Resposta da Tarefa 2 — Definindo Computação Açúcar Sintático Código como Dados, Dados como Código

Análise e Complexidade de Algoritmos
PPComp — Campus Serra, Ifes

Pablo Simões Nascimento

16 de dezembro de 2021

1 Problema 1

Parte I - (05pts)

Escreva um programa em AON-CIRC (variante de NAND-CIRC que usa as funções AND/OR/NOT) que implemente a função GT6, que recebe uma entrada de 6 bits, $x = x_0x_1x_2x_3x_4x_5$, que retorne 1 se, e somente se, o número binário representado por $x_0x_1x_2$ for maior do que o número binário representado por $x_3x_4x_5$. Considere que os números estão em binário, com o bit menos significativo à direita.

Parte II - (10pts)

Prove que existe uma constante $c \in \mathbb{N}$ tal que, para todo $c \in \mathbb{N}$ existe um circuito Booleano C, com portas lógicas AND, OR e NOT, com no máximo $c.n$ portas lógicas que computa a função $GT_{2n} : \{0,1\}^{2n} \to 0,1$ tal que $GT_{2n}(x0 \dots x_{n-1}x_n \dots x_{2n-1}) = 1$ se e somente se o número representado por $x0 \dots xn - 1$ for maior do que o número representado por $x_n \dots x_{2n-1}$. Considere que os números estão em binário, com o bit menos significativo à direita.

Parte III - (05pts)

Prove que existe uma constante $c \in \mathbb{N}$ tal que, para todo $c \in \mathbb{N}$ existe um circuito Booleano C, com portas lógicas NAND, com no máximo $c.n$ portas lógicas que computa a função $GT_{2n} : \{0,1\}^{2n} \to 0,1$ tal que $GT_{2n}(x0 \dots x_{n-1}x_n \dots x_{2n-1}) = 1$ se e somente se o número representado por $x0 \dots xn - 1$ for maior do que o número representado por $x_n \dots x_{2n-1}$. Considere que os números estão em binário, com o bit menos significativo à direita.

1.1 Solução - parte I

Para conseguirmos definir a implementação na linguagem AON-CIRC que dará solução a este problema, iremos buscar definir primeiro uma expressão apenas em termos de *and*, *or* e *not*.

Sejam os números $A = a_2a_1a_0$, sendo $a_2a_1a_0 = x_0x_1x_2$ e $B = b_2b_1b_0$, sendo $b_2b_1b_0 = x_3x_4x_5$, respectivamente. Enumeramos de 2 a 0 dos bits mais significantes para os menos significantes a fim de facilitar intuitivamente o raciocínio.

A função GT_6 é verdadeira quando os bits $a_2a_1a_0$ indicam um número binário maior que o número formado por $b_2b_1b_0$, ou seja, quando $A > B$. Portanto, vamos analisar quais os casos em que $A > B$.

$A > B$ nos seguintes casos:

- se $a_2 > b_2$ ou
- se $a_2 = b_2$ e $a_1 > b_1$ ou
- se $a_2 = b_2$ e $a_1 = b_1$ e $a_0 > b_0$

Conforme os casos acima, há duas relações que precisaremos expressar na forma de conjunção e disjunção: a relação *maior que* e a relação de *igualdade* entre dois dígitos binários. Assim, considere a seguinte tabela verdade:

a	b	a > b	a = b
0	0	0	1
0	1	0	0
1	0	1	0
1	1	0	1

Tabela 1: Tabela verdade para identificação de casos para $a > b$ e $a = b$

Podemos identificar pela Tabela 1 que $a > b \equiv a \wedge \neg b$ e que $a = b \equiv (\neg a \wedge \neg b) \vee (a \wedge b)$. Agora, podemos redefinir os casos em que $A > B$ apenas utilizando os operadores $\wedge$, $\vee$ e $\neg$, conforme abaixo:

$A > B$ nos seguintes casos:

