Problema 2 – Sortnet (Mihai Pătraşcu)

Se observă că cele N intrări de 0 şi 1 pot fi interpretate ca un număr în baza 2, între 0 şi 2^N-1. Se poate aplica convenţia $1 \rightarrow$LSB (least-significant-bit), $N \rightarrow$MSB.

Soluţia 1

Generăm toate cele 2^N posibile intrări, şi simulăm reţeaua de sortare pentru fiecare în parte. Timpul de rulare este $O(2^N \cdot N \cdot M)$. Această soluţie va obţine aproximativ 20 de puncte.

Soluţia 2

Încercăm să reducem timpul de rulare, refolosind informaţiile de la o simulare anterioară pentru a calcula ieşirile corespunzătoare unei noi configuraţii. Dacă păstrăm starea completă a reţelei şi în intrare se modifică o singură poziţie, putem calcula noua stare a reţelei (şi implicit ieşirea) în $O(M)$. Dacă în intrare se modifică un singur bit, trebuie urmărită propagarea modificării în toată reţeaua; evident, în ieşire fiecărui ciclu se va modifica un singur bit, a cărui modificare trebuie urmărită în continuare. Pseudocodul pentru această soluţie este:

```
cnt-ok=1          - dacă toate valorile de intrare sunt 0, ieşirea este evident sortată
for i=1,N
     for all bits that differ in i and (i-1)
          trace change of one input value
     if(output-is-sorted) cnt-ok++
```

Calcularea numărului de apeluri ale funcţiei `trace` este o aplicaţie clasică la analiza amortizată. Numărul amortizat de comutări la incrementarea unui contor binar este 2. Ajungem la un număr total de apeluri de $O(2^{N+1})=O(2^N)$, deci un timp de rulare de $O(2^N \cdot M)$. Această soluţie va obţine aproximativ 60-70 de puncte.

Soluţia 3

În loc să considerăm numerele între 0 şi 2^N în ordine crescătoare, putem considera o permutare a lor cu proprietatea că două numere vecine diferă printr-un singur bit. Astfel, va fi nevoie de un singur apel al funcţiei `trace` pentru fiecare configuraţie considerată, şi timpul total de rulare se va reduce aproximativ la jumătate (deşi asimptotic nu se modifică).

O permutare cu proprietatea de mai sus se numeşte cod Gray. Codul Gray se poate obţine în timp $O(2^N)$ astfel:

```
consider an empty stack
for i=N downto 1
     push i
first configuration is 0,0,0,... (which always gets sorted)
while(stack-not-empty)
     pop k
     change bit k and trace change
     for i=k-1 downto 1
          push i
```

Testarea ieşirii (sau de ce programarea este o artă)

Mai sus, nu am discutat modul în care se testează dacă ieşirea reţelei este sortată pentru fiecare configuraţie. Este trivial să scriem un algoritm în timp `O(N)` pentru acest test (şi acest algoritm este suficient de rapid). Totuşi, pentru că ieşirea se poate reprezenta ca întreg (ca şi intrarea), putem realiza şi un algoritm în timp constant.

Următoarele linii au fost copiate din soluţia oficială scrisă în ANSI C:

```
#define CLR_LS1B(x)      ((x) & ((x) - 1))
#define GET_LS1B(x)      ((x) ^ CLR_LS1B(x))
#define SORTED(x)        !((x + GET_LS1B(x)) & max_in)
...
long max_in = (1l << nwires) - 1;
```

Dacă vă întrebaţi, `LS1B` este o prescurtare (dubioasă) pentru *least-significant one bit*.

- `CLR_LS1B(x)` întoarce întregul `x` cu ultimul bit de unu transformat în zero. Explicaţia este simplă:

```
x          = ?..?10..0
x-1        = ?..?01..1
x&(x-1)    = ?..?00..0
```

- `GET_LS1B(x)` întoarce întregul `x` după ce anulează toţi biţii de unu, mai puţin ultimul. Explicaţia este simplă: `XOR`-ul anulează toţi biţii care coincid între cei doi operanzi.

- `SORTED(x)` testează dacă argumentul (văzut ca întreg pe `N` biţi) are biţii sortaţi (toţi biţii de unu se găsesc pe poziţii mai mari decât biţii de zero). Este apelat pentru a vedea dacă ieşirea reţelei este sortată corect.