Análise de Algoritmos

NP-Completude

NP-Completude

- Quase todos os algoritmos que estudamos fazem parte do conjunto de Problemas Polinomiais; tempo do pior caso na ordem de $O(n^K)$, para alguma constante k.
- Existem problemas que têm solução mas não em tempo polinomial, e sim tempo não-polinomial ou superpolinomial.
- Geralmente pensamos em **algoritmos polinomiais** como sendo **tratáveis ou fáceis**, e **problemas superpolinomial** como **intratáveis ou difíceis**.

Problemas NP

- Classe de problemas NP-completo: contém problemas cujo o status é desconhecido.
- Problemas para os quais nenhum algoritmo de tempo polinomial foi encontrado como solução, e também não há uma prova que tal algoritmo não exista para eles.
- Essa questão P ≠ NP é um dos problemas de pesquisa em aberto mais profundos e descordantes em teoria da ciência da computação desde que foi colocada pela primeira vez em 1971.

Problemas P e NP - Exemplos

- Alguns problemas Não-Polinomiais são, superficialmente, parecidos com outro problema que tem uma solução polinomial, e isto nos provoca a tentar solucioná-los. Alguns deles estão listados a seguir.
- **O mais curto X O mais longo caminho em um grafo**: sabemos que mesmo com pesos de arestas negativos, podemos encontrar caminhos mais curtos de uma única fonte em um grafo direcionado G=(V,E) em tempo O(VE). No entanto, encontrar um caminho simples mais longo entre dois vértices é difícil. Apenas determinar se um grafo contém um caminho simples com pelo menos um determinado número de arestas é NP-completo.

Problemas P e NP - Exemplos

- **Tour de Euler x Ciclo Hamiltoniano**: um passeio de Euler em um grafo direcionado G=(V, E) é um ciclo que atravessa cada aresta de G exatamente uma vez, embora seja permitido visitar cada vértice mais de uma vez, é solucionado em tempo O(E). Um ciclo hamiltoniano do grafo direcionado G=(V, E) é um ciclo simples que contem todos os vértices em V. Determinar se um grafo direcionado tem um ciclo hamiltoniano é NP-completo.

Problemas P e NP - Exemplos

- **2-CNF satisfabilidade x 3-CNF satisfabilidade**: Uma fórmula booleana contém variáveis cujos valores são 0 ou 1; conectivos booleanos como ^ (AND), V (OR) e ! (NOT); e parênteses.
- Uma fórmula booleana é satisfazível se existir alguma atribuição dos valores 0 e 1 à suas variáveis que a levem a ser avaliada como 1.
- Informalmente, uma fórmula booleana está em forma normal k-conjuntiva, ou k-CNF, se for o AND de cláusulas de ORs de exatamente k variáveis ou suas negações.
- Por exemplo, a fórmula (x1 V !x2) ^ (!x1 V x3) ^ (!x2 V !x3) está em 2-CNF. (Tem a atribuição satisfatória (x1=1, x2=0, x3=1).
- Embora possamos determinar em tempo polinomial se uma fórmula de 2-CNF é satisfazível, veremos mais adiante que determinar se uma 3-CNF fórmula é satisfazível é NP-completa.

Classes de Problemas

- Classe P: consiste daqueles problemas que têm solução em tempo polinomial. Mais especificamente, problemas que podem ser solucionados em tempo $O(n^k)$, para alguma constante k, e n é o tamanho da entrada do problema.
- Classe NP: consiste daqueles problemas que são **VERIFICÁVEIS** em tempo polinomial. Se de alguma forma nos fosse dado um "**certificado**" de uma solução, então poderíamos verificar se o certificado está correto no tempo polinomial no tamanho da entrada para o problema.

Classes de Problemas

- Classe NP-Completo: Informalmente, um problema está na classe NPC se estiver em NP e for tão “difícil” quanto qualquer problema em NP. Vamos definir formalmente o que significa ser tão difícil quanto qualquer problema em NP mais adiante.
- Um grande esforço já foi dedicado até agora para provar que os problemas NP-completos são intratáveis sem um resultado conclusivo; logo não podemos descartar a possibilidade de que os problemas NP-completos sejam de fato solucionáveis em tempo polinomial.

Visão geral de como mostrar problemas como NP-completos

- Quando demonstramos que um problema é NP-completo, estamos fazendo uma afirmação sobre o quão difícil é (ou pelo menos o quão difícil pensamos que é), em vez de quão fácil é.
- Não estamos tentando provar a existência de um algoritmo eficiente, mas sim que nenhum algoritmo eficiente provavelmente existirá.

Visão geral de como mostrar problemas como NP-completos

- Contamos com três conceitos-chave para mostrar que um problema é NP-completo:
  - Problemas de Decisão vs. Problemas de Otimizações.
  - Reduções
  - Um primeiro problema NP-Completo

Problemas de Decisão vs Problemas de Otimização

- Muitos problemas de interesse são **problemas de otimização**, nos quais cada solução viável (ou seja, “legal”) tem um valor associado e desejamos encontrar uma solução viável com o melhor valor.
- A NP-completude aplica-se diretamente não a problemas de otimização, mas a **problemas de decisão**, em que a resposta é simplesmente “sim” ou “não” (ou, mais formalmente, “1” ou “0”).

Problemas de Decisão vs Problemas de Otimização

- Embora os problemas NP-completos estejam confinados ao domínio dos problemas de decisão, podemos tirar vantagem de uma relação conveniente entre problemas de otimização e problemas de decisão. Normalmente, podemos lançar um determinado problema de otimização como um problema de decisão relacionado impondo um limite ao valor a ser otimizado.