- se $a_2 \wedge \neg b_2$ OU
- se $[((\neg a_2 \wedge \neg b_2) \vee (a_2 \wedge b_2)) \wedge (a_1 \wedge \neg b_1)]$ OU
- se $[((\neg a_2 \wedge \neg b_2) \vee (a_2 \wedge b_2)) \wedge ((\neg a_1 \wedge \neg b_1) \vee (a_1 \wedge b_1)) \wedge (a_0 \wedge \neg b_0)]$

Unindo todas 3 partes em uma única expressão, temos:

$$\begin{aligned} GT_6(a_2a_1a_0b_2b_1b_0) \equiv & (a_2 \wedge \neg b_2) \vee \\ & [((\neg a_2 \wedge \neg b_2) \vee (a_2 \wedge b_2)) \wedge (a_1 \wedge \neg b_1)] \vee \\ & [((\neg a_2 \wedge \neg b_2) \vee (a_2 \wedge b_2)) \wedge ((\neg a_1 \wedge \neg b_1) \vee (a_1 \wedge b_1)) \wedge (a_0 \wedge \neg b_0)] \end{aligned}$$

Finalmente, interpretando essa expressão na linguagem AON-CIRC temos o programa conforme solução no arquivo src/probl1/PROG_AON.txt e orientações de execução no README.

1.2 Solução - parte II

Na solução anterior temos que $GT_{2n} = GT_6$, onde $n = 3$ e o programa final possui 19 portas lógicas. Supondo $c.n = 19$ temos que $c = 19/3 = 6.33$. Como queremos provar que existe certo $c \in \mathbb{N}$, então, aceitamos o natural mais próximo que é 7. Portanto, temos que nossa solução GT_6 possui no máximo 7k portas lógicas.

Vamos abordar a prova como um caso de prova por indução. Analisando o próximo valor de n, ou seja, $n = 4$, temos $GT_8(a_3a_2a_1a_0b_3b_2b_1b_0)$. Para que o número representado por $a_3a_2a_1a_0$ seja maior que o número representado por $b_3b_2b_1b_0$ temos que $a_3 > b_3$ ou o resultado é verdadeiro se $a_3 = b_3$ e a avaliação da comparação entre $a_2a_1a_0$ e $b_2b_1b_0$ é verdadeira. Note que a comparação dos bits restantes é dada pela função GT_6 já definida. Temos, portanto, que a função GT_8 é verdadeira conforme o seguinte cenário:

1. Se $a_3 > b_3$ ou
2. Se $a_3 = b_3$ e $GT_6(a_2a_1a_0b_2b_1b_0)$

Por equivalência, podemos construir uma expressão para a função GT_8 conforme abaixo:

$$GT_8(a_3a_2a_1a_0b_3b_2b_1b_0) \equiv (a_3 \wedge \neg b_3) \vee (((\neg a_3 \wedge \neg b_3) \vee (a_3 \wedge b_3)) \wedge GT_6(a_2a_1a_0b_2b_1b_0))$$

Um pseudo-programa que daria solução a esta solução seria da seguinte forma:

a2 = X[0]
a1 = X[1]
a0 = X[2]
b2 = X[3]
b1 = X[4]
b0 = X[5]

notb3 = NOT(b3)
nota3 = NOT(a3)
and1 = AND(nota3,notb3)
and2 = AND(a3,b3)
or1 = OR(and1,and2)
and3 = AND(or1,$GT_6(a_2a_1a_0b_2b_1b_0)$)

and4 = AND(a3,notb3)
Y[0] = OR(and4,and3)

Note que precisamos de 8 portas lógicas a mais que em GT_6 para conseguirmos computar GT_8. Isso significa que a cada incremento de k, 8 portas lógicas a mais serão necessárias para computar a função. Se $c.4 = 8$, $c = 8/4 = 2$. Logo, aumentamos em

2k o numero de portas necessárias para computar a função GT_8. Portanto temos que o número de portas lógicas para computar GT_8 é dado por $2k + 7k = 9k$ portas.

