Tema LFA 2021-2022
Sincronizarea automatelor

Ianuarie 2022

Rezumat

Tema constă în implementarea, într-un limbaj de programare la alegere, a unui algoritm de sincronizare pentru un automat cu informație limitată despre starea curentă.

1 Specificații temă

1.1 Cerință

Să se implementeze un program care primește la intrare reprezentarea unui automat și găsește secvențe de sincronizare în cazul în care există. In continuare vom descrie fiecare task pe rand.

1.2 Stări accesibile (accessible)

Notăm cu $A = \mathcal{A}(K', \delta)$ mulțimea stărilor accesibile pe tranzițiile din δ din stările din K'.

$\forall q \in K, (q \in A \Leftrightarrow (\exists w \in \Sigma^*, \exists p \in K', (p, w) \vdash^* (q, e)))$

O stare q este accesibilă dacă și numai dacă există un drum dintr-o stare din K' până în q.

La primirea *problem =accessible*, programul va afișa toate stările accesibile din stările date la intrare, K', câte una pe linie. Parametrii sunt $s \neq 0, f = 0$.

1.3 Stări productive (productive)

Notăm cu $P = \mathcal{P}(K'', \delta)$ mulțimea stărilor productive relativ la K''.

$\forall p \in K, (p \in P \Leftrightarrow (\exists w \in \Sigma^*, \exists q \in K'', (p, w) \vdash^* (q, e)))$

O stare p este productivă dacă și numai dacă există un drum din p într-o stare din K''.

La primirea *problem =productive*, programul va afișa toate stările productive relativ la stările date la intrare, K'', câte una pe linie. Parametrii sunt $s = 0, f \neq 0$.

1.4 Stări utile (utils)

Notăm cu $U = \mathcal{U}(K', K'', \delta)$ mulțimea stărilor utile pe rutele de la stări din K' la stări din K''.

$\forall u \in K, (u \in U \Leftrightarrow$
$(\exists w_1 \in \Sigma^*, \exists w_2 \in \Sigma^*, \exists p \in K', \exists q \in K'', (p, w_1 w_2) \vdash^* (u, w_2) \vdash^* (q, e)))$

O stare u este utilă dacă și numai dacă există un drum dintr-o stare din K' până în u și din u într-o stare din K''. O stare utilă este accesibilă și productivă.

La primirea *problem* =*useful*, programul va afișa toate stările utile luând în considerare restricțiile date la intrare. Parametrii sunt $s \neq 0, f \neq 0$. La intrare se specifică atât stările din care se poate porni (K') cât și stările în care trebuie să se ajungă (K'').

1.5 Sincronizare (synchronize)

Considerând că am pierdut configurația unui automat și nu știm starea în care se află, putem da un șir la intrare astfel încât să știm în ce stare s-ar afla după consumarea lui?

Fie K o mulțime de stări, Σ un alfabet și $\delta : K \times \Sigma \to K$ o funcție de tranziție.

Spunem că δ admite sincronizare dacă există un cuvânt care dat la intrare duce automatul în aceiasi stare, indiferent din ce stare a pornit.

δ admite sincronizare $\Leftrightarrow$
$\exists w \in \Sigma^*, \exists q \in K, \forall p \in K, (p, w) \vdash^* (q, e)$

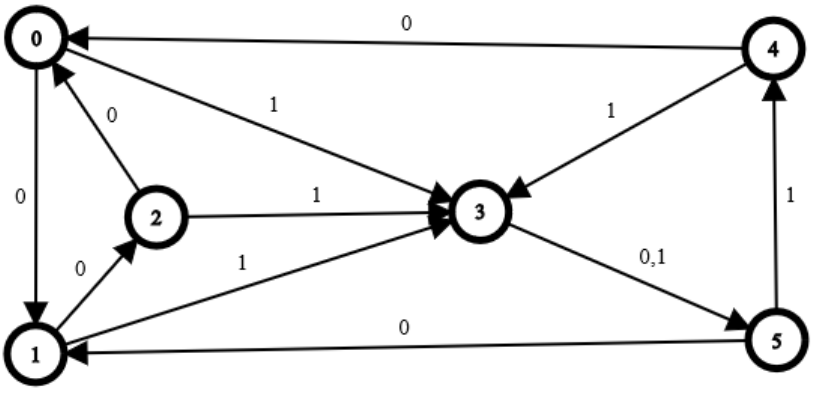
Pentru partea de sincronizare, se garanteaza ca automatul de intrare va avea urmatoarele caracteristici:

1. este orientabil
2. toate stările sunt productive
3. există un cuvânt de sincronizare

La primirea *problem* =*synchronize*, programul va afișa un cuvânt care sincronizează δ. Parametrii sunt $s = 0, f = 0$.

Exemplu de sincronizare

Fie un automat cu următoarele tranziții



δ admite sincronizare pentru că secvența 001 duce automatul din orice stare în starea 3.

