Léo Vieira Peres

Complexidade de circuitos booleanos

Florianópolis/SC
2015

Léo Vieira Peres

Complexidade de circuitos booleanos

Orientadora:
Prof. Dr. Jerusa Marchi

Proposta de trabalho de conclusão
de curso para a obtenção do grau de
bacharel em ciências da computação
pela Universidade Federal de Santa Catarina

Florianópolis/SC
2015

Sumário

Resumo

Podemos dizer que complexidade computacional busca descobrir o quão dificil é resolver problemas computacionais. Por exemplo, uma forma de descrever o problema em aberto mais importante da teoria da computação, $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, é perguntar se o problema da sastifazibilidade booleana necessita de tempo "mais do que polinomial" para ser decidido no caso geral. No entanto, até agora não se obteve muito sucesso em provar limites inferiores para complexidade de problemas.

Classificar problemas pela sua complexidade de circuitos é uma duas principais frentes de pesquisa para provar limites inferior de problemas computacionais e por muitos anos pesquisadores acreditaram que complexidade de circuitos era a chave para provar problemas como $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, onde a complexidade de circuito de um problema é basicamente o número mínimo de portas lógicas necessárias para implementar um circuito que decida este problema.

Nós veremos que as técnicas conhecidas até recentemente para provar limites inferiores são limitadas e não são suficientes para provar que $\mathsf{P} \neq \mathsf{NP}$. Entretanto, resultados recentes conseguiram se esquivar destas limitações e abriram caminho para novos tópicos de pesquisa.

A princípio, o objetivo do trabalho é realizar um estudo sobre os resultados mais recentes em complexidade de circuitos.

Palavras chaves: complexidade computacional, complexidade de circuitos.

Capítulo 1

Introdução

O objetivo central da área de complexidade computacional é saber a dificuldade intrínseca de problemas computacionais, diferentemente de design de algoritmos que busca encontrar soluções eficientes para um problema. Quando falamos de complexidade de problemas computacionais estamos querendo dizer o recurso necessário para resolver tais problemas.

Quando analisamos a complexidade de problemas computacionais nós devemos levar em conta o modelo computacional e o recurso em questão. Às vezes estamos interessados em avaliar o tempo necessário para computar um certo problema em uma máquina de acesso aleatório ou talvez o número de bits que dois processadores enviam um ao outro. Alguns resultados antigos mostraram que dar mais tempo ou espaço para máquinas de Turing aumenta o número de problemas que elas podem resolver, porém estes resultados não apresentaram limites inferiores para problemas naturais [**?**]. Meyer e Stockmeyer provaram que certos problemas são completos para a classe de problemas que necessitam de tempo exponencial em máquinas de Turing [**?**], o que também significa que estes problemas não podem ser decididos em tempo polinomial. O limite inferior de $\Omega(nlogn)$ passos para algoritmos de ordenação de lista que usa comparações é um exemplo de limite inferior em algoritmos.

Em complexidade de circuitos procura-se saber o tamanho ou profundidade dos circuitos necessários para decidir uma linguagem. Circuitos booleanos são matematicamente mais simples do que máquinas de Turing e alguns resultados em complexidade de circuitos resolveriam também problemas em aberto em outros modelos de computação. Como exemplo, se conseguirmos provar que problemas que são fáceis de se verificar uma solução não têm circuitos de tamanho polinomial ($\mathsf{NP} \not\subseteq \mathsf{P/poly}$) então $\mathsf{P} \neq \mathsf{NP}$. Porém, provar que $\mathsf{NP} \not\subseteq \mathsf{P/poly}$ é extremamente dificil e por isso o foco de pesquisa hoje em dia é provar problemas mais fracos, na maioria das vezes restrigindo a classe de circuitos (como por exemplo, circuitos de profundidade logaritmica). Nos anos 80 houve avanços nesse sentido quando pesquisadores conseguiram mostrar que certos problemas não podiam ser decididos por classes mais restritas de circuitos. No entanto, em 1994, Razborov e Rudich mostraram que, sob algumas hipóteses de complexidade computacional, as técnicas usadas até então para provar limites inferiores,

as quais eles chamaram de provas naturais, não seriam suficiente para provar que $\mathsf{P} \neq \mathsf{NP}$ [**?**]. E por isso, para que qualquer estratégia de prova possa ser levada a frente é necessário de alguma forma passar pelas limitações das provas naturais.

Nos últimos anos, novos limites inferiores em complexidade de circuitos foram obtidos usando uma estratégia de prova que liga algoritmos "rápidos" a limite inferiores [**?**, **?**]. Para uma determinada classe de circuitos C, se você conseguir mostrar que o problema C–SAT (o problema de avaliar se um circuito em C não computa a função $f(x) = 0$, para todo x) tem um algoritmo mais rápido do que o algoritmo mais óbvio (tentar todas as 2^n entradas possíveis), então você consegue mostrar que a classe de problemas cuja uma solução pode ser verificada em tempo exponencial (NEXP) não tem circuitos em C. Estudar a conexão entre algoritmos e limites inferiores em circuitos booleanos é um assunto interessante para alguém que deseja realizar pesquisas em complexidade computacional.

1.1 Objetivo

Este trabalho visa apresentar um estudo teórico acerca da área de complexidade de circuitos e suas aplicações ao estudo da complexidade computacional. Primeiramente será feito um estudo sobre complexidade computacional. Depois serão pesquisados tópicos em complexidades de circuitos com foco principal nos tópicos mais recentes e ferramentas/técnicas comumente usadas em provas em complexidade de circuitos.

Capítulo 2

Fundamentos

Neste capítulo nós introduzimos algumas convenções e conceitos fundamentais para entender este trabalho. Muitas das convenções usadas aqui são as mesmas encontradas em alguns dos principais livros de teoria da computação [**?**, **?**, **?**, **?**, **?**].

Introduzimos primeiro linguagens e como representar problemas computacionais como linguagens. Depois falamos de máquinas de Turings e suas variantes, e mostramos alguns resultados de fundamental importância. Na seção 2.3 nós vemos circuitos Booelanos como um modelo de computação. Nas seções 2.4 e 2.5 nós discutiremos complexidade computacional pela primeira vez em algum detalhe. Finalmente, na seção 2.6 veremos um introdução elementar à análise de Fourier de funções Booleanas.

2.1 Linguagens

Um *alfabeto* Σ é um conjunto finito e não vazio de símbolos como $\{0,1\}$ ou $\{a,b,c\}$. Uma *palavra* construída sobre um alfabeto Σ é uma sequência de símbolos de Σ. Como exemplo, se Σ for o alfabeto binário $\{0,1\}$, então 0011 e 0101 são palavras sobre Σ. Finalmente, denotamos por Σ^* o conjunto de todas as palavras formada por símbolos de Σ e definimos uma *linguagem* como um subconjunto qualquer de Σ^*.

Permitimos uma palavra vazia que não contém nenhum símbolo e denotamos esta palavra por ε e temos que $\varepsilon \in \Sigma^*$, para qualquer alfabeto Σ. O tamanho de uma palavra w é o número de símbolos que a compõem e é denotada por $|w|$ — desta forma $|\varepsilon| = 0$. Para representar o i-ésimo símbolo que compõe uma palavra w nós escreveremos w_i. Para algum inteiro $n \geq 0$, Σ^n denota o conjunto de todas as palavras de tamanho n sobre o alfabeto Σ.

Nós também queremos representar objetos como grafos, vetores, etc, através de palavras. Neste caso, se x é um objeto qualquer, então sua representação em binário será denotada por $\langle x \rangle$.

O que mais nos importa aqui é que podemos representar problemas computacionais através de linguagens, o que nos chamamos de problemas de decisão. Após problemas de decisão, nós veremos problemas de busca que diferem de problemas de decisão na maneira em que ele são representados e no número de soluções admitidas.

2.1.1 Problemas de decisão

Em problemas de decisão nós queremos decidir se um determinado elemento pertence a um conjunto S ou não. Como exemplo de um problema de decisão: dado um número natural p, nós queremos decidir se p é primo. Neste caso S é o conjunto de todos os números primos.

Para solucionar o problema de decisão de $S \subseteq \{0,1\}^*$ nós usamos uma função $f : \{0,1\}^* \to \{0,1\}$ tal que $S = \{x \mid f(x) = 1\}$. Chamamos f de *função característica* de S. Dessa forma, solucionar um problema de decisão é análogo a decidir se uma palavra pertence à uma linguagem, dado uma representação das instâncias do problema como strings binárias.

2.1.2 Problemas de busca

Em problemas de busca, é dada uma instância e queremos achar uma solução do problema para aquela instância. Por exemplo, dado um grafo G e vértices u e v, achar o caminho mais curto entre u e v seguindo as arestas de G.

Nós vamos representar um problema de busca por uma relação $R \subseteq \{0,1\}^* \times \{0,1\}^*$, onde $R(x,y)$ sse y é uma solução para a instância x, e para cada x temos $R(x) = \{y \mid R(x,y)\}$, ou seja, $R(x)$ é conjunto de todas as soluções para x. Um solucionador para R é uma função $f : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$ onde

$$f(x) = \begin{cases} y \in R(x) & \text{se } R(x) \neq \emptyset \\ \perp & \text{caso contrário.} \end{cases}$$

2.2 Máquinas de Turing

Uma visão intuitiva de uma máquina de Turing é a de um matemático que tem consigo uma folha de rascunho em que ele pode escrever os resultados parciais de sua computação e um conjunto finito de instruções que ele deve seguir. Formalmente, uma máquina de Turing é composta por três unidades:

- k fitas infinitas à direita que contêm células adjacentes e um cabeçote que em um dado momento se localiza em uma única célula de sua fita e que pode realizar as seguintes funções: a) escrever ou apagar um símbolo na célula em que ele se encontra b) se mover para uma das células adjacentes à sua célula atual;

- um registrador que guarda o estado atual da computação;
- um conjunto de instruções.

A computação inicia com os k cabeçotes na célula mais à esquerda de suas respectivas fitas e em um estado inicial que é o mesmo para todas as entradas. Daí em cada passo da computação os k cabeçotes irão ler o conteúdo atual das células em que eles se encontram e conforme o estado atual e o conjunto de instruções eles decidem se escrevem ou apagam um símbolo na sua célula atual (sendo que o símbolo escrito pode ser o mesmo que já se encontra naquela célula) e para qual direção eles irão se movimentar (ou se permanecerão na mesma célula). Após cada passo o registrador de estado passa a guardar um novo estado (ou seja, o próximo estado da computação) que depende do estado atual e o símbolo lido pelos cabeçotes. A computação termina quando o registrador de estado guarda um estado de parada.

Das k fitas da máquina de Turing, a primeira é de somente leitura e a chamaremos de *fita de entrada*. As últimas $k-1$ fitas são de escrita e leitura e elas são chamadas de *fitas de trabalho*, sendo a última fita a *fita de saída*.

A seguir nós vemos uma definição formal de máquina de Turing.

Definição 2.1. *(Máquinas de Turing)*

Uma máquina de Turing M é uma tripla (Γ, Q, δ) onde Γ é o alfabeto de fita, Q é o conjunto de estados de M que contém o estado inicial q_0 e o estado de parada q_h e $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{E, N, D\}^k$ é a função de transição.

A função de transição e interpretada como $\delta(q, (\sigma_1, \ ..., \ \sigma_k)) = (q', (\sigma'_2, \ ..., \ \sigma'_k), z)$, $z \in \{E, N, D\}^k$, significando que quando o estado atual de M for q e os símbolos sendo lidos pelos cabeçotes das k fitas forem $\sigma_1, \ ..., \ \sigma_k$ então M muda o seu estado atual para q', muda o conteúdo das suas últimas $k-1$ fitas para $\sigma'_2, \ ..., \ \sigma'_k$ e os k cabeçotes da fita se movimentam conforme z (a i-ésima fita se move para a esquerda, permanece na mesma célula ou se move para direita se o valor de z_i for E, N ou D, respectivamente). Sempre que um cabeçote que estiver na célula mais à esquerda de sua fita tentar se mover para esquerda, este permanecerá na mesma célula.

A entrada de uma máquina de Turing é o conteúdo da fita de entrada antes do ínicio da computação. Denotamos o resultado da computação de M sobre uma entrada x por $M(x)$.

Neste trabalho vamos na maior parte das vezes assumir que $\Gamma = \{0, 1, \triangleright, \square\}$, onde $\triangleright$ é o símbolo que marca o começo das fitas e $\square$ é um símbolo que denota uma célula vazia.

Nós podemos representar cada passo da computação de uma máquina de Turing levando em conta o conteúdo atual das k fitas, as posições dos cabeçotes e o estado atual. Essa representação de um passo da computação de uma máquina de Turing é chamada de *configuração* e podemos mapear uma configuração para uma palavra em $\{0, 1\}^*$. No ínicio da computação a máquina de Turing se encontra na *configuração inicial*. A sequência de todas as configurações que uma máquina de Turing entra

durante a computação é chamada de *história de computação*. Essa visão dos passos da computação de uma máquina de Turing é útil quando queremos representar toda a computação de uma máquina de Turing como uma string. Sem nos preocurpamos com os detalhes de uma representação das configurações, vamos convencionar que a configuração inicial de qualquer computação terá o seguinte:

- Todas as fitas têm o símbolo $\triangleright$ em sua célula mais à esquerda;
- A primeira fita irá conter uma string $x \in \{0,1\}^*$ após a sua primeira célula;
- Todas as outras células de todas as fitas serão marcadas com $\square$.

2.2.1 Máquina de Turing universal

Note que precisamos somente da função de transição para descrever uma máquina de Turing. Dessa forma podemos representar máquinas de Turing como strings binárias e fazemos duas suposições:

- Cada string $\alpha \in \{0,1\}^*$ descreve uma máquina de Turing
- Cada máquina de Turing é descrita por infinitas strings

A primeira condição pode ser alcançada se mapearmos todas as strings que não são descrição válidas de máquinas de Turing para uma máquina canônica qualquer — como a máquina de Turing que rejeita todas as entradas. A segunda condição pode ser obtida se concatenarmos uma sequência de símbolos inúteis ao fim da descrição da máquina de Turing, isto não irá mudar o conjunto de instruções sendo representado se usarmos alguma sequência de bits para demarcar o fim da descrição.

De acordo com a nossa notação, denotaremos a string que descreve uma máquina de Turing M por $\langle M \rangle$. Se α é uma string, então denotaremos por M_α a máquina de Turing descrita por α.

Essa representação de máquinas de Turing como strings é útil quando queremos usar descrições de máquinas de Turing como entrada para uma máquina de Turing. O teorema a seguir nos diz que existe uma máquina de Turing capaz de simular a execução de qualquer máquina de Turing sobre uma entrada arbitrária.

Teorema 2.2. *(Máquina de Turing universal)*

Existe uma máquina de Turing $\mathcal{U}$ que ao receber $\langle \alpha, x \rangle$ em sua fita de entrada, $\mathcal{U}$ dá como saída o resultado da computação de M_α sobre a entrada x.

Demonstração. Precisamos apenas nos convencer que uma vez que podemos extrair da descrição de M_α (ou seja, a string α) o seu conjunto de estados e sua função de transição temos então toda informação necessária para simular a execução de M_α sobre a entrada x usando as fitas de trabalho de $\mathcal{U}$.

Porém, o número de fitas de $\mathcal{U}$ é finito (somente 3 fitas são necessárias), e $\mathcal{U}$ deve ser capaz de simular qualquer máquina de Turing com um número arbitrário de fitas. Se k é o número de fitas de M_α, então é possível fazer isto guardando o conteúdo de $k-1$ fitas (nós podemos usar a fita de saída de $\mathcal{U}$ para simular a fita de saída de M_α) de M_α em uma das fitas de trabalho de $\mathcal{U}$ particionando esta fita em $k-1$ espaços $E_1, E_2, \ldots, E_{k-1}$, onde cada espaços consecutivos são separados por um símbolo especial (como '#'). Sempre que a i-ésima fita de M_α precisar de mais espaço, movemos todos símbolos que aparecem após a última célula de E_i uma posição para a direita.

Dessa forma, após a simulação teremos $M_\alpha(x)$ escrito sobre a fita de saída de $\mathcal{U}$.

□

Um ponto importante sobre o resultado acima é que a simulação pode ser feita de forma eficiente. No capítulo seguinte iremos definir o que queremos dizer por eficiente e também veremos em detalhe uma máquina de Turing universal ainda mais eficiente do que a máquina de Turing esboçada na prova do teorema anterior.

2.2.2 Máquina de Turing não-determinística

Na nossa definição de máquinas de Turing acima, o próximo passo de uma máquina de Turing é definido somente pelos símbolos sendo lidos pelos seus cabeçotes de fita e o estado atual da máquina. Nós chamamos estas máquinas de Turing cujo o próximo passo é estritamento único de máquinas de Turing determinísticas. Por outro lado, uma máquina de Turing não-determinística tem sempre duas alternativas de próximos passos que ela deve decidir tomar.

Definição 2.3. *(Máquina de Turing não-determinística)*

Uma máquina de Turing não-determinística N é uma máquina de Turing convencional como definida em 2.1 mas com duas funções de transições δ_1 e δ_2. A cada passo de sua execução N deve escolher usar uma de suas duas funções.

Dizemos que N aceita a entrada x se existe pelo menos uma sequência de escolha das funções de transição tal que $N(x) = 1$.

O conjunto de linguagens decididas por máquinas de Turing não-determinística é o mesmo que o conjunto de linguagens decididas por máquinas de Turing determinística, isso segue pois podemos simular uma máquina de Turing não-determinística N por uma máquina de Turing M que tenta todas as possíveis sequência de escolhas da função de transição que N faz. Além disso, máquinas de Turing determinística são uma classe específica de máquinas de Turing não-determinísticas (onde δ_1 e δ_2 são idênticas).

2.2.3 Máquinas de Turing com oráculo

Um oráculo O para uma linguagem L é um dispositivo que recebe uma entrada x e dá como resposta 1 se $x \in L$ e 0 caso contrário. Nós não estamos preocupados com o funcionamento interno de um oráculo, nós vemos oráculos como "caixas pretas" donde nós simplesmente colocamos a entrada em um lado e recebemos a saída em outro lado. Máquinas de Turing com oráculo são máquinas de Turing convencionais que têm acesso a um oráculo.

Definição 2.4. *(Máquina de Turing com oráculo)*

Uma máquina de Turing M com acesso a um oráculo para L é uma máquina de Turing convencional com a adição de uma fita que chamaremos de fita de oráculo *e três estados $q_{consulta}$, q_{sim} e $q_{não}$. Sempre que M quiser consultar o oráculo para saber se uma string x' pertence a L ou não, M escreve x' sobre sua fita de oráculo e muda seu estado para $q_{consulta}$. Daí, o próximo estado de M será q_{sim} caso $x' \in L$, ou $q_{não}$ caso contrário.*

A partir de agora denotaremos uma máquina de Turing M com acesso a um oráculo para uma linguagem L por M^L e o resultado da computação de M^L sobre x por $M^L(x)$.

Se uma linguagem L' é decidida por uma máquina de Turing com acesso a um oráculo O nós dizemos que L é decidível em relação a O.

A seguir nós vemos que, como esperado, a adição de um oráculo nos dar um poder adicional em relação a máquinas de Turing convencionais.

