06. Grafy – algoritmy

Obsah

- Procházení grafu (prohledávání do šířky a do hloubky)
- Testování souvislosti grafu a určení komponent
- Nalezení cesty v grafu
- Nalezení nejkratší cesty v neohodnoceném grafu
- Nalezení nejkratší cesty v ohodnoceném grafu
- Určení minimální kostry grafu
- Rekurze při práci s grafy
- Efektivita grafových algoritmů
- Příklady využití

Procházení grafu

Existují dva základní způsoby systematického procházení grafu: - Prohledávání do šířky (BFS - Breadth-First Search) - Prohledávání do hloubky (DFS - Depth-First Search)

Prohledávání do šířky (BFS)

BFS je algoritmus, který systematicky prochází všechny vrcholy grafu po úrovních.

Princip: 1. Začneme v počátečním vrcholu 2. Navštívíme všechny jeho sousedy 3. Pak navštívíme všechny dosud nenavštívené sousedy těchto sousedů 4. Pokračujeme, dokud nenavštívíme všechny dostupné vrcholy

Implementace: - Používá datovou strukturu fronta (FIFO) - Každý vrchol je zpracován pouze jednou - Vstupní vrchol vložíme do fronty - V každém kroku: - Vyjmeme vrchol z fronty - Zpracujeme ho - Všechny jeho nenavštívené sousedy označíme jako navštívené a vložíme do fronty

Pseudokód BFS:

```
BFS(G, s):                  // G je graf, s je počáteční vrchol
    pro každý vrchol u v G:
        u.visited = false
        u.distance = ∞
        u.parent = NULL

    s.visited = true
    s.distance = 0
    fronta = prázdná fronta
    přidej s do fronty

    dokud fronta není prázdná:
        u = vyjmi vrchol z fronty
        pro každého souseda v vrcholu u:
            pokud v.visited == false:
                v.visited = true
                v.distance = u.distance + 1
                v.parent = u
                přidej v do fronty
```

Vlastnosti BFS: - Nalezne nejkratší cestu v neohodnoceném grafu - Časová složitost: O(V + E), kde V je počet vrcholů a E je počet hran - Prostorová složitost: O(V) - "Vlnové" procházení grafu - navštěvujeme nejdříve bližší vrcholy

Prohledávání do hloubky (DFS)

DFS je algoritmus, který prochází graf tak, že jde co nejhlouběji podél jedné větve, než se vrátí zpět.

Princip: 1. Začneme v počátečním vrcholu 2. Rekurzivně navštívíme jeden z jeho nenavštívených sousedů 3. Pokračujeme tímto způsobem co nejhlouběji 4. Když nelze jít dál, vracíme se (backtracking) a zkoušíme jinou cestu

Implementace: - Používá zásobník (implicitně rekurzí nebo explicitně) - Každý vrchol je zpracován pouze jednou - Pro každý vrchol si pamatujeme: - Čas objevení (discovery time) - Čas dokončení (finish time)

Pseudokód DFS (rekurzivní):

```
DFS(G):                       // G je graf
    pro každý vrchol u v G:
        u.visited = false
        u.parent = NULL
    time = 0

    pro každý vrchol u v G:
        pokud u.visited == false:
            DFS-Visit(G, u)

DFS-Visit(G, u):
    time = time + 1
    u.discovery_time = time
    u.visited = true

    pro každého souseda v vrcholu u:
        pokud v.visited == false:
            v.parent = u
            DFS-Visit(G, v)

    time = time + 1
    u.finish_time = time
```

Vlastnosti DFS: - Identifikuje komponenty grafu - Detekuje cykly - Může být použit pro topologické řazení - Časová složitost: O(V + E) - Prostorová složitost: O(V) v nejhorším případě (zásobník rekurze)

Testování souvislosti grafu a určení komponent

Graf je souvislý, pokud existuje cesta mezi libovolnými dvěma vrcholy.

Testování souvislosti grafu

Algoritmus testování souvislosti: 1. Spustíme BFS nebo DFS z libovolného vrcholu 2. Po dokončení zkontrolujeme, zda byly všechny vrcholy navštíveny 3. Pokud ano, graf je souvislý, jinak není

Pseudokód:

```
isConnected(G):                  // G je graf
    spusť BFS nebo DFS z libovolného vrcholu

    pro každý vrchol v v G:
        pokud v.visited == false:
            return false    // Graf není souvislý

    return true             // Graf je souvislý
```

Určení komponent grafu

Komponenta grafu je maximální souvislý podgraf. Nesouvislý graf má více komponent.

