BFS, DFS, FRONTA, ZÁSOBNÍK, PRIORITNÍ FRONTA, HALDA

Petr Ryšavý

19. září 2017

Katedra počítačů, FEL, ČVUT

PROHLEDÁVÁNÍ GRAFŮ

Proč prohledávání grafů

- Zkontrolovat, zda je síť spojitá.
- Hledání nejkratší cesty, plánování cest.
- Prohledávání stavového prostoru, formulování plánu (např. jak vyřešit sudoku).
- Spočíst komponenty grafu.

Požadavky na algoritmus prohledávající graf

- Algoritmus musí umět nalézt cestu do všech vrcholů, které jsou dosažitelné.
- Nechceme procházet žádné části grafu vícekrát.
  - Požadujeme lineární čas běhu ($\mathcal{O}(m+n)$).

Obecný algoritmus

function GENERIC-GRAPH-SEARCH(*graph*, s)

  Označ s jako navštívené

  while lze nalézt hranu (u, v), že u bylo navštíveno a v ne **do**

    $(u, v) \leftarrow$ libovolná hrana, kde u bylo navštíveno a v ne

    Označ v jako navštívené

  end while

end function

Prohledávání je úplné

Tvrzení *Na konci je vrchol* v *navštívený právě tehdy, když* G *obsahuje cestu z* s *do* v.

Důkaz

Rozdíly mezi BFS a DFS

Algoritmy se odlišují podle toho v jakém pořadí vrcholy prohledávají

Rozdíly mezi BFS a DFS

Algoritmy se odlišují podle toho v jakém pořadí vrcholy prohledávají

- BFS (Breadth-First Search, prohledávání do šířky)
  - Vrcholy jsou prohledávány po úrovních.
  - Sousedi jedné úrovně tvoří další úroveň.
  - Může počítat nejkratší cesty a komponenty souvislosti.
  - $\mathcal{O}(m+n)$ s frontou.

Rozdíly mezi BFS a DFS

Algoritmy se odlišují podle toho v jakém pořadí vrcholy prohledávají

- BFS (Breadth-First Search, prohledávání do šířky)
  - Vrcholy jsou prohledávány po úrovních.
  - Sousedi jedné úrovně tvoří další úroveň.
  - Může počítat nejkratší cesty a komponenty souvislosti.
  - $\mathcal{O}(m+n)$ s frontou.
- DFS (Depth-First Search, prohledávání do hloubky)
  - Prohledáváme stále hlouběji, backtrackujeme jen když musíme.
  - Počítá topologické očíslování a silné komponenty souvislosti.
  - $\mathcal{O}(m+n)$ se zásobníkem.

Zásobník (anglicky *stack*)

- Datová struktura pro uchovávání dat
- Data přicházejí a odcházejí v pořadí *last in first out*
- Data přidáváme i odebíráme z konce
- Implementace obvykle pomocí pole

Fronta (anglicky *queue*)

- Datová struktura pro uchovávání dat
- Data přicházejí a odcházejí v pořadí *first in first out*
- Data přidáváme na konec, odebíráme ze začátku
- Implementace obvykle pomocí kruhového bufferu nebo spojového seznamu

Čas na programování

- Proveďte úkoly 1.-5.
- Popis i implementaci v Javě naleznete na `http://introcs.cs.princeton.edu/java/43stack/`. Zde je implementace pomocí spojového seznamu.
- Implementace fronty v C++ naleznete na `http://www.programming-techniques.com/2011/11/queue-is-order-collection-of-items-from.html` a zásobníku na `http://www.algolist.net/Data_structures/Stack/Array-based_implementation`.
- Každopádně obě struktury zkuste naimplementovat sami a neinspirujte se referencí.

Přestávka

BFS

Pseudokód

```
function BREADTH-FIRST-SEARCH(graph, s)
    Q ← new FIFO queue
    ADD(Q, s)
    MARK-VISITED(s)
    while SIZE(Q) ≠ 0 do
        v ← REMOVE(Q)
        for all edges (v, w) do
            if UNVISITED(w) then
                MARK-VISITED(w)
                ADD(Q, w)
            end if
        end for
    end while
end function
```

Příklad

BFS splňuje požadavky na prohledávací algoritmus

Tvrzení *Na konci BFS navštíví* $v \quad \Leftrightarrow \quad$ *v* G *existuje cesta z* s *do* v*.*

BFS splňuje požadavky na prohledávací algoritmus

Tvrzení *Na konci BFS navštíví* $v \quad \Leftrightarrow \quad$ *v* G *existuje cesta z* s *do* v*.*

Důkaz

BFS splňuje požadavky na prohledávací algoritmus

Tvrzení *Na konci BFS navštíví* $v \quad \Leftrightarrow \quad$ *v* G *existuje cesta z* s *do* v.