Assim, podemos definir nossa hipótese de indução da seguinte forma: GT_{2k} é computável usando no máximo 9k portas lógicas.

No passo de indução, podemos construir uma lógica semelhante ao apresentado anteriormente. O número de portas lógicas necessário para computar $GT_{2(k+1)}$ é 8 mais o número de portas lógicas para computar GT_{2k}. Assim, temos que $8 + 9k < 9(k + 1) \equiv 8 + 9k < 9k + 9$.

Portanto, provamos que existe uma constante $c \in \mathbb{N}$ tal que, para todo $n \in \mathbb{N}$ existe um circuito Booleano C com no máximo $c.n$ portas lógicas.

1.3 Solução - parte III

Por definição, temos que "Para cada circuito Booleano C de s portas lógicas, existe um circuito NAND C' de no máximo $3.s$ portas lógicas que computa a mesma função que C".

Do exercício anterior, provamos que existe uma constante $c \in \mathbb{N}$, tal que, para todo $n \in \mathbb{N}$ existe um circuito Booleano C com no máximo $c.n$ portas lógicas. Vimos que, em AON-CIRC, um número de portas lógicas necessário para computar $GT_{2(k+1)}$ é 8 mais o número de portas lógicas para computar GT_{2k}. Neste caso, portanto, aplicando a definição, temos que o número de portas lógicas para computar GT_{2k} em NAND-CIRC é no máximo 3 vezes maior que o número necessário em AON-CIRC. Portanto, existe uma nova constante c, múltipla de 3, aplicável sobre uma constante anterior que já provamos existir. Logo, também provamos que existe tal constante c aplicável a linguagem NAND-CIRC.

Se quisermos verificar a prova, mais uma vez, podemos provar por indução. Vamos assumir nosso caso base como sendo o mapeamento do programa GT_6 em AON-CIRC para GT_6 em NAND-CIRC.

Pela definição 3.12, podemos definir nossa hipótese de indução da seguinte forma: GT_{2k} é computável usando no máximo 3(9k) portas lógicas, ou seja, 27k portas lógicas.

No passo de indução, podemos construir uma lógica semelhante. O número de portas lógicas necessário para computar $GT_{2(k+1)}$ em NAND-CIRC é 3 vezes maior que o necessário em AON-CIRC. Assim, temos que $3(8 + 9k) < 27(k + 1) \equiv 24 + 27k < 27k + 27$.

Portanto, provamos que existe uma constante $c \in \mathbb{N}$ tal que, para todo $n \in \mathbb{N}$ existe um circuito Booleano C com no máximo $c.n$ portas lógicas também em NAND-CIRC.

2 Problema 2

Este problema trabalha com uma representação da linguagem NAND-CIRC-IF em Clojure. Tipicamente, ao se processar uma linguagem, o código fonte é convertido para alguma representação interna. Esta representação interna costuma ser chamada de Abstract Syntax Tree (AST). Neste problema você receberá o AST correspondente a um programa em NAND-CIRC-IF.

O AST será representado por um mapa em Clojure. A Tabela 1 apresenta exemplos das estruturas de dados usadas para representar os elementos dos programas em NAND-CIRC-IF. Note que os casos apresentados naquela tabela são apenas exemplos, os programas NAND-CIRCIF podem ser bem mais complexos. A sintaxe completa da linguagem NAND-CIRC-IF é dada na Subseção A.4.

Parte I - (10pts)

Escreva uma função em Clojure que receba o AST de um programa NAND-CIRC-IF P como entrada e produza um programa P' em NAND-CIRC, i.e., sem a estrutura de controle $if : \ldots else : \ldots end$, que compute a mesma função que P. A sua função em Clojure pode assumir a existência da função IF em NAND-CIRC.

