalszych wierzchołków) tak długo, jak to możliwe – stąd nazwa algorytmu
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - W każdym kroku algorytmu badamy krawędzie wychodzące z ostatnio odwiedzonego wierzchołka. Mówimy, że krawędź jest przeglądana, jeśli wychodzące z niej krawędzie nie były jeszcze przeglądane. Przy badaniu wszystkich krawędzi wychodzących z danego wierzchołka krawędzie są przeglądane w kolejności wierzchołków, z którymi są incydentne.

Algorytmy grafowe

Przeszukiwanie grafu w głąb (DFS):

- Przeszukiwanie w głąb polega na sięganiu "głębiej" w graf (do dalszych wierzchołków) tak długo, jak to możliwe – stąd nazwa algorytmu
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - W każdym kroku algorytmu badamy krawędzie ostatniego odwiedzonego wierzchołka, który ma jeszcze niezbadane wychodzące krawędzie. Po zbadaniu wszystkich krawędzi wychodzących z danego wierzchołka v wracamy do wierzchołka, z którego v został odwiedzony.

Algorytmy grafowe

Przeszukiwanie grafu w głąb (DFS):

- Przeszukiwanie w głąb polega na sięganiu "głębiej" w graf (do dalszych wierzchołków) tak długo, jak to możliwe – stąd nazwa algorytmu
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - W każdym kroku algorytmu badamy krawędzie ostatniego odwiedzonego wierzchołka, który ma jeszcze niezbadane wychodzące krawędzie. Po zbadaniu wszystkich krawędzi wychodzących z danego wierzchołka v wracamy do wierzchołka, z którego v został odwiedzony.

Algorytmy grafowe

Przeszukiwanie grafu w głąb (DFS):

- Przeszukiwanie w głąb polega na sięganiu "głębiej" w graf (do dalszych wierzchołków) tak długo, jak to możliwe – stąd nazwa algorytmu
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - W każdym kroku algorytmu badamy krawędzie ostatniego odwiedzonego wierzchołka, który ma jeszcze niezbadane wychodzące krawędzie. Po zbadaniu wszystkich krawędzi wychodzących z danego wierzchołka v wracamy do wierzchołka, z którego v został odwiedzony.

Algorytmy grafowe

Przeszukiwanie grafu w głąb (DFS):

- Przeszukiwanie w głąb polega na sięganiu "głębiej" w graf (do dalszych wierzchołków) tak długo, jak to możliwe – stąd nazwa algorytmu
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek u posiada następujące atrybuty:
  - `u.visited` - true lub false; mówi, czy wierzchołek u był już odwiedzony
  - `u.prev` - poprzednik wierzchołka u na ścieżce z wierzchołka s
  - W każdym kroku algorytmu badamy krawędzie ostatniego odwiedzonego wierzchołka, który ma jeszcze niezbadane wychodzące krawędzie. Po zbadaniu wszystkich krawędzi wychodzących z danego wierzchołka v wracamy do wierzchołka, z którego v został odwiedzony.

Algorytmy grafowe

```
procedure DFS(G)
 for each u ∈ V
  u.visited = false
  u.prev = NIL
 for each u ∈ V
  if u.visited = false
   DFS-VISIT(u)
procedure DFS-VISIT(u)
 u.visited = true
 for each v ∈ A[u]
  if v.visited = false
   v.prev = u
   DFS-VISIT(v)
```

Algorytmy grafowe

```
procedure DFS(G)
 for each u ∈ V
  u.visited = false
  u.prev = NIL
 for each u ∈ V
  if u.visited = false
   DFS-VISIT(u)
procedure DFS-VISIT(u)
 u.visited = true
 for each v ∈ A[u]
  if v.visited = false
   v.prev = u
   DFS-VISIT(v)
```

Algorytmy grafowe

- DFS można zaimplementować bez rekurencji, wykorzystując stos do przechowywania wierzchołków, które zostały już odwiedzone, ale mają niezbadane wychodzące krawędzie
- Krawędzie postaci $(v.prev, v)$ tworzą las przeszukiwania w głąb
- W przedstawionym pseudokodzie DFS odwiedza wszystkie wierzchołki grafu G. Można jednak ograniczyć się do odwiedzenia wszystkich wierzchołków osiągalnych z danego źródła s, podobnie jak w przedstawionym pseudokodzie algorytmu BFS
- Złożoność algorytmu: $O(|V| + |E|)$