Teorema 2.5. *Existe uma linguagem que é decidível em relação a algum oráculo mas que não é decidível por uma máquina de Turing sem acesso a nenhum oráculo.*

Demonstração. Considere a seguinte linguagem:

$$\mathsf{HALT} = \{\langle \alpha, x \rangle |\ M_\alpha \text{ para após um número finito de passos quando recebe x como entrada}\}$$

Podemos decidir HALT com um oráculo para HALT. M^{HALT} simplesmente copia o conteúdo de sua fita de entrada para a sua fita de oráculo e faz uma consulta ao oráculo. Após isso M^{HALT} escreve em sua fita de saída 1 se ela estiver no estado q_{sim}, ou 0 caso esteja no estado $q_{\text{não}}$.

Pelo teorema seguinte nós vemos que nenhuma máquina de Turing convencional decide HALT.

□

Teorema 2.6. HALT *não é decidida por nenhuma máquina de Turing sem acesso a um oráculo.*

Demonstração. Assuma que H decida HALT, então existe uma máquina de Turing H' que se comporta da seguinte maneira:

$$H'(\alpha) = \begin{cases} 1 & \text{caso } H(\alpha, \alpha) = 0, \\ 1 - M_\alpha(\alpha) & \text{caso contrário.} \end{cases}$$

Então temos duas possibilidades para quando H' recebe $\langle H' \rangle$ como entrada:

1. Se $H(\langle H' \rangle, \langle H' \rangle) = 1$, então $H'(\langle H' \rangle) = 1 - H'(\langle H' \rangle)$.
2. Se $H(\langle H' \rangle, \langle H' \rangle) = 0$, então $H'(\langle H' \rangle) = 1$.

Daí chegamos a uma contradição.

□

A técnica de prova usada acima se chama *diagonalização*. Esta técnica foi inventada por Georg Cantor que a usou para provar que existe diferente níveis de infinito. Mais precisamente, ele provou que a cardinalidade do conjuntos de todas as string binárias de tamanho infinito tem cardinalidade maior do que o conjunto de todos os números naturais, apesar de ambos os conjuntos serem infinitos.

2.3 Circuitos booleanos

Agora nós vamos ver circuitos booleanos que é o principal modelo de computação para o propósito deste trabalho. Nós também iremos ver como circuito booleanos estão naturalmente relacionados com fórmulas booleanas. Ambos os modelos são "flexíveis" no sentido em que eles não estão somente restritos a um conjunto fixo de operações permitidas. Também iremos ver que os dois modelos são equivalentes dado que as operações primitivas permitidas são as mesmas.

Um circuito booleano é um grafo direcionado acíclico. Nós particionamos os vértices do circuito em três partes: 1) n entradas do cirtuito 2) k portas lógicas 3) uma porta de saída. As entradas do circuito têm grau de entrada zero e os vértice de saída têm grau de saída também zero.

Uma base Ω é uma coleção finita e não vazia de funções booleanas. Cada porta lógica de um circuito (incluindo a porta de saída) deve computar uma função booleana tirada de uma base Ω. Os vértices de entrada guardam algum valor booleano (0 ou 1).

O valor da computação de um circuito vai depender dos valores das variáveis de entrada e de uma sequência de valores de funções tirada de Ω que dependem das variáves de entrada e/ou de funções previamente computadas.

Definição 2.7. *(Circuitos booleanos)*

Um circuito booleano C sobre uma base Ω é um grafo direcionado acíclico com m vértices donde n vértices de grau de entrada zero são as variáveis de entrada $v_1, \ldots, v_n$, e todos os outros vértices

$v_{n+1}, \ldots, v_m$ são portas lógicas que computam alguma função em Ω e que têm grau de entrada e grau de saída maior ou igual a um com a exceção de v_m que é a saída do circuito e tem grau de saída zero.

Cada vértice v do circuito terá um valor associado a ele que denotamos por $val(v)$. As arestas que chegam em uma porta lógica são suas entradas enquanto que as arestas que saem dela são as suas saídas. Dessa forma, se dois vértices u e v, onde u é uma porta lógica ou uma variável de entrada e v é uma porta lógica, são ligados por uma aresta que sai de u e chega em v, então temos que $val(v)$ depende de $val(u)$. Mais precisamente, se v é uma porta lógica que computa uma função $g \in \Omega$ e $u_1, \ldots, u_l$ são todos os vértices que são predecessores de v, então o valor de v é definido por $val(v) = g(val(u_1), \ldots, val(u_l))$.

Como um circuito é um grafo direcionado acíclico e os n primeiros vértices v_1 até v_n são fontes e o último vértice v_m é um sumidouro, podemos assumir que o ordenamento $(v_1, v_2, \ldots, v_m)$ é um ordenamento topológico dos vértices do circuito. Portanto podemos formalizar o funcionamento do circuito da seguinte maneira: Assume-se que os vértices de entradas $v_1, \ldots, v_n$ recebem valores booleanos arbitrários e $x = val(v_1) \cdot val(v_2) \ldots val(v_{n-1}) \cdot val(v_n)$ é a entrada do circuito e queremos computar o valor $C(x) = val(v_m)$. Para isso segue-se em m - n passos onde no i-ésimo passo é computado $val(v_{n+i})$. Note que em cada passo os valores dos vértices dos quais v_{n+i} depende já foram decididos por causa da nossa hipótese que $(v_1, \ldots, v_m)$ é um ordenamento topológico dos vértices do circuito.

Se $f : \{0,1\}^n \to \{0,1\}$ é a função booleana $f(x) = C(x)$, para todo $x \in \{0,1\}^n$, então é dito que C computa f.

O fan-in de uma porta lógica é o número de entradas que ela aceita e o fan-out é o número de saídas (ou seja, o grau de saída). Geralmente o fan-in das porta lógicas de um circuito vão ser limitados por uma constante mas em alguns casos nós vamos considerar classes de circuitos onde não há nenhuma restrição quanto ao fan-in máximo das portas lógicas.

Para superar a limitação de circuitos aceitarem somente entradas de um tamanho fixo nós definimos uma sequência infinita de circuitos onde o n-ésimo circuito da sequência computa uma função com entradas de tamanho n. Desta forma podemos falar de circuitos (ou família de circuitos) que decidem uma dada linguagem, ao invés de somente computar uma função com domínio nas string binárias de um determinado tamanho.

Definição 2.8. *(Família de circuitos)*

Uma família de circuitos é uma sequência $\{C_n\}_{n\in\mathbb{N}}$[1] de circuitos booleanos onde cada C_n computa uma função $f_n : \{0,1\}^n \to \{0,1\}$.

Dizemos que $\{C_n\}_{n\in\mathbb{N}}$ computa uma função $f : \{0,1\}^* \to \{0,1\}$ se $f(x) = C_{|x|}(x)$, $\forall x \in \{0,1\}^*$.

[1] A notação $\{\}_{n\in\mathbb{N}}$ significa uma sequência $\{C_1, C_2, \ldots\}$

Adicionalmente, se uma família de circuitos $\{C_n\}_{n\in\mathbb{N}}$ computa a função característica de uma linguagem L qualquer, então dizemos que $\{C_n\}_{n\in\mathbb{N}}$ decide L.

2.3.1 Fórmulas booleanas

Uma *fórmula booleana* é um circuito onde todas as porta lógicas têm fan-out igual a 1. Todos circuitos booleanos podem ser convertidos para uma fórmula se substituirmos todas portas lógicas com fan-out maior do que um por um número suficiente de cópias dessas portas lógica com somente uma saída. E como fórmulas são um caso especial de circuitos temos que os dois modelos são equivalentes. Geralmente descrevemos fórmulas lógicas através de variáves, conectivos e parênteses para denotar a sequência correta de operações. Por exemplo, considere os seguintes operadores lógicos:

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	$\neg a$
0	1
1	0

O operador $\vee$ é chamado de OU, $\wedge$ é chamado de E e $\neg$ de NÃO. A base formada por $\vee, \wedge$ e $\neg$ é a mais "popular" no contexto de operação lógicas. Se x_1 e x_2 são variáveis então $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$ é um exemplo de fórmula lógica.

Alternativamente, podemos definir fórmulas logicas sobre a base $\{\vee, \wedge, \neg\}$ recursivamente da seguinte forma:

- Se x é uma variável então x é uma fórmula
- Se ϕ e ψ são fórmulas então também são $\phi \vee \psi$, $\phi \wedge \psi$ e $\neg\phi$.

A seguir nós vemos algumas forma normais de se representar fórmulas lógicas que vão ser bastante úteis para nós.

Definição 2.9. *(Forma normal conjuntiva (FNC))*

Uma fórmula lógica sobre as variáveis $x_1, \ldots, x_n$ é dita estar na forma normal conjuntiva (ou FNC) se ela é o E de OUs de variáveis em $\{x_1, \ldots, x_n\}$ ou as suas negações.

Ou seja, uma fórmula na FNC pode ser escrita como

$$\bigwedge_{i=1}^{m} c_i$$

Onde os $c_i = x_{i_1} \vee \cdots \vee x_{i_{k(i)}}$ são chamados de cláusulas e m é o número de claúsulas na fórmula.

Por exemplo, se ϕ é uma fórmula sobre as variáves x_1, x_2, x_3 e x_4, então

$$\phi = (x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_3) \wedge (\overline{x_2} \vee x_4)$$

está na forma normal conjuntiva.

Uma fórmula é dita ser uma k-FNC se ela estar na forma normal conjuntiva e cada cláusula estiver restrita a no máximo k literais.

Definição 2.10. *(Forma normal disjuntiva)*

Uma fórmula lógica sobre as variáveis $x_1, \ldots, x_n$ é dita estar na forma normal disjuntiva (ou FND) se ela é o OU de Es de variáveis em $\{x_1, \ldots, x_n\}$ ou as suas negações.

Os Es são chamados de termos. Se o número de termos na fórmula for m e $c_i = x_{i_1} \wedge \cdots \wedge x_{i_{k(i)}}$, $i \in [m]$[2], forem termos então uma fórmula na FND pode ser escrita como

$$\bigvee_{i=1}^{m} c_i$$

Por exemplo, a fórmula $\phi = (x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_4)$ está na forma normal disjuntiva.

Assim como k-FNCs, uma k-FND é uma fórmula na forma normal disjuntiva com a restrição que cada termo deva ter no máximo k literais.

Definição 2.11. *Um* mintermo *é o E de todas as variáves de uma fórmula ou suas negações. Por exemplo, $x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4$ é um mintermo sobre as variáveis x_1, x_2, x_3 e x_4.*

Considere a seguinte operação sobre uma variável booleana x:

$$x^b = \begin{cases} x & \text{caso } b = 1, \\ \overline{x} & \text{caso } b = 0. \end{cases}$$

Então podemos escrever um mintermo sobre as variáveis $x_1, \ldots, x_n$ de uma fórmula booleana como $x_i^{b_1} \wedge \cdots \wedge x_n^{b_n}$, onde cada b_i é 0 ou 1. Daí fica óbvio que um mintermo é verdadeiro se e somente se $x_i = b_i$, para cada $i \in [n]$. Se $x = (b_1, \ldots, b_n)$ é uma atribuição às variáveis então associamos o seguinte mintermo a esta atribuição:

$$\bigwedge_{i=1}^{n} x_i^{b_i}$$

[2] $[n]$ é simplesmente o conjunto de todos os números naturais menores ou igual a n

Então, para cada função booleana f com n variáveis, o n-FND onde cada termo é um mintermo associado às atribuições que satisfazem $f(x) = 1$ é uma fórmula que computa f, consequentemente todas funções booleanas podem ser computadas por um circuito booleano e todas linguagens em $\{0,1\}^*$ são decididas por alguma família de circuitos, incluindo a linguagem HALT que não é computável por máquinas de Turing convencionais.

Algumas vezes nós vamos chamar mintermos de monômios e podemos denotar eles por algo como $x_1^{\alpha_1} \dots x_n^{\alpha_n}$, onde $\alpha_i \in \{0,1\}$.

2.4 Complexidade computacional

Na seção anterior nós vimos que Turing nos deu uma definição formal do que nós intuitivamente pensamos ser computável. Porém, no mundo real, um problema ter um processo computacional finito que o resolva não é suficiente. Também queremos que a computação seja feita num tempo que seja útil para nós. O que foi observado é que o tempo de execução de algoritmos em computadores cresce a medida em que o tamanho da entrada também cresce. Pense no tamanho da entrada sendo medido como, por exemplo, o número de bits na representação binária de um número ou o número de vértices em um grafo. Como normalmente é desejável que um algoritmo seja eficiente para entradas de tamanho razoavelmente grande (em alguns casos o tamanho da entrada pode ser 10^6, por exemplo), nós queremos que a função de crescimento do algoritmo não cresça muito rápido — para que até para entradas de tamanho "razoavelmente grande" o número de passos necessários para realizar o algoritmo não seja muito grande. Portanto foi importante definir uma forma de medir a complexidade de algoritmos e também o que nos queremos dizer por uma "função eficiente"para o tempo de execução de um algoritmo.

Talvez o primeiro artigo que definiu uma medição de complexidade para problema computacionais foi [**?**], onde Hartmarnis e Stearns definiram que uma sequência binária α é computável em tempo T, onde T é uma função computável monotônica crescente de $\mathbb{N}$ para $\mathbb{N}$, se existe uma máquina de Turing que dá como saída o n-ésimo bit de α em menos do que $T(n)$ passos. Em [**?**], Edmonds propõe que devemos considerar funções polinomiais como sendo sinônimo de eficiência.

Nós dizemos que uma máquina de Turing M roda em tempo $T(n)$ se M, ao receber uma entrada de tamanho n, executa no máximo $T(n)$ passos (um passo da computação da máquina de Turing envolve escrever símbolos em suas fitas de trabalho, movimentar os cabeçotes de suas fitas e mudar o seu estado atual).

Nós vimos no teorema 2.2 que existe uma máquina de Turing que pode simular a execução de todas as outras máquinas de Turing sobre qualquer entrada. Também foi dito que a simulação é "eficiente" e que segundo Edmonds isto deveria significar que a simulação pode ser feito em tempo polinomial. E nós podemos verificar que o número de passos que $\mathcal{U}$ precisa para simular uma máquina de Turing M de tempo $T(n)$ é $\mathcal{O}(T(n)^2)$. Para provar isto temos que ver quanto passos $\mathcal{U}$ necessita

para simular um único passo de M. Em cada simulação de um passo, $\mathcal{U}$ visita cada "espaço" que representa uma fita de M, e como uma computação que executa menos do que $T(n)$ passos não pode usar mais do que $T(n)$ células de sua fita, temos que cada espaço contém no máximo $T(n)$ células. Então para simular um passo de M, $\mathcal{U}$ visita algo em torno de $kT(n)$ células de sua fita de trabalho, onde k é o número de fitas de M, e portanto o "slowdown" de simular M é apenas $\mathcal{O}(T(n))$.

Nós podemos fazer melhor do que $T(n)^2$, nós podemos simular uma máquina de Turing com "slowdown" logarítmico.

Teorema 2.12. *Existe uma máquina de Turing $\mathcal{U}^*$ que sobre a entrada (α, x), $\mathcal{U}^*$ dá como saída $M_\alpha(x)$. Além disso, se $T(|x|)$ é o tempo que M_α leva para executar sua computação sobre a entrada x, então $\mathcal{U}^*$ roda em tempo $\mathcal{O}(T(|x|)\log T(|x|))$ ao receber (α, x) em sua fita de entrada.*

Uma prova do teorema 2.12 pode ser encontrada em [**?**] (Teorema 1.9). O resultado foi originalmente proposto por Hennie e Stearns [**?**].

Construindo sobre o resultado acima nós podemos provar o seguinte resultado que nos será útil mais para frente.

Definição 2.13. *Uma máquina de Turing* oblivious *é uma máquina de Turing cuja o movimento de seus cabeçotes de fita só dependem do tamanho da entrada, e não no conteúdo das células e estado atual.*

Desta forma, a função de transição de uma máquina de Turing *oblivious* $A = \{\Gamma, Q, \delta\}$ de k fitas é $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1}$. Nós podemos imaginar que existe uma função $m : \mathbb{N} \to \{\{E, N, D\}^k\}^*$ tal que ao receber uma entrada x, os movimentos dos cabeçotes das fitas de A é dado por $m(|x|) = (z_1, \ldots, z_{T(|x|)})$, onde cada $z_i \in \{E, N, D\}^k$ representa os movimentos dos cabeçotes de fita de A no i-ésimo passo e $T(|x|)$ é o tempo em que A para ao receber entradas de tamanho $|x|$.

Teorema 2.14. *Se M é uma máquina de Turing que roda em tempo $T(n)$ para entradas de tamanho n, então existe uma máquina de Turing* oblivious *A de duas fitas que roda em tempo $\mathcal{O}(T(n)\log T(n))$ tal que $A(x) = M(x)$, para todo $x \in \{0,1\}^*$.*

Demonstração. Nós podemos modificar a MT $\mathcal{U}^*$ no teorema 2.12 de forma que ela seja uma máquina de Turing *oblivious* sem aumentar significativamente o seu tempo de execução. E além disso, $\mathcal{U}^*$ pode ser construida usando somente 2 fitas.

Então, A simplesmente executa a simulação de $\mathcal{U}^*$ sobre entradas $(\langle M\rangle, x)$, para qualquer $x \in \{0,1\}^*$.

□

2.4.1 Classes de complexidade

Uma das contribuições de Hartmanis e Stearns em [**?**] foi que eles mostraram como podemos agrupar problemas computacionais de acordo com o número de passos que um máquina de Turing necessita para resolve-los. Nesta seção iremos nos preocupar apenas com o tempo e espaço necessários para resolver problemas computacionais. Algumas classes de complexidade de tempo são definidas a seguir, e no fim desta seção iremos ver algumas classes de complexidade de espaço. Ao longo deste trabalho nós sempre vamos deixar implícito que todas as funções T que usamos para definir uma classe de complexidade é *tempo-construtivel*, o que significa que $T(n) \geq n$ e existe uma máquina de Turing que computa o valor $T(|x|)$ sobre a entrada x em menos do que $T(|x|)$ passos.

Definição 2.15. *Para uma função $T : \mathbb{N} \to \mathbb{N}$, nós definimos as seguintes classes de problemas:*

- $\mathsf{DTIME}(T(n))$*: a classe de todas linguagens L tal que existe uma máquina de Turing determinística M de tempo $T(n)$ que decide L.*
- $\mathsf{NTIME}(T(n))$*: a classe de todas linguagens L tal que existe uma máquina de Turing não-determinística N que decide L e que N executa no máximo $T(n)$ passos ao receber uma entrada de tamanho n, indepedente das escolhas das funções de transição que N faça.*
- $\mathsf{coDTIME}(T(n))$*: a classe de todas linguagens L tal que $\overline{L} \in \mathsf{DTIME}(T(n))$, onde $\overline{L}$ é o complemento da linguagem L (ou seja, $x \in L \iff x \notin \overline{L}$). Da mesma forma nós definimos* $\mathsf{coNTIME}(T(n))$.

As classes P e NP

Como já vimos, iremos usar tempo polinomial como sinônimo de eficiência. Uma linguagem L é decidida em tempo polinomial se existe um polinômio p tal que o tempo necessário para decidir a pertinência de uma string x em L é menor do que $p(|x|)$. A classe de linguagens decididas em tempo polinomial é chamada de P.