1.5.1 Sincronizare parțială

Există funcții de tranziție pentru care nu există cuvânt de sincronizare. Totuși, există situații în care dacă dispunem de informație parțială despre starea curentă, putem găsi un cuvânt de sincronizare.

Fie $K' \subseteq K$, o submulțime a lui K.

δ admite sincronizare din $K' \Leftrightarrow$

$\exists w \in \Sigma^*, \exists q \in K, \forall p \in K', (p, w) \vdash^* (q, e)$

Programul va afișa un cuvânt care sincronizează δ pornind din stările specificate la intrare, K'. Parametrii sunt $s \neq 0, f = 0$.

1.5.2 Sincronizare restricționată

Există situații pentru care nu este suficient să ajungem într-o stare oarecare, ci ne interesează ca destinația să fie intr-o submulțime specificată a lui K (in cazurile practice această submulțime va fi formată din mulțimea stărilor finale).

Fie $K'' \subseteq K$, o submulțime a lui K.

δ admite sincronizare în $K'' \Leftrightarrow$

$\exists w \in \Sigma^*, \exists q \in K'', \forall p \in K, (p, w) \vdash^* (q, e)$

Programul va afișa un cuvânt care sincronizează δ ajungând într-una din stările specificate la intrare, K''. Parametrii sunt $s = 0, f \neq 0$.

1.5.3 Sincronizare parțială restricționată

Sincronizarea poate fi parțială și restrictionată simultan.

Fie $K' \subseteq K$, $K'' \subseteq K$, submulțimi ale lui K.

δ admite sincronizare din K' în $K'' \Leftrightarrow$

$\exists w \in \Sigma^*, \exists q \in K'', \forall p \in K', (p, w) \vdash^* (q, e) \Leftrightarrow$

Programul va afișa un cuvânt care sincronizează δ cu restricțiile specificate la intrare, K' și K''. Parametrii sunt $s \neq 0, f \neq 0$.

1.6 Labirint (Bonus)

Se dă un labirint codificat *lab* ca în sectiunea 1.8.4.

1. În interiorul labirintului sunt r roboți.
2. Obiectivul roboților este să ajungă la una din e ieșiri din labirint.
3. Roboții nu pot comunica între ei, dar vor executa instrucțiuni.
4. O instrucțiune este executată de toți roboții simultan, în cazul în care pot face asta.
5. Instrucțiunile sunt direcții în care să se miște roboții, reprezentate ca numere în felul următor: est = 0, nord = 1, vest = 2, sud = 3.
6. Dacă un robot primește o instrucțiune de a se deplasa în direcția unui perete, rămâne în celula curentă.

Codificați labirintul ca un automat și configurația roboților și ieșirilor ca parametri pentru o problemă de sincronizare parțială restricționată.

Programul apelat de regula *make labyrinth* va primi la intrare codificarea *lab* și va scrie la ieșire codificarea automatului și parametrii de sincronizare astfel încât secvența de sincronizare calculată pe ieșirea programului să ducă toți roboții la aceeași ieșire. Se garantează că există o soluție.

1.7 Conținutul arhivei

Arhiva trebuie să conțină:

- surse, a căror organizare nu vă e impusă
- un fișier Makefile care să aibă target de build și run
- un fișier README care să conțină linii de maxim 80 de caractere în care să descrieți sumar reprezentarea și algoritmii aplicați. Cu cât mai scurt, cu atât mai bine!

Arhiva trebuie să fie zip. Nu rar, 7z, ace sau alt format ezoteric. Fișierul Makefile și fișierul README trebuie să fie în rădăcina arhivei, nu în vreun director.

Fișierul trebuie să se numească README, nu readme, ReadMe, README.txt, readme.txt, read-me.doc, rEADME, README.md sau alte variante asemănătore sau nu.

Nerespectarea oricărui aspect menționat mai sus va duce la nepunctarea temei.

1.8 Specificații program

1.8.1 Rularea

Programul primește un singur argument în linia de comandă, numele problemei rezolvate.
Numele problemei poate fi:

- accessible (vezi 1.2)
- productive (vezi 1.3)
- useful (vezi 1.4)
- synchronize (vezi 1.5)

Pentru a permite apelarea executabilului cu argumente, regula de run din Makefile trebuie să aibă următoarea formă:

```
run: build
    ./<exec> $(problem)
```

Exemplu de rulare:

```
make run problem=productive
```

1.8.2 Bonus

Pentru bonus, va exista o regulă de make separată, *labyrinth*, care va transforma intrarea de bonus în intrare compatibilă cu sincronizarea.

```
labyrinth: build
    ./<lab-exec> $(lab)
```

Checker-ul va rula următoarele comenzi, necesitând program funcțional de sincronizare (vezi 1.5).