Důkaz

Tvrzení *Čas běhu BFS je* $\mathcal{O}(m_s + n_s)$.

BFS splňuje požadavky na prohledávací algoritmus

Tvrzení *Na konci BFS navštíví* $v \quad \Leftrightarrow \quad$ *v* G *existuje cesta z* s *do* v.

Důkaz

Tvrzení *Čas běhu BFS je* $\mathcal{O}(m_s + n_s)$.

Důkaz

Aplikace na hledání nejkratších cest

Cíl: spočíst $\operatorname{dist}(v)$, což je nejmenší počet hran na cestě z s do v.

Aplikace na hledání nejkratších cest

Cíl: spočíst $\mathrm{dist}(v)$, což je nejmenší počet hran na cestě z s do v.

```
function Breadth-First-Search(graph, s)
    Q ← new FIFO queue
    Add(Q, s)
    Mark-Visited(s)
    dist(v) = { 0, pro v = s,
              { ∞  jinak.
    while Size(Q) ≠ 0 do
        v ← Remove(Q)
        for all edges (v, w) do
            if Unvisited(w) then
                Mark-Visited(w)
                Add(Q, w)
                dist(w) = dist(v) + 1
            end if
        end for
    end while
```

Příklad

BFS hledá nejkratší cesty

Tvrzení *Na konci BFS platí* $\mathrm{dist}(v) = i \quad \Leftrightarrow \quad v$ *leží v* i*-té vrstvě.*

BFS hledá nejkratší cesty

Tvrzení *Na konci BFS platí* $\mathrm{dist}(v) = i \quad \Leftrightarrow \quad v$ *leží v* i*-té vrstvě.*

Důkaz

Programování

- Proveďte úkoly 6.-8.
- Vzorovou implementaci prohledávání do šířky naleznete na `http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/`. Opět se ale snažte vše naimplementovat a odladit sami.

DFS

Příklad

DFS prohledává graf více agresivně a vrací se co nejméně.

Pseudokód

Nahradíme frontu zásobníkem

```
function DEPTH-FIRST-SEARCH(graph, s)
    Q ← new LIFO stack
    ADD(Q, s)
    MARK-VISITED(s)
    while SIZE(Q) ≠ 0 do
        v ← REMOVE(Q)
        for all edges (v, w) do
            if UNVISITED(w) then
                MARK-VISITED(w)
                ADD(Q, w)
            end if
        end for
    end while
end function
```

Pseudokód, rekurzivně

```
function DEPTH-FIRST-SEARCH(graph, s)
    MARK-VISITED(s)
    for all edges (s, v) do
        if UNVISITED(v) then
            DEPTH-FIRST-SEARCH(graph, v)
        end if
    end for
end function
```

DFS splňuje požadavky na prohledávací algoritmus

Tvrzení *Na konci DFS navštíví* $v \quad \Leftrightarrow \quad$ *v* G *existuje cesta z* s *do* v*.*

DFS splňuje požadavky na prohledávací algoritmus

Tvrzení *Na konci DFS navštíví v* $\quad\Leftrightarrow\quad$ *v G existuje cesta z s do v.*

Důkaz

DFS splňuje požadavky na prohledávací algoritmus

Tvrzení *Na konci DFS navštíví* $v \quad \Leftrightarrow \quad$ *v* G *existuje cesta z* s *do* v.

Důkaz

Tvrzení *Čas běhu DFS je* $\mathcal{O}(m_s + n_s)$.

DFS splňuje požadavky na prohledávací algoritmus

Tvrzení *Na konci DFS navštíví* $v \quad \Leftrightarrow \quad$ *v* G *existuje cesta z* s *do* v.

Důkaz

Tvrzení *Čas běhu DFS je* $\mathcal{O}(m_s + n_s)$.

Důkaz

Programování

- Proveďte úkoly 9.-11. Tentokrát ale použijte prohledávání do hloubky místo do šířky.
- Vzorovou implementaci prohledávání do hloubky naleznete na `http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/`. Opět se ale snažte vše naimplementovat a odladit sami.