Exemplo

- Seja um problema de otimização que chamamos de SHORTEST-PATH, onde recebemos um grafo não direcionado G e os vértices u e v, e desejamos encontrar um caminho de u a v que use o menor número de arestas. Em outras palavras, SHORTEST-PATH é o problema do caminho mais curto de par único em um grafo não ponderado e não direcionado.

- Um problema de decisão relacionado a SHORTEST-PATH é PATH: dado um grafo direcionado G, vértices u e v, e um inteiro k, existe um caminho de u para v consistindo de no máximo k arestas?

Problemas de Decisão vs Problemas de Otimização

- A relação entre um problema de otimização e seu problema de decisão relacionado funciona a nosso favor quando tentamos mostrar que o problema de otimização é “difícil”. Isso ocorre porque o problema de decisão é, em certo sentido, “mais fácil”, ou pelo menos “não mais difícil”.
- Como exemplo específico, podemos resolver PATH resolvendo SHORTEST-PATH e então comparando o número de arestas no caminho mais curto encontrado com o valor do parâmetro do problema de decisão k.
- **Em outras palavras, se um problema de otimização é fácil, seu problema de decisão relacionado também é fácil.**

Reduções

- A noção acima de mostrar que um problema não é mais difícil ou mais fácil do que outro se aplica mesmo quando ambos os problemas são problemas de decisão. Aproveitamos essa ideia em quase todas as provas de NP-completude, como segue.

Reduções

- Consideremos um problema de decisão A, que gostaríamos de resolver em tempo polinomial. Chamamos a entrada de um problema específico de instância desse problema; por exemplo, em PATH, uma instância seria um grafo particular G, vértices particulares u e v de G, e um inteiro k particular.
- Agora suponha que já sabemos como resolver um problema de decisão diferente B em tempo polinomial.
- Finalmente, suponha que temos um procedimento que transforma qualquer instância α de A em alguma instância β de B com as seguintes características:
  - A transformação leva tempo polinomial.
  - As respostas são as mesmas. Ou seja, a resposta para α é “sim” se e somente se a resposta para β também for “sim”.

Reduções

- Chamamos tal procedimento de **algoritmo de redução** em tempo polinomial e ele nos fornece uma maneira de resolver o problema A em tempo polinomial:
  - Dada uma instância α do problema A, use um algoritmo de redução de tempo polinomial para transformá-la em uma instância β de um problema B.
  - Executar o algoritmo de decisão de tempo polinomial para B sobre β.
  - Usar a resposta para β como a resposta para α.

Um primeiro problema NP-Completo

- Como a técnica de redução depende de ter um problema já conhecido como NP-completo para provar um problema diferente NP-completo, precisamos de um “primeiro” problema NP-completo. O problema que usaremos é o problema da satisfação do circuito, no qual nos é dado um circuito combinacional booleano composto de portas AND, OR e NOT, e desejamos saber se existe algum conjunto de entradas booleanas para este circuito que faz com que sua saída seja 1.

Formalizando

- Problemas Abstratos
  - Definimos um **problema abstrato** Q como uma relação binária em um conjunto I de instâncias de problemas e um conjunto S de soluções de problemas.
  - Por exemplo, uma instância para SHORTEST-PATH é uma tripla que consiste em um grafo e dois vértices. Uma solução é uma sequência de vértices no grafo, com talvez a sequência vazia denotando que não existe caminho. O problema SHORTEST-PATH em si é a relação que associa cada instância de um grafo e dois vértices a um caminho mais curto no grafo que conecta os dois vértices. Como os caminhos mais curtos não são necessariamente únicos, uma determinada instância do problema pode ter mais de uma solução.

Formalizando...

- Como vimos acima, a teoria da NP-completude restringe a atenção aos problemas de decisão: aqueles que têm uma solução sim/não. Nesse caso, podemos ver um **problema de decisão abstrato** como uma função que mapeia o conjunto de instâncias I para o conjunto de soluções {0, 1}.

Formalizando...

- Para algoritmos de computadores solucionarem problemas abstratos é preciso representar suas instâncias de forma que o programa entenda, geralmente isso é uma codificação de um conjunto A de objetos abstratos em um conjunto de strings binárias.
- **Problema Concreto** é aquele cujo conjunto de instâncias é um conjunto de strings binárias.

Formalizando….

- Formalmente, definimos….
- **A classe P de complexidade** é o conjunto de **problemas de decisão concretos** solucionáveis em tempo polinomial.

Linguagens Formais

- Vamos revisar algumas definições da teoria de Linguagens Formais:
  - Um alfabeto Σ é um conjunto finito de símbolos.
  - Uma linguagem L sobre Σ é qualquer conjunto de strings compostas de símbolos de Σ.
  - Por exemplo, se Σ = {0, 1}, o conjunto L = {10, 11, 101, 111, 1011, 1101, 10001, … } é a linguagem de representações binárias de números primos.
  - Denotamos a string vazia por ε,
  - a linguagem vazia por 0,
  - e a linguagem de todas as strings sobre Σ por Σ*.