Parte II - (10pts)

Seja P1 um programa NAND-CIRC com $s1$ linha que computa a função $f : \{0,1\}^n \to \{0,1\}$, e seja P2 um programa NAND-CIRC com $s2$ linhas que computa a função $g : \{0,1\}^n \to \{0,1\}$. Prove que existe um programa P' em NAND-CIRC com no máximo $s1 + s2 + 10$ linhas que computa a função $h : \{0,1\}^{n+1} \to \{0,1\}$, onde:

$$h(x_0, \ldots, x_{n-1}, x_n) = \begin{cases} f(x_0, \ldots, x_{n-1}) & \text{se } x_n = 0 \\ g(x_0, \ldots, x_{n-1}) & \text{se } x_n = 1 \end{cases} \quad (1)$$

2.1 Solução - parte I

Não finalizado. Implementei o início e testes básicos em Clojure, mas demorei demais e gerenciei a lista, por fim, não deu tempo de finalizar.

2.2 Solução - parte II

Para solução deste problema, observamos primeiramente que a condição para que seja definido se f ou g será executado é dado por x_n. Ou seja, a saída da função h será igual à de f ou g conforme o valor de x_n. Seja $a = g(x_0, \ldots, x_{n-1})$, onde $a \in \{0,1\}$ é o resultado da computação da função g; e seja $b = f(x_0, \ldots, x_{n-1})$, onde $b \in \{0,1\}$ é o resultado da computação da função f. Note que a função h é equivalente a um condicional $if(x_n, a, b)$ que verifica o condicional x_n e retorna a saída a caso verdadeiro ou b caso falso. Uma abstração da ideia pode ser interpretada da seguinte forma:

```
if(xn):
    g(x)
else
    f(x)
end
```

onde xn equivale a x_n e x equivale a toda a cadeia de entrada $x_0, \ldots, x_{n-1}$ para f ou g.

Em NAND-CIRC não temos o comando if como açúcar sintático, mas podemos implementá-lo apenas com comandos NAND conforme o seguinte:

```
def IF(cond,a,b):
    notcond = NAND(cond,cond)
    temp = NAND(b,notcond)
    temp1 = NAND(a,cond)
    return NAND(temp,temp1)
```

Essa implementação nos possibilita construirmos um programa P' em NAND-CIRC unindo as linhas de código do programa P_1 (todos comandos estão em NAND-CIRC) com as linhas de código do programa P_2, respectivamente, e tratarmos as saídas. Para isso, ao final do programa P_1, acrescentamos um comando de atribuição de uma variável a como saída do programa P_1; semelhantemente, ao final do programa P_2 acrescentamos um comando de atribuição de uma variável b e, por fim, adicionamos as 4 linhas de código do IF equivalente em NAND-CIRC passando o condicional x_n. Ao final, nosso programa P' terá executado o programa P_1, o programa P_2 e verificado quem deverá ser a saída de P' avaliando x_n. Portanto, teríamos um programa da seguinte forma:

$$P' = \begin{cases} P1 & \text{código NAND-CIRC} \\ & \vdots \\ a = p1 & \text{guarda o resultado da função g} \\ P2 & \text{código NAND-CIRC} \\ & \vdots \\ b = p2 & \text{guarda o resultado da função f} \\ notcond = NAND(X[n], X[n]) & \text{início do tratamento if} \\ temp = NAND(b, notcond) & \\ temp1 = NAND(a, X[n]) & \\ Y[0] = NAND(temp, temp1) & \text{resultado conforme condição do if} \end{cases} \tag{2}$$

onde $p1, p2 \in \{0,1\}$ são os bits de saída do programa P_1 e P_2, respectivamente.

Note que conseguimos dessa forma um código em NAND-CIRC que equivale a função h, ou seja, dependendo do valor de x_n oferecerá como saída o equivalente à computação de g ou de f.

Por fim, o número de linhas que computará o programa P' será dado por $s_1 + s_2 + 6$, sendo s_1 o número de linhas de P_1, s_2 o número de linhas de P_2, 4 comandos para implementar o if em NAND-CIRC e 2 comandos de atribuição de variável para a e b recebendo as saídas de P_1 e P_2.

