Wstęp do Programowania potok funkcyjny

Marcin Kubica

2010/2011

Outline

1. Grafy
   - Grafy
   - Przeszukiwanie grafów

Podstawowe pojęcia

Definition

- Graf = wierzchołki + krawędzie.
- Krawędzie muszą mieć różne końce.
- Między dwoma wierzchołkami może być co najwyżej jedna krawędź.
- Inne warianty: grafy z pętelkami, multi-grafy.
- Nieskierowane i skierowane.
- Krawędź nieskierowaną utożsamiamy z parą krawędzi skierowanych w obu kierunkach.
- Zakładamy, że wierzchołki są ponumerowane liczbami całkowitymi od 0 do $n-1$.
- m = liczba krawędzi.

Implementacja grafów

Example

Interfejs modułu implementującego grafy nieskierowane z etykietowanymi krawędziami:

```
module type GRAPH_LABELS = sig type label end;;

module type GRAPH =
  sig
    type label
    type graph
    val init : int -> graph
    val size : graph -> int
    val insert_directed_edge :
                              graph -> int -> int -> label -> unit
    val insert_edge : graph -> int -> int -> label -> unit
    val neighbours : graph -> int -> (int*label) list
  end;;

module type GRAPH_FUNC =
  functor (L : GRAPH_LABELS) -> GRAPH with type label = L.label;;
```

Implementacja grafów

Dwa sposoby reprezentacji grafów:

- Macierz sąsiedztwa:
  - Macierz $n \times n$ wartości logicznych.
  - $E[v, w] \equiv$ istnieje krawędź z v do w.
  - Grafy nieskierowane — macierz jest symetryczna.
  - Złożoność pamięciowa $\Theta(n^2)$.
  - Dodanie, usunięcie, sprawdzenie krawędzi — czas $\Theta(1)$.
  - Lista sąsiadów — czas $\Theta(n)$.
- Listy incydencji:
  - Lista wszystkich sąsiadów danego wierzchołka.
  - Graf = tablica list sąsiedztwa.
  - Dodanie nowej krawędzi — czas $\Theta(1)$.
  - Złożoność pamięciowa $\Theta(n + m)$.
  - Usunięcie, sprawdzenie krawędzi — czas $\Theta(\text{stopień})$.
  - Lista sąsiadów — czas $\Theta(1)$.

Zastosujemy listy sąsiedztwa.

Implementacja grafów

Example

```
module Graph : GRAPH_FUNC = functor (L : GRAPH_LABELS) ->
  struct
    type label = L.label
    type graph = {n : int; e : (int*label) list array}
    let init s = {n = s; e = Array.make s []}
    let size g = g.n
    let insert_directed_edge g x y l =
      g.e.(x) <- (y,l)::g.e.(x)
    let insert_edge g x y l =
      insert_directed_edge g x y l;
      insert_directed_edge g y x l
    let neighbours g x =
      g.e.(x)
  end;;
```

Przeszukiwanie grafów

Definition

Problem przeszukiwania grafu:

- Odwiedzamy wierzchołki grafu w określonej kolejności.
- Po kolei przechodzimy spójne składowe grafu.
- Kolejności w obrębie składowej przyjrzymy się dalej.
- W każdym wierzchołku wywołujemy procedurę `visit`, która jest parametrem procedury przeszukującej.
- Każdą składową oznaczamy innym „kolorem" — liczbą całkowita od 0 wzwyż.
- Kolor ów jest parametrem procedury `visit`.
- Wynikiem procedury przeszukującej graf jest liczba spójnych składowych.

Przeszukiwanie grafów

Example

Przeszukiwanie grafu możemy zrealizować jako funktor:

```
module GraphSearch (Q : QUEUE) (G : GRAPH) =
  struct
    open G
    let search visit g =
      let visited = Array.make (size g) false
      in let walk x color = ...
         in let k = ref 0
            in for i = 0 to size g - 1 do
                 if not visited.(i) then begin
                   walk i (!k);
                   k := !k+1
                 end;
               done; !k
  end;;
```

Przeszukiwanie grafów

Example

```
let walk x color =
  let q = ref Q.empty
  in begin
    q := Q.insert !q x;
    visited.(x) <- true;
    while not (Q.is_empty !q) do
      let v = Q.front !q
      in begin
        visit v color;
        q := Q.remove !q;
        List.iter
          (fun (y, _) ->
            if not (visited.(y)) then begin
              q := Q.insert !q y;
              visited.(y)<- true
            end)
          (neighbours g v);
      end
    done
  end
```

Przeszukiwanie grafów

Example

- n = liczba wierzchołków w grafie.
  m = liczba krawędzi w grafie.
- Każdy wierzchołek jest rozpatrywany tyle razy ile wychodzi z niego krawędzi plus co najwyżej jedno wywołanie `search`.
- Rozmiar kolejki nie przekroczy n.
- Złożoność pamięciowa $= \Theta(n + m)$.
- Złożoność czasowa zależy od czasu, w jakim działają operacje na kolejce.
  Zakładając, że działają w czasie stałym (zamortyzowanym) mamy złożoność czasową $\Theta(n + m)$.