Linguagens Formais

- Por exemplo, se $\Sigma = \{0, 1\}$, então $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ é o conjunto de todas as strings binárias. Cada linguagem L sobre Σ é um subconjunto de Σ^*.
- Do ponto de vista da teoria da linguagem, o conjunto de instâncias para qualquer problema de decisão Q é simplesmente o conjunto Σ^*, onde $\Sigma=\{0,1\}$.
- Como Q é inteiramente caracterizado por instâncias de problemas que produzem uma resposta 1 (sim), podemos ver Q como uma linguagem L sobre $\Sigma=\{0, 1\}$, onde $L = \{x \in \Sigma^* : Q(x) = 1\}$

Linguagens Formais

- Por exemplo, o problema de decisão PATH tem a linguagem correspondente
- PATH = {<G, u, v, k> : G =(V, E) é um grafo não dirigido ;
  u e v ϵ V;
  k ≥ é um inteiro, e
  existe um caminho de u a v em G
  consistindo de no máximo k arestas.}

Linguagens Formais

- A estrutura de linguagem formal nos permite expressar de forma concisa a relação entre problemas de decisão e algoritmos que os resolvem.
- Dizemos que um algoritmo A aceita uma string $x \in \{0, 1\}^*$ se, dada a entrada x, a saída do algoritmo A(x) é 1.
- A linguagem aceita por um algoritmo A é o conjunto de strings $L = \{x \in \{0, 1\}^* : A(x)= 1\}$, ou seja, o conjunto de strings que o algoritmo aceita.
- Um algoritmo A rejeita uma string x se A(x) = 0.

Linguagens Formais

- Mesmo que a linguagem L seja aceita por um algoritmo A, o algoritmo não rejeitará necessariamente uma string x que não pertença a L fornecida como entrada para ele. Por exemplo, o algoritmo pode entrar em um loop indefinido.
- Uma linguagem L é **decidida** por um algoritmo A se toda string binária em L é aceita por A e toda string binária que não está em L é rejeitada por A.
- Uma linguagem L é aceita em tempo polinomial por um algoritmo A se é aceita por A e se além disso existe uma constante k tal que para qualquer string $x \in L$ de comprimento n, o algoritmo A aceita x em tempo $O(n^k)$.

Linguagens Formais

- Podemos definir informalmente uma classe de complexidade como um conjunto de linguagens cuja participação é determinada por uma medida de complexidade, como o tempo de execução, de um algoritmo que determina se uma determinada string x pertence à linguagem L. A definição real de uma classe de complexidade é um pouco mais técnico.

  Usando essa estrutura teórica da linguagem, podemos fornecer uma definição alternativa da classe de complexidade P:

- P = {L ⊂ {0, 1}*: existe um algoritmo A que decide L em tempo polinomial }

  De fato, **P também é a classe de linguagens** que podem ser aceitas em tempo polinomial.

Verificação em Tempo Polinomial.

Algoritmos Verificadores, algoritmos que verificam a associação em linguagens.

Por exemplo, Sejam dados:

- instância (G; u; v; k) do problema de decisão PATH,
- p de u para v.

Podemos facilmente verificar se p é um caminho em G e se o comprimento de p é no máximo k, e se for, assumimos p como um "**certificado**" de que a instância realmente pertence a PATH.

Verificação em Tempo Polinomial.

- Problema do Grafo Hamiltoniano (HAM-CYCLE).

  Um grafo G=(V,E) é dito Hamiltoniano se ele é não direcionado e contém um ciclo simples com todos os vértices em V. (Ciclo Hamiltoniano).

- Podemos definir a Linguagem Formal do problema como:

  HAM-CYCLE={<G> : G é um grafo hamiltoniano}.

Verificação em Tempo Polinomial.

- Um ALGORITMO de DECISÃO para o problema, dada uma instância G=(V,E), deve:
  - listar todas as permutações dos vértices de G ; e
  - verificar se cada permutação é um caminho hamiltoniano.
- Tempo de execução: com uma codificação razoável do grafo, sua matriz de adjacências:
  - São |V|=m → m! Permutações,
  - m = Ω(√n) , onde n =<G>, o tamanho da codificação de G
- Logo temos $\Omega(m!) = \Omega(\sqrt{n}\,!) = \Omega(2^{\sqrt{n}})$ que não é $O(n^k)$.

  Esse algoritmo ingênuo não roda em tempo polinomial.

Verificação em Tempo Polinomial.

- Definimos um algoritmo de verificação como sendo um algoritmo de dois argumentos A:
  - uma string de entrada comum x; e
  - uma string binária y chamada de certificado.
- A verifica se para a string de entrada x, existe um certificado y tal que A(x,y) = 1.
- A linguagem verificada por um algoritmo de verificação A é

  $L=\{x \in \{0,1\}^* : \text{existe } y \in \{0,1\}^* \text{ tal que } A(x,y) = 1\}$.

Verificação em Tempo Polinomial.

- Intuitivamente, um algoritmo A verifica uma linguagem L se para qualquer sequência x Є L, existe um certificado y que A pode usar para provar que x Є L.
- Além disso, para qualquer sequência x não pertencente a L, não deve haver certificado provando que x Є L. Ou seja, se a entrada x não pertence a L o algoritmo A não pode ser enganado por um certificado errado.
- Assim se um grafo não é hamiltoniano, não pode haver lista de vértices que engane o algoritmo de verificação a acreditar que o grafo é hamiltoniano, uma vez que o algoritmo de verificação verifica cuidadosamente o “ciclo” proposto para ter certeza.