Portanto, provamos que existe um programa P' em NAND-CIRC com no máximo $s1 + s2 + 10$ linhas que computa a função $h : \{0,1\}^{n+1} \to \{0,1\}$.

3 Problema 3

Parte I - (5pts)

Escreva um programa em NAND-CIRC-FOR, conforme definida na Subseção A.5, que compute $MULT3 : \{0,1\}^6 \to \{0,1\}^6$, ou seja, a função que multiplica dois números de 3 bits. As entradas $X[0]$, $X[1]$ e $X[2]$ representarão o primeiro número, sendo X[0] o bit de mais alta ordem, e as entradas $X[3]$, $X[4]$ e $X[5]$ representarão o segundo bit, sendo X[3] o bit de mais alta ordem. O resultado da multiplicação será representado pelas saídas $Y[0]$ a $Y[5]$ sendo $Y[0]$ o bit de mais alta ordem.

Parte II - (10pts)

Escreva uma função em Clojure que receba um número n como entrada e produza como saída um programa em NAND-CIRC padrão que calcule a função $MULT_n : \{0,1\}^{2n} \to \{0,1\}^{2n}$. Sua função deve retornar uma string com o código do programa NAND-CIRC.

Parte III - (5pts)

Demonstre ou refute a afirmação de que o programa $MULT_n$ em NAND-CIRC gerado pela sua função sempre terá menos do que $1000 \cdot n^2$ linhas de código.

3.1 Solução - parte I

Para solução deste problema, utilizei o for para gerar as multiplicações bit a bit e também for para resolver as somas dos números gerados pelas multiplicações. Fiz a multiplicação bit a bit utilizando a função AND.

Consegui testar esse código interpretando-o com desugar do for, adicionando algumas definições de função (XOR, AND, ZERO...) e executando com variante nand e dialeto proc. Os resultados foram corretos.

Arquivo com a solução do problema está em src/probl3/solucao_mult3.txt e orientações para execução em README.md.

3.2 Solução - parte II

A função "imprime-multn"recebe um número inteiro e escreve em um arquivo multn.txt o código NAND-CIRC. Foi adicionado ao arquivo da solução as definições para as funções NOT, AND, OR, XOR, ONE e ZERO como procedimentos em NAND-CIRC para utilização como açúcar sintático.

Arquivo com a implementação da solução do problema está em src/probl3/multn.clj e orientações para execução em README.md.

Tive o resultado esperado, para os testes de diversos tamanhos de n a função está correta.

3.3 Solução - parte III

Para analisar o crescimento de linhas de código da minha função com a função proposta plotei um gráfico com ambas conforme os seguintes passos:

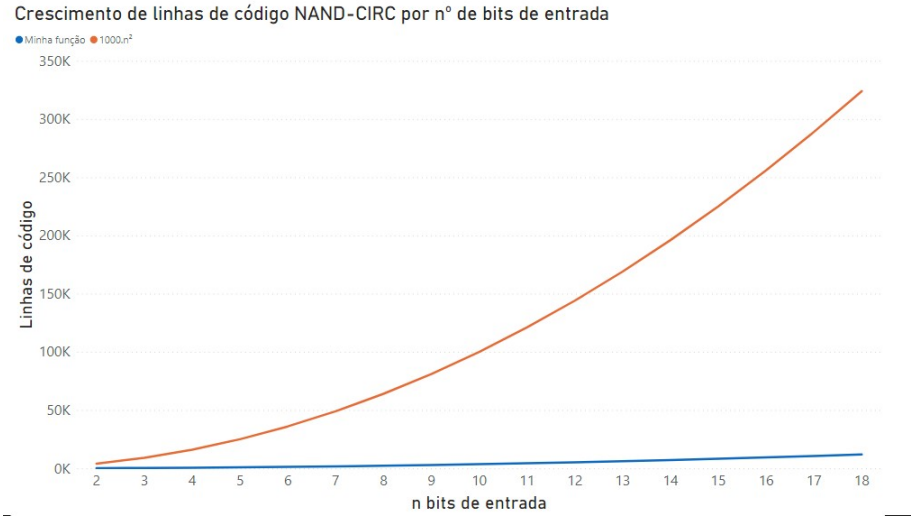
- Executei a função multn (implementada em Clojure) para valores de n entre 2 e 18 bits. (Obs: tentei gerar até 100, mas ao aplicar o desugar do próximo passo acima de 18 bits deu erro no interpretador)
- Apliquei o desugar desses arquivos para código NAND-CIRC puro
- Contei quantas linhas de código esses programas geraram
- Plotei o gráfico comparando o número de linhas de código que meus arquivos tiveram com o equivalente na fórmula $1000.n^2$ para cada n bits de entrada, conforme tabela 2.