Definição 2.16 (A classe P)**.** *Uma linguagem L é dita estar em* P *se e somente se existe $c \geq 1$ tal que $L \in \mathsf{DTIME}(n^c)$.*

Um dos grandes objetivos de designers de algoritmos é provar que um determinado problema está em P pois então geralmente ele pode ser implementado eficientemente em um computador. Alguém poderia dizer que talvez exista um problema (natural) que esteja em P mas o tempo de execução do algoritmo para este problema é algo do tipo $10^{1000}n$ ou n^{1000}, o que com certeza não seria eficiente nem mesmo para $n = 2$. É verdade que um problema estar em P não implica necessariamente em ele poder ser resolvido eficientemente. Na verdade, nem mesmo a não existência de um algoritmo de tempo polinomial para um problema implica em ele não poder ser resolvido eficientemente na prática. Mas

usar a convenção de tempo polinomial = eficiência é conveniente quando estamos estudando classes de complexidade e a relação entre elas, por alguns motivos como por exemplo algumas modificações na definição de máquinas de Turing e até mesmo outros modelos computacionais mais realistas (como máquinas de acesso aleatório) não alteram a classe P, entre outros motivos.

Enquanto que P procura capturar linguagens que podem ser decididas eficientemente, a classe NP por sua vez procura capturar linguagens cuja suas instâncias sejam eficientemente verificáveis.

Definição 2.17 (A classe NP)**.** *Uma linguagem L está em NP se e somente se existe um polinômio p e uma máquina de Turing de tempo polinomial M tal que $x \in L \iff \exists u \in \{0,1\}^{p(|x|)} M(x,u) = 1$.*

Dizemos que u é um certificado da pertinência de x em L. Nós podemos definir NP de uma outra forma:

Definição 2.18. $\mathsf{NP} = \bigcup_{c \geq 1} \mathsf{NTIME}(n^c)$.

Para ver que as duas definições são equivalentes note que as escolhas da máquina de Turing não-determinística podem servir como um certificado, enquanto que uma máquina de Turing não-determinística poderia "adivinhar" um certificado para x.

A questão em aberto mais importante em complexidade computacional pergunta se as classes P e NP são iguais. Esse problema tem alguma importância histórica já que vários problemas que são importante em aplicações práticas que estão em NP não parecem, pelo que sabemos até agora, ter solução melhor do que tentar exaustivamente todas as possibilidades, a busca por uma solução para esses problemas melhor do que a busca exaustiva esteve no coração de algumas das primeiras pesquisas em complexidade computacional.

Definição 2.19 (A classe coNP)**.** $\mathsf{coNP} = \bigcup_{c \geq 1} \mathsf{coNTIME}(n^c)$.

Note que $\mathsf{P} = \mathsf{coP}$, já que um procedimento que decide eficientemente a pertinência de uma *string* em uma linguagem também pode ser usada para decidir a não pertinência (simplesmente inverta a saída), e portanto $\mathsf{P} = \mathsf{NP}$ implica em $\mathsf{NP} = \mathsf{coNP}$.

Reduções

Um dos principais conceitos em teoria da computação é o de uma *redução*. Uma redução é basicamente um procedimento que transforma uma instância de um problema A em uma instância de um outro problema B.

Definição 2.20 (Reduções)**.** *Uma redução de um problema $L \subseteq \{0,1\}^*$ para um problema $L' \subseteq \{0,1\}^*$ é uma função $f : \{0,1\}^* \to \{0,1\}^*$ tal que $x \in L \iff f(x) \in L'$, para todo $x \in \{0,1\}^*$.*

Além disso, L é dita ser redutível em tempo polinomial para L', o que denotamos por $L \leq_p L'$, se a redução f pode ser computada em tempo polinomial.

Reduções de tempo polinomial vão ser útil quando formos ver o próximo assunto. Se L é redutível em tempo polinomial para L', então um algoritmo de tempo polinomial para L' implica em um algoritmo de tempo polinomial para L, já que podemos usar a redução f para mapear uma string $x \in \{0,1\}^*$ em uma instâcia $f(x)$ de L' e depois usamos o algoritmo A de tempo polinomial que decida L' para computar $A(f(x))$. Se o tempo necessário para computar A sobre entradas de tamanho n for $p(n)$, onde p é um polinômio, e q for o tempo necessário para computar f então acabamos de mostrar que podemos decidir L em tempo menor do que $q(|x|) + p(q(|x|))$.

NP-completude e o teorema de Cook-Levin

Algumas linguagens em uma determinada classe de complexidade tem uma propriedade interessante em que elas capturam toda a dificuldade daquela classe. Um linguagem L é completa para uma classe sobre uma determinada "classe de reduções" $\leq_R$ (por exemplo, reduções em tempo polinomial como vimos na definição 2.20) se ela pertence à classe e todos os outros problemas dentro desta classe são redutíveis através de $\leq_R$ para L.

Definição 2.21 (NP-completude)**.** *Uma linguagem L é dita ser* NP*-difícil sse para todas linguagens $A \in$ NP, $A \leq_p L$.*

Se além de ser NP*-difícil L também está em* NP *então dizemos que L é* NP*-completa.*

Problemas NP-completos (que sejam naturais) existem, como foi provado por Stephen Cook e Leonid Levin, independentemente, no começo da década de 70. [**?**, **?**] O primeiro problema que foi provado ser NP-completo foi o problema da satisfazibilidade booleana.

Definição 2.22 (O problema da satisfazibilidade booleana)**.** *No problema da satisfazibilidade booleana, que chamaremos de* SAT*, é dado uma fórmula ϕ com variáveis $x_1, \ldots, x_n$ e queremos de decidir se existe uma atribuição $(x'_1, \ldots, x'_n)$ às variáves $x_1, \ldots, x_n$ tal que $\phi(x'_1, \ldots, x'_n) = 1$.*

Teorema 2.23 (Teorema de Cook-Levin)**.** SAT *é* NP*-completo.*

Apesar de poder ser um pouco longa, a prova do teorema 2.23 é bem simples de entender. Basicamente, nós temos que se A é uma linguagem em NP, então existe uma máquina de Turing M (que podemos assumir ter apenas uma fita) que aceita uma entrada $x \in \{0,1\}^n$ com um certificado $u \in \{0,1\}^{\text{poly}(n)}$ se e somente se $x \in A$ e u é um certificado da pertinência de x em A. A função $f : \{0,1\}^* \to \{0,1\}^*$ que transforma x em uma fórmula ϕ_x que é satisfazível se e somente se $x \in A$ faz o seguinte:

1. Se para todo $n > 0$ M roda em tempo menor do que $T(n)$ sobre entradas de tamanho n então f constroi um tableau $T(|x|) \times T(|x|)$ onde a i-ésima linha deste tableau guardará a configuração de M no seu i-ésimo passo.

2. A fórmula ϕ_x tem $T(|x|)^2$ variáveis que chamaremos de v_{ij}, $1 \leq i, j \leq T(|x|)$. O valor da variável v_{ij} é o conteúdo da célula na linha i e coluna j do tableau.

3. Pela computação de uma máquina de Turing ser local, o que significa dizer que o conteúdo de uma das células em um passo da computação depende somente do estado atual, da posição do cabeçote da fita e do conteúdo das duas células adjacente à ela, podemos construir uma fórmula booleana que decide o valor da variável v_{ij} em função das variáveis $v_{(i-1)(j-1)}, v_{(i-1)j}$ e $v_{(i-1)(j+1)}$. Esta fórmula depende somente da função de transição de M e portanto tem tamanho constante.

4. Precisamos assegurar algumas outras coisas, como por exemplo que a primeira linha do tableau é uma configuração inicial válida e que a última linha é uma configuração de aceitação (isto é, uma configuração onde o estado atual é q_{aceita}).

Pela natureza "repetitiva" da redução e pela fórmula ter tamanho polinomial ($\mathcal{O}(T(n)^2)$) podemos ver que ela pode ser feita em tempo polinomial. Finalmente, SAT $\in$ NP já que uma atribuição das variáveis que satisfazem uma fórmula pode servir como certificado.

Agora que nós temos um único problema que sabemos ser NP-completo, nós podemos provar que outros problemas são também NP-completo mostrando que SAT é redutível em tempo polinomial para eles. Isso segue pois a relação $\leq_p$ é transitiva. Por exemplo, podemos provar que a linguagem 3-SAT, que pergunta se uma fórmula na 3-FNC é satisfazível, é NP-completa. Em 1972, Richard Karp publicou [**?**] onde 21 problemas importantes foram provados serem NP-completos e deste então milhares de problemas que aparecem em aplicações práticas já foram provados serem NP-completos. O livro de Garey e Johnson é uma excelente referência para o fenômeno da NP-completude. [**?**]

Como já observamos antes, se existe uma linguagem NP-completa em P então P = NP, pois poderiamos usar a redução de tempo polinomial para L e depois o seu algoritmo de tempo polinomial para decidir qualquer outra linguagem em NP em tempo polinomial.

Hierarquia polinomial

Considere o seguinte problema em NP:

CLIQUE. *Dado um inteiro $k > 0$ e um grafo G, aceite se G tem um clique de tamanho maior ou igual a k.*

Podemos ver que CLIQUE está em NP pois um clique de tamanho maior ou igual a k em G é obviamente um certificado que G tem um clique de tamanho maior ou igual a k. Mas se ao invés de decidir se G tem um clique de tamanho pelo menos k nós queremos decidir se o maior clique em G tem tamanho k, como no problema MAX-CLIQUE:

MAX-CLIQUE. *Dado um inteiro $k > 0$ e um grafo G, aceite se o maior clique em G tem tamanho igual a k.*

Agora não fica tão óbvio para nós o que seria um certificado para MAX-CLIQUE. Além de termos que mostrar um clique de tamanho k, também devemos mostrar que nenhum subconjunto de tamanho maior do que k dos vértices de G formam um clique. Porém, adicionando um quantificador $\forall$ parece ser o suficiente para nós podermos capturar problemas como MAX-CLIQUE, o que a classe NP não parece conseguir fazer pelo o que nós sabemos até agora.

Definição 2.24 (A classe Σ_2^p)**.** *Uma linguagem L é dita estar em Σ_2^p se e somente se existe um polinômio p e uma máquina de Turing M que roda em tempo $p(n)$ tal que L pode ser escrita como:*

$$\textit{Para todo } x \in \{0,1\}^*,\ x \in L \iff \exists x_1 \in \{0,1\}^{p(|x|)} \forall x_2 \in \{0,1\}^{p(|x|)} M(x, x_1, x_2) = 1$$

Se as instâncias MAX-CLIQUE são da forma $\langle G, k\rangle$, onde G é um grafo e $k > 0$ um inteiro, então dizer $\langle G, k\rangle \in$ MAX-CLIQUE é o mesmo que dizer que *existe* um clique de tamanho k e que *todos* subconjuntos de tamanho maior do que k dos vértices de G não formam um clique.

Nós podemos ainda generalizar as classes NP e Σ_2^p, o que nós chamamos de hierarquia polinomial:

Definição 2.25 (Hierarquia polinomial)**.** *Para $k \geq 1$, uma linguagem L é dita estar em Σ_k^p se L pode ser expressa da seguinte forma:*

$$x \in L \iff \exists x_1 \in \{0,1\}^{p(|x|)} \forall x_2 \in \{0,1\}^{p(|x|)} \dots Q_k x_k \in \{0,1\}^{p(|x|)} M(x, x_1, x_2, \dots, x_k) = 1$$

Onde Q_k é $\exists$ se k é ímpar ou $\forall$ se k é par. M é uma máquina de Turing de tempo $p(n)$.

A hierarquia polinomial é $\mathsf{PH} = \bigcup_{k \geq 1} \Sigma_k^p$.

Note que $\mathsf{NP} = \Sigma_1^p$ e também podemos chamar P de Σ_0^p.

Assim como fizemos com NP, também podemos generilizar a classe coNP através de quantificadores alternantes. A diferença é que o primeiro quantificador é um $\forall$.

Definição 2.26. *Para todo $k \geq 1$ a classe $\mathsf{co}\Sigma_k^p$ consiste de todas as linguagens L que podem ser expressas como:*

$$x \in L \iff \forall x_1 \in \{0,1\}^{p(|x|)} \exists x_2 \in \{0,1\}^{p(|x|)} \dots Q_k x_k \in \{0,1\}^{p(|x|)} M(x, x_1, x_2, \dots, x_k)$$

Onde agora Q_k é $\forall$ se k é ímpar e $\exists$ caso contrário. E de novo, M é uma máquina de Turing de tempo $p(n)$. Para cada k, nós chamamos $\mathsf{co}\Sigma_k^p$ de Π_k^p.

E nós temos que $\mathsf{coNP} = \Pi_1^p$.

É fácil ver que para todo $k \geq 1$ nós temos as seguintes desigualdades:

$$\Sigma_k^p \subseteq \Pi_{k+1}^p \subseteq \Sigma_{k+2}^p$$

Portanto $\mathsf{PH} = \bigcup_{k \geq 1} \Pi_k^p$

Nós dizemos que a hierarquia polinomial *colapsa* se para algum k, $\mathsf{PH} = \Sigma_k^p$. Neste caso dizemos que a hierarquia polinomial colapsa para o seu k-ésimo level e também temos que $\Sigma_k^p = \Sigma_l^p$, para todo $l > k$.

Teorema 2.27. *Para todo $k \geq 1$, se $\Sigma_k^p = \Pi_k^p$ então $\mathsf{PH} = \Sigma_k^p$.*

Demonstração.

Assuma que para algum $k \geq 1$, $\Sigma_k^p = \Pi_k^p$.

Nós mostramos por indução que para todo $l > k$, $\Sigma_l^p = \Pi_l^p = \Sigma_k^p$. Para isso só precisamos mostrar que $\Sigma_l^p \subseteq \Sigma_k^p$, pois por nós termos assumido que $\Sigma_k^p = \Pi_k^p$, $\Sigma_l^p = \Sigma_k^p$ implica em Σ_l^p estar fechada sob complemento.

Para $l = k + 1$, nós temos que toda linguagem L em Σ_{k+1}^p pode ser expressa como:

$$x \in L \iff \exists x_1 \in \{0,1\}^{p(|x|)} \forall x_2 \in \{0,1\}^{p(|x|)} \dots Q_{k+1} x_{k+1} \{0,1\}^{p(|x|)} M(x, x_1, x_2, \dots, x_{k+1}) = 1 \tag{2.1}$$

para alguma polinômio p e máquina de Turing de tempo polinomial M.

Então considere a seguinte linguagem L':

$$(x, x_1) \in L' \iff \forall x_2 \in \{0,1\}^{p(|x|)} \dots Q_{k+1} x_{k+1} \{0,1\}^{p(|x|)} M(x, x_1, x_2, \dots, x_{k+1}) = 1$$

L' está em $\Pi_k^p = \Sigma_k^p$ portanto podemos reescrever L' como:

$$(x, x_1) \in L' \iff \exists y_1 \in \{0,1\}^{p(|x|)} \forall y_2 \in \{0,1\}^{p(|x|)} \dots Q_k x_k \in \{0,1\}^{p(|x|)} M'(x, x_1, y_1, y_2, \dots, y_k) = 1 \tag{2.2}$$

Então podemos trocar toda parte a partir do primeiro quantificador $\forall$ de 2.1 pelo lado direito de 2.2 e temos o seguinte:

$$x \in L \iff \exists x_1, y_1 \in \{0,1\}^{p(|x|)} \forall y_2 \in \{0,1\}^{p(|x|)} \dots Q_k y_k \in \{0,1\}^{p(|x|)} M'(x, x_1, y_1, y_2, \dots, y_k) = 1$$

Portanto $L \in \Sigma_k^p$.

Para provar para outros valores de $l > k+1$, nós provamos da mesma maneira mas assumindo que $\Sigma_{l-1}^p = \Pi_{l-1}^p$ que agora sabemos que são iguais a Σ_k^p.

□

Assim como vimos que a classe NP tem problemas completos, podemos provar que cada level da hierarquia tem seu próprio problema completo. Uma linguagem L é Σ_k^p-completa se e somente se $L \in \Sigma_k^p$ e para todo $L' \in \Sigma_k^p$, $L' \leq_p L$.

Cada level da hierarquia polinomial tem a sua própria versão do problema SAT.

Definição 2.28. *Para todo $k > 0$, a linguagem Σ_k^pSAT consiste de todas as fórmulas lógicas ϕ tal que:*

$$\exists u_1 \in \{0,1\}^{p(|\langle\phi\rangle|)} \forall u_2 \in \{0,1\}^{p(|\langle\phi\rangle|)} \exists \ldots Q_k u_k \in \{0,1\}^{p(|\langle\phi\rangle|)} \phi(u_1, u_2, \ldots, u_k)$$

é verdadeiro, onde Q_k é $\exists$ se k é ímpar e $\forall$ se k é par.

Da mesma forma, uma fórmula lógica está em Π_k^pSAT se e somente o seguinte predicato quantificado é verdadeiro:

$$\forall u_1 \in \{0,1\}^{p(|\langle\phi\rangle|)} \exists u_2 \in \{0,1\}^{p(|\langle\phi\rangle|)} \forall \ldots Q_k u_k \in \{0,1\}^{p(|\langle\phi\rangle|)} \phi(u_1, u_2, \ldots, u_k)$$

Onde Q_k é $\forall$ se k é ímpar e $\exists$ se k é par.

E como é de se esperar, cada Σ_k^pSAT é Σ_k^p-completo. A prova que Π_k^pSAT é Π_k^p-completo, para cada $k \geq 1$, é análoga.

Teorema 2.29. *Para todo $k \geq 1$, Σ_k^pSAT é Σ_k^p-completo.*

Demonstração. Para algum $k \geq 1$, seja L uma linguagem em Σ_k^p. Para toda string $x' \in \{0,1\}^*$, nós temos que mostrar que existe uma redução em tempo polinomial que transforma x' em uma instância $\phi_{x'}$ de Σ_k^pSAT tal que $\phi_{x'} \in \Sigma_k^p\mathsf{SAT} \iff x' \in L$. Sabemos que podemos expressar L como:

$$x \in L \iff \exists x_1 \in \{0,1\}^{p(|x|)} \forall x_2 \in \{0,1\}^{p(|x|)} \exists \ldots Q_k x_k \in \{0,1\}^{p(|x|)} M(x, x_1, x_2, \ldots, x_k) = 1$$

Usando a redução do teorema 2.23, nós podemos transformar a máquina de Turing M em uma fórmula ϕ tal que $\phi(x, x_1, x_2, \ldots, x_k) = 1 \iff M(x, x_1, x_2, \ldots, x_k) = 1$ em tempo polinomial. Para cada $x' \in \{0,1\}^*$ nós criamos a fórmula $\phi_{x'}(x_1, x_2, \ldots, x_k) = \phi(x', x_1, x_2, \ldots, x_k)$ e temos que

$$\exists x_1 \in \{0,1\}^{p(|x|)} \forall x_2 \in \{0,1\}^{p(|x|)} \exists \ldots Q_k x_k \in \{0,1\}^{p(|x|)} \phi_{x'}(x_1, x_2, \ldots, x_k)$$

é verdade se e somente se

$$\exists x_1 \in \{0,1\}^{p(|x|)} \forall x_2 \in \{0,1\}^{p(|x|)} \exists \ldots Q_k x_k \in \{0,1\}^{p(|x|)} M(x', x_1, x_2, \ldots, x_k)$$

também é verdade. Ou seja, $\phi_{x'} \in \Sigma_k^p\mathsf{SAT}$ se e somente se $x' \in L$, como queriamos mostrar.