Classe NP

- **A classe de complexidade NP é a classe de linguagens que pode ser verificada por um algoritmo de tempo polinomial.**
- Mais precisamente, uma linguagem L pertence a NP se e somente se existir um algoritmo de tempo polinomial de duas entradas A e uma constante c tal que

  $L=\{x \in \{0,1\}^*$:existe um certificado y com $|y|=O(|x|^c)$ tal que $A(x,y)=1\}$
- Dizemos que o algoritmo A verifica a linguagem L em tempo polinomial.

Redutibilidade

Intuitivamente, um problema Q pode ser reduzido a outro problema Q' se qualquer instância de Q puder ser "facilmente reformulada" como uma instância de Q', cuja solução fornece uma solução para a instância de Q.

Por exemplo, o problema de resolver equações lineares em um x indeterminado se reduz ao problema de resolver equações quadráticas.

Dada uma instância $ax+b=0$, nós a transformamos em $0x^2 +ax+b = 0$, cuja solução fornece uma solução para $ax + b = 0$. Assim, se um problema Q se reduz a outro problema Q', então Q é, em certo sentido, "não mais difícil de resolver" do que Q'

Redutibilidade

- Retornando à nossa estrutura de linguagem formal para problemas de decisão, dizemos que :
  - uma linguagem L_1 é redutível em tempo polinomial a uma linguagem L_2 , escrita $L_1 \leq_P L_2$, se existir uma função computável em tempo polinomial $f : \{0; 1\}^* \rightarrow \{0; 1\}^*$ tal que :
  - para todo $x \in \{0; 1\}^*$, $x \in L_1$ se e somente se $f(x) \in L_2$}.
- Chamamos a função f de **função de redução**, e um algoritmo de tempo polinomial F que calcula f é um **algoritmo de redução.**

NP-Completude

- Reduções em tempo polinomial fornecem uma maneira formal para apresentar que um problema é pelo menos tão difícil que outro, por um fator polinomial;
- isto é, se $L_1 \leq_P L_2$, (L_1 é redutivel polinomialmente a L_2) então L_1 não é mais que um fator polinomial difícil que L_2, é por isso que a notação "menor ou igual a" para redução é mnemônica.

Classe NP-Completo

- Formalmente, O conjunto de Linguagens NP-Completo, que são os problemas mais difíceis em NP:
- Uma linguagem L $\subseteq$ {0,1}* é NP-Completo se :
  - 1) L $\in$ NP; e
  - 2) L' $\leq_p$ L para todo L' $\in$ NP
- Se uma linguagem satisfaz 2) mas não satisfaz 1), dizemos que L é NP-Difícil (*NP-Hard* em inglês).
- Definimos NPC para ser as linguagens NP-Completas.

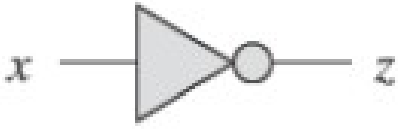
Problema da Satisfazibilidade do circuito

- Infelizmente, a prova formal de que o problema da satisfazibilidade de circuito é NP-completo requer detalhes além do escopo deste texto. em vez disso, descreveremos informalmente uma prova que se baseia em um entendimento básico de circuitos combinacionais booleanos.

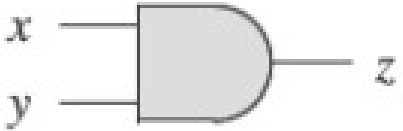
Circuito Combinacional Lógico

- **Consiste de um ou mais elementos combinacionais interconectados por fios**.
- Elementos combinacionais lógicos que usamos no problema de satisfazibilidade de circuitos computam simples funções lógicas e são chamadas de **portas lógicas**.
- Descrevemos a operação de cada porta lógica, e qualquer elemento combinacional lógico, por uma **tabela da verdade**. A tabela da verdade mostra o resultado esperado de cada elemento combinacional para cada possível configuração de entrada.

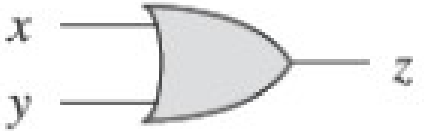
Circuito Combinacional Lógico



x	$\neg x$
0	1
1	0



x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1



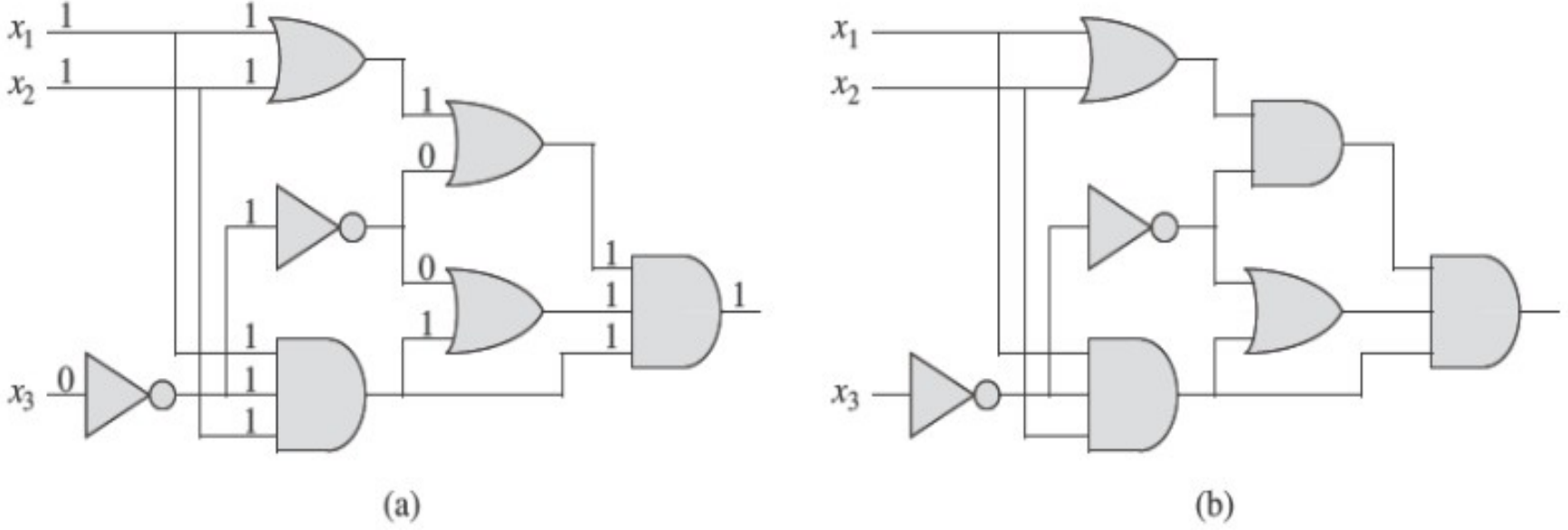
x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Circuito Combinacional Lógico

