Mulțimi

parcurgeri de grafuri, diagrame de decizie, unificare, funcții ca date

Marius Minea

17 martie 2010

(Re)vizităm: parcurgerea unui graf

$Visited = \emptyset$; $Frontier = \{s_{init}\}$
while $Frontier \neq \emptyset$ **do**
  $s = extract(Frontier)$
  $add(Visited, s)$
  for all $s' \in succ(s)$ **do**
    if $s' \notin Visited \cup Frontier$ **then**
      $add(Frontier, s')$

Implementarea lui *add*/*extract* determină parcurgerea:
  coadă (FIFO): prin cuprindere (breadth-first)
  stivă (LIFO): în adâncime (depth-first)
dar parcurgerea e *completă* indiferent de ordine

Q: De ce nu am marcat explicit nodurile $s.mark = 0$, $s.mark = 1$?

Grafuri și programe

Parcurgerea folosind *mulțimi* de stări

Algoritmul anterior *iterează artificial* prin mulțimea succesorilor
$\Rightarrow$ soluție mai naturală: *uniunea* mulțimilor

$Visited = \emptyset$; $Frontier = \{s_{init}\}$
while $Frontier \neq \emptyset$ **do**
  $s = extract(Frontier)$
  $add(Visited, s)$
  $Frontier = Frontier \cup (succ(s) \setminus Visited)$

Radical: folosim *doar* mulțimi

add/*extract* și *succ* încă folosesc stări individuale (s)

$\Rightarrow$ modificăm și *folosim integral mulțimi*

Succ dă mulțimea de succesori pentru o mulțime de noduri

$$Succ(S) = \{s' \mid s \in S \wedge s' \in succ(s)\}$$

$Visited = \emptyset$; $Frontier = \{s_{init}\}$

while $Frontier \neq \emptyset$ **do**

$Visited = Visited \cup Frontier$

$Frontier = Succ(Frontier) \setminus Visited$

Radical: folosim *doar* mulțimi

add/*extract* și *succ* încă folosesc stări individuale (s)

$\Rightarrow$ modificăm și *folosim integral mulțimi*

Succ dă mulțimea de succesori pentru o mulțime de noduri

$$Succ(S) = \{s' \mid s \in S \wedge s' \in succ(s)\}$$

$Visited = \emptyset$; $Frontier = \{s_{init}\}$

while $Frontier \neq \emptyset$ **do**

$Visited = Visited \cup Frontier$

$Frontier = Succ(Frontier) \setminus Visited$

invarianți (la iterația k)

$Frontier$ = mulțimea nodurilor accesibile în k dar nu $k-1$ pași

$Visited$ = mulțimea nodurilor accesibile în $\leq k$ pasi

Alternativă, folosind doar mulțimi

$Oldvis = \emptyset$; $Visited = \{s_{init}\}$
while $Visited \neq Oldvis$ **do**
  $Oldvis = Visited$
  $Visited = Oldvis \cup Succ(Oldvis)$

invarianți (la iterația k)
$Oldvis$ = mulțimea nodurilor accesibile în $< k$ pași
$Visited$ = mulțimea nodurilor accesibile în $\leq k$ pasi

Se poate implementa practic ?

Cum reprezentăm mulțimile ?

Se poate implementa practic ?

Cum reprezentăm mulțimile ?

- tablouri

Se poate implementa practic ?

Cum reprezentăm mulțimile ?

- tablouri
- liste

Se poate implementa practic ?

Cum reprezentăm mulțimile ?

- tablouri
- liste
- arbori

Se poate implementa practic ?

Cum reprezentăm mulțimile ?

- tablouri
- liste
- arbori
- tabele de dispersie

Se poate implementa practic ?

Cum reprezentăm mulțimile ?

- tablouri
- liste
- arbori
- tabele de dispersie

toate enumeră explicit elementele

$\Rightarrow$ în toate, operațiile necesită *parcurgerea individuală* a elementelor

Se poate reprezenta o mulțime fără a *enumera explicit* elementele ?

Se poate implementa practic ?

Cum reprezentăm mulțimile ?

- tablouri
- liste
- arbori
- tabele de dispersie

toate enumeră explicit elementele

$\Rightarrow$ în toate, operațiile necesită *parcurgerea individuală* a elementelor

Se poate reprezenta o mulțime fără a *enumera explicit* elementele ?

Da! În matematică: $\{x \in \mathbb{R} \mid x \geq 5 \wedge x < 7\}$

$\Rightarrow$ *mulțime = formulă*

Mulțimi și funcții

O mulțime e reprezentată prin *funcția caracteristică*

$$f(x) = \text{ if } x \in S \text{ then } 1 \text{ else } 0$$

Dacă mulțimea S e finită
putem codifica fiecare valoare $v \in S$ pe $\lceil \log |S| \rceil$ biți.

$\Rightarrow$ problema: reprezentarea unei *funcții boolene* de n variabile

$$f : \mathbb{B}^n \to \mathbb{B}$$

Date și cod

Prin BDD, am reprezentat o *funcție* într-o structură de date.

Evaluăm funcția *parcurgând* datele într-un anume fel.
Invers, putem reprezenta date prin *cod* care le generează.

Șirul numerelor naturale

```
public class Nats {
  int val = 0;

  public int next() { return val++; }

  public static void main(String[] args) {
    Nats a = new Nats();
    System.out.println(a.next());
    System.out.println(a.next());
  }
}
```

În Java, trebuie (evident) să creem o clasă.
și, fiind limbaj *imperativ*, soluția se bazează pe *efect lateral*:
inițial, a reprezintă $\{n \mid n \geq 0\}$
după `a.next(); a.next();` , *același* obiect a reprezintă *altceva*,
mulțimea $\{n \mid n \geq 2\}$ (membrul `val` s-a modificat).

Într-un limbaj funcțional ...

... funcțiile sunt *first-class citizens* (entități de prim rang)
$\Rightarrow$ pot fi atribuite, transmise ca argumente și returnate

Reprezentăm *direct* un *șir infinit* prin:
primul element
restul șirului ?

Într-un limbaj funcțional ...

... funcțiile sunt *first-class citizens* (entități de prim rang)
$\Rightarrow$ pot fi atribuite, transmise ca argumente și returnate

Reprezentăm *direct* un *șir infinit* prin:
primul element
~~restul șirului~~ *o funcție* care generează restul șirului