□

Teoremas de hierarquia

Como foi dito na introdução, um dos principais desafios da complexidade computacional é demonstrar limites inferiores para problemas computacionais. Intuitivamente podemos acreditar que problemas intrinsecamente difíceis devem existir pois se dermos mais recursos para uma máquina de Turing computar deveriamos também sermos capaz de decidir mais linguagens. Os teoremas de hierarquia é uma série de teoremas que provam exatamente isso. Geralmente, iremos usar o fato que máquinas de Turing que usam mais tempo podem simular máquinas de Turing que usam menos tempo para montar uma máquina de Turing "diagonalizadora", e então mostramos que essa máquina deve discordar com todas as máquinas que usam menos tempo em pelo menos um ponto.

Teorema 2.30. *(Teorema da hierarquia de tempo determinístico [?])*

Para todas funções $f, g : \mathbb{N} \to \mathbb{N}$ tempo-construtíveis satisfazendo $g(n)\log g(n) = o(f(n))$, temos que $\mathsf{DTIME}(g(n)) \subsetneq \mathsf{DTIME}(f(n))$.

Demonstração. Seja $L \in \mathsf{DTIME}(g(n))$, note que pelo teorema 2.12 existe uma máquina de Turing que simula a execução de uma máquina de Turing que decide L em tempo $\mathcal{O}(g(n)\log g(n)) = o(f(n))$. Então temos que deve existir uma máquina de Turing $\mathcal{D}$ que funciona da seguinte maneira:

- Sobre entrada $\langle M \rangle$:
- Simule M sobre a entrada $\langle M \rangle$ por $g(|\langle M \rangle|)$ passos.
- Se em algum momento M aceita, rejeite a entrada; caso contrário, aceite.

Note que $\mathcal{D}$ roda em tempo $\mathcal{O}(g(n)\log g(n))$ e portanto $L(\mathcal{D}) \in \mathsf{DTIME}(f(n))$.

Nós afirmamos que $L(\mathcal{D}) \notin \mathsf{DTIME}(g(n))$. Assuma o contrário, que $L(\mathcal{D}) \in \mathsf{DTIME}(g(n))$ e chame de $\mathcal{D}'$ uma máquina de Turing que decida $L(\mathcal{D})$ em tempo $g(n)$. Então usamos o mesmo argumento que usamos para mostrar que HALT não é decidível para mostrar que $L(\mathcal{D}) \notin \mathsf{DTIME}(g(n))$ (ou equivalentemente, que tal máquina de Turing $\mathcal{D}'$ não pode existir): nós rodamos $\mathcal{D}'$ sobre a sua própria descrição e vemos o que acontece.

- Se $\mathcal{D}'(\langle \mathcal{D}' \rangle) = 1$ então $\mathcal{D}(\langle \mathcal{D}' \rangle) = 0$

- Se $\mathcal{D}'(\langle\mathcal{D}'\rangle) = 0$ então $\mathcal{D}(\langle\mathcal{D}'\rangle) = 1$

Nos dois itens acima nós temos que assumir que a descrição de $\mathcal{D}'$ seja grande o suficiente para que D possa simular a toda a tua execução sobre a entrada $\langle\mathcal{D}'\rangle$, mas isso não é problema já que D' possui descrições arbitrariamente grandes.

□

Agora nós vamos considerar o caso não-determinístico. Se tentarmos provar da mesma forma um teorema de hierarquia de tempo não-determinístico nós esbarrariamos no seguinte problema: não é óbvio o modo como podemos negar uma computação não-determinística, pois deveriamos ter um conhecimento "universal" sobre todas os ramos da computação. A única forma óbvia de fazer isso seria simular todos os ramos da computação, o que leva tempo exponencial. Porém, ainda pode-se usar essa simulação determinística para diagonalizar máquinas de Turing não-determinística, dado que a simulação seja feita numa entrada exponencialmente menor do que a entrada original. O truque é fazer que uma máquina "diagonalizadora" $\mathcal{D}$ ou discorde com uma máquina de Turing não-determinística M em alguma string unária de tamanho em um intervalo de comprimento exponencial ou que ela discorde nos extremos deste intervalo. Na prova do teorema abaixo nós usamos o fato que uma simulação não-deterministica pode ser feita com "slowdown" constante.

Teorema 2.31. *(Teorema da hierarquia de tempo não-determinístico [?])*

Sejam $f, g : \mathbb{N} \to \mathbb{N}$ funções tempo-construtíveis satisfazendo $g(n+1) = o(f(n))$, então $\mathsf{NTIME}(g(n)) \subsetneq \mathsf{NTIME}(f(n))$.

Demonstração. Nesta prova, $\{M_i\}_{i\in\mathbb{N}}$ representa uma enumeração de todas as máquinas de Turing não-determinísticas.

Considere a função h definida como $h(1) = 2$ e $h(i+1) = 2^{g(h(i)+1)}$, para $i > 1$. Nós construimos uma máquina de Turing não-determinística $\mathcal{D}$ que inicialmente assumismo concordar com M_i em todas as strings unárias 1^n satisfazendo $h(i) < n \leq h(i+1)$, daí ela diagonaliza na entrada $1^{h(i+1)}$ e a partir disso nós obtemos uma contradição. Como $h(i+1)$ é exponencialmente maior do que $h(i)+1$, $\mathcal{D}$ pode simular todos os ramos da computação de M_i sobre a entrada $1^{h(i)+1}$ deterministicamente. A máquina de Turing $\mathcal{D}$ é definida da seguinte maneira (qualquer entrada que não seja da forma 1^n para $n \in \mathbb{N}$ é imediatamente rejeitada):

1. Sobre a entrada 1^n, ache i tal que $h(i) < n \leq h(i+1)$.
2. Se $h(i) < n < h(i+1)$, $\mathcal{D}$ não-deterministicamente simula M_i sobre a entrada 1^{n+1} por $g(n+1)$ passos e aceita se e somente se M_i aceita.
3. Se $n = h(i+1)$, $\mathcal{D}$ deterministicamente simula M_i sobre a entrada $1^{h(i)+1}$ por $g(h(i)+1)$ passos e aceita se e somente se M_i rejeita.

Dessa forma, como estamos assumindo que $\mathcal{D}$ e M_i concordam em todas as entradas 1^n com $h(i) < n < h(i+1)$, nós temos que $M_i(1^{h(i)+1}) = \mathcal{B}(1^{h(i+1)})$, uma contradição.

□

Limites da diagonalização

Podemos nos perguntar se a estratégia de diagonalização usadas nas provas dos teoremas de hierarquia pode também nos dar limites inferiores mais interessantes. Podemos separar as classes P e NP usando diagonalização?

Vamos imaginar que nós temos uma prova que $\mathsf{P} \neq \mathsf{NP}$ que usa diagonalização da forma que usamos para provar o teoremas de hierarquia. Ou seja, nós temos uma máquina de Turing $\mathcal{D}$ construida de forma que ela difere de todas as máquina de Turing M "em P" em pelo menos um ponto, simulando a execução de M e depois invertendo a saída de M sobre alguma entrada. Como podemos também enumerar e simular máquinas de Turing com qualquer oráculo $\mathcal{O}$ da mesma forma que podemos enumerar e simular máquinas de Turing convencionais, ao adicionar o oráculo $\mathcal{O}$ à $\mathcal{D}$ nós podemos também separar $\mathcal{D}^{\mathcal{O}}$ de todas as outras máquinas de Turing em $\mathsf{P}^{\mathcal{O}}$ de forma análoga. Portanto uma prova que $\mathsf{P} \neq \mathsf{NP}$ que usa diagonalização deve também provar que $\mathsf{P}^{\mathcal{O}} \neq \mathsf{NP}^{\mathcal{O}}$. Porém, o objetivo desta seção é mostrar que existem oráculos $\mathcal{A}$ e $\mathcal{B}$ tal que $\mathsf{P}^{\mathcal{A}} = \mathsf{NP}^{\mathcal{A}}$ e $\mathsf{P}^{\mathcal{B}} \neq \mathsf{NP}^{\mathcal{B}}$.

Teorema 2.32. *Existem oráculos $\mathcal{A}$ e $\mathcal{B}$ tais que:*

1. $\mathsf{P}^{\mathcal{A}} = \mathsf{NP}^{\mathcal{A}}$.
2. $\mathsf{P}^{\mathcal{B}} \neq \mathsf{NP}^{\mathcal{B}}$.

Demonstração. Para provar 1 nós fazemos $\mathcal{A}$ ser um oráculo para a linguagem SAT. SAT obviamente está em $\mathsf{P}^{\mathsf{SAT}}$, portanto podemos reduzir qualquer instância de uma linguagem $L \in \mathsf{NP}^{SAT}$ para uma instância de SAT em tempo polinomial, colocando L em $\mathsf{P}^{\mathsf{SAT}}$.

O oráculo B nós vamos contruí-lo de forma que nenhuma máquina de Turing que executa menos do que $2^n/10$ passos possa decidir a linguagem $L_{\mathcal{B}}$ definida como

$$L_{\mathcal{B}} = \{x | \exists y \in \mathcal{B} \text{ satisfazendo } |y| = |x|\}$$

Note que $L_{\mathcal{B}} \in \mathsf{NP}^{\mathcal{B}}$.

Nós contruimos $\mathcal{B}$ em estágios onde no i-ésimo estágio nós apenas decidimos a pertinência de um número finito de strings em $\mathcal{B}$. Daí, seja n o menor inteiro tal que nenhuma string de tamanho pelo menos n esteja em $\mathcal{B}$ no começo do estágio i, então consideramos a execução de M_i sobre a entrada 1^n e para cada consulta ao oráculo $\mathcal{B}$ à pertinência de uma string de tamanho pelo menos n, declaramos x como não estando em $\mathcal{B}$. Se M_i para em menos do que $2^n/10$ passos sobre a entrada 1^n nós temos o seguinte:

1. M_i aceita $1^n \rightarrow$ declare cada string de tamanho n como não estando em $\mathcal{B}$.
2. M_i rejeita $1^n \rightarrow$ declare alguma das (pelo menos) $2^n - 2^n/10$ strings de tamanho n que não foram consultadas como estando em $\mathcal{B}$.

□

2.5 Complexidade de circuitos

Agora discutimos complexidade de circuitos.

O *tamanho* de um circuito booleano C é o número de portas lógicas que o compõem e a sua profundidade é o maior caminho de uma das variáveis de entrada até a saída de C. Denotamos o tamanho de C por $|C|$.

Definição 2.33. *(Medição de complexidade de circuitos)*

Para funções $f, g : \mathbb{N} \rightarrow \mathbb{N}$ dizemos que uma linguagem L é dita estar em $\mathsf{SIZE}(f(n))$ se existe uma família de circuitos $\{C_n\}_{n\in\mathbb{N}}$ tal que $|C_n| \leq f(n)$, para todos n, e que L é dita estar em $\mathsf{DEPTH}(g(n))$ se a profundidade de C_n é menor ou igual a $g(n)$, para todos n.

Uma observação quanto a representação de circuitos booleanos por strings. Como um circuito é um grafo não direcionado acíclico, nós podemos representá-lo pela sua lista de adjacência. A lista de adjacência de um circuito de tamanho S tem S linhas onde cada linha irá conter:

- O label da porta lógica.
- O índice das portas lógicas que alimentam a entrada desta porta lógica.

Se assumirmos que o fan-in de todas portas lógicas é limitado por uma constante então precisamos de $c \log S$ bits para formar uma das linhas da lista de adjacência de C, para alguma constante c, portanto ao todo precisamos de $\mathcal{O}(S \log S)$ bits para representar um circuito pela sua lista de adjacência.

Algumas vezes nós vamos querer usa máquinas de Turing para simular a computação de um circuito C sobre a entrada x. Considere o seguinte problema:

Definição 2.34 (Problema da avaliação de circuito)**.** *O problema da avaliação de circuito, que nós chamaremos de* CIRCUIT-EVAL*, consiste de todos pares $\langle C, x\rangle$ onde C é um circuito booleano e x uma string binária tal que $C(x) = 1$.*

Podemos verificar que existe uma máquina de Turing que decide CIRCUIT-EVAL em tempo linear usando basicamente o procedimento descrito na definição 2.7 em que avaliamos o valor de cada porta lógica de C seguindo um ordenamento topológico.

Circuitos de tamanho polinomial

Assim como fizemos na seção anterior, nós queremos definir uma classe que procura capturar todas as linguagens que são decididas por circuitos "pequenos". Assim como fizemos com máquinas de Turing, nós usamos complexidade polinomial como sinônimo de eficiência.

Definição 2.35 ($\mathsf{P/poly}$)**.** *Uma linguagem L é dita estar em* $\mathsf{P/poly}$ *se e somente se existe $c > 0$ tal que $L \in \mathsf{SIZE}(n^c)$.*

Ou seja, $L \in \mathsf{P/poly}$ se e somente se existe uma família de circuitos $\{C_n\}_{n\in\mathbb{N}}$ que decide L onde todos os circuitos em $\{C_n\}_{n\in\mathbb{N}}$ têm tamanho polinomial.

Nós sabemos que vários problemas úteis têm circuitos pequenos, como adição e outras operações aritméticas. Na verdade, todos problemas que podem ser resolvidos eficientemente por uma máquina de Turing tem um família de circuito de tamanho polinomial como mostra o teorema a seguir.

Teorema 2.36. $\mathsf{P} \subseteq \mathsf{P/poly}$.

Podemos provar este teorema mostrando que para todas máquinas de Turing que param em menos do que $T(n)$ passos existe um circuito de tamanho $\mathcal{O}(T(n)\log T(n))$.

Demonstração. Seja L uma linguagem em $\mathsf{DTIME}(T(n))$. Nós sabemos que existe uma máquina de Turing oblivious $A = (\Gamma, Q, \delta)$ de duas fitas que computa L em menos do que $T'(n) = T(n)\log T(n)$ passos. Nós podemos usar A para construir circuitos C_n de tamanho $\mathcal{O}(T'(n))$ que decidem L restrita à strings de tamanho n, para todos $n > 0$. C_n tem $T'(n)$ níveis em que cada nível tem um número constante de portas lógicas e portanto $|C_n| = \mathcal{O}(T'(n)) = \mathcal{O}(T(n)\log T(n))$.

Cada nível de C_n é composto por um subcircuito que computa a função de transição de A. Chamaremos este subcircuito de C_δ e nós escreveremos C_δ^i quando queremos especificar o subcircuito que se encontra no i-ésimo nível. O que nós queremos é que o i-ésimo nível de C_n compute a configuração que A entra no $(i+1)$-ésimo passo. As entradas do subcircuito C_δ é dividida em três partes que representam o estado atual e os símbolos sendo lidos pelos dois cabeçotes de A, no total a entrada tem $\log|Q| + 3$ bits.[3] As saídas são particionada em duas partes que representam o estado de A no próximo passo e o símbolo escrito na fita de trabalho. O estado na entrada de C_δ^i é o estado na saída de C_δ^{i-1}, o símbolo na fita de entrada de C_δ^i vem de uma das entradas de C_n e o símbolo na fita de trabalho vem da saída de C_δ^j, onde j é o maior número menor do que i tal que no passo j o cabeçote da fita de trabalho esteve na mesma posição em que ele se encontra no passo i. A entrada de C_δ^1 é o estado inicial de A e caso o "j" não exista então o símbolo sendo lido na fita de trabalho é $\square$, ambos podem ser implementados usando as constantes '1' e '0' usando um número constante de portas lógicas. Como C_δ depende somente da função de transição de A temos que $|C_\delta| = \mathcal{O}(1)$.

$\square$

[3]Para representar o símbolo sendo lido na fita de entrada precisamos de somente um bit, para representar o símbolo sendo lido pela fita de trabalho precisamos de um bit adicional porque este símbolo pode ser $\square$.

O problema CIRCUIT-SAT

No problema CIRCUIT-SAT é dado um circuito C com n entradas e queremos decidir se existe $x \in \{0,1\}^n$ tal que $C(x) = 1$. Dizendo de outra maneira, queremos decidir se C computa ou não a função constante $C(x) = 0$. A seguir nós vemos que CIRCUIT-SAT é NP-completo e portanto ainda não se sabe se existe um algoritmo eficiente para computá-lo.

Teorema 2.37. CIRCUIT-SAT *é* NP*-completo.*

Demonstração. CIRCUIT-SAT está em NP pois uma atribuição às variáveis de entrada de C servem como certificado.

Se $L \in$ NP então existe uma máquina de Turing oblivious M de tempo polinomial e polinômio p tal que $M(x,u) = 1$ sempre que $x \in L$ e $u \in \{0,1\}^{p(|x|)}$ é um certificado para x. Daí, pela construção na prova do teorema 2.36 nós podemos construir um circuito C que computa $M_x(u) = M(x,u)$. C é satisfazível se e somente se existe u tal que $M(x,u) = 1$, que é o mesmo que dizer que $x \in L$.

A redução pode ser feita em tempo polinomial pois para construir cada nível do circuito nós precisamos somente da descrição de C_δ e a posição dos cabeçotes nas duas fitas de M_x. Para decidir a posição dos cabeçotes de M_x no passo i em tempo polinomial nós simplesmente simulamos M_x sobre uma entrada de tamanho $p(|x|)$ por i passos.

□

Famílias de circuitos uniforme

Famílias de circuitos como vimos até agora são um modelo de computação meio que irrealista. Tome por exemplo a linguagem HALT. Não só existe uma família de circuito que decide HALT mas também temos que HALT está em P/poly se modificarmos ela um pouco. É só considerar a redução de HALT para a linguagem unária $\{1^n |$ a n-ésima string binária na ordem lexicográfica pertence a HALT$\}$, que chamamos de UHALT. Todas linguagens unárias $\mathcal{U}$ são decididas por uma família de circuito de tamanho polinomial: para cada n tal que $1^n \notin \mathcal{U}$, C_n é simplesmente o circuito que computa a função constante 0, e se $1^n \in \mathcal{U}$ então C_n computa o mintermo $x_1 \wedge \cdots \wedge x_n$. Portanto UHALT $\in$ P/poly mesmo ela sendo claramente indecidível (isto é, não existe nenhuma máquina de Turing que a decida).

Para contornar este problema podemos exigir que cada circuito de uma família de circuitos seja computável de alguma forma. Além disso, podemos também exigir que eles sejam eficientemente computáveis como veremos a seguir.

Definição 2.38. *(Família de circuitos* P*-uniforme)*

Uma família de circuitos $\{C_n\}_{n\in\mathbb{N}}$ é dita ser P*-uniforme sse existe uma máquina de Turing A de tempo polinomial tal que sobre a entrada 1^n, A dá como saída o circuito C_n.*

E nós podemos provar que a classe de linguagens decididas por famílias de circuitos P-uniforme coincide com P.

Teorema 2.39. *Uma linguagem L é computável por uma família de circuitos* P*-uniforme sse* $L \in \mathsf{P}$.

Demonstração.

- L é computável por uma família de circuitos P-uniforme $\Rightarrow L \in \mathsf{P}$:

  Seja A a máquina de turing que computa C_n a partir de 1^n. Nós contruimos uma máquina de turing B que sobre uma entrada x, simula A sobre a entrada $1^{|x|}$ para obter $C_{|x|}$. Daí podemos computar $C_{|x|}(x)$ em tempo polinomial.