Algorytmy grafowe

- DFS można zaimplementować bez rekurencji, wykorzystując stos do przechowywania wierzchołków, które zostały już odwiedzone, ale mają niezbadane wychodzące krawędzie
- Krawędzie postaci $(v.prev, v)$ tworzą las przeszukiwania w głąb
- W przedstawionym pseudokodzie DFS odwiedza wszystkie wierzchołki grafu G. Można jednak ograniczyć się do odwiedzenia wszystkich wierzchołków osiągalnych z danego źródła s, podobnie jak w przedstawionym pseudokodzie algorytmu BFS
- Złożoność algorytmu: $O(|V| + |E|)$

Algorytmy grafowe

- DFS można zaimplementować bez rekurencji, wykorzystując stos do przechowywania wierzchołków, które zostały już odwiedzone, ale mają niezbadane wychodzące krawędzie
- Krawędzie postaci $(v.prev, v)$ tworzą las przeszukiwania w głąb
- W przedstawionym pseudokodzie DFS odwiedza wszystkie wierzchołki grafu G. Można jednak ograniczyć się do odwiedzenia wszystkich wierzchołków osiągalnych z danego źródła s, podobnie jak w przedstawionym pseudokodzie algorytmu BFS
- Złożoność algorytmu: $O(|V| + |E|)$

Algorytmy grafowe

- DFS można zaimplementować bez rekurencji, wykorzystując stos do przechowywania wierzchołków, które zostały już odwiedzone, ale mają niezbadane wychodzące krawędzie
- Krawędzie postaci $(v.prev, v)$ tworzą las przeszukiwania w głąb
- W przedstawionym pseudokodzie DFS odwiedza wszystkie wierzchołki grafu G. Można jednak ograniczyć się do odwiedzenia wszystkich wierzchołków osiągalnych z danego źródła s, podobnie jak w przedstawionym pseudokodzie algorytmu BFS
- Złożoność algorytmu: $O(|V| + |E|)$

Algorytmy grafowe

Przykładowe zastosowania algorytmu DFS:

- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- wyznaczanie silnie spójnych składowych grafu skierowanego
- sprawdzanie, czy graf jest acykliczny
- sortowanie topologiczne acyklicznego grafu skierowanego

Algorytmy grafowe

Przykładowe zastosowania algorytmu DFS:

- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- wyznaczanie silnie spójnych składowych grafu skierowanego
- sprawdzanie, czy graf jest acykliczny
- sortowanie topologiczne acyklicznego grafu skierowanego

Algorytmy grafowe

Przykładowe zastosowania algorytmu DFS:

- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- wyznaczanie silnie spójnych składowych grafu skierowanego
- sprawdzanie, czy graf jest acykliczny
- sortowanie topologiczne acyklicznego grafu skierowanego

Algorytmy grafowe

Przykładowe zastosowania algorytmu DFS:

- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- wyznaczanie silnie spójnych składowych grafu skierowanego
- sprawdzanie, czy graf jest acykliczny
- sortowanie topologiczne acyklicznego grafu skierowanego

Algorytmy grafowe

Przykładowe zastosowania algorytmu DFS:

- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- wyznaczanie silnie spójnych składowych grafu skierowanego
- sprawdzanie, czy graf jest acykliczny
- sortowanie topologiczne acyklicznego grafu skierowanego

Algorytmy grafowe

Przykładowe zastosowania algorytmu DFS:

- sprawdzanie spójności grafu
- znajdowanie wszystkich składowych spójności grafu
- wyznaczanie silnie spójnych składowych grafu skierowanego
- sprawdzanie, czy graf jest acykliczny
- sortowanie topologiczne acyklicznego grafu skierowanego

Algorytmy grafowe

Graf ważony to uporządkowana trójka $G = (V, E, w)$, gdzie

- V to skończony zbiór wierzchołków grafu G
- E to zbiór krawędzi grafu G
- w to funkcja przyporządkowująca każdej krawędzi liczbę rzeczywistą zwaną jej wagą

Wagi krawędzi możemy interpretować np. jako długości dróg między wierzchołkami lub koszty podróży między nimi.