Algoritmus nalezení komponent: 1. Inicializujeme počítadlo komponent na 0 2. Pro každý nenavštívený vrchol: a. Zvýšíme počítadlo komponent b. Spustíme BFS nebo DFS z tohoto vrcholu c. Všechny navštívené vrcholy patří do aktuální komponenty

Pseudokód:

```
findComponents(G):                // G je graf
    pro každý vrchol u v G:
        u.visited = false
        u.component = -1

    componentCount = 0

    pro každý vrchol u v G:
        pokud u.visited == false:
            componentCount = componentCount + 1
            markComponent(G, u, componentCount)

    return componentCount

markComponent(G, u, componentId):
    u.visited = true
    u.component = componentId

    pro každého souseda v vrcholu u:
        pokud v.visited == false:
            markComponent(G, v, componentId)
```

Nalezení cesty v grafu

Nalezení cesty mezi dvěma vrcholy je základní úloha v teorii grafů.

Nalezení libovolné cesty

K nalezení libovolné cesty mezi vrcholy s a t můžeme použít BFS nebo DFS:

1. Spustíme BFS nebo DFS z vrcholu s
2. Pokud t je dosažitelný, můžeme zrekonstruovat cestu pomocí předchůdců

Rekonstrukce cesty:

```
reconstructPath(s, t):            // s je počáteční vrchol, t je cílový vrchol
    pokud t.visited == false:
        return "Cesta neexistuje"

    cesta = prázdný seznam
    aktuální = t

    dokud aktuální != s:
        přidej aktuální na začátek cesty
        aktuální = aktuální.parent

    přidej s na začátek cesty
    return cesta
```

Nalezení nejkratší cesty v neohodnoceném grafu

V neohodnoceném grafu je nejkratší cesta ta s nejmenším počtem hran. BFS přirozeně nachází nejkratší cestu.

Algoritmus: 1. Spustíme BFS z počátečního vrcholu s 2. Pro každý vrchol si pamatujeme vzdálenost od s a předchůdce 3. Po dokončení BFS můžeme: - Zjistit vzdálenost k libovolnému vrcholu - Zrekonstruovat nejkratší cestu

Nalezení nejkratší cesty v ohodnoceném grafu

Pro nalezení nejkratší cesty v ohodnoceném grafu (kde hrany mají různé váhy) používáme specializované algoritmy.

Dijkstrův algoritmus

Dijkstrův algoritmus nachází nejkratší cesty z jednoho vrcholu do všech ostatních v grafu s nezápornými vahami hran.

Princip: 1. Každému vrcholu přiřadíme prozatímní vzdálenost (počáteční vrchol 0, ostatní ∞) 2. Opakovaně vybíráme vrchol s nejmenší prozatímní vzdáleností 3. Aktualizujeme vzdálenosti jeho sousedů 4. Označíme vybraný vrchol jako navštívený (fixovaný)

Pseudokód:

```
Dijkstra(G, s):                 // G je graf, s je počáteční vrchol
    pro každý vrchol v v G:
        v.distance = ∞
        v.visited = false
        v.parent = NULL

    s.distance = 0
    prioritní_fronta = všechny vrcholy G seřazené podle distance

    dokud prioritní_fronta není prázdná:
        u = vyjmi vrchol s nejmenší distance z prioritní_fronty

        pokud u.distance == ∞:
            break    // Zbývající vrcholy jsou nedosažitelné

        u.visited = true

        pro každou hranu (u, v) s váhou w:
            pokud v.visited == false a u.distance + w < v.distance:
                v.distance = u.distance + w
                v.parent = u
                aktualizuj pozici v v prioritní_frontě
```

Časová složitost: - S binární haldou: O((V + E) log V) - S Fibonacciho haldou: O(E + V log V)

Bellman-Fordův algoritmus

Bellman-Fordův algoritmus nachází nejkratší cesty z jednoho vrcholu do všech ostatních a funguje i pro grafy se zápornými vahami hran (pokud neobsahují záporné cykly).