n bits	minha função	$1000.n^2$
2	100	4000
3	266	9000
4	508	16000
5	826	25000
6	1220	36000
7	1690	49000
8	2236	64000
9	2858	81000
10	3556	100000
11	4330	121000
12	5180	144000
13	6106	169000
14	7108	196000
15	8186	225000
16	9340	256000
17	10570	289000
18	11876	324000

Tabela 2: Número de linhas de código por n bits de entrada para minha solução e para a expressão $1000.n^2$

Como podemos perceber na tabela 2, o número de linhas de código limite dado pela expressão $1000.n^2$ cresce muito mais rápido que o número de linhas de código resultante da minha função. Este resultado pode ser visualizado também pela Figura 1. Assim, demonstramos que o resultado da função $MULT_n$ em NAND-CIRC sempre terá menos que $1000.n^2$ linhas de código.

Figura 1: Comparação entre aumento de linhas de código.



4 Problema 4

Considere a representação de programas NAND-CIRC dada na Definição 5.7 de Barak [2021], e repetida abaixo.

4 Definição 5.7: Representação em Lista de Tuplas Seja P um programa NAND-CIRC de n entradas, m saídas e com s linhas de código, e seja t o número de variáveis distintas usadas em P (incluindo entradas e saídas). A *representação em lista de tuplas de P* é um trio (n, m, L) onde L é uma lista de trios (i, j, k) para $i, j, j \in [t]$. Atribuímos números às variáveis de P da seguinte forma:

- Para cada $i \in [n]$, a variável $X[i]$ é associada ao número i.
- Para cada $j \in [m]$, a variável $Y[j]$ é associada ao número j.
- Cada uma das outras variáveis é associada a um número $\{n, n+1, ..., t-m-1\}$ na ordem em que elas aparecem no programa P.

Parte I - (10pts)

Escreva uma função em Clojure que receba como entrada o AST de um programa NAND-CIRC, i.e., uma sequência de dicionários de expressões do tipo ":assign", conforme exemplificado na Tabela 1, e retorne a representação em lista de tuplas deste programa.

Parte II - (10pts)

Escreva uma função em Clojure que receba um trio com a representação em lista de tuplas de P, um programa NAND-CIRC, e uma sequência de 0's e 1's representando as entradas de P, e retorna uma sequência de 0's e 1's representando as saídas de P.

4.1 Solução - parte I

A solução executa um loop na lista de entrada onde cada item é um comando NAND-CIRC em AST. Para cada comando NAND, extraio as variáveis envolvidas e gero uma saída com os trios: as variáveis encontradas no left-side e as duas no rigth-side do comando NAND.

Dessa lista, separo as variáveis de entrada X, as variáveis de saída Y e as variáveis internas ao programa. Ordeno as variáveis de entrada e saída e as combino em um vetor conforme a ordem: variáveis de entrada - variaveis internas - variáveis de saída, respectivamente.

Por fim, percorro todo o AST montando a tupla com os resultados obtidos na função que busca apenas o n de entrada, o m de saída e a lista L.