BFS

Example

Przeszukiwanie grafu wszerz:

- Idea algorytmu przypomina pożar prerii.
- Jeśli ogień dojdzie do jakiegoś miejsca, to rozprzestrzenia się na sąsiednie miejsca, na których jest jeszcze niewypalona trawa.
- Przyłożenie ognia w jednym punkcie, spala cały „spójny obszar” suchej trawy.
- *Źródło* — pierwszy odwiedzony wierzchołek w każdej spójnej składowej.
- Kolejność odwiedzania: najpierw źródło, potem jego sąsiedzi, potem ich sąsiedzi itd.

BFS

Example

- Wierzchołki są odwiedzane w kolejności rosnącej odległości od źródła.
- Wierzchołki trzymamy w kolejce FIFO.

```
module BFS = GraphSearch (Fifo);;
```

BFS

Example

- Operacje na kolejce FIFO są wykonywane w stałym czasie zamortyzowanym.
- Tak więc złożoność czasowa przeszukiwania $= \Theta(n + m)$.
- W każdej chwili, w kolejce q znajdują się:
  - albo wierzchołki położone w tej samej odległości l od źródła,
  - albo najpierw wierzchołki położone w odległości l a dalej w odległości $l + 1$ od źródła.
- W każdej spójnej składowej wierzchołki są odwiedzane zgodnie z rosnącymi odległościami od źródła.

DFS

Example

Przeszukiwanie grafu w głąb:

- Przeszukiwanie z nawrotami.
- Procedurę przeszukującą wywołujemy dla źródła.
  Następnie jest ona rekurencyjnie wywoływana dla jego sąsiadów, ich sąsiadów itd.
  Przy pierwszym wywołaniu wierzchołek jest odwiedzany.
- Implementacja tej procedury nie musi być rekurencyjna.
  Może być iteracyjna, jeżeli zastosujemy stos, czyli kolejkę LIFO (ang. *last in, first out*).

DFS

Example

```
module Lifo : QUEUE =
  struct
    exception EmptyQueue
    type 'a queue = 'a list
    let empty = []
    let is_empty q = q = []
    let insert q x = x::q
    let front q =
      if q = [] then raise EmptyQueue
      else hd q
    let remove q =
      if q = [] then raise EmptyQueue
      else tl q
  end;;

module DFS = GraphSearch (Lifo);;
```

DFS

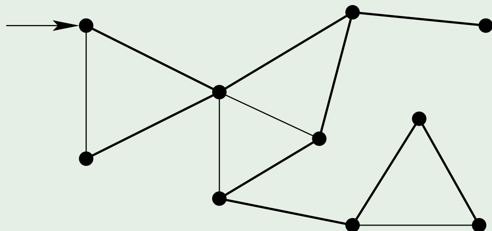
Example

- Algorytm DFS ma wiele ciekawych własności.
  Pozwala on na znalezienie w grafie cykli prostych, a także na podział grafu na dwuspójne składowe.
- Wyobraźmy sobie drzewa zbudowane z tych krawędzi, które łączą odwiedzane wierzchołki i wierzchołki, które przy tej okazji są wstawiane do kolejki.
- Są to drzewa rozpinające spójne składowe grafu.
- Korzeniem drzewa jest źródło.
- Zastanówmy się jak mogą być położone względem takiego drzewa pozostałe krawędzie grafu.

DFS

Example

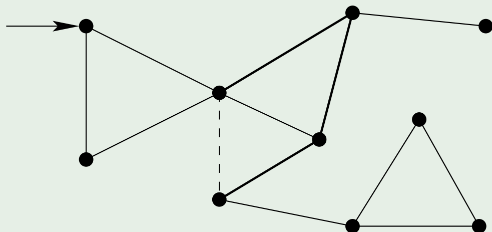
- Krawędzie spoza drzewa muszą prowadzić w górę drzewa.
- Krawędź spoza drzewa nie może łączyć dwóch gałęzi drzewa. Gdyby tak było, to w trakcie obchodzenia pierwszej gałęzi powinniśmy wstawić do kolejki wierzchołek leżący na drugim końcu krawędzi.



DFS

Example

- Z każdą krawędzią spoza drzewa związany jest cykl, złożony z niej samej oraz ścieżki w drzewie łączącej jej końce.
- Okazuje się, że dowolny cykl prosty w grafie możemy przedstawić jednoznacznie, jako zbiór krawędzi spoza drzewa rozpinającego DFS.