Algorytmy grafowe

Graf ważony to uporządkowana trójka $G = (V, E, w)$, gdzie

- V to skończony zbiór wierzchołków grafu G
- E to zbiór krawędzi grafu G
- w to funkcja przyporządkowująca każdej krawędzi liczbę rzeczywistą zwaną jej wagą

Wagi krawędzi możemy interpretować np. jako długości dróg między wierzchołkami lub koszty podróży między nimi.

Algorytmy grafowe

Graf ważony to uporządkowana trójka $G = (V, E, w)$, gdzie

- V to skończony zbiór wierzchołków grafu G
- E to zbiór krawędzi grafu G
- w to funkcja przyporządkowująca każdej krawędzi liczbę rzeczywistą zwaną jej wagą

Wagi krawędzi możemy interpretować np. jako długości dróg między wierzchołkami lub koszty podróży między nimi.

Algorytmy grafowe

Graf ważony to uporządkowana trójka $G = (V, E, w)$, gdzie

- V to skończony zbiór wierzchołków grafu G
- E to zbiór krawędzi grafu G
- w to funkcja przyporządkowująca każdej krawędzi liczbę rzeczywistą zwaną jej wagą

Wagi krawędzi możemy interpretować np. jako długości dróg między wierzchołkami lub koszty podróży między nimi.

Algorytmy grafowe

Graf ważony to uporządkowana trójka $G = (V, E, w)$, gdzie

- V to skończony zbiór wierzchołków grafu G
- E to zbiór krawędzi grafu G
- w to funkcja przyporządkowująca każdej krawędzi liczbę rzeczywistą zwaną jej wagą

Wagi krawędzi możemy interpretować np. jako długości dróg między wierzchołkami lub koszty podróży między nimi.

Algorytmy grafowe

Graf ważony to uporządkowana trójka $G = (V, E, w)$, gdzie

- V to skończony zbiór wierzchołków grafu G
- E to zbiór krawędzi grafu G
- w to funkcja przyporządkowująca każdej krawędzi liczbę rzeczywistą zwaną jej wagą

Wagi krawędzi możemy interpretować np. jako długości dróg między wierzchołkami lub koszty podróży między nimi.

Algorytmy grafowe

- Wagą ścieżki $p = [v_0, v_1, \ldots, v_k]$ w grafie ważonym nazywamy sumę wag tworzących ją krawędzi

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Najkrótszą ścieżką z wierzchołka u do v nazywamy ścieżkę o minimalnej wadze prowadzącą z u do v
- Może istnieć wiele najkrótszych ścieżek z wierzchołka u do v
- Jeśli wierzchołek v nie jest osiągalny z u, to najkrótsza ścieżka z u do v nie istnieje, a wagę najkrótszej ścieżki definiujemy jako ∞

Algorytmy grafowe

- Wagą ścieżki $p = [v_0, v_1, \ldots, v_k]$ w grafie ważonym nazywamy sumę wag tworzących ją krawędzi

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Najkrótszą ścieżką z wierzchołka u do v nazywamy ścieżkę o minimalnej wadze prowadzącą z u do v
- Może istnieć wiele najkrótszych ścieżek z wierzchołka u do v
- Jeśli wierzchołek v nie jest osiągalny z u, to najkrótsza ścieżka z u do v nie istnieje, a wagę najkrótszej ścieżki definiujemy jako ∞

Algorytmy grafowe

- Wagą ścieżki $p = [v_0, v_1, \ldots, v_k]$ w grafie ważonym nazywamy sumę wag tworzących ją krawędzi

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Najkrótszą ścieżką z wierzchołka u do v nazywamy ścieżkę o minimalnej wadze prowadzącą z u do v
- Może istnieć wiele najkrótszych ścieżek z wierzchołka u do v
- Jeśli wierzchołek v nie jest osiągalny z u, to najkrótsza ścieżka z u do v nie istnieje, a wagę najkrótszej ścieżki definiujemy jako ∞

Algorytmy grafowe

- Wagą ścieżki $p = [v_0, v_1, \ldots, v_k]$ w grafie ważonym nazywamy sumę wag tworzących ją krawędzi

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Najkrótszą ścieżką z wierzchołka u do v nazywamy ścieżkę o minimalnej wadze prowadzącą z u do v
- Może istnieć wiele najkrótszych ścieżek z wierzchołka u do v
- Jeśli wierzchołek v nie jest osiągalny z u, to najkrótsza ścieżka z u do v nie istnieje, a wagę najkrótszej ścieżki definiujemy jako ∞