Arquivo com a implementação da solução do problema está em src/probl4/ast_to_nand.clj e orientações para execução em README.md.

Tive o resultado esperado e os testes executados corretamente.

4.2 Solução - parte II

A solução foi separar as variáveis em um vetor independente com as variáveis de entrada de forma única, identificando seus nomes (números), em ordem crescente e com seu respectivo valor a ser armazenado à frente da posição da variavel. Ex: [001020314150] Neste vetor, as variaveis 0, 1, 2 e 5 estão com valor 0 e as variaveis 3 e 4 estão com valor 1. Dessa forma ficou simples manter um vetor atualizado perpetuando a cada novo passo do loop atualizando os valores das variáveis à medida em que os comandos nand vão sendo interpretados. Ao final, retorno apenas os valores das últimas m variáveis, que são as de saída Y.

Arquivo com a implementação da solução do problema está em src/probl4/exec_nand.clj e orientações para execução em README.md.

Tive o resultado esperado e os testes executados corretamente.

5 Problema 5

(20pts)

Para qualquer $n \in \mathbb{N}$ suficientemente grande, seja $E_n : \{0,1\}^{n^2} \to \{0,1\}$ a função que recebe cadeias de tamanho n2 que codificam pares (P, x), onde $x \in \{0,1\}^n$ e P é um programa NAND-CIRC, tal que P recebe n entradas e gera uma saída, e P tem no máximo $n^{1.1}$ linhas de código, e retorna a saída de P sobre x. Ou seja, $E_n(P, x) = P(x)$.

Prove que para todo valor suficientemente grande de $n \in \mathbb{N}$, não existe um circuito XOR C que compute a função E_n, onde um circuito XOR contém a porta lógica XOR e as constantes 0 e 1. Ou seja, prove que existe alguma constante n0 tal que para todo n > n0 e todo circuito XOR C de n2 entradas e uma saída, existe um par (P, x) tal que $C(P, x) \neq E_n(P, x)$.

5.1 Solução 5

A ideia geral do problema é mostrar que, utilizando somente a porta XOR, não é possível sempre construir um circuito capaz de executar qualquer programa definido em NAND-CIRC. Sempre haverá pelo menos um programa em NAND-CIRC (P,x) que não poderá ser expresso em circuito XOR C(P,x) para qualquer tamanho de n bits na entrada do programa.

Vejamos a seguinte tabela verdade para n = 2, onde $n_0 = a$ e $n_1 = b$:

a	b	NAND	XOR
0	0	1	0
0	1	1	1
1	0	1	1
1	1	0	0

Tabela 3: Tabela verdade para NAND e XOR

NAND e XOR são iguais quando (a,b) é (0,1), (1,0), e (1,1) e diferem quando (a,b) é (0,0). Note que o XOR é zero quando ambos bits são iguais e 1 quando são diferentes. Essa é a única distinção que o XOR faz sobre os bits: se são iguais ou diferentes. Não importa para o XOR se os bits são ambos 1 ou ambos 0 e aqui está a primeira parte do problema. Sem fazer esta distinção, não conseguimos criar um circuito apenas com XOR, 0 e 1 que reconheça essa distinção (pois, cada vez que o XOR for utilizado, essa distinção será ignorada) e, consequentemente, não conseguimos implementar um simples NAND em XOR.

Vamos verificar. A função XOR recebe dois bits de entrada e oferece como saída 2 pares de possibilidades: 2 casos em que o resultado é 0 e 2 casos em que o resultado é 1. A função NAND, por outro lado, recebe dois bits de entrada, porém, fornece como saída 3 casos em que o resultado é 1 e um caso em que o resultado é zero. Ou seja, a quantidade de zeros e uns do XOR é par e a quantidade de zeros e uns do NAND é ímpar. Se a quantidade de zeros e uns do NAND é ímpar e as saídas possíveis do XOR são pares, não temos como representar um NAND com um XOR. Vamos analisar porque também não é possível representar um NAND mesmo que utilizemos mais de um XOR.