- Podemos generalizar as portas AND e OR para mais que duas entradas, e assim a porta AND resulta 1 se todas as entradas forem igual a 1, e resulta 0 caso contrário. Já a porta OR resulta 1 se pelo menos uma entrada for 1, e resulta 0 se nenhuma entrada for 1.

- O número de elementos de entrada alimentados por um fio é chamado de **fan-out**.

Circuito Combinacional Lógico



Circuito Combinacional Lógico

- **Um circuito combinacional lógico NÃO contém ciclos.**
- Assim, se criarmos um grafo direcionado G=(V,E) com um vértice para cada elemento combinacional e k arestas direcionadas para cada fio, onde k é o fan-out;
- O grafo contém uma aresta dirigida (*u*,*v*) se um fio conecta a saída do elemento *u* à entrada do elemento *v*.
- Então G dever ser acíclico.

Circuito Combinacional Lógico

- Uma **Atribuição de Verdade** para um CCL é um conjunto de valores lógicos de entrada.
- Dizemos que um CCL é **satisfatório** / satisfazível se ele tem uma **atribuição satisfatória** / satisfazível, ou seja uma atribuição de verdade que causa uma saída igual a 1.

Problema de Satisfazibilidade do Circuito.

- Dado um circuito combinacional lógico composto de portas AND, OR ou NOT, ele é satisfazível?
- Para colocar essa questão formalmente, no entanto, devemos concordar com uma codificação padrão para circuitos.
  - podemos criar um gráfico como a codificação que mapeia qualquer circuito C em uma string binária <C> cujo comprimento é polinomial no tamanho do próprio circuíto.
- Como uma linguagem formal, podemos, portanto, definir

  CIRCUIT-SAT={ <C>: C é um circuito combinacional lógico satisfazível }

Satisfazibilidade de Circuito

- Dado um circuito C, devemos tentar determinar se ele é satisfazível simplesmente verificando todas as possíveis atribuições para a entrada.
- Infelizmente, se o circuito tem k entradas, então é preciso avaliar 2^k possíveis atribuições
- De fato, como afirmamos, há fortes evidências de que não existe nenhum algoritmo de tempo polinomial que resolva o problema da satisfação do circuito porque a satisfação do circuito é NP-completa.
- Dividimos a prova desse fato em duas partes, com base nas duas partes da definição de NP-completude.

Satisfazibilidade de Circuito

Lema 0

O problema de satisfazibilidade de circuito pertence à classe NP.

Prova: Forneceremos um algoritmo A de tempo polinomial de duas entradas que pode verificar CIRCUIT-SAT. Uma das entradas para A é (uma codificação padrão do) circuito combinacional lógico C. A outra entrada é um certificado correspondente a uma atribuição de valores lógico aos fios em C.

Satisfazibilidade de Circuito

- Construímos o algoritmo A da seguinte forma.
  - Para cada porta lógica do circuito, verifica se o valor fornecido pelo certificado no fio de saída é calculado corretamente em função dos valores nos fios de entrada.
  - Então, se a saída de todo o circuito for 1, o algoritmo retorna 1, uma vez que os valores atribuídos às entradas de C fornecem uma atribuição satisfatória. Caso contrário, A gera 0.
- Sempre que um circuito satisfazível C é inserido no algoritmo A, existe um certificado cujo comprimento é polinomial ao tamanho de C e que faz com que A produza um 1.
- Sempre que um circuito não satisfazível é inserido, nenhum certificado pode enganar A fazendo-o acreditar que o circuito é satisfazível.
- O algoritmo A é executado em tempo polinomial: com uma boa implementação, o tempo linear é suficiente. Assim, podemos verificar CIRCUIT-SAT em tempo polinomial, e CIRCUIT-SAT pertence a NP.

Satisfazibilidade de Circuito

- A segunda parte da prova que CIRCUIT-SAT é NP-completo, é mostrar que a linguagem é NP-difícil.
- Ou seja, devemos mostrar que toda linguagem em NP é redutível em tempo polinomial a CIRCUIT-SAT.
  - A prova real deste fato está cheia de complexidades técnicas, e assim vamos nos contentar com um esboço da prova baseada em alguma compreensão do funcionamento do hardware do computador.