Algorytmy grafowe

Problemy ścieżkowe:

- najkrótsze ścieżki z jednym źródłem - należy znaleźć najkrótsze ścieżki z ustalonego źródła s do wszystkich wierzchołków grafu
- najkrótsze ścieżki z jednym wierzchołkiem docelowym - należy znaleźć najkrótsze ścieżki z wszystkich wierzchołków grafu do ustalonego wierzchołka docelowego t
- najkrótsza ścieżka między parą wierzchołków - należy znaleźć najkrótszą ścieżkę z ustalonego wierzchołka u do ustalonego wierzchołka v
- najkrótsze ścieżki między wszystkimi parami wierzchołków - dla każdej pary wierzchołków u, v należy znaleźć najkrótszą ścieżkę z u do v

Algorytmy grafowe

Problemy ścieżkowe:

- najkrótsze ścieżki z jednym źródłem - należy znaleźć najkrótsze ścieżki z ustalonego źródła s do wszystkich wierzchołków grafu
- najkrótsze ścieżki z jednym wierzchołkiem docelowym - należy znaleźć najkrótsze ścieżki z wszystkich wierzchołków grafu do ustalonego wierzchołka docelowego t
- najkrótsza ścieżka między parą wierzchołków - należy znaleźć najkrótszą ścieżkę z ustalonego wierzchołka u do ustalonego wierzchołka v
- najkrótsze ścieżki między wszystkimi parami wierzchołków - dla każdej pary wierzchołków u, v należy znaleźć najkrótszą ścieżkę z u do v

Algorytmy grafowe

Problemy ścieżkowe:

- najkrótsze ścieżki z jednym źródłem - należy znaleźć najkrótsze ścieżki z ustalonego źródła s do wszystkich wierzchołków grafu
- najkrótsze ścieżki z jednym wierzchołkiem docelowym - należy znaleźć najkrótsze ścieżki z wszystkich wierzchołków grafu do ustalonego wierzchołka docelowego t
- najkrótsza ścieżka między parą wierzchołków - należy znaleźć najkrótszą ścieżkę z ustalonego wierzchołka u do ustalonego wierzchołka v
- najkrótsze ścieżki między wszystkimi parami wierzchołków - dla każdej pary wierzchołków u, v należy znaleźć najkrótszą ścieżkę z u do v

Algorytmy grafowe

Problemy ścieżkowe:

- najkrótsze ścieżki z jednym źródłem - należy znaleźć najkrótsze ścieżki z ustalonego źródła s do wszystkich wierzchołków grafu
- najkrótsze ścieżki z jednym wierzchołkiem docelowym - należy znaleźć najkrótsze ścieżki z wszystkich wierzchołków grafu do ustalonego wierzchołka docelowego t
- najkrótsza ścieżka między parą wierzchołków - należy znaleźć najkrótszą ścieżkę z ustalonego wierzchołka u do ustalonego wierzchołka v
- najkrótsze ścieżki między wszystkimi parami wierzchołków - dla każdej pary wierzchołków u, v należy znaleźć najkrótszą ścieżkę z u do v

Algorytmy grafowe

Problemy ścieżkowe:

- najkrótsze ścieżki z jednym źródłem - należy znaleźć najkrótsze ścieżki z ustalonego źródła s do wszystkich wierzchołków grafu
- najkrótsze ścieżki z jednym wierzchołkiem docelowym - należy znaleźć najkrótsze ścieżki z wszystkich wierzchołków grafu do ustalonego wierzchołka docelowego t
- najkrótsza ścieżka między parą wierzchołków - należy znaleźć najkrótszą ścieżkę z ustalonego wierzchołka u do ustalonego wierzchołka v
- najkrótsze ścieżki między wszystkimi parami wierzchołków - dla każdej pary wierzchołków u, v należy znaleźć najkrótszą ścieżkę z u do v

Algorytmy grafowe

Algorytm Dijkstry:

- Algorytm Dijkstry znajduje najkrótsze ścieżki z jednym źródłem s w grafie $G = (V, E, w)$ (skierowanym lub nieskierowanym) z nieujemnymi wagami krawędzi
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek v ma dwa atrybuty:
  - $v.d$ – est. $v.prev$ – NIL
- Na początku algorytmu dla każdego wierzchołka v, $v \neq s$ mamy $v.d = \infty$, $v.prev = NIL$. Dla wierzchołka s mamy $s.d = 0$, $s.prev = NIL$