- $L \in \mathsf{P} \Rightarrow L$ é computável por uma família de circuitos P-uniforme:

  Como $L \in \mathsf{P}$ temos que existe uma máquina de Turing oblivious M que computa L em tempo $p(n) \log p(n) = \mathsf{poly}(p(n))$, para algum polinômio p. Usando a redução na prova do teorema 2.36, que sabemos que roda em tempo polinomial, podemos construir C_n a partir de 1^n.

□

Nós podemos melhorar o resultado no teorema 2.39 restrigindo ainda mais o recurso usado para computar C_n:

Definição 2.40 (Família de circuitos logspace-uniforme)**.** *Uma família de circuitos é dita ser logspace-uniforme sse existe uma máquina de Turing que computa C_n a partir de 1^n usando somente $\mathcal{O}(\log n)$ espaços de memória.*

Teorema 2.41. *Uma linguagem L é computada por uma família de circuitos logspace-uniforme sse* $L \in \mathsf{P}$.

Demonstração.

A ida é consequência de toda computação que usa espaço logaritmo poder ser feita em tempo polinomial.

Para provar a volta vamos mostrar que a redução na prova do teorema 2.36 pode ser feita em espaço logaritmo. Seja L uma linguagem em P e A uma máquina de Turing oblivious de duas fitas que decide L em menos do que $p(n)$ passos, onde p é um polinômio. Nós queremos mostrar que existe uma máquina de Turing M que usa não mais do que $\mathcal{O}(\log n)$ espaços de memória que dado 1^n, M escreve em sua fita de saída a descrição do circuito C_n usando a construção na prova do teorema 2.36. Para construir C_n a partir de 1^n nós precisamos calcular $p(n)$, a descrição do circuito C_δ (δ é a função de transição de A) e também deve ser possível, dado um inteiro $i \leq p(n)$, calcular as células em que os dois cabeçotes de A se encontram no i-ésimo passo. Calcular $p(n)$ pode ser feito em espaço

logaritmo e calcular a descrição de C_δ pode ser feito em tempo constante. Para decidir as posições do cabeçotes em um dado passo, nós simulamos A sem nos preocuparmos com o estado e os símbolos lidos para que não precisamos ter que guardar o conteúdo das fitas de A em algum lugar. Podemos ignorar o contéudo das fitas e o estado atual porque os cabeçotes se movimentam em função apenas do tamanho da entrada. Dessa forma só precisamos manter três contadores que contam o número de passos executados e as posições dos cabeçotes das duas fitas de A.

□

Teorema de Karp-Lipton

Uma das maiores motivações para o estudo de complexidade de circuitos é a perspectiva de podermos separar P e NP usando resultados nesta área. Nós já vimos que $\mathsf{P} \subseteq \mathsf{P/poly}$, portanto se provarmos que $\mathsf{NP} \not\subseteq \mathsf{P/poly}$ nós teremos provado que $\mathsf{P} \neq \mathsf{NP}$. Nós iremos ver agora que $\mathsf{NP} \subseteq \mathsf{P/poly}$ implica no colapso da hierarquia polinomial.

Lema 2.42. *Se* $\mathsf{SAT} \in \mathsf{P/poly}$ *então existe um polinômio* p *tal que para todo* $n > 0$ *existe uma máquina de Turing* M_n *que ao receber a descrição de uma fórmula* ϕ *satisfazível de tamanho* n*,* M_n *retorna um* y *tal que* $\phi(y) = 1$ *em menos do que* $p(n)$ *passos.*

Demonstração.

Seja $\{C_n\}_{n\in\mathbb{N}}$ uma família de circuitos de tamanho polinomial que decida SAT.

A máquina M_n tem a capacidade de escrever em uma das suas fitas de trabalho a descrição de C_k, $1 \leq k \leq n$. Ao receber $\langle\phi\rangle$ em sua fita de entrada, M_n primeiro verifica se $|\langle\phi\rangle| = n$, e se não for o caso que $|\langle\phi\rangle| = n$, M_n imediatemente rejeita a entrada.

Se $|\langle\phi\rangle| = n$ então primeiro verificamos se ϕ é satisfazível usando o circuito C_n, rejeitando a entrada $\langle\phi\rangle$ caso não seja. Daí, nós podemos achar uma atribuição $y \in \{0,1\}^m$ tal que $\phi(y) = 1$, m é o número de entradas de ϕ, decidindo sequencialmente o valor de cada variável de ϕ na atribuição y.

Primeiro faça $x_1 = 1$ e verifique se ϕ é satisfazível quando trocamos cada aparição de x_1 em ϕ pela constante 1. Se sim, então fixe $x_1 = 1$ e se não fixe $x_1 = 0$. Uma das duas restrições ($x_1 = 1$ ou $x_1 = 0$) deve manter ϕ satisfazível. Restringir o valor de algumas variáveis de ϕ resulta em uma fórmula menor, e por M_n poder escrever em sua fita de trabalho o circuito C_k, para todo $k \leq n$, podemos usar um destes circuitos para decidir a satisfazibilidade da fórmula restrita. Nós podemos continuar usando o mesmo procedimento que usamos para decidir o valor de x_1 para todas as outras variáveis, sempre diminuindo o tamanho do circuito. No fim nós teremos obtido a atribuição y.

Para ver que M_n roda em tempo menor do que $p(n)$, para algum polinômio p, nós simplesmente notamos que a operação mais custosa na computação de M_n é escrever a descrição de C_k, $1 \leq k \leq n$, leva tempo polinomial pois C_k tem tamanho polinomial.

□

Teorema 2.43 (Teorema de Karp-Lipton [?])**.** $\mathsf{NP} \subseteq \mathsf{P/poly} \implies \mathsf{PH} = \Sigma_2^p$.

Demonstração.

Assuma que $\mathsf{NP} \subseteq \mathsf{P/poly}$.

Pelo teorema 2.27 sabemos que para mostrar que $\mathsf{PH} = \Sigma_2^p$, é suficiente mostrar que $\Pi_2^p \subseteq \Sigma_2^p$. E para isso nós mostramos que a linguagem Π_2^p-completa $\Pi_2^p\mathsf{SAT}$ está em Σ_2^p.

Lembrando que a linguagem $\Pi_2^p\mathsf{SAT}$ contém todas fórmulas ϕ tal que

$$\forall x \in \{0,1\}^n \ \exists y \in \{0,1\}^n \phi(x,y) \tag{2.3}$$

é verdadeira. Ou seja, para cada $x \in \{0,1\}^n$, fixar o primeiro parâmetro de ϕ mantém a fórmula ϕ satisfazível. Denotamos $\phi(x,.)$ por $\phi_x(.)$ e dizer que $\phi \in \Pi_2^p\mathsf{SAT}$ é equivalente a dizer que para todos $x \in \{0,1\}^n$, ϕ_x é satisfazível. Isso é o mesmo que dizer que para um circuito C que computa atribuições que satisfazem fórmulas satisfazíveis:

$$\forall x \phi_x(C(\langle\phi\rangle, x)) = 1$$

Já que $\mathsf{NP} \subseteq \mathsf{P/poly}$ implica em SAT ter circuitos de tamanho polinomial, pelo lema 2.42 temos que tal circuito C deve ter tamanho polinomial, então o seguinte é verdadeiro se assumirmos que ϕ_x é satisfazível para todo x (ou seja, $\phi \in \Pi_2^p\mathsf{SAT}$):

$$\exists C' \in \{0,1\}^{p(n)} \forall x \in \{0,1\}^n \phi_x(C'(\langle\phi\rangle, x)) = 1 \tag{2.4}$$

Onde p é um polinômio.

Se 2.3 é verdadeiro então o circuito C irá, para todas strings $x \in \{0,1\}^n$, apresentar em tua saída um $y \in \{0,1\}^n$ tal que $\phi_x(x,y) = 1$ e portanto 2.4 também é verdadeiro.

E por outro lado, se 2.4 é verdadeiro então para todo $x \in \{0,1\}^n$, ϕ_x é satisfazível e portanto 2.3 também é verdadeiro.

Ou seja, 2.4 é uma formulação equivalente do problema $\Pi_2^p\mathsf{SAT}$ que pode ser visto como uma linguagem em Σ_2^p.

□

Limites inferior e superior no tamanho de circuitos

Nós já vimos que usando a base $\Omega = \{\vee, \wedge, \neg\}$ nós podemos implementar qualquer função booleana de n entradas usando circuitos de tamanho $\mathcal{O}(n2^n)$ através de mintermos. E também acabamos de ver que acredita-se que algumas linguagens em NP exigem circuitos de tamanho superpolinomial.

Agora nós vamos ver que nem todas funções podem ser computadas por circuitos de tamanho polinomial. E também veremos um limite superior melhor do que $\mathcal{O}(n2^n)$. Na verdade, vamos ver que ambos o limites difereciam entre si por um fator constante.

O limite inferior foi descoberto por Shannon em dos primeiros resultados em complexidade de circuitos. Em 1949, Shannon estava interessado no problema de minimização de circuitos quando ele publicou o resultado que nos vamos ver em seguida em [**?**], a prova que eu apresento aqui foi tirada de [**?**].

Teorema 2.44. *Para todo $n > 1$, existem uma constante d e funções booleanas $f : \{0,1\}^n \to \{0,1\}$ tal que f não é computada por circuitos de tamanho menor do que $\frac{2^n}{dn}$.*

Demonstração. Nós vimos que todos circuitos de tamanho S podem ser representados através de sua lista de adjacência usando no máximo $cS \log S$ bits, para alguma constante $c > 1$. Então, o número de circuitos de tamanho no máximo S é menor ou igual ao número de strings de tamanho $cS \log S$ que é igual a $2^{cS \log S}$. Por outro lado, o número de funções booleanas de n entradas é 2^{2^n}. Fazendo $S = \frac{2^n}{dn}$, para alguma constante $d > c$, nós temos o seguinte:

$$\begin{aligned} 2^{cS \log S} &= 2^{c\frac{2^n}{dn} \log(\frac{2^n}{dn})} \\ &= 2^{\frac{c2^n}{dn}(n - \log dn)} \\ &< 2^{\frac{2^n cn}{dn}} \\ &< 2^{2^n} \end{aligned}$$

O número de funções booleanas de n entradas que têm circuitos de tamanho $\frac{2^n}{dn}$ é menor do que do que o número total de funções booleanas de n entradas. Portanto deve existir funções booleanas com n entradas que exigem circuitos com no mínimo $\frac{2^n}{dn}$ portas lógicas.

□

Nós também podemos notar que $\frac{2^{\frac{c}{d}2^n}}{2^{2^n}} = 2^{2^n(\frac{c}{d}-1)} = 2^{-\Theta(2^n)}$, o que significa que a fração de funções booleanas que têm circuitos de tamanho menor do que $\frac{2^n}{dn}$ é bem pequena.

Esse limite inferior é bem preciso na verdade. Em [**?**] Lupanov apresenta uma representeção de funções booleanas que nos levam a um limite superior de $(1 + o(1))\frac{2^n}{n}$.

Definição 2.45 (Representação de Lupanov)**.** *Uma (k, s)-representação de Lupanov de uma fórmula booleana é descrita a seguir.*

- *Particione as n variáveis de f em duas partes: $x_1, \ldots, x_k$ e $x_{k+1}, \ldots, x_n$*

- *Construa uma matrix* $2^k \times 2^{n-k}$ *onde as linhas da matriz são indexadas pelas atribuições às* k *primeiras variáveis de* f *e as colunas são indexadas pelas atribuições às* $n-k$ *últimas variáveis. A entrada* (x, y), $x \in \{0,1\}^k$ *e* $y \in \{0,1\}^{n-k}$, *desta matriz é* $f(x,y)$.

- *Seja* $p = \lceil \frac{2^k}{s} \rceil$, *temos* p *conjuntos* A_i, $1 \leq i \leq p$, *onde* A_i *contém as* s *linhas* $(i-1)s$ *até* $is-1$ *da matriz que nós construimos.* A_p *pode ter menos do que* s *linhas caso* s *não seja um divisor de* 2^k.

Daí nós definimos para cada $1 \leq i \leq p$ *e* $w \in \{0,1\}^s$ *as seguintes funções:*

$$g_{iw}(x_1, \ldots, x_k) = \begin{cases} 1 & \text{se } x_1, \ldots, x_k \text{ indexa a } j\text{-ésima linha de } A_i \text{ e } w_j = 1 \\ 0 & \text{caso contrário} \end{cases}$$

O "j" pode ser qualquer inteiro menor do que $|A_i|$.

$$h_{iw}(x_1, \ldots, x_{n-k}) = \begin{cases} 1 & \text{se a coluna de } A_i \text{ indexada por } x_1, \ldots, x_{n-k} \text{ for } w \\ 0 & \text{caso contrário} \end{cases}$$

Então podemos escrever $f(x_1, \ldots, x_n)$ *como*

$$\bigvee_{i=1}^{p} \bigvee_{s \in \{0,1\}^s} (g_{iw}(x_1, \ldots, x_k) \wedge h_{iw}(x_{k+1}, \ldots, x_n)) \tag{2.5}$$

Vamos nos convencer que f realmente pode ser escrita como 2.5. Particione $x \in \{0,1\}^n$ de forma que $x^1 = x_1 \ldots x_k$ e $x^2 = x_{k+1} \ldots x_n$, e suponha que x^1 indexe a j-ésima linha de A_i. Suponha que $f(x_1, \ldots, x_n) = 1$, o que implica na entrada (x^1, x^2) da tabela ser 1. Seja w a string que representa a coluna indexada por x^2 em A_i. Temos que $g_{iw}(x^1) = 1$ pois $(x^1, x^2) = w_j = 1$ e também $h_{iw}(x^2) = 1$ pela forma como escolhemos w, e portanto 2.5 é verdadeira também.

Indo na outra direção, se 2.5 é verdadeira então existe um w tal que $g_{iw}(x^1) = 1$ e $h_{iw}(x^2) = 1$, o que significa que uma string que representa a coluna indexada por x^2 tem um 1 na j-ésima linha de A_i, a linha indexada por x^1, e portanto a entrada (x^1, x^2) da tabela é 1 e consequentemente $f(x) = 1$.

Lema 2.46. *Para* $n > 1$, *podemos computar todos os monômios* $x_1^{\alpha_1} \ldots x_n^{\alpha_n}$ *usando* $\mathcal{O}(2^n)$ *portas lógicas.*

Demonstração.

Provamos por indução que podemos computar todos monômios de n variáveis usando $2^{n+1} + n - 5$ portas lógicas.

Para computar $x_1^{\alpha_1}x_2^{\alpha_2}$, $(\alpha_1,\alpha_2)\in\{0,1\}^2$, precisamos usar apenas 4 portas $\wedge$ e 1 porta $\neg$ somando ao todo $5 = 2^{2+1}+2-5$ portas lógicas.

Assumimos agora que para todo $2 < k < n$ é verdade que podemos computar todos monômios $x_1^{\alpha_1}\dots x_k^{\alpha_k}$ usando $2^{k+1}+k-5$ portas lógicas. Então em particular, podemos calcular todos monômios de $n-1$ variáveis usando $2^n+(n-1)-5$ portas lógicas. Para computar os monômios de n variáveis nós podemos usar os circuitos que computam os monômios de $n-1$ variáveis e ligamos x_n e $\overline{x_n}$ a cada uma das 2^{n-1} saídas dos monômios de $n-1$ variáveis adicionando 2^n portas $\wedge$ e uma porta $\neg$. Portanto ao todo nós usamos $(2^n+(n-1)-5)+2^n+1 = 2^{n+1}+n-5$ portas lógicas.

□

Agora mostramos como usamos a (k, s)-representação de Lupanov para alcançar um limite superior próximo do limite inferior de Shannon. A prova usada aqui pode ser encontrada em [**?**] e [**?**]

Teorema 2.47. *Para todas as funções booleanas f, $f \in$ SIZE$((1+o(1))\frac{2^n}{n})$.*

Demonstração. Vamos analizar o número de portas lógicas que precisamos para representar uma função booleana em sua (k, s)-representação de Lupanov com $k = \lceil 3\log n\rceil$ e $s = \lceil n - 5\log n\rceil$.

Primeiro nós computamos todos os monômios $x_1^{\alpha_1}\dots x_k^{\alpha_k}$ e $x_{k+1}^{\alpha_{k+1}}\dots x_n^{\alpha_n}$, o que nós já vimos em 2.46 que podemos fazer usando $\mathcal{O}(2^k+2^{n-k})$ portas lógicas. Substituindo k por $\lceil 3\log n\rceil$:

$$\mathcal{O}(2^{\lceil 3\log n\rceil}+2^{n-\lceil 3\log n\rceil}) = \mathcal{O}(n^3+\frac{2^n}{n^3}) = \mathcal{O}(\frac{2^n}{n^3})$$

Agora nós podemos computar cada g_{iw} e h_{iw} usando suas representações por $\vee$s de mintermos. Como todos os mintermos já foram previamente computados nós só precisamos adicionar portas $\vee$ para implementar essas funções.

O número de atribuições às variáveis $x_1,\dots,x_k$ que satisfazem g_{iw} é igual ao número de 1s em w então precisamos em média de $s/2$ portas $\vee$ para implementar cada g_{iw}. Então o número total de portas $\vee$ necessárias para implentar todos os g_{iw} é $sp2^{s-1}$. Como $p = \lceil\frac{2^k}{s}\rceil < (\frac{2^k}{s}+1)$:

$$\begin{aligned} sp2^{s-1} &= s\lceil\frac{2^k}{s}\rceil 2^{s-1} \\ &< s(\frac{2^k}{s}+1)2^{s-1} \\ &= 2^{k+s-1}+s2^{s-1} \end{aligned}$$

Substituindo $k = \lceil 3\log n\rceil$ e $s = \lceil n-5\log n\rceil$ isso vira:

$$2^{\lceil 3\log n\rceil+\lceil n-5\log n\rceil-1}+(\lceil n-5\log n\rceil)2^{\lceil n-5\log n\rceil-1}<\frac{2^{n+1}}{n^2}+(n-5\log n+1)\frac{2^n}{n^5}=\mathcal{O}(\frac{2^n}{n^2})$$

Cada atribuição de valores às variáveis $x_{k+1},\ldots,x_n$ satisfaz exatamente p funções h_{iw} e portanto o número de portas $\vee$ necessárias para implementar todas as h_{iw} é $p2^{n-k}$. Daí nós temos:

$$\begin{aligned}p2^{n-k}&=\lceil\frac{2^k}{s}\rceil 2^{n-k}\\&<(\frac{2^k}{s}+1)2^{n-k}\\&=\frac{2^n}{s}+2^{n-k}\end{aligned}$$

De novo substituindo os valores de k e s nós obtemos:

$$\frac{2^n}{\lceil n-5\log n\rceil}+2^{n-\lceil 3\log n\rceil}<\frac{2^n}{n-5\log n-1}+\frac{2^{n+1}}{n^3}=\mathcal{O}(\frac{2^n}{n})$$

E finalmente precisamos de $3p2^s=\mathcal{O}(\frac{2^{k+s}}{s})=\mathcal{O}\big(\frac{2^n}{n^3}\big)$ portas $\vee$ e $\wedge$ que aparecem na fórmula 2.5.