Princip: 1. Inicializujeme vzdálenosti (počáteční vrchol 0, ostatní ∞) 2. Relaxujeme všechny hrany V-1 krát (V je počet vrcholů) 3. Kontrolujeme, zda existují záporné cykly

Pseudokód:

```
Bellman-Ford(G, s):             // G je graf, s je počáteční vrchol
    pro každý vrchol v v G:
        v.distance = ∞
        v.parent = NULL

    s.distance = 0

    // Relaxace hran
    pro i od 1 do |V| - 1:
        pro každou hranu (u, v) s váhou w:
            pokud u.distance + w < v.distance:
```

```
                v.distance = u.distance + w
                v.parent = u

    // Detekce záporných cyklů
    pro každou hranu (u, v) s váhou w:
        pokud u.distance + w < v.distance:
            return "Graf obsahuje záporný cyklus"

    return "Nejkratší cesty nalezeny"
```

Časová složitost: O(V×E)

Určení minimální kostry grafu

Minimální kostra grafu (MST - Minimum Spanning Tree) je podgraf, který: - Spojuje všechny vrcholy - Je acyklický (strom) - Má minimální celkovou váhu

Pro nalezení MST existují dva hlavní algoritmy:

Kruskalův algoritmus

Kruskalův algoritmus buduje MST postupným přidáváním hran od nejmenší váhy, přičemž se vyhýbá vytvoření cyklu.

Princip: 1. Seřadíme hrany podle váhy vzestupně 2. Inicializujeme každý vrchol jako samostatnou komponentu 3. Procházíme hrany a přidáváme ty, které nespojují vrcholy ze stejné komponenty

Pseudokód:

```
Kruskal(G):                      // G je ohodnocený graf
    T = prázdná množina hran     // Budoucí MST

    seřaď hrany G vzestupně podle váhy
    inicializuj disjunktní množiny pro každý vrchol

    pro každou hranu (u, v) seřazenou podle váhy:
        pokud Find-Set(u) != Find-Set(v):
            přidej hranu (u, v) do T
            Union(u, v)

    return T
```

Časová složitost: O(E log E) nebo O(E log V)

Primův algoritmus

Primův algoritmus buduje MST postupným rozšiřováním souvislého podgrafu.

Princip: 1. Začneme s libovolným vrcholem 2. Opakovaně přidáváme nejlehčí hranu, která spojuje strom s novým vrcholem

Pseudokód:

```
Prim(G, r):                      // G je ohodnocený graf, r je počáteční vrchol
    pro každý vrchol u v G:
        u.key = ∞
        u.parent = NULL
        u.inMST = false

    r.key = 0
    prioritní_fronta = všechny vrcholy G

    dokud prioritní_fronta není prázdná:
```

```
        u = vyjmi vrchol s nejmenší key z prioritní_fronty
        u.inMST = true

        pro každého souseda v vrcholu u s váhou w(u,v):
            pokud v.inMST == false a w(u,v) < v.key:
                v.parent = u
                v.key = w(u,v)
                aktualizuj pozici v v prioritní_frontě
```

Časová složitost: - S binární haldou: O(E log V) - S Fibonacciho haldou: O(E + V log V)

Rekurze při práci s grafy

Rekurze je přirozený způsob práce s grafy díky jejich rekurzivní struktuře. Mnoho grafových algoritmů lze elegantně implementovat rekurzivně.

Výhody rekurze u grafových algoritmů

- Přirozený způsob procházení do hloubky
- Jednodušší implementace některých algoritmů
- Implicitní využití zásobníku pro udržování stavu

Nevýhody rekurze

- Riziko přetečení zásobníku pro velké grafy
- Vyšší režie při volání funkcí
- Může být méně efektivní než iterativní řešení

Vhodné užití rekurze

- DFS a algoritmy založené na DFS
- Dynamické programování na stromech a DAG
- Backtracking algoritmy

Převod rekurze na iteraci

Pro větší grafy je často lepší převést rekurzivní algoritmus na iterativní: - Explicitně udržujeme zásobník - Ručně spravujeme stav algoritmu

Efektivita grafových algoritmů

Algoritmus	Časová složitost	Prostorová složitost
BFS	O(V + E)	O(V)
DFS	O(V + E)	O(V)
Dijkstra (binární halda)	O((V + E) log V)	O(V)
Bellman-Ford	O(V × E)	O(V)
Floyd-Warshall	O(V^3)	O(V^2)
Kruskal	O(E log E)	O(V)
Prim (binární halda)	O(E log V)	O(V)

Příklady využití

Prohledávání do šířky (BFS)

- Hledání nejkratší cesty v neohodnoceném grafu
- Test bipartitnosti grafu
- Hledání komponent v neorientovaném grafu
- Rozpoznávání vzorů v sítích

Prohledávání do hloubky (DFS)

- Topologické řazení
- Detekce cyklů
- Hledání silně souvislých komponent (Tarjanova algoritmus)
- Řešení labyrintů

Dijkstrův algoritmus

- GPS navigace
- Směrování v počítačových sítích
- Plánování cest

Algoritmy pro minimální kostru

- Návrh telekomunikačních sítí
- Clustrování dat
- Aproximační algoritmy pro NP-těžké problémy