Algorytmy grafowe

Algorytm Dijkstry:

- Algorytm Dijkstry znajduje najkrótsze ścieżki z jednym źródłem s w grafie $G = (V, E, w)$ (skierowanym lub nieskierowanym) z nieujemnymi wagami krawędzi
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek v ma dwa atrybuty:
  - $v.d$ - ograniczenie wagi najkrótszej ścieżki z s do v (wagi najkrótszej dotychczas znalezionej ścieżki)
  - $v.prev$ - poprzednik wierzchołka v na najkrótszej z dotychczas znalezionych ścieżek z s do v
- Na początku algorytmu dla każdego wierzchołka $v \neq s$ mamy $v.d = \infty$ i $v.prev = NIL$. Dla wierzchołka s mamy $s.d = 0$ i $s.prev = NIL$

Algorytmy grafowe

Algorytm Dijkstry:

- Algorytm Dijkstry znajduje najkrótsze ścieżki z jednym źródłem s w grafie $G = (V, E, w)$ (skierowanym lub nieskierowanym) z nieujemnymi wagami krawędzi
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek v ma dwa atrybuty:
  - $v.d$ - górne ograniczenie wagi najkrótszej ścieżki z s do v (waga najkrótszej dotychczas znalezionej ścieżki)
  - $v.prev$ - poprzednik wierzchołka v na najkrótszej znalezionej dotychczas ścieżce z s do v
- Na początku algorytmu dla każdego wierzchołka $v \neq s$ mamy $v.d = \infty$ i $v.prev = NIL$. Dla wierzchołka s mamy $s.d = 0$ i $s.prev = NIL$

Algorytmy grafowe

Algorytm Dijkstry:

- Algorytm Dijkstry znajduje najkrótsze ścieżki z jednym źródłem s w grafie $G = (V, E, w)$ (skierowanym lub nieskierowanym) z nieujemnymi wagami krawędzi
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek v ma dwa atrybuty:
  - `v.d` - górne ograniczenie wagi najkrótszej ścieżki z s do v (waga najkrótszej dotychczas znalezionej ścieżki)
  - `v.prev` - poprzednik wierzchołka v na najkrótszej znalezionej dotychczas ścieżce z s do v
- Na początku algorytmu dla każdego wierzchołka $v \neq s$ mamy $v.d = \infty$ i $v.prev = NIL$. Dla wierzchołka s mamy $s.d = 0$ i s.prev = NIL

Algorytmy grafowe

Algorytm Dijkstry:

- Algorytm Dijkstry znajduje najkrótsze ścieżki z jednym źródłem s w grafie $G = (V, E, w)$ (skierowanym lub nieskierowanym) z nieujemnymi wagami krawędzi
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek v ma dwa atrybuty:
  - `v.d` - górne ograniczenie wagi najkrótszej ścieżki z s do v (waga najkrótszej dotychczas znalezionej ścieżki)
  - `v.prev` - poprzednik wierzchołka v na najkrótszej znalezionej dotychczas ścieżce z s do v
- Na początku algorytmu dla każdego wierzchołka $v \neq s$ mamy $v.d = \infty$ i $v.prev = NIL$. Dla wierzchołka s mamy $s.d = 0$ i s.prev = NIL

Algorytmy grafowe

Algorytm Dijkstry:

- Algorytm Dijkstry znajduje najkrótsze ścieżki z jednym źródłem s w grafie $G = (V, E, w)$ (skierowanym lub nieskierowanym) z nieujemnymi wagami krawędzi
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek v ma dwa atrybuty:
  - `v.d` - górne ograniczenie wagi najkrótszej ścieżki z s do v (waga najkrótszej dotychczas znalezionej ścieżki)
  - `v.prev` - poprzednik wierzchołka v na najkrótszej znalezionej dotychczas ścieżce z s do v
- Na początku algorytmu dla każdego wierzchołka $v \neq s$ mamy $v.d = \infty$ i $v.prev = NIL$. Dla wierzchołka s mamy $s.d = 0$ i s.prev = NIL

Algorytmy grafowe

Algorytm Dijkstry:

- Algorytm Dijkstry znajduje najkrótsze ścieżki z jednym źródłem s w grafie $G = (V, E, w)$ (skierowanym lub nieskierowanym) z nieujemnymi wagami krawędzi
- Graf G jest reprezentowany przez listy sąsiedztwa: lista $A[u]$ zawiera wszystkie wierzchołki v, dla których $(u, v) \in E$
- Każdy wierzchołek v ma dwa atrybuty:
  - `v.d` - górne ograniczenie wagi najkrótszej ścieżki z s do v (waga najkrótszej dotychczas znalezionej ścieżki)
  - `v.prev` - poprzednik wierzchołka v na najkrótszej znalezionej dotychczas ścieżce z s do v
- Na początku algorytmu dla każdego wierzchołka $v \neq s$ mamy $v.d = \infty$ i $v.prev = NIL$. Dla wierzchołka s mamy $s.d = 0$ i s.prev = NIL

Algorytmy grafowe

- Wierzchołki grafu przechowywane są w kolejce priorytetowej typu min Q. Priorytetem (kluczem) wierzchołka u jest $u.d$
- W każdym głównym kroku algorytmu pobieramy z kolejki wierzchołek u o najmniejszym kluczu. Wartość $u.d$ jest wówczas długością najkrótszej ścieżki z s do u
- Przeglądamy wszystkie krawędzie (u, v) i wykonujemy ich relaksację: sprawdzamy, czy przechodząc do wierzchołka v bezpośrednio z wierzchołka u znajdziemy krótszą niż dotychczas ścieżkę do v. Jeśli tak, to aktualizujemy wartości $v.d$ i $v.prev$
- Po wykonaniu algorytmu znalezione najkrótsze ścieżki można wypisać w taki sam sposób, jak ścieżki znajdowane przez algorytm BFS

Algorytmy grafowe

- Wierzchołki grafu przechowywane są w kolejce priorytetowej typu min Q. Priorytetem (kluczem) wierzchołka u jest $u.d$
- W każdym głównym kroku algorytmu pobieramy z kolejki wierzchołek u o najmniejszym kluczu. Wartość $u.d$ jest wówczas długością najkrótszej ścieżki z s do u
- Przeglądamy wszystkie krawędzie (u, v) i wykonujemy ich relaksację: sprawdzamy, czy przechodząc do wierzchołka v bezpośrednio z wierzchołka u znajdziemy krótszą niż dotychczas ścieżkę do v. Jeśli tak, to aktualizujemy wartości $v.d$ i $v.prev$
- Po wykonaniu algorytmu znalezione najkrótsze ścieżki można wypisać w taki sam sposób, jak ścieżki znajdowane przez algorytm BFS

Algorytmy grafowe

- Wierzchołki grafu przechowywane są w kolejce priorytetowej typu min Q. Priorytetem (kluczem) wierzchołka u jest $u.d$
- W każdym głównym kroku algorytmu pobieramy z kolejki wierzchołek u o najmniejszym kluczu. Wartość $u.d$ jest wówczas długością najkrótszej ścieżki z s do u
- Przeglądamy wszystkie krawędzie (u, v) i wykonujemy ich relaksację: sprawdzamy, czy przechodząc do wierzchołka v bezpośrednio z wierzchołka u znajdziemy krótszą niż dotychczas ścieżkę do v. Jeśli tak, to aktualizujemy wartości $v.d$ i $v.prev$
- Po wykonaniu algorytmu znalezione najkrótsze ścieżki można wypisać w taki sam sposób, jak ścieżki znajdowane przez algorytm BFS

Algorytmy grafowe

- Wierzchołki grafu przechowywane są w kolejce priorytetowej typu min Q. Priorytetem (kluczem) wierzchołka u jest $u.d$
- W każdym głównym kroku algorytmu pobieramy z kolejki wierzchołek u o najmniejszym kluczu. Wartość $u.d$ jest wówczas długością najkrótszej ścieżki z s do u
- Przeglądamy wszystkie krawędzie (u, v) i wykonujemy ich relaksację: sprawdzamy, czy przechodząc do wierzchołka v bezpośrednio z wierzchołka u znajdziemy krótszą niż dotychczas ścieżkę do v. Jeśli tak, to aktualizujemy wartości $v.d$ i $v.prev$
- Po wykonaniu algorytmu znalezione najkrótsze ścieżki można wypisać w taki sam sposób, jak ścieżki znajdowane przez algorytm BFS