Considere o seguinte programa XOR e a tabela para cada combinação das entradas a e b:

```
xab  = XOR(a,b)
xza  = XOR(0,a)
xua  = XOR(1,a)
xaab = XOR(a,xab)
```

Observe que todas as saídas XOR, mesmo usando 0 ou 1 fixos ou com XOR no parâmetro (chamada recursiva) sempre retornam quantidades pares de zeros e uns. Isso mostra que a função XOR, necessariamente, apenas implementa circuitos cujas saídas sejam sempre quantidades pares de zeros e uns, independentemente da quantidade de portas XOR utilizadas, uma vez que, a última porta avaliada será uma porta XOR e dará o resultado em pares de zeros e uns.

a	b	XOR(a,b)	XOR(0,a)	XOR(1,a))	XOR(a, XOR(a,b))
0	0	0	0	1	0
0	1	1	0	1	1
1	0	1	1	0	0
1	1	0	1	0	1

Tabela 4: Tabela verdade para NAND e XOR

Assim, vimos que um circuito XOR não consegue implementar uma porta NAND(a,b). Agora vamos considerar a prova de que sempre existe um circuito C(P,x) em XOR para x de n bits que não implementa a função $E_n(P, x)$.

Tomemos como exemplo um programa semelhante ao exercício 1.2, GT_{2n} em NAND-CIRC. Seja P um programa que computa a função $GT_{2n} : \{0,1\}^{2n} \rightarrow 0,1$ tal que $GT_{2n}(x0 \dots x_{n-1}x_n \dots x_{2n-1}) = 1$ se e somente se o número representado por $x0 \dots xn-1$ for maior do que o número representado por $x_n \dots x_{2n-1}$. Considere que os números estão em binário, com o bit menos significativo à direita.

Seja n=1, logo $GT_2 : \{0,1\}^2 \rightarrow \{0,1\}$. Neste caso, basta verificar se a $>$ b que pode ser expressado por $a \wedge \neg b$, conforme a seguinte tabela verdade.

a	b	a $>$ b	$a \wedge \neg b$
0	0	0	0
0	1	0	0
1	0	1	1
1	1	0	0

Tabela 5: Tabela verdade para a $>$ b

Assim, em NAND-CIRC o programa poderia ser definido como:

```
def GT2_NAND(a,b):
    notb = NAND(b,b)
    n1 = NAND(notb, a)
    Y[0] = NAND(n1, n1)
```

Ao tentarmos aplicar uma solução com porta XOR, temos:

```
def GT2_XOR(a,b):
    notb = XOR(1,b)
    n1 = ...
    Y[0] = XOR(1, n1)
```

Note que não é possível implementar o passo 2 para atribuir um equivalente NAND(not, a) com portas XOR conforme explicado anteriormente. Uma outra solução diferente também não é possível, pois a saída da expressão $a > b$ é de um número ímpar de zeros e uns, que já vimos não ser possível de expressar apenas com portas XOR, o que nos impossibilita escrevermos esse programa em XOR.

Conforme visto no exercício 1.2, a generalização da função GT_{2n} é simples. A cada novo par de bits na entrada, comparamos se o bit mais significativo do primeiro número dado por $x0 \dots x_{n-1}$ é maior que o bit mais significativo do segundo número dado por $xn \dots x_{2n-1}$ e, dependendo do resultado, repetimos o processo para o próximo bit de cada um dos dois números até encontrarmos a saída. Note que nesse processo, cada comparação entre dois bits necessariamente passa pelo primeiro caso, ou seja $GT_2(a, b)$ em algum momento para avaliar se $a > b$ (ver solução 1.2).

Neste ponto sempre esbarramos no problema de não conseguirmos finalizar essa comparação apenas usando XOR, o que implica não existir um circuito XOR capaz de implementar a função $E_n(P, x)$ para todo $n \in \mathbb{N}$.