Então o número total de portas lógicas necessárias para implementar a (k, s)-representação de Lupanov de f é:

$$\mathcal{O}(\frac{2^n}{n^3})+\mathcal{O}(\frac{2^n}{n^2})+\mathcal{O}(\frac{2^n}{n})+\mathcal{O}(\frac{2^n}{n^3})$$

Ignorando os termos de menores ordem dentro dos $\mathcal{O}()$ nós temos que o número de portas lógicas é algo da ordem de $(1+\frac{1}{n}+\frac{2}{n^2})\frac{2^n}{n}=(1+o(1))\frac{2^n}{n}$.

□

Circuitos de profundidade logaritmica e P vs NC

Mas pra frente nós vamos discutir um pouco mais sobre circuitos com profundidade logaritmica, nesta subseção nós vamos conhecer duas classes de circuitos com circuitos de profundidade logaritmica e além disso nós iremos discutir como elas se relacionam com computação paralela.

Nós dizemos que um problema é eficientemente paralelizável se ele pode ser computado em tempo polilogaritmico usando um número polinomial de processadores. Nós vimos na seção 2.5 que um problema é eficientemente computável se existe uma máquina de Turing que é capaz de decidir tal problema em tempo polinomial. Agora, o que se espera é que aumentar o número de "máquinas de

Turing“ nos permite decidir um dado problema mais rapidamente. O quão mais rapidamente? Saber se todos os problemas em P podem ser *altamente paralelizáveis* ainda é um problema em aberto pelo menos quase tão interessante quanto a questão P vs NP. A seguir nós vemos que a classe NC proposta por Pippenger - o primeiro nome dele é Nick, daí que veio o nome NC: *Nick's class* - que é equivalente à classe de problemas eficientemente paralelizáveis.

Definição 2.48. *(A classe NC)*

Uma linguage L é dita estar em NC^i se L é decidida por uma família de circuitos $\{C_n\}_{n\in\mathbb{N}}$ de tamanho polinomial tal que $DEPTH(C_n) = \mathcal{O}(\log^i n)$, para todo n.

A classe NC é $\bigcup_{i\geq 1} NC^i$.

Se $L \in NC$ então L é eficientemente paralelizável: se L é decidida por uma família de circuitos $\{C_n\}_{n\in\mathbb{N}}$ com profundidade $\mathcal{O}(\log^d n)$ então para entradas de tamanho n nós temos um único processador que computa a descrição de C_n e daí para cada porta lógica p de C_n nós temos um processador que irá computar $val(p)$ e enviar o resultado para o processador de cada porta lógica que depende de p. Nós precisamos de tempo $\mathcal{O}(\log^d n)$ para computar a descrição de C_n e também para computar o valor de cada porta lógica. E pelo tamanho de C_n ser polinomial temos que o número de processadores também é polinomial.

Para ver que a outra direção também é verdade nós contruimos um circuito que nós vemos como um grid onde a porta lógica c_{ij} simula o i-ésimo processador no j-ésimo passo. Este circuito tem tamanho $\mathcal{O}(n^c \log^d n)$ e profundidade $\mathcal{O}(\log^d n)$.

Definição 2.49. *Uma linguagem L é dita estar em AC^i se L é decidida por uma família de circuitos $\{C_n\}_{n\in\mathbb{N}}$ de tamanho polinomial tal que $DEPTH(C_n) = \mathcal{O}(\log^i n)$, para todo n, e além disso o fan-in das portas lógicas são arbitrários.*

A classe AC é $\bigcup_{i\geq 1} AC^i$

2.6 Análise de Fourier de Funções Booleanas

Uma função Booleana é uma função $f : \{0,1\}^n \to \{0,1\}$, porém nesta seção nós vamos na maior parte das vezes considerar funções com domínio $\{-1,1\}^n$ e contradomínio $\{-1,1\}$. O motivo para isso é que os resultados desta seção são geralmente mais intuitivos quando consideramos $\{-1,1\}$ ao invés de $\{0,1\}$. Para passar uma string $x \in \{0,1\}^n$ para uma string $x' \in \{-1,1\}^n$ usamos a transformação $x'_i = (-1)^{x_i}$, então para cada função $f : \{-1,1\}^n \to \{-1,1\}$ temos a função $f^* : \{0,1\}^n \to \{0,1\}$ onde $f^*(x) = 1 - 2f(x')$ que é "equivalente“ à f e também estruturalmente semelhante como vamos ver.

Alguém interessado em saber mais sobre a Análise de funções Booleanas pode ler o livro do Ryan O'Donnell [**?**].

Qualquer função $f : \{-1,1\}^n \to \mathbb{R}$ pode ser representada da seguinte forma:

$$f(x) = \sum_{x' \in \{-1,1\}} g(x', x) f(x') \tag{2.6}$$

Onde $g(x', x) = 1$ quando $x' = x$ e 0 quando $x' \neq x$, e é fácil verificar que $g(x', x) = \prod_{i=1}^{n} \frac{1}{2}(1 + x_i x'_i)$ satisfaz exatamente isso. Seja $\mathcal{V}$ o espaço vetorial de todas funções de $\{-1,1\}^n$ para $\mathbb{R}$ com produto interno $\langle f, g \rangle = 2^{-n} \sum_{x \in \{-1,1\}^n} f(x)g(x) = \mathsf{E}_{x \sim \{-1,1\}^n}[f(x)g(x)]$, nós queremos mostrar que as *funções paridade* $\chi_S(x) = \prod_{i \in S} x_i$, $S \subseteq [n]$ e com $\chi_\emptyset = 1$, formam uma base de $\mathcal{V}$. Ou seja, podemos escrever f como

$$f(x) = \sum_{S \subseteq [n]} \widehat{f}(S) \chi_S(x) \tag{2.7}$$

Onde $\widehat{f}(S)$ é a coordenada de f na "direção S", o que nós vamos chamar de coeficiente de Fourier em S de f.

Proposição 2.50. *Seja $f : \{-1,1\}^n \to \mathbb{R}$. Podemos escrever f como 2.7 onde para cada $S \subseteq [n]$, $\widehat{f}(S) = \langle f, \chi_S \rangle$.*

Demonstração. Nós expandimos 2.6 com $g(x, x') = \prod_{i=1}^{n} \frac{1}{2}(1 + x_i x'_i)$.

$$\begin{aligned}
f(x) &= \sum_{x' \in \{-1,1\}^n} f(x') \prod_{i=1}^{n} \frac{1 + x_i x'_i}{2} \\
&= \sum_{x' \in \{-1,1\}^n} \sum_{S \subseteq [n]} 2^{-n} f(x') \chi_S(x') \chi_S(x) \\
&= \sum_{S \subseteq [n]} \chi_S(x) \left(2^{-n} \sum_{x' \in \{-1,1\}^n} f(x') \chi_S(x') \right) \\
&= \sum_{S \subseteq [n]} \langle f, \chi_S \rangle \chi_S(x)
\end{aligned}$$

Então, para cada $S \subseteq [n]$, fazemos $\widehat{f}(S) = \langle f, \chi_S \rangle$ e obtemos 2.6.

□

Além disso, como existem 2^n funções paridades temos que elas formam uma base de $\mathcal{V}$. A base formada pelas funções paridade é ortornomal, basta observar que $\mathsf{E}_{x \sim \{-1,1\}^n}[\chi_S(x)] = 0$ para qualquer

$S \subseteq [n]$ que não seja $\emptyset$ e $\chi_S\chi_{S'}$ é uma função paridade diferente de $\chi_\emptyset$ (mais especificamente, $\chi_{S \triangle S'}$) sempre que S e S' não são iguais. Quando $S = S'$ temos $\mathsf{E}_{x\sim\{-1,1\}^n}[\chi_S(x)\chi_S(x)] = \mathsf{E}_{x\sim\{-1,1\}^n}[\chi_\emptyset] = 1$.

Nós denotamos por $\|f\|_2$ o valor $\sqrt{\langle f, f\rangle}$, e em geral $\|f\|_p = \mathsf{E}_{x\sim\{-1,1\}^n}[|f(x)|^p]^{1/p}$.

Note que

$$\begin{aligned}
\mathop{\mathsf{E}}_{x\sim\{-1,1\}^n}[f(x)g(x)] &= \mathop{\mathsf{E}}_{x\sim\{-1,1\}^n}[\left(\sum_{S\subseteq[n]} \widehat{f}(S)\chi_S(x)\right)g(x)] \\
&= \sum_{S\subseteq[n]} \widehat{f}(S) \mathop{\mathsf{E}}_{x\sim\{-1,1\}^n}[\chi_S(x)g(x)] \\
&= \sum_{S\subseteq[n]} \widehat{f}(S)\widehat{g}(S)
\end{aligned}$$

Este resultado é o *teorema de Plancherel*. Do teorema de Plancherel podemos obter o *teorema de Parseval*: $\mathsf{E}_{x\sim\{-1,1\}^n}[f(x)^2] = \sum_{S\subseteq[n]} \widehat{f}(S)^2$. No caso especial em que $f : \{-1,1\}^n \to \{-1,1\}$ segue do teorema de Parseval que $\sum_{S\subseteq[n]} \widehat{f}(S)^2 = 1$.

O valor esperado $\mathsf{E}_{x\sim\{-1,1\}^n}[f(x)]$ de $f : \{-1,1\}^n \to \mathbb{R}$ é igual a $\widehat{f}(\emptyset)$. Isto segue pela fórmula do coeficiente de Fourier: $\mathsf{E}_{x\sim\{-1,1\}^n}[f(x)] = \mathsf{E}_{x\sim\{-1,1\}^n}[f(x).1] = \mathsf{E}_{x\sim\{-1,1\}^n}[f(x)\chi_\emptyset] = \widehat{f}(\emptyset)$. A variância de f é $Var[f] = \mathsf{E}[f^2] - \mathsf{E}[f]^2 = \sum_{S\neq\emptyset} \widehat{f}(S)^2$. O peso de f em k, onde $0 \leq k \leq n$, é $\sum_{S\subseteq[n],|S|=k} \widehat{f}(S)^2$ que denotamos por $W^k[f]$. Daí podemos reescrever a fórmula para a variância de f como $Var[f] = \sum_{k>0} W^k[f]$. Nós iremos dizer que $f : \{-1,1\}^n \to \{-1,1\}$ é balanceada se $\mathsf{E}_{x\sim\{-1,1\}^n}[f(x)] = \widehat{f}(\emptyset) = 0$, se dissermos que f é essencialmente balanceada nós geralmente queremos dizer que f não está "próxima" de uma das funções constantes, o que pode significar algo como $1/10 \leq \Pr_{x\sim\{-1,1\}^n}[f(x) = -1] \leq 9/10$ (ou, $|\widehat{f}(\emptyset)| \leq 4/5$). Se $f : \{0,1\}^n \to \{0,1\}$ então f é balanceada sse $\mathsf{E}_{x\sim\{0,1\}^n}[f(x)] = \widehat{f}(\emptyset) = 1/2$, e essencialmente balanceada é definida analogamente ao caso $\{-1,1\}$. O grau de f, que denotamos por $deg(f)$, é o maior valor $d \in [n]$ tal que $W^d[f] > 0$.

Influência individual e total

Nós dizemos que uma variável i é *pivotal* para uma string $x \in \{-1,1\}$ em f se $f(x) \neq f(x^{\oplus i})$, onde $x^{\oplus i}$ é a string x com a i-ésima coordenada invertida.

Definição 2.51. *(Influência individual)*

A influência de i em f, denotada por $Inf_i[f]$, é:

$$\Pr_{x\sim\{-1,1\}^n}[i \text{ é pivotal para } x \text{ em } f\,].$$

Ou seja,

$$Inf_i[f] = \Pr_{x \sim \{-1,1\}^n}[f(x) \neq f(x^{\oplus i})]$$

Para funções Booleanas $f : \{-1,1\}^n \to \{-1,1\}$ nós fazemos a seguinte definição.

Definição 2.52. *Seja $f : \{-1,1\}^n \to \{-1,1\}$. A derivada na direção i de f é definida como*

$$D_i f(x) = \frac{f(x^{(i \to 1)}) - f(x^{(i \to -1)})}{2}$$

onde $x^{(i \to b)}$ é a string x com a coordenada i "forçada" como b.

Note que para todo $i \in [n]$, $D_i f(x)^2$ é 1 quando a coordenada i é pivotal para x em f e 0 caso contrário, portanto $\mathsf{E}[D_i f(x)^2] = Inf_i[f]$ e podemos generalizar a noção de influência para funções $g : \{-1,1\}^n \to \mathbb{R}$ definindo $Inf_i[g] = \mathsf{E}[D_i g(x)^2]$.

Proposição 2.53. *Seja $f : \{-1,1\}^n \to \mathbb{R}$ e $i \in [n]$, então:*

1. $D_i f(x) = \sum_{S \ni i} \widehat{f}(S) \chi_{S \setminus \{i\}}(x)$;
2. $Inf_i[f] = \sum_{S \ni i} \widehat{f}(S)^2$.

Demonstração. Primeiros nós provamos (1). Como D_i é um operador linear, precisamos apenas mostrar que $D_i \chi_S$ é igual a $\chi_{S \setminus \{i\}}$ quando $i \in S$ e 0 caso contrário.

Se $i \in S$ então $\chi_S(x^{(i \to 1)}) = 1 \times \prod_{i \in S \setminus \{i\}} x_i = \chi_{S \setminus \{i\}}(x)$. Na mesma forma vemos que $\chi_S(x^{(i \to -1)}) = -\chi_{S \setminus \{i\}}(x)$. Portanto

$$\begin{aligned} D_i \chi_S(x) &= \frac{1}{2}(\chi_{S \setminus \{i\}}(x) - (-\chi_{S \setminus \{i\}}(x))) \\ &= \frac{1}{2}(2\chi_{S \setminus \{i\}}(x)) \\ &= \chi_{S \setminus \{i\}}(x) \end{aligned}$$

Por outro lado, se $i \notin S$ então $\chi_S(x^{(i \to 1)}) = \chi_S(x^{(i \to -1)})$ e portanto $D_i \chi_S(x) = 0$. O item (2) é só uma aplicação do teorema de Parseval e $Inf_i[f] = E[D_i f(x)^2]$.

□

Nós podemos meio que generalizar o operador D_i para funções $f : \{0,1\}^n \to \mathbb{R}$ com o operador de Laplace definido a seguir.

Definição 2.54. *(Operador de Laplace)*

Seja $f : \{0,1\}^n \to \mathbb{R}$. O operador de expectativa E_i é definido como

$$E_i f(x) = \mathsf{E}_{b\in\{0,1\}}[f(x^{(i\to b)})]$$

O operador de Laplace L_i é

$$L_i f = f - E_i f$$

Ou seja, o operador de Laplace subtrai de f a parte de f que não depende da i-ésima coordenada, então parece razoável medir a "importância" da i-ésima coordenada usando L_i, assim como fizemos com D_i. Nós podemos provar o seguinte.

Proposição 2.55. *Seja $f : \{0,1\} \to \mathbb{R}$, então*

$$L_i f = \sum_{S\ni i} \widehat{f}(S)\chi_S$$

Demonstração. Para isso precisamos apenas mostrar que

$$E_i f = \sum_{S\not\ni i} \widehat{f}(S)\chi_S \tag{2.8}$$

e portanto o resultado segue pela definição do operador de Laplace.

Para mostrar que a fórmula 2.8 é verdadeira nós só precisamos considerar o caso em que f é uma das funções paridades por E_i se tratar de um operador linear. Então, seja $S \subseteq [n]$. Se $S = \emptyset$ então $E_i\chi_\emptyset = 1 = \chi_\emptyset$. Se $S \neq \emptyset$:

$$\begin{aligned} E_i\chi_S(x) &= \mathsf{E}_{b\in\{0,1\}}[\chi_S(x^{(i\to b)})] \\ &= \frac{1}{2}\big(\chi_S(x^{(i\to 0)}) + \chi_S(x^{(i\to 1)})\big) \end{aligned}$$

Se $i \in S$ então $\chi_S(x^{(i\to 0)}) + \chi_S(x^{(i\to 1)}) = 0$ e caso contrário é igual a $2\chi_S(x)$, e portanto

$$E_i\chi_S = \begin{cases} \chi_S \text{ se } i \notin S \\ 0 \text{ caso contrário} \end{cases}$$

□

Note que se $f : \{-1,1\}^n \to \mathbb{R}$ então $L_i f = x_i D_i f$ e portanto $\mathsf{E}[L_i f^2] = \mathsf{E}[D_i f^2] = Inf_i[f]$. Para funções $f : \{0,1\}^n \to \{0,1\}$ temos somente que $E[L_i f^2] = \frac{1}{4} Inf_i[f]$, se adaptarmos a forma como influências individuais foram definidas em 2.51.

Proposição 2.56. *Seja $f : \{0,1\}^n \to \{0,1\}$ então $\mathsf{E}[L_i f^2] = \frac{1}{4} \Pr_{x\sim\{0,1\}^n}[f(x) \neq f(x^{\oplus i})]$.*

Demonstração. Primeiro note que $\Pr_{x\in\{0,1\}^n}[f(x) \neq f(x^{\oplus i})] = \mathsf{E}_{x\in\{0,1\}^n}\left[\left(f(x) - f(x^{\oplus i})\right)^2\right]$. Portanto

$$\begin{aligned}
\Pr_{x\in\{0,1\}^n}[f(x) \neq f(x^{\oplus i})] &= \mathsf{E}_{x\in\{0,1\}^n}\left[\left(f(x) - f(x^{\oplus i})\right)^2\right] \\
&= 4\mathsf{E}_{x\in\{0,1\}^n}\left[\left(\frac{f(x) - f(x^{\oplus i})}{2}\right)^2\right] \\
&= 4\mathsf{E}[L_i f^2]
\end{aligned}$$

□

Por convenção, neste trabalho nós iremos usar $\mathsf{E}[L_i f^2]$ como a definição de $Inf_i[f]$ para funções $f : \{0,1\}^n \to \{0,1\}$, ao invés de $\Pr_{x\in\{0,1\}^n}[f(x) \neq f(x^{\oplus i})]$. O mesmo vale para funções de $\{0,1\}^n$ para $\mathbb{R}$. A vantagem disso é podermos usar a fórmula que aparece em 2.53 como a definição da influência da i-ésima coordenada independente se o domínio é $\{0,1\}^n$ ou $\{-1,1\}^n$.

A influência total de f é a soma das influências individuais de todas as coordenadas: $\mathrm{I}[f] = \sum_{i\in[n]} Inf_i[f]$. Levando em conta que $Inf_i[f] = \sum_{i\ni S} \widehat{f}(S)^2$, cada coeficiente de Fourier $\widehat{f}(S)$ aparece $|S|$ vezes na soma da influência total, portanto temos a seguinte fórmula: $\mathrm{I}[f] = \sum_{k>0} kW^k[f]$.

Estabilidade de ruído

Seja $\rho \in [0,1]$ e considere a string y formada a partir de x onde:

$$y_i = \begin{cases} x_i \text{ com probabilidade } \rho \\ \text{uniformente distribuido com probabilidade } 1-\rho \end{cases}$$

Dizemos que x e y são ρ-correlacionados e denotamos $y \sim N_\rho(x)$.