Exemplu de rulare:

```
make labyrinth
make run problem=synchronize
```

1.8.3 Intrări

La intrare vor fi $4 + n \cdot m + s + f$ numere întregi:

1. Pe prima linie patru numere n, m, s și f, unde n e numărul de stări, m e numărul de simboluri, s e numărul de stări de pornire și f e numărul de stări finale.
2. Următoarele n linii conțin câte m numere, unde al q-lea număr de pe linia i reprezintă starea destinație pentru tranziția din starea i pe simbolul q
3. Următoarea linie conține s stări
4. Următoarea linie conține f stări

Restricții:

- $1 < n \leq 2^{13}$
- $0 < m \leq 2^{13}$
- Stările sunt numerotate de la 0 la $n - 1$, $0 \leq i < n$
- Simbolurile sunt numerotate de la 0 la $m - 1$, $0 \leq q < m$
- $0 \leq s \leq n$, în cazul în care există restricții pe sursă
- $0 \leq f \leq n$, în cazul în care există restricții pe destinație

1.8.4 Intrări (bonus)

Un labirint este o matrice de dimensiune $l \cdot c$, care conține celule care pot fi separate de pereți.

Specificații pentru intrarea care descrie structura labirintului

1. Prima linie conține patru numere, l, c, r și o unde l e numărul de linii, c e numărul de coloane, r e numărul de roboți și o e numărul de ieșiri
2. Următoarele l linii conțin câte c numere: unde al q-lea număr de pe linia i reprezintă codificarea pereților celulei de la coordonatele (i, q)

3. Următoarea linie conține r perechi de numere l_i c_i separate prin spații: robotul i se află la coordonatele (l_i, c_i)
4. Următoarea linie conține o perechi de numere l_q c_q separate prin spații: ieșirea q se află la coordonatele (l_q, c_q)

Restricții:

- $1 < l, c \leq 2^8$
- $1 < r \leq l \cdot c$
- $1 < o \leq l \cdot c$
- $0 \leq l_i, l_q \leq l$
- $0 \leq c_i, c_q \leq c$

O celulă are patru pereți: est, nord, vest, sud. Fiecare perete poate fi reprezentat ca un bit. Informația despre configurația pereților unei celule poate fi reprezentată printr-un număr pe patru biți.

Exemple:

- $12 = 1010_{(2)}$: est(0), nord(1), vest(0), sud(1), o cameră cu pereți la nord și sud
- $0 = 0000_{(2)}$: est(0), nord(0), vest(0), sud(0), o cameră fără pereți

1.8.5 Ieșiri

Programul va afișa la ieșirea standard răspunsul la problema *problem* conform cerințelor din secțiunea 1.1, câte un element pe linie.

1.8.6 Suport

Pentru această temă nu există schelet de cod.

Vi se pune la dispoziție un model de Makefile.

1.8.7 Versiuni

Mașina de test are instalat Ubuntu 20.04 LTS si următoarele versiuni:

- make: 4.2.1
- gcc/g++: 9.3.0
- clang: 10.0.0
- bison: 3.5.1
- libboost: 1.71
- javac: 14.0.2
- python: 2.7.18, 3.8.5
- python numpy: 1.16.5(2.7), 1.19.5(3.8)
- python ply: 3.11

1.8.8 Limbaje acceptate

Limbajele acceptate inițial sunt C, C++, Java și Python, urmând ca ulterior lista sa poată fi extinsă, dacă este cazul.

1.8.9 Observație

Dacă aveți nevoie de modificări, puteți posta pe forum (moodle). Dacă cererea va fi aprobată, vom actualiza mașina pe care sunt testate temele și vom anunța acest lucru pe forum în momentul în care se fac modificarile.

1.8.10 Încărcare temă

Tema trebuie încărcată pe vmchecker [1].

2 Elemente de teorie

2.1 Automate orientabile

Un automat este orientabil (sau monoton) dacă există o ordine ciclica a stărilor care este pastrată de toate tranzițiile automatului (pentru mai multe detalii consultați [5](Pagina 22) și [6], secțiunea Definitions and Lemmas).

2.2 Sincronizare

Pentru mai multe detalii legate de sincronizare incluzand descrierea unui algoritm puteti consulta [4] (Algorithm 2, Theorem 1.14, 1.15) si [6], secțiunea Definitions and Lemmas si Theorem 1.

3 Checker

Vi se pune la dispoziție un checker cu teste publice, care dă punctaje de până la 256(200 + 56 bonus).

200 de puncte (fără bonus) pot fi obținute prin implementarea funcțiilor de sincronizare.

Punctajele sunt distribuite astfel:

- stări accesibile - 20p
- stări productive - 20p
- stări utile - 20p
- sincronizare generală - 80p
- sincronizare parțială - 20p
- sincronizare restricționată - 20p
- sincronizare parțială restricționată - 20p
- bonus (labirint) - 56p

Deadline: 23 Ianuarie 2021 , 23:59. Upload-ul va rămâne deschis pâna la ora 05:00 a doua zi.

Bibliografie

[1] vmchecker LFA

[2] elf tema

[3] Laborator 1 SO: Makefile

[4] Sven Sandberg, Homing and Synchronizing Sequences

[5] Mikhail V. Volkov, Synchronizing Automata and the Cerny Conjecture

[6] David Eppstein, Reset Sequences for Monotonic Automata