Satisfazibilidade de Circuito

- Um programa é armazenado na memória de um computador como uma sequência de instruções; e o registrador **pc** (*program counter*) é responsável por indicar a próxima instrução a ser executada, geralmente a subsequente;
- Normalmente, uma instrução é composta por dois operandos (endereços de memória) e um local de memória que armazena o resultado;
- Durante a execução do programa instruções podem modificar o valor do **pc**, mudando a direção da execução e permitindo laços.

Satisfazibilidade de Circuito

- Em qualquer ponto durante a execução de um programa, a memória do computador mantém todo o estado da computação.
  - (Consideramos que a memória inclui o próprio programa, o contador de programa, armazenamento de trabalho e qualquer um dos vários bits de estado que um computador mantém para escrituração.) Chamamos qualquer estado particular da memória do computador de **configuração**.
- Podemos ver a execução de uma instrução como o mapeamento de uma **configuração** para outra.
- O hardware do computador que realiza esse mapeamento pode ser implementado como um circuito combinacional lógico, que denotamos por M na prova do seguinte lema.

Satisfazibilidade de Circuito

Lema: O problema da satisfação do circuito é NP-difícil.

- **Prova:** Seja L qualquer linguagem em NP. Descreveremos um algoritmo de tempo polinomial *F* computando uma função de redução *f* que mapeia cada string binária x para um circuito C=f(x) tal que x Є L se e somente se C Є CIRCUIT-SAT.

  Uma vez que L Є NP, deve existir um algoritmo *A* que verifica L em tempo polinomial. O algoritmo *F* que construiremos usa o algoritmo *A* de duas entradas para calcular a função de redução *f* .

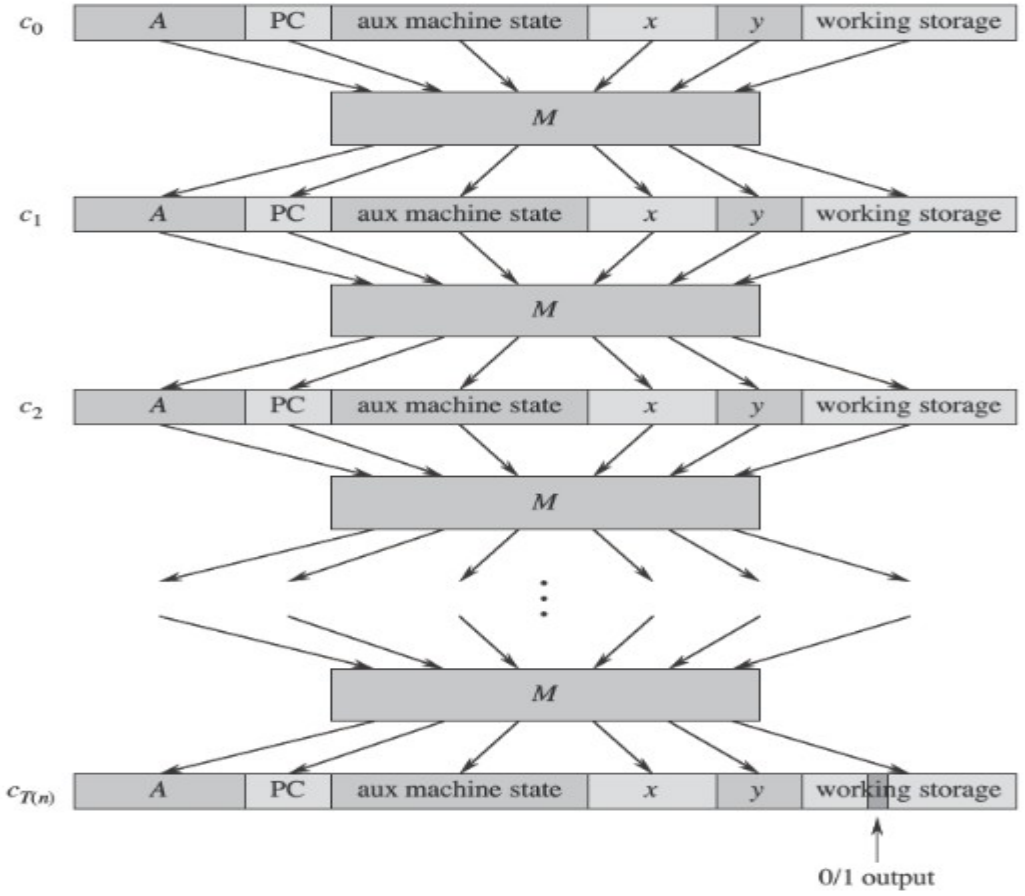
Satisfazibilidade de Circuito

- Seja T(n) o tempo de execução do pior caso do algoritmo A com strings de entrada de comprimento n,
- e seja $k \geq 1$ uma constante tal que $T(n) = O(n^k)$
- e o comprimento do certificado seja $O(n^k)$
- O **tempo de execução de A** é na verdade um **polinômio no tamanho total da entrada**, que inclui
  - a string de entrada e um certificado, mas como o comprimento do certificado é polinomial no comprimento n da string de entrada, o tempo de execução é polinômio em n.

Satisfazibilidade de Circuito

- A ideia básica da prova é representar a computação de A como uma sequência de configurações.
- Como a figura a seguir ilustra, podemos dividir cada configuração em partes que consistem no programa para A,
  - o contador de programa e
  - o estado da máquina auxiliar,
  - a entrada x,
  - o certificado y e
  - o armazenamento de trabalho.
- O circuito combinacional M, que implementa o hardware do computador, mapeia cada configuração c_i para a próxima configuração c_{i+1} , a partir da configuração inicial c_0 . O algoritmo A escreve sua saída — 0 ou 1 — em algum local designado no momento em que termina de executar, e presumimos que depois que A parar, o valor nunca muda. Assim, se o algoritmo for executado por no máximo T(n) passos, a saída aparece como um dos bits em $c_{T(n)}$.