Definição 2.57. *(Estabilidade de ruído)*

Seja $f : \{-1,1\}^n \to \mathbb{R}$ e $\rho \in [0,1]$. A estabilidade de ruído de f sobre ρ é

$$Stab_\rho[f] = \mathop{\mathsf{E}}_{\substack{x\sim\{-1,1\}^n \\ y\sim N_\rho(x)}}[f(x)f(y)]$$

Pela definição da estabilidade de ruído, nós temos o seguinte:

$$\begin{aligned} Stab_\rho[f] &= \mathsf{E}[f(x)f(y)] \\ &= \Pr[f(x) = f(y)] - \Pr[f(x) \neq f(y)] \\ &= 1 - 2\Pr[f(x) \neq f(y)] \end{aligned}$$

Onde x e y são strings ρ-correlacionadas. Esse valor, $\Pr[f(x) \neq f(y)]$ é a *sensitividade de ruído* de f que denotamos por $NS_\rho[f]$. Ou seja,

$$NS_\rho[f] = \frac{1}{2} - \frac{Stab_\rho[f]}{2}.$$

Assim como fizemos para a influência também é conveniente ter uma interpretação baseada na expansão de Fourier para a estabilidade de ruído e sensitividade de ruído de uma função. Para isso consideramos o operador de ruído T_ρ definido como

$$T_\rho f(x) = \mathsf{E}_{y \sim N_\rho(x)}[f(y)].$$

Proposição 2.58. *Seja $f : \{-1,1\}^n \to \mathbb{R}$, a expansão de Fourier de $T_\rho f$ é dada por:*

$$T_\rho f = \sum_{S \subseteq [n]} \rho^{|S|} \widehat{f}(S) \chi_S$$

Demonstração. De novo, como T_ρ é um operador linear precisamos apenas provar para o caso $f = \chi_S$, $S \subseteq [n]$:

$$T_\rho \chi_S(x) = \mathop{\mathsf{E}}_{y \sim N_\rho(x)}[\chi_S(y)] = \prod_{i \in S} \mathop{\mathsf{E}}_{y \sim N_\rho(x)}[y_i] = \prod_{i \in S} (\rho x_i) = \rho^{|S|} \chi_S(x)$$

Donde nós usamos que os bits y_i são mutualmente independentes e têm expectativa ρx_i.

□

Daí nós temos que

$$Stab_\rho[f] = \mathop{\mathsf{E}}_{\substack{x \sim \{-1,1\}^n \\ y \sim N_\rho(x)}}[f(x)f(y)] = \mathop{\mathsf{E}}_{x \sim \{-1,1\}^n}\Big[f(x) \mathop{\mathsf{E}}_{y \sim N_\rho(x)}[f(y)]\Big] = \mathop{\mathsf{E}}_{x \sim \{-1,1\}^n}[f(x) T_\rho f(x)] = \langle f, T_\rho f \rangle$$

E portanto, pelo teorema de Plancherel e proposição 2.58:

$$Stab_\rho[f] = \langle f, T_\rho f\rangle = \sum_{S\subseteq[n]} \widehat{f}(S)\widehat{T_\rho f}(S) = \sum_{S\subseteq[n]} \rho^{|S|}\widehat{f}(S)^2 = \sum_{k=0}^{n} \rho^k W^k[f]$$

Tribes$_N$

A função $\text{Tribes}_{w,s} : \{0,1\}^{ws} \to \{0,1\}$ é definida como sendo $\bigvee_{i=1}^{s}\bigwedge_{j=1}^{w} x_{i,j}$. Nós chamamos cada termo de *tribo* e portanto $Tribes_{w,s} = 1$ se e somente se pelo menos uma tribo é unanimamente 1.

Essa função (até onde eu saiba) apareceu primeiro em [**?**], em que Ben-Or e Linial a usaram como exemplo de uma função que é ao mesmo tempo equilibrada e que tem influência máxima pequena, mais ou menos próxima do mínimo imposto pela desigualdade de Poincaré: $\max_{i\in[n]}\{Inf_i[f]\} \geq \frac{1}{n}$.

Fixando $w \geq 1$, nós vamos considerar a função $\text{Tribes}_n = \text{Tribes}_{s,w}$, onde $n = sw$ e s é o maior inteiro tal que $(1-2^{-w})^s \geq 1/2$. Com essas escolhas de s e w nós temos o seguinte.

Proposição 2.59. *Seja $w \geq 1$, s o maior inteiro satisfazendo $(1-2^{-w})^s \geq 1/2$ e $n = ws$, então:*

- $w = logn - loglogn - o(1)$.
- $s = \Theta(\frac{n}{logn})$.

Demonstração. Primeiro nós achamos uma expressão para s. Usando a desigualdade $e^x \geq 1+x$, para todo $x \in \mathbb{R}$, temos que $e^{-s2^{-w}} \geq (1-2^{-w})^s \geq 1/2$, e portanto quanto tiramos o logaritmos de ambos os lados obtemos $-s2^{-w} \geq -ln(2)$ e rearranjando:

$$s \leq 2^w ln(2)$$

E também, como escolhemos s de forma que $(1-2^{-w})^{s+1} < 1/2$:

$$s + 1 \geq \frac{-ln(2)}{ln(1-2^{-w})} \geq (2^w - 1)ln(2)$$

Donde nós usamos que $ln(1-2^{-w}) \geq \frac{1}{1-2^w}$. Juntando tudo temos que

$$2^w ln(2) - ln(2) - 1 \leq s \leq 2^w ln(2)$$

Então, para algum $\alpha_w \in [0,2]$ apropriado, temos que $s = 2^w ln(2) - \alpha_w$.

Agora, $n = ws = w2^w ln(2) - w\alpha_w$ e consequentemente $n \leq w2^w ln(2)$ e $n \geq w2^w ln(2) - 2w$, ou seja:

$$\frac{n}{ln(2)} \leq w2^w \leq \frac{n}{ln(2) - 2^{-w+1}}$$

E vemos que com w tendendo ao infinito, $n/ln(2) = w2^w$ e portanto $w = logn - loglogn - o(1)$. Quando substituimos este valor w na expressão que encontramos para s temos que $s = \Theta(\frac{n}{logn})$.

□

Da forma que definimos n para um w fixo nós temos na verdade uma sequência $\{n_w\}_{w\geq 1}$, e podemos verificar que $n_{w+1} > 2w_w$.

Agora também podemos computar a influência total de Tribes_n.

Proposição 2.60. *Para todo $i \in [n]$, $Inf_i[Tribes_n] = \mathcal{O}(\frac{logn}{n})$.*

Demonstração. Usando a nossa definição para as influências individuais para funções de $\{0,1\}^n$ para $\{0,1\}$, temos que

$$Inf_i[\text{Tribes}_n] = \mathsf{E}[L_i\text{Tribes}_n^2] = \frac{1}{4}\Pr_{x\sim\{0,1\}^n}[\text{Tribes}_n(x) \neq \text{Tribes}_n(x^{\oplus i})]$$

Para que uma coordenada i seja pivotal para uma entrada $x \in \{0,1\}^n$ é suficiente e necessário que todas as outras coordenadas na mesma tribo que i sejam 1 e que nenhuma outra tribo seja 1 unanimamente. Traduzindo:

$$Inf_i[\text{Tribes}_n] = \frac{1}{4}(2^{-w+1}(1-2^{-w})^{s-1}) = \frac{1}{2^{w+1}-2}Pr[Tribes_n(x) = 0]$$

Agora, seja $s' \in \mathbb{R}$ tal que $(1-2^{-w})^{s'} = 1/2$ e $\epsilon \in [0,1)$ tal que $s' = s + \epsilon$. Então temos:

$$\begin{aligned} Pr[Tribes_n(x) = 1] &= (1-2^{-w})^s \\ &= (1-2^{-w})^{s'}(1-2^{-w})^{-\epsilon} \\ &= \frac{1}{2}(1 + \epsilon 2^{-w} + \mathcal{O}(2^{-2w})) \end{aligned}$$

E como $w = logn - loglogn - o(1)$,

$$Pr[Tribes_n(x) = 1] = \frac{1}{2} + \mathcal{O}(\frac{logn}{n})$$

E como $\frac{1}{2^{w+1}-2} = \mathcal{O}(\frac{logn}{n})$ o resultado segue.

□

Para cada $i \in [s]$ denotaremos por T_i o conjunto de índices das variáveis na i-ésima tribo e definimos f_i como

$$f_i(x) = \begin{cases} 1 \text{ se a } i\text{-ésima tribo não é unanimamente 1 sobre a entrada } x \\ 0 \text{ caso contrário} \end{cases}$$

Desta forma $\text{Tribes}_n(x) = 1 - \prod_{i=1}^{s} f_i(x)$ e usamos esta forma de representar Tribes_n para calcular seus coeficientes de Fourier.

Proposição 2.61. *Seja $S \subseteq [n]$, $(S_1, S_2, \ldots, S_s)$ uma partição de S tal que $S_i = S \cap T_i$, para cada $i \in [s]$, e $k = \#\{i | S_i \neq \emptyset\}$ então*

$$\widehat{Tribes_n}(S) = \begin{cases} 1 - (1 - 2^{-w})^s \text{ se } S = \emptyset \\ (-1)^{|S|+k+1} 2^{-kw} (1 - 2^{-w})^{s-k} \text{ se } S \neq \emptyset \end{cases}$$

Demonstração. O caso $S = \emptyset$ é verdade pois

$$\widehat{\text{Tribes}_n}(\emptyset) = \mathsf{E}_{x \sim \{0,1\}^n}[\text{Tribes}_n(x)] = \Pr_{x \sim \{0,1\}^n}[\text{Tribes}_n(x) = 1] = 1 - (1 - 2^{-w})^s$$

Para o caso geral nós temos

$$\begin{aligned} \widehat{Tribes_n}(S) &= \mathsf{E}_{x \sim \{0,1\}^n}[\text{Tribes}_n(x)\chi_S(x)] \\ &= \mathsf{E}_{x \sim \{0,1\}^n}\big[(1 - \prod_{i=1}^{n} f_i(x))\chi_S(x)\big] \\ &= -\prod_{i=1}^{s} \mathsf{E}_{x \sim \{0,1\}^n}[f_i(x)\chi_{S_i}(x)] \\ &= -\prod_{i=1}^{s} \widehat{f_i}(S_i) \end{aligned}$$

Então só precisamos analizar $\widehat{f_i}(S_i)$.

$$\begin{aligned} \widehat{f_i}(S_i) &= 2^{-n} \sum_{x \in \{0,1\}^n} f_i(x)\chi_{S_i}(x) \\ &= 2^{-n} \sum_{\substack{x \in \{0,1\}^n \\ x_j = 1 \text{ para todo } j \in T_i}} (-1)^{|S_i|} f_i(x) + 2^{-n} \sum_{\substack{x \in \{0,1\}^n \\ x_j \neq 1 \text{ para algum } j \in T_i}} \chi_{S_i}(x) \end{aligned}$$

Cada termo da primeira soma é 1 sempre que pelo menos uma das $w - |S_i|$ variáveis em $T_i \setminus S_i$ é 1, o que acontece com probabilidade $(1 - 2^{-w+|S_i|})$. A segunda soma pode ser decomposta pela quantidade de variáveis com índices em S_i que são 0:

$$\begin{aligned}
\widehat{f_i}(S_i) &= (-1)^{|S_i|}2^{-|S_i|}(1 - 2^{-w+|S_i|}) + 2^{-n}\sum_{k=1}^{|S_i|}(-1)^{|S_i|-k}\binom{|S_i|}{k}2^{n-|S_i|} \\
&= (-1)^{|S_i|+1}2^{-w} + (-1/2)^{|S_i|}\sum_{k=0}^{|S_i|}(-1)^{k}\binom{|S_i|}{k} \\
&= (-1)^{|S_i|+1}2^{-w}
\end{aligned}$$

Lembrando que $k = \#\{i|S_i \neq \emptyset\}$, podemos concluir:

$$\begin{aligned}
\widehat{Tribes_n}(S) &= -\prod_{i=1}^{s}\widehat{f}_i(S_i) \\
&= -\left(\prod_{i:S_i\neq\emptyset}(-1)^{|S_i|+1}2^{-w}\right)(1 - 2^{-w})^{s-k} \\
&= (-1)^{|S|+k+1}2^{-kw}(1 - 2^{-w})^{s-k}
\end{aligned}$$

□

Capítulo 3

Computação relativizada e complexidade de circuitos

Neste capítulo nós vemos como circuitos e complexidade "relativizada" se relacionam. Antes de tudo definiremos rapidamente o que queremos dizer por complexidade de espaço.

Definição 3.1. *(Complexidade de espaço)*

Uma linguagem L é dita estar em SPACE($T(n)$) para uma função $T : \mathbb{N} \to \mathbb{N}$ se existe uma máquina de Turing M que decide L usando menos do que $T(n)$ células da fita.

Definição 3.2. *(Classe PSPACE)*

Uma linguagem L é dita estar em PSPACE se e somente se $L \in$ SPACE($p(n)$), para algum polinômio p.

Uma linguagem L é PSPACE-completa se $L \in$ PSPACE e todas linguagens em PSPACE são redutíveis à L em tempo polinomial. Um exemplo de linguagem PSPACE-completa é QUANT-SAT, satisfazibilidade de fórmulas quantificadas. Aqui nós percebemos uma relação entre a classe PSPACE e a hierarquia polinomial.

Teorema 3.3. *$PH \subseteq PSPACE$.*

Isso é verdade pois todas as linguagens em PH podem ser expressas através de uma fórmula quantificada mais a redução de Cook-Levin.

3.1 Uma prova alternativa do teorema 2.32

Nós já vimos que existem oráculos A e B satisfazendo $\mathsf{P}^A = \mathsf{NP}^A$ e $\mathsf{P}^B \neq \mathsf{NP}^B$. Neste capítulo nós vemos uma outra prova do segundo resultado que mostra a idéia geral em como conectamos limites

inferiores a resultados em complexidade relativizada. Basicamente, nós usamos a incapacidade de certos dispositivos computacionais (por exemplo, circuitos de baixa profundidade) de computar certas funções mais o fato que certas classes de complexidade poderem ser expressas por estes mesmos dispositivos computacionais para, ironicamente, diagonalizar e provar separação por oráculo. Note que essa é a mesma idéia que usamos anteriormente para provar o teorema 2.32, porém se tentassemos generalizar aquele argumento para provar que $\mathsf{PH}^A \neq \mathsf{PSPACE}^A$ nós esbarrariamos em alguns problemas como por exemplo ter que simular cada máquina de Turing um número exponencial de vezes o que destroi a nossa hipótese que uma máquina de Turing de tempo polinomial só faz uma quantidade polinomial de consultas ao oráculo, e também podemos ver que trabalhar com modelos que apresentam uma estrutura combinatórica como circuitos Booleanas é muito menos problemático.

Vamos considerar de novo a função Tribes_n que nós vimos no capítulo anterior. Uma das propriedades de Tribes_n é que ela é *evasiva*, o que significa que ter conhecimento apenas parcial dos bits da entrada não é suficiente para avaliar a função. Para formalizar isso, suponha que nós temos uma função $f : \{0,1\}^n \to \{0,1\}$ e queremos computar $f(x)$ para alguma entrada x. Nós podemos fazer isso somente consultando cada bit de x sem nos preocuparmos com a complexidade das computações intermediárias. Desta formas podemos definir diversas medidas de complexidade de f. A medida que nos interessa no momento é a complexidade de consulta determinística:

Definição 3.4. *A complexidade de consulta determinística de um algoritmo A é o maior número de consulta aos bits da entrada sobre todas as entradas $x \in \{0,1\}^n$ que A faz.*

Seja $f : \{0,1\}^n \to \{0,1\}$. A complexidade de consulta determinística de f, que denotamos por $D(f)$, é o mínimo da complexidade de consulta determinística de todos os algoritmos que computam f.

A partir de agora quando dissermos complexidade de consulta fica implícito que estamos falando da complexidade de consulta determinística.

Nós podemos representar as consultas feita pelo algoritmo através de uma árvore de decisão, onde em cada consulta o algoritmo ramifica para a esquerda ou para direita dependendo do valor do bit consultado. Desta forma $D(f)$ é a profundidade mínima entre todas as árvores de decisão que computam f.

Para ver que Tribes_n é evasiva bastar considerar o cenário em que nenhuma consulta revela o valor de uma das tribos (isto é, a consulta sempre retorna 1) até que o último bit da tribo seja consultado e acaba sendo 0, assim mesmo que todas exceto uma tribo tenha todos seus bits revelados o valor desta última tribo vai ser incerto até que a última consulta seja feita. Isso pode ser traduzido como $D(\text{Tribes}_n) = n$. Também poderiamos ter usado o seguinte fato:

Fato 3.5. *Para todas funções $f : \{0,1\}^n \to \{0,1\}$, $deg(f) \leq D(f)$.*

E como vimos em 2.61, $\deg(\text{Tribes}_n) = \text{n}$.

Agora, seja M uma máquina de Turing que pode fazer consultas a algum oráculo, e seja $x \in \{0,1\}^*$. O que M faz antes de tua primeira consulta é independente de qual oráculo ela tem acesso, e o que M faz entre a primeira e segunda consulta só depende da resposta à primeira consulta, e por aí vai. Isto sugere que a computação de M sobre a entrada x pode ser representada por um algoritmo de consulta que ramifica sobre as respostas às tuas consultas ao oráculo.

Seja $\mathcal{X}$ um subconjunto finito de $\{0,1\}^*$ e $A \subseteq \{0,1\}^*$ um oráculo que inicialmente está definido apenas nas strings que não estão em $\mathcal{X}$. Sejam M e x os mesmos do parágrafo anterior, então o algoritmo de consulta $T^{\mathcal{X}}_{M^A,x}$ ramifica sobre consultas ao oráculo por strings em $\mathcal{X}$. Podemos ver a entrada de $T^{\mathcal{X}}_{M^A,x}$ como um vetor característica v onde $v_j = 1$ significa que a j-ésima string na ordem lexicográfica em $\mathcal{X}$ está em A. Desta forma, temos que $T^{\mathcal{X}}_{M^A,x}(v) = 1 \iff M^{A \cup v}(x) = 1$, permitindo um pequeno abuso de notação. Em particular, se M roda em tempo polinomial e $\mathcal{X} = \{0,1\}^{|x|}$ então $T^{\mathcal{X}}_{M^A,x}$ tem complexidade de consulta polilogarítmica se $|x|$ for suficientemente grande.

Então, para cada oráculo $A \subseteq \{0,1\}^*$, considere a seguinte linguagem.

$$L_A = \{1^n | \text{Tribes}_N(A) = 1\}$$

Com N satisfazendo $2^{n-1} \leq N \leq 2^n$ e $\text{Tribes}_N(A)$ significando a função Tribes_N com o vetor característica de $A^{=n}$ em seu argumento, mas como N pode ser menor do que 2^n nós na verdade truncamos a string para que ela tenha tamanho igual a N. $L_A \in \mathsf{NP}^A$ para todos $A \subseteq \{0,1\}^*$ pois precisamos apenas verificar que para algum i múltiplo de w, $x_i, x_{i+1}, \ldots, x_{i+w-1} \in A$, em que x_i é a i-ésima string de tamanho n na ordem lexicográfica. Com isso dito, podemos provar o teorema 2.32 novamente:

Proposição 3.6. *Existe um oráculo $A \subseteq \{0,1\}^*$ tal que $L_A \notin P^A$.*

Demonstração. Seja $\{M_i\}_{i \geq 1}$ uma enumeração de todas as máquinas de Turing de tempo polinomial, e $\{p_i\}_{i \geq 1}$ o tempo de execução das respectivas máquinas de Turing.