Algorytmy grafowe

```
procedure DIJKSTRA(G, s)
 for each u ∈ V
  u.d = ∞
  u.prev = NIL
 s.d = 0
 Q = V
 while Q ≠ ϕ
  u = EXTRACT-MIN(Q)
  for each v ∈ A[u]
   RELAX(u,v,w)
procedure RELAX(u,v,w)
 if v.d > u.d + w(u,v)
  v.d = u.d + w(u,v)
  v.prev = u
```

Algorytmy grafowe

```
procedure DIJKSTRA(G, s)
 for each u ∈ V
  u.d = ∞
  u.prev = NIL
 s.d = 0
 Q = V
 while Q ≠ ϕ
  u = EXTRACT-MIN(Q)
  for each v ∈ A[u]
   RELAX(u,v,w)
procedure RELAX(u,v,w)
 if v.d > u.d + w(u,v)
  v.d = u.d + w(u,v)
  v.prev = u
```

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje $O(|V|)$
  - Znalezienie minimum zajmuje $O(|V|)$
  - Zmniejszenie klucza zajmuje $O(1)$
  - Całkowity czas działania $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:
  - Inicjalizacja wierzchołków i stworzenie Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN wymaga czasu $O(\lg |V|)$
  - Aktualizacja $v.d$ to operacja HEAP-DECREASE-KEY, więc zajmuje czas $O(\lg |V|)$
  - Złożoność algorytmu: $O((|V| + |E|) \lg |V|)$. Jeśli wszystkie wierzchołki są osiągalne z s, to $|V| = O(|E|)$, więc algorytm ma złożoność $O(|E| \lg |V|)$.

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:
  - Inicjalizacja wierzchołków i stworzenie Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN wymaga czasu $O(\lg |V|)$
  - Aktualizacja $v.d$ to operacja HEAP-DECREASE-KEY, więc zajmuje czas $O(\lg |V|)$
  - Złożoność algorytmu: $O((|V| + |E|) \lg |V|)$. Jeśli wszystkie wierzchołki są osiągalne z s, to $|V| = O(|E|)$, więc algorytm ma złożoność $O(|E| \lg |V|)$.

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:
  - Inicjalizacja wierzchołków i stworzenie Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN wymaga czasu $O(\lg |V|)$
  - Aktualizacja $v.d$ to operacja HEAP-DECREASE-KEY, więc zajmuje czas $O(\lg |V|)$
  - Złożoność algorytmu: $O((|V| + |E|) \lg |V|)$. Jeśli wszystkie wierzchołki są osiągalne z s, to $|V| = O(|E|)$, więc algorytm ma złożoność $O(|E| \lg |V|)$.

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:
  - Inicjalizacja wierzchołków i stworzenie Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN wymaga czasu $O(\lg |V|)$
  - Aktualizacja $v.d$ to operacja HEAP-DECREASE-KEY, więc zajmuje czas $O(\lg |V|)$
  - Złożoność algorytmu: $O((|V| + |E|) \lg |V|)$. Jeśli wszystkie wierzchołki są osiągalne z s, to $|V| = O(|E|)$, więc algorytm ma złożoność $O(|E| \lg |V|)$.

Algorytmy grafowe

Złożoność algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Pętla while ma $|V|$ iteracji. Każda krawędź (u, v) jest badana dokładnie raz w trakcie działania algorytmu:

- Jeśli Q jest tablicą:
  - Inicjalizacja wierzchołków i stworzenie tablicy Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN to wyszukanie minimum w tablicy, więc wymaga czasu $O(|V|)$
  - Aktualizacja $v.d$ zajmuje czas $O(1)$
  - Złożoność algorytmu: $O(|V|^2 + |E|) = O(|V|^2)$
- Jeśli Q jest kopcem binarnym:
  - Inicjalizacja wierzchołków i stworzenie Q zajmuje czas $O(|V|)$
  - Operacja EXTRACT-MIN wymaga czasu $O(\lg |V|)$
  - Aktualizacja $v.d$ to operacja HEAP-DECREASE-KEY, więc zajmuje czas $O(\lg |V|)$
  - Złożoność algorytmu: $O((|V| + |E|) \lg |V|)$. Jeśli wszystkie wierzchołki są osiągalne z s, to $|V| = O(|E|)$, więc algorytm ma złożoność $O(|E| \lg |V|)$.