Satisfazibilidade de Circuito



Satisfazibilidade de Circuito

- O algoritmo de redução *F* constrói um único circuito combinacional que computa todas as configurações produzidas por uma dada configuração inicial. A ideia é colar T(n) cópias do circuito M. A i-ésima saída do circuito, que produz a configuração c_i, alimenta diretamente a entrada do (i+1)-ésimo circuito. Assim, as configurações, em vez de serem armazenadas na memória do computador, simplesmente residem como valores nos fios que conectam as cópias de M.
- Lembre-se do que o algoritmo de redução de tempo polinomial *F* deve fazer. Dada uma entrada x, ela deve calcular um circuito C=f(x) que seja satisfazível se e somente se existir um certificado y tal que A(x, y)=1. Quando *F* obtém uma entrada x, primeiro calcula n = |x| e constrói um circuito combinacional C' consistindo em T(n) cópias de M. A entrada para C' é uma configuração inicial correspondente a um cálculo em A(x, y), e a saída é a configuração $c_{T(n)}$

Satisfazibilidade de Circuito

- O algoritmo F modifica ligeiramente o circuito C' para construir o circuito C=f(x).
  - Primeiro, ele conecta as entradas a C' correspondentes ao programa para A, o contador inicial do programa, a entrada x e o estado inicial da memória diretamente a esses valores conhecidos. Assim, as únicas entradas restantes do circuito correspondem ao certificado y.
  - Segundo, ele ignora todas as saídas de C', exceto um bit de $c_{T(n)}$ correspondente à saída de A.
  - Este circuito C, assim construído, calcula C(y)=A(x,y) para qualquer entrada y de comprimento $O(n^k)$.
- O algoritmo de redução F, quando fornecido uma string de entrada x, calcula tal circuito C e o produz.

Satisfazibilidade de Circuito

- Precisamos provar duas propriedades.
  - Primeiro, devemos mostrar que F calcula corretamente uma função de redução f. Isto é, devemos mostrar que C é satisfazível se e somente se existe um certificado y tal que A(x,y) = 1.
  - Em segundo lugar, devemos mostrar que F roda em tempo polinomial.
- Para mostrar que F calcula corretamente uma função de redução, vamos supor que existe um certificado y de comprimento $O(n^k)$ tal que A(x, y)=1.
- Então, se aplicarmos os bits de y às entradas de C , a saída de C será C(y)=A(x,y)=1.
- Assim, se existe um certificado, então C é satisfazível.
- Na outra direção, suponha que C seja satisfazível. Portanto, existe uma entrada y para C tal que C(y)=1, da qual concluímos que A(x, y)=1.
- Assim, F calcula corretamente uma função de redução.

Satisfazibilidade de Circuito

- Para completar o esboço de prova, precisamos apenas mostrar que F é executado em tempo polinomial em n = |x|.
  - A primeira observação que fazemos é que o número de bits necessários para representar uma configuração é polinomial em n. O próprio programa para A tem tamanho constante, independente do comprimento de sua entrada x.
  - O comprimento da entrada x é n, e o comprimento do certificado y é $O(n^k)$. Como o algoritmo executa no máximo $O(n^k)$ passos, a quantidade de armazenamento de trabalho exigida por A também é polinomial em n.
  - O circuito combinacional M que implementa o hardware do computador tem tamanho polinomial no comprimento de uma configuração, que é $O(n^k)$; portanto, o tamanho de M é polinomial em n. (A maior parte deste circuito implementa a lógica do sistema de memória.)
  - O circuito C consiste em no máximo $t=O(n^k)$ cópias de M e, portanto, possui polinômio de tamanho em n.
- O algoritmo de redução F pode construir C a partir de x em tempo polinomial, pois cada passo da construção leva tempo polinomial.

Provas de NP-Completude

- Provamos que o problema de satisfação de circuitos é NP-completo por uma prova direta de que

  $L \leq_P$ CIRCUIT-SAT para cada linguagem $L \in$ NP.

- Vamos mostrar como provar que as linguagens são NP-completas sem reduzir diretamente todas as linguagens em NP à linguagem dada.
- Ilustraremos essa metodologia provando que vários problemas de satisfação de fórmulas são NP-completos.

Provas de NP-Completude

Lema 1

Se L é uma linguagem tal que $L' \leq_P L$ para algum $L' \in NPC$, então L é NP-difícil. Se, além disso, $L \in NP$, então $L \in NPC$.

Prova: Como L' é NPC, para todo $L'' \in NP$, temos $L'' \leq_P L'$. Por suposição, $L' \leq_P L$, e portanto por transitividade, temos $L'' \leq_P L$, o que mostra que L é NP-difícil. Se $L \in NP$, também temos $L \in NPC$.

Em outras palavras, ao reduzir uma linguagem NP-completa conhecida L' para L, reduzimos implicitamente todas as linguagens em NP para L.

Provas de NP-Completude

Assim, o Lema 1 nos dá um método para provar que uma linguagem L é NP-completa:

1. Prove L ∈ NP.
2. Selecione uma linguagem NP-completa conhecida L'.
3. Descreva um algoritmo que calcula uma função f mapeando cada instância x ∈ {0, 1}* de L' para uma instância f(x) de L.
4. Prove que a função f satisfaz x ∈ L' se e somente se f(x) ∈ L para todo x ∈ {0, 1}* .
5. Prove que o algoritmo que calcula f é executado em tempo polinomial.