Sortowanie topologiczne

Definition

Problem sortowania topologicznego:

- Mamy do wykonania n różnych czynności.
- Niektóre z nich muszą być wykonane wcześniej, a niektóre później.
- Mamy danych m par zależności postaci: czynność a musi być wykonana wcześniej niż czynność b.
- Należy ustalić taką kolejność wykonywania czynności, żeby wszystkie zależności były spełnione, lub stwierdzić, że nie jest to możliwe.
- Dane możemy przedstawić jako graf skierowany. Czynności to wierzchołki, a zależności to krawędzie.

Sortowanie topologiczne

Example

- Do rozwiązania problemu możemy użyć zmodyfikowanej wersji algorytmu DFS.
- Możemy przechodzić wzdłuż krawędzi tylko zgodnie z ich kierunkiem.
- Każdy wierzchołek odwiedzamy dwa razy:
  - najpierw do niego „*wchodzimy*”,
  - następnie odwiedzamy wszystkich jego sąsiadów,
  - na koniec z niego „*wychodzimy*”.
- Obejście *kartezjańskie* drzewa.
  Wzdłuż każdej krawędzi przechodzimy dwa razy.

Sortowanie topologiczne

Example

- Jak mogą być położone krawędzie spoza drzewa rozpinającego DFS?
- Krawędzie drzewa rozpinającego DFS są zorientowane od korzenia w dół.
- Jeśli istnieje krawędź prowadzącą w górę drzewa, to istnieje cykl w grafie i rozwiązanie nie istnieje.
- Krawędź taka prowadzi do wierzchołka, do którego weszliśmy, ale z którego jeszcze nie wyszliśmy.
- Jeśli napotkamy krawędź prowadzącą do wierzchołka, z którego już wyszliśmy, to jest to wierzchołek leżący na innej gałęzi drzewa rozpinającego.

Sortowanie topologiczne

Example

- Reprezentowaną przez niego czynność należy wykonać wcześniej.
  Dotyczy to również wszystkich innych wierzchołków, do których można z niego dojść.
  Zauważmy, że one również zostały już odwiedzone i z nich wyszliśmy.
- Jeśli przechodzimy krawędź i odwiedzamy nowy wierzchołek, to wejdziemy do niego później, ale wyjdziemy z niego wcześniej, niż z wierzchołka, w którym jesteśmy.

Sortowanie topologiczne

Example

- Z wierzchołków reprezentujących czynności, które należy wykonać później wyjdziemy wcześniej niż z danego wierzchołka.
- Wystarczy wykonywać czynności w odwrotnej kolejności wychodzenia z wierzchołków.

Algorytm Dijkstry

Example

- Graf nieskierowany, w którym krawędzie mają długości.
- Długości krawędzi są nieujemnymi liczbami rzeczywistymi.
- Algorytm Dijkstry służy do wyznaczania odległości od źródła do pozostałych wierzchołków,
- Odwiedzane wierzchołki trzymamy w kolejce priorytetowej. Priorytety odpowiadają odległości od źródła.

Algorytm Dijkstry

Example

- Wierzchołki zaznaczamy jako odwiedzone dopiero po wyjęciu z kolejki.
  Wyjmując upewniamy się, czy wcześniej już danego wierzchołka nie odwiedziliśmy. Wierzchołki mogą być wstawiane do kolejki priorytetowej wielokrotnie, jednak nie więcej niż łącznie m razy.
- Może się zdarzyć, że wstawiając wierzchołek do kolejki po raz kolejny, nadamy mu mniejszą wagę i zostanie on wcześniej wyjęty z kolejki.
- Patrząc na pierwsze wyjęcia z kolejki wierzchołków, wyjmujemy je w kolejności rosnącej odległości od źródła.

Algorytm Dijkstry

Example

```
module FloatLabel =
  struct
    type label = float
  end;;

module FloatGraph = Graph(FloatLabel);;
```

Algorytm Dijkstry

Example

```
module Dijkstra :
  sig
    val search : FloatGraph.graph -> int -> float array
  end = struct
    open FloatGraph

    module Order =
      struct
        type t = int * float
        let porownaj (v1, d1) (v2, d2) =
          if (d1, v1) < (d2, v2) then Wieksze
          else if (d1, v1) = (d2, v2) then Rowne
          else Mniejsze
      end

    module Q = PriQueue (Order)
```

Algorytm Dijkstry

Example

```
    let search g v =
      let visited = Array.make (size g) false
      and dist = Array.make (size g) infinity
      and q = ref Q.empty
      in begin
           q := Q.put (v, 0.0) !q;
           while not (Q.is_empty !q) do
             let (v, d) = Q.getmax !q
             in if not visited.(v) then begin
                  List.iter
                    (fun (w, l) -> q := Q.put (w, d +. l) !q)
                    (neighbours g v);
                  dist.(v) <- d;
                  visited.(v)<- true
                end;
             q := Q.removemax !q
           done;
           dist
         end
  end;;
```

Algorytm Dijkstry

Example

- Koszt czasowy operacji na kolejce priorytetowej jest rzędu $\Theta(\log n)$.
- Koszt czasowy algorytmu jest rzędu $\Theta((n+m)\log n)$.