Assim como fizemos na primeira prova do teorema 2.32, vamos construir o oráculo A em estágios. Inicialmente $A = \emptyset$ e n_1 é o menor inteiro tal que $2^{n_1} > p_1(n_1)$ para todos $n \geq n_1$. Consideramos então o i-ésimo estágio e fazemos as seguintes definições:

- $A(i-1)$ é o conjuto (finito) de todas as strings que já foram declaradas em A após o estágio $i-1$.
- n_i é um inteiro que satisfaz
  1. é maior do que o tamanho de todas as strings já declaradas em $A(i-1)$;
  2. $2^{n_i} > p_i(n_i)$;

3. $2^{n_i-1} \leq N \leq 2^{n_i}$, para algum N que pode ser definido como em 2.59 (lembrando que este N é único).

- $D(i) = \bigcup_{k=n_i}^{p_i(n_i)} \{0,1\}^k$.

Como o algoritmo de consulta de M_i sobre a entrada 1^{n_i} tem complexidade de consulta polilogaritmica e Tribes_N é evasiva, segue que existe um oráculo $A' \subseteq \{0,1\}^*$ tal que $T^{\{0,1\}^{n_i}}_{M_i^{A(i-1)},1^{n_i}}(A') \neq \text{Tribes}_N(A')$. Seja $B = A' \cap D_i$ e $A(i) = A(i-1) \cup B$, nós podemos argumentar que M_i não pode ser uma máquina de Turing que decide $L_{A(i)}$ pois:

- Se $M_i^{A'}(1^{n_i}) = 0 \Rightarrow \text{Tribes}_N(A') = 1$ e portanto $1^{n_i} \in L_{A(i)}$.
- Se $M_i^{A'}(1^{n_i}) = 1 \Rightarrow \text{Tribes}_N(A') = 0$ e portanto $1^{n_i} \notin L_{A(i)}$.

Por fim, nós fazemos $A = \bigcup_{i \geq 1} A(i)$ e pelo nosso argumento todas as máquinas de Turing de tempo polinomial com acesso a A deixam de computar L_A corretamente em pelo menos uma entrada e portanto $L_A \notin P^A$.

□

Na prova que acabamos de ver nós usamos dois fatos a respeito da função Tribes_N: 1) ela é evasiva (na verdade só precisamos do fato que Tribes_n não pode ser computada por um algoritmo que faz $\Omega(n^c)$ consultas, para qualquer $c > 0$) e 2) ela admite 1-certificados (em algum lugar eu explico o que é isso) pequenos. Portanto poderiamos substituir Tribes_n por qualquer função que satisfaça 1 e 2, como por exemplo a função Or_n, a disjunção de N variáveis. Porém, a função Or_n não é balanceada, uma propriedade que nós vamos precisar na seção seguinte.

3.2 P ≠ NP para oráculos aleatórios

Um dos objetivos de estudar complexidade relativizada é enteder a relação entres classes de complexidades quando não conseguimos dizer nada de muito útil no mundo não-relativizado. Porém, como vemos pelo teorema 2.32, provar que P ≠ NP para algum oráculo não é nenhuma evidência que P ≠ NP no mundo não-relativizado, até porque a construção do oráculo A em ambas provas que vimos é só uma especialização do método da diagonalização que por sua vez é basicamente só uma forma de "trapacear o sistema" construido uma asserção que codifica a tua própria falsidade. Então, com o objetivo de conseguir algum tipo de evidência que P ≠ NP poderiamos nos perguntar se é o caso que P e NP são diferentes no mundo relativizado típico. E de fato, o próximo teorema é o assunto desta subseção:

Teorema 3.7. $Pr_A[\mathsf{P}^A \neq \mathsf{NP}^A] = 1$.

Antes de ver a prova do teorema 3.7 nós vamos discutir o que queremos dizer por $\mathsf{P}^A \neq \mathsf{NP}^A$ com probabilidade 1.

A lei zero-um de Kolmogorov

A lei zero-um de Kolmogorov nos diz que certos eventos sempre acontecem com probabilidade 0 ou com probabilidade 1. O que isso quer dizer, e que eventos são esses?

Prova do teorema 3.7

A idéia principal do teorema 3.7 é que a função $Tribes_n$ não pode nem mesmo ser aproximada por algoritmos de consulta com complexidade de consulta polilogarítmica.

Proposição 3.8. *Seja A qualquer algoritmo de consulta com complexidade de consulta $o(\frac{n}{logn})$, então:*

$$\Pr_{x\sim\{0,1\}^n}[A(x) = Tribes_n(x)] < 0,65$$

Demonstração. É suficiente provar que qualquer algoritmo de consulta que faz consultas a no máximo $\frac{1}{10}s$ das tribos não pode aproximar Tribes_n.

Seja A tal algoritmo de consultas. Nós usamos o fato que para qualquer entrada x em que A não "descobre" uma tribo que é unanimamente 1, o melhor que A pode fazer é adivinhar o valor de $Tribes_n(x)$. Chamamos tal entrada x de "má". Então:

$$\Pr_{x\in\{0,1\}^n}[x \text{ é má}] \geq 9/10$$

Se x é má então $Tribes_n(x) = 1$ se e somente se pelo menos uma das $\frac{9}{10}s$ tribos restantes é unanimamente 1. Isso acontece com probabilidade $1 - (1 - 2^{-w})^{\frac{9}{10}s}$, mas temos que:

$$1/2 \leq (1 - 2^{-w})^{\frac{9}{10}s} \leq 2^{-\frac{9}{10}} < 0,55$$

Para w suficientemente grande. Ou seja, condicionando em nenhuma das primeiros $\frac{1}{10}s$ tribos serem unanimamente 1, o melhor que A pode fazer é adivinhar que $\text{Tribes}_n(x) = 0$ e ainda assim A não poderá fazer melhor do que acertar numa fração maior do que 0,55 das vezes. Então:

$$\Pr_{x\sim\{0,1\}^n}[A(x) = \text{Tribes}_n(x)] < 1/10 + 0,55 = 0,65.$$

□

Alternativamente podemos provar o teorema anterior da seguinte forma:

Demonstração. De novo nós provamos que qualquer algoritmo A que só faz consultas às primeiras $\frac{1}{10}s$ tribos não pode aproximar a função Tribes_n. Podemos notar que neste caso o melhor que podemos fazer é fazer A computar a função $\text{Tribes}_{\frac{1}{10}s,w}$, ou seja, se g é a função computada por A, então

$$g(x) = 1 \iff \text{pelo menos uma das primeiras } \frac{1}{10}s \text{ tribos é unanimamente 1.}$$

Daí temos que

$$\Pr_{x\sim\{0,1\}^n}[\text{Tribes}_n(x) \neq g(x)] = \mathsf{E}_{x\sim\{0,1\}^n}[(\text{Tribes}_n(x) - g(x))^2].$$

Mas como $\text{Tribes}_n(x) \geq g(x)$, para todos $x \in \{0,1\}^n$, temos que

$$\begin{aligned}
\Pr_{x\sim\{0,1\}^n}[\text{Tribes}_n(x) \neq g(x)] &= \mathsf{E}_{x\sim\{0,1\}^n}[\text{Tribes}_n(x) - g(x)] \\
&= \mathsf{E}_{x\sim\{0,1\}^n}[\text{Tribes}_n(x)] - \mathsf{E}_{x\sim\{0,1\}^n}[g(x)] \\
&= \widehat{\text{Tribes}_n}(\emptyset) - \widehat{g}(\emptyset) \\
&= 1 - (1-2^{-w})^s - 1 + (1-2^{-w})^{\frac{1}{10}s} \\
&= (1-2^{-w})^{\frac{1}{10}s} - (1-2^{-w})^s
\end{aligned}$$

Com w tendendo ao infinito isso é igual a $2^{-\frac{1}{10}} - 1/2 \approx 0,433$, e portanto

$$\Pr_{s\sim\{0,1\}^n}[\text{Tribes}_n(x) = A(x)] \leq 1 - 0,433 = 0,567.$$

□

Na verdade, como nós só precisamos provar que algoritmos que fazem uma quantidade polilogarítmica de consultas não podem aproximar Tribes_n nós podemos mudar a fração de tribos consultadas para algo como $\frac{\log^d n}{n}$ e daí temos que a probabilidade que A coincide com Tribes_n seria algo como $1/2 + o(1)$.

Então podemos provar o teorema 3.7.

Demonstração. (Prova do teorema 3.7)

Pela proposição 3.8 nós temos que para uma fração significativa das entradas, $T^{\{0,1\}^{n_i}}_{M^{A(i-1)},1^{n_i}}(A') \neq Tribes_N(A')$ e portanto o conjunto $\{A \subseteq \{0,1\}^* | \mathsf{P}^A \neq \mathsf{NP}^A\}$ tem medida (de Lebesgue?) positiva.

□

3.3 PH vs PSPACE

Agora nós generalizamos a idéia que usamos para separar P e NP nas seções anteriores para podermos separar PSPACE e a hierarquia polinomial. Da mesma forma que usamos árvores de decisão para representar P nós iremos usar circuitos AC^0 para representar PH.

Fixe $k \geq 1$. Seja $L \subseteq \{0,1\}^*$ uma linguagem em Σ_k^p, o que significa que existe uma máquina de Turing M de tempo polinomial tal que para todo $x \in \{0,1\}^*$, $x \in L \iff \exists y_1 \forall y_2 \ldots Q_k y_k M(x, y_1, y_2, \ldots, y_k) = 1$. Fixando $y_1, y_2, \ldots, y_k$, podemos representar a computação de M sobre $x, y_1, y_2, \ldots, y_k$ por uma árvore de decisão da forma que vimos anteriormente. Então o circuito AC^0 $C^{\mathcal{X}}_{M,x}$ que usamos para representar L tem profundidade $k+1$ onde cada quantificador $\exists$ é representado por uma porta $\vee$ e $\forall$ é representado por uma porta $\wedge$, a profundidade é $k+1$ porque no nível mais baixo nós temos árvores de decisão que podem ser representadas por fórmulas FNC ou FND. Também vemos que o fan-in das portas no primeiro nível é $\mathsf{polylog}(n)$ e o fan-in nas demais portas é $2^{p(n)}$, o tamanho do circuito é $N^{\mathsf{polylog}(N)}$, onde $N = 2^n$, o tamanho da entrada de $C^{\mathcal{X}}_{M,x}$.

Teorema 3.9. *Existe $A \subseteq \{0,1\}^*$ tal que $\mathsf{PSPACE}^A \neq \mathsf{PH}^A = \bigcup_{k \geq 1} \Sigma_k^{p,A}$.*

Nós provamos 3.6 a partir de um limite inferior para a função Tribes_N. Para provar o teorema 3.9 nós iremos usar um limite inferior para a função paridade de n variáveis.

Definição 3.10. *(Paridade)*

Para $n \geq 1$ a função paridade de n variáveis é 1 se e se somente se $\sum_{i=1}^n x_1 \pmod 2 = 1$.

Consideremos para cada oráculo $A \subseteq \{0,1\}^*$ a seguinte linguagem:

$$L(A) = \{1^n | A(x) = 1 \text{ para um número ímpar de } x \in \{0,1\}^n\}.$$

E temos que $L(A) \in \mathsf{PSPACE}^A$ já que só precisamos de espaço para escrever cada string de n bits na fita de oráculo mais uma célula para guardar a paridade de $|\{x \in \{0,1\}^n | A(x) = 1)\}|$.

O teorema a seguir nos dá o limite inferior necessário para podermos provar que $L(A) \notin \mathsf{PH}^A$ para algum oráculo $A \subseteq \{0,1\}^*$.

Teorema 3.11. *Seja $d > 0$ um inteiro. Para n suficientemente grande temos que qualquer circuito de profundidade d com fan-in $\mathsf{polylog}(n)$ no teu primeiro nível e tamanho $< 2^{\mathcal{O}\left(n^{\frac{1}{d-1}}\right)}$ não pode computar a função paridade de n variáveis corretamente em todas as entradas.*

Nós iremos provar o teorema 3.11 no próximo capítulo. Na verdade, iremos ganhar de grátis o seguinte teorema.

Teorema 3.12. *Seja $d > 0$ um inteiro. Então qualquer circuito com profundidade d, fan-in do primeiro nível* $\mathsf{polylog}(n)$ *que computa a função paridade de n variáveis corretamente numa fração maior do que $1/2 + \Omega(n^{-d})$ das entradas deve ter tamanho maior do que $2^{\Theta\left(n^{\frac{1}{d-1}}\right)}$.*

E portanto temos o seguinte.

Teorema 3.13. $\Pr_A[\mathsf{PSPACE}^A \neq \mathsf{PH}^A] = 1$.

Demonstração. (Prova do Teorema 3.9)

Nós usamos o argumento usual.

Seja $\{M_i\}_{i\geq 1}$ uma enumeração de todas as máquinas de Turing de tempo polinomial e $\{p_i\}_{i\geq 1}$ o tempo de execução dessas máquinas de Turing. Fixe algum $k \geq 1$ e provaremos que existe $A \subseteq \{0,1\}^*$ tal que $\mathsf{PSPACE}^A \neq \Sigma_k^{p,A}$.

Inicialmente faça $A = \emptyset$ e seja n_1 tal que $p_1(n_1) < 2^{n_1}$, seja $D(1) = \bigcup_{l=n_1}^{p_1(n_1)} \{0,1\}^l$. Seja A' tal que $C_{M,1^{n_1}}^{\{0,1\}^{n_1}}(A') \neq \mathrm{Parity}_{p_1(n_1)}(A')$ (tal A' deve existir por $C_{M,1^{n_1}}^{\{0,1\}^{n_1}}$ ter tamanho $\mathsf{subexp}(n_1)$ e pelo Teorema 3.11). Daí fazemos $A(1) = A \cup (D(1) \cap A')$ e finalizamos o primeiro estágio.

Estágio i:

Escolha n_i tal que $n_i > p_{i-1}(n_{i-1})$ e $p_i(n_i) < 2^{n_i}$, seja $A(i-1)$ o conjunto de todas as strings declarada em A antes do i-ésimo estágio e $D(i) = \bigcup_{l=n_i}^{p_i(n_i)}$. De novo, escolha A' tal que $C_{M_i,1^{n_i}}^{\{0,1\}^{n_i}}(A') \neq \mathrm{Parity}_{p_i(n_i)}(A')$. Então fazemos $A(i) = A(i-1) \cup (D(i) \cap A')$.

Argumentaremos que $A = \bigcup_{i=1}^{\infty} A(i)$ satisfaz $L(A) \notin \Sigma_k^{p,A}$.

Suponha que uma das $M_i s$ decida $L(A)$ com k quantificadores alternantes e com acesso a A. Então:

- $1^{n_i} \in L(A) \Rightarrow C_{M_i,1^{n_i}}(A) = 1 \Rightarrow C_{M_i,1^{n_i}}(A(i)) = 1 \Rightarrow \mathrm{Parity}_{p_i(n_i)} = 0.$
- $1^{n_i} \notin L(A) \Rightarrow C_{M_i,1^{n_i}}(A) = 0 \Rightarrow C_{M_i,1^{n_i}}(A(i)) = 0 \Rightarrow \mathrm{Parity}_{p_i(n_i)} = 1.$

Ambos os casos são contradições pois $1^n \in L(A) \iff \mathrm{Parity}_{2^n}(A) = 1$. □

3.4 Separando a hierarquia polinomial

Como vimos na seção anterior, podemos representar Σ_k^p por circuitos AC^0 com profundidade $k+1$. Portanto, se queremos provar que existe $A \subseteq \{0,1\}^*$ tal que $\Sigma_k^{p,A} \not\subseteq \Sigma_{k-1}^{p.A}$ nós temos que demonstrar a existência de uma "hierarquia de profundidade", ou seja, que existe, para cada $k > 1$ uma função f_k tal que existe um circuito de tamanho polinomial e profundidade $k+1$ que computa f_k mas que

qualquer circuito com profundidade k que computa f_k tem tamanho exponencial. Note que pelo teorema 3.11 a função paridade não pode ser computada por circuitos AC^0 de tamanho polinomial e profundidade k para *todas* as constantes $k \geq 1$ e portanto temos que provar limites inferior para funções diferentes da função paridade. Para isso, definiremos as funções de Sipser:

Definição 3.14. *(As funções de Sipser)*

Para $d \geq 2$ a função de Sipser $f^{d,n}$ é uma fórmula monotônica e read-once *onde o nível mais baixo tem fan-in $\sqrt{\frac{1}{2}dn\log n}$, as portas lógicas nos níveis 2 até d - 1 têm fan-in n e a porta lógica no nível mais alto tem fan-in $\sqrt{\frac{n}{\log n}}$. Ou seja, podemos escrever $f^{d,n}$ como*

$$\bigwedge_{i_d=1}^{\sqrt{\frac{n}{\log n}}} \bigvee_{i_{d-1}=1}^{n} \cdots \bigvee_{i_2=1}^{n} \bigwedge_{i_1=1}^{\sqrt{\frac{1}{2}dn\log n}} x_{i_1,i_2,\ldots,i_d}, \text{ se } d \text{ é par.} \tag{3.1}$$

e

$$\bigwedge_{i_d=1}^{\sqrt{\frac{n}{\log n}}} \bigvee_{i_{d-1}=1}^{n} \cdots \bigwedge_{i_2=1}^{n} \bigvee_{i_1=1}^{\sqrt{\frac{1}{2}dn\log n}} x_{i_1,i_2,\ldots,i_d}, \text{ se } d \text{ é ímpar.} \tag{3.2}$$

Segue direto da definição que o número m de variáveis de entrada da função $f^{d,n}$ é

$$m = n^{d-2}\sqrt{\frac{n}{\log n}}\sqrt{\frac{1}{2}dn\log n} = n^{d-1}\sqrt{d/2}.$$

Também segue que pela definição de $f^{d,n}$ exposta em 3.1 e 3.2 que ela pode ser computada por um circuito de profunidade d e tamanho

$$S = 1 + \sum_{i=0}^{d-2} n^i\sqrt{\frac{n}{\log n}} = 1 + \Big(\frac{n^{d-1}-1}{n-1}\Big)\sqrt{\frac{n}{\log n}},$$

e como $m = \mathsf{poly}(n)$ temos que $S = \mathsf{poly}(m)$. Agora o resultado que nós precisamos para separar cada nível da hierarquia polinomial relativa à um oráculo segue do seguinte resultado, também provado por Håstad em tua tese de doutorado.

Teorema 3.15. *Seja $d > 2$ e n suficientemente grande, qualquer circuito de tamanho $< 2^{\Theta\left(\sqrt{\frac{n}{d\log n}}\right)}$ não computa a função $f^{d,n}$ corretamente em todas as entradas.*

De novo, deixaremos a prova do teorema 3.15 para o próximo capítulo, por enquanto só estamos preocupados na seguinte aplicação deste teorema.

Teorema 3.16. *Existe um oráculo $A \subseteq \{0,1\}^*$ tal que para todo $k \geq 2$, $\Sigma_k^{p,A} \neq \Sigma_{k-1}^{p,A}$.*

Ou seja, para este oráculo A, $PH^A \neq \Sigma_k^{p,A}$, para todos $k \geq 1$. Logo, em particular, $P^A \neq PH^A \Rightarrow P^A \neq NP^A$, então podemos ver a prova do teorema 3.16 como a terceira prova do teorema de Baker-Gill-Solovay em 2.32 e 3.6 que iremos ver, mas desta vez também estaremos provando algo bem mais forte usando basicamente a mesma estratégia.