Přestávka

HALDA

Prioritní fronta (anglicky *priority queue*)

- Kontejner na objekty, které jsou identifikovány klíčem
  - Osoby, odkazy v internetu, události s časy, atd.
- `insert`- přidej nový objekt do prioritní fronty
- `extract-min`- odeber z haldy objekt s nejmenším klíčem (pflichta se řeší libovolně)[1]

[1] Někdy prioritní fronta místo `extract-min` nabízí operaci `extract-max`. Nikdy ale obě zároveň.

Halda (anglicky *heap*)

- Podporuje obě operace v $\mathcal{O}(\log n)$.
- Nejčastější implementace prioritní fronty.
- Navíc operace
  - `heapify`- inicializace haldy v lineárním čase
  - `delete`- odstranění libovolného prvku z haldy $\mathcal{O}(\log n)$.

Užití haldy

- Pokud objevíme v algoritmu opakované počítání minima.
  - Např. rozdíl mezi *selection-sort* a *heap-sort*.

Užití haldy

- Pokud objevíme v algoritmu opakované počítání minima.
  - Např. rozdíl mezi *selection-sort* a *heap-sort*.
- Plánovač úloh
  - Například události s časovým klíčem, např. ve hře.

Užití haldy

- Pokud objevíme v algoritmu opakované počítání minima.
  - Např. rozdíl mezi *selection-sort* a *heap-sort*.
- Plánovač úloh
  - Například události s časovým klíčem, např. ve hře.
- *Median Maintenance*
  - sekvence $x_1, x_2, \ldots, x_n$, dodáváme prvky jeden po druhém,
  - v každém kroku je třeba říci medián z čísel $x_1, x_2, \ldots, x_i$,
  - podmínkou je, že to stíháme v $\mathcal{O}(\log n)$.

Užití haldy

- Pokud objevíme v algoritmu opakované počítání minima.
  - Např. rozdíl mezi *selection-sort* a *heap-sort*.
- Plánovač úloh
  - Například události s časovým klíčem, např. ve hře.
- *Median Maintenance*
  - sekvence $x_1, x_2, \ldots, x_n$, dodáváme prvky jeden po druhém,
  - v každém kroku je třeba říci medián z čísel $x_1, x_2, \ldots, x_i$,
  - podmínkou je, že to stíháme v $\mathcal{O}(\log n)$.
- zrychlení Dijkstrova algoritmu
  - Naivně $\mathcal{O}(mn)$, s haldou $\mathcal{O}(m \log n)$.

Programování - halda uživatelsky

- Proveďte úkoly 12.-15.

Přestávka

IMPLEMENTACE HALDY

Definice haldy

2 pohledy - jako strom a jako pole

- Halda je kořenový, binární strom, který je co nejvíce vyvážený.
- Poslední úroveň je zaplňována zleva.
- Je splněná vlastnost haldy:

$$\forall x : \mathrm{key}[x] \leq \text{ klíče potomků uzlu } x.$$

Důsledek

V každé haldě je minimální prvek uložen v kořeni.

Implementace v poli

- Očíslujme uzly podle úrovní
- Efektivnější na paměť (nepotřebujeme žádné ukazatele navíc)
- Efektivní vzorce pro výpočet adres rodiče a potomků

Implementace v poli

- Očíslujme uzly podle úrovní
- Efektivnější na paměť (nepotřebujeme žádné ukazatele navíc)
- Efektivní vzorce pro výpočet adres rodiče a potomků

$$\mathrm{parent}(i) = \begin{cases} \frac{i}{2} & i \text{ sudé}, \\ \lfloor \frac{i}{2} \rfloor & i \text{ liché}, \end{cases}$$

$$\mathrm{children}(i) = \{2i, 2i+1\}$$

`insert` a probublávání nahoru

- Vložíme nový klíč na konec haldy.
- Probubláváme ho nahoru tak dlouho, odkud není vlastnost haldy splněna.
- Čas běhu je $\log(n)$.

`extract-min` a probublávání nahoru

- Odebereme kořen.
- Nahradíme ho posledním uzlem.
- Probubláváme nový kořen dolů.
- Vždy prohazujeme s menším z potomků.

Programování - vlastní implementace haldy

- Proveďte úkol 16. Pokud neprogramujete v Javě, ale v C++, zkuste dodržet strukturu třídy.
- Vlastní implementaci haldy budeme potřebovat zítra v Dijkstrově algoritmu, dejte si tedy záležet.
- Popis a implementaci lze opět nalézt na `http://algs4.cs.princeton.edu/24pq/`.

Děkuji za pozornost.

Čas na otázky!