Provas de NP-Completude

- (Os passos 2–5 mostram que L é NP-difícil.) Essa metodologia de redução de uma única linguagem NP-completa conhecida é muito mais simples do que o processo mais complicado de mostrar diretamente como reduzir de cada linguagem em NP. Provar CIRCUIT-SAT ∈ NPC nos deu um "pé na porta". Como sabemos que o problema de satisfação de circuito é NP-completo, agora podemos provar muito mais facilmente que outros problemas são NP-completos. Além disso, à medida que desenvolvemos um catálogo de problemas NP-completos conhecidos, teremos mais e mais opções de linguagens das quais reduzir.

Satisfação da fórmula

Ilustramos a metodologia de redução fornecendo uma prova de NP-completude para o problema de determinar se uma fórmula booleana, **não um circuito**, é satisfatória.

- Este problema tem a honra histórica de ser o primeiro problema já demonstrado como NP-completo.

Satisfação da fórmula

Formulamos o problema de satisfação (fórmula) em termos da linguagem SAT como segue:

- Uma instância de SAT é uma fórmula booleana composta de:

  1. n variáveis booleanas: x1. x2 , ... , xn ;

  2. m conectivos booleanos: qualquer função booleana com uma ou duas entradas e uma saída, como ^ (AND), V(OR), !(NOT), → (implicação), ↔ (se e somente se); e

  3. parênteses. (Sem perda de generalidade, assumimos que não há parênteses redundantes, ou seja, uma fórmula contém no máximo um par de parênteses por conectivo booleano.)

Satisfação da fórmula

- Podemos facilmente codificar uma fórmula booleana em um comprimento que é polinomial em n + m.
- Como em circuitos combinacionais lógicos, **uma atribuição de verdade** para uma fórmula booleana Φ é um conjunto de valores para as variáveis de Φ, e
- uma **atribuição satisfatória** é uma atribuição de verdade que faz com que ela avalie como 1.
- Uma fórmula com uma atribuição satisfatória é uma **fórmula satisfatória**.
- O problema de satisfação pergunta se uma dada fórmula booleana é satisfatória; em termos de linguagem formal,

$$SAT = \{\langle \Phi \rangle : \Phi \text{ é uma fórmula booleana satisfatória}\}$$

Satisfação da fórmula

Exemplo: seja a fórmula

$\Phi = ((x1 \rightarrow x2) \vee !((!x1 \leftrightarrow x3) \vee x4)) \wedge !x2$

tem a atribuição satisfatória {x1 = 0; x2 = 0; x3= 1; x4 = 1}, uma vez que

$\Phi = ((0 \rightarrow 0) \vee !((!0 \leftrightarrow 1) \vee 1)) \wedge !0$ Equação 2.

$\Phi = (1 \vee !(1 \vee 1)) \wedge 1$

$\Phi = (1 \vee 0) \wedge 1$

$\Phi = 1$

E assim a fórmula pertence a SAT.

Satisfação da fórmula

O algoritmo ingênuo para determinar se uma fórmula booleana arbitrária é satisfatória não é executado em tempo polinomial.

- Uma fórmula com n variáveis tem 2^n atribuições possíveis. Se o comprimento de <Φ> for polinomial em n, então verificar cada atribuição requer tempo $\Omega(2^n)$, que é superpolinomial no comprimento de <Φ>.
- Como o teorema a seguir mostra, é improvável que exista um algoritmo de tempo polinomial.

Satisfação da fórmula

Teorema 1

A satisfazibilidade das fórmulas booleanas é NP-completa.

Prova: Começamos argumentando que SAT $\in$ NP. Então provamos que SAT é NP-difícil mostrando que CIRCUIT-SAT $\leq_P$ SAT; pelo Lema 1, isso provará o teorema.

- Para mostrar que SAT pertence a NP, mostramos que um certificado consistindo de uma atribuição satisfatória para uma fórmula de entrada pode ser verificado em tempo polinomial.
- O algoritmo de verificação simplesmente substitui cada variável na fórmula por seu valor correspondente e então avalia a expressão, assim como fizemos na equação 2 acima.
- Essa tarefa é fácil de fazer em tempo polinomial. Se a expressão for avaliada como 1, então o algoritmo verificou que a fórmula é satisfatória.

- Assim, a primeira condição do Lema 1 para NP-completude é válida.

Satisfação da fórmula

Para provar que SAT é NP-difícil, mostramos que CIRCUIT-SAT $\leq_P$ SAT. Em outras palavras, precisamos mostrar como reduzir qualquer instância de satisfação de circuito a uma instância de satisfação de fórmula em tempo polinomial.

- Podemos usar indução para expressar qualquer circuito combinacional booleano como uma fórmula booleana. Simplesmente olhamos para a porta que produz a saída do circuito e expressamos indutivamente cada uma das entradas da porta como fórmulas. Então obtemos a fórmula para o circuito escrevendo uma expressão que aplica a função da porta às fórmulas de suas entradas.
- Infelizmente, esse método direto não equivale a uma redução de tempo polinomial, pois subfórmulas compartilhadas — que surgem de portas cujos fios de saída têm fan-out de 2 ou mais — podem fazer com que o tamanho da fórmula gerada cresça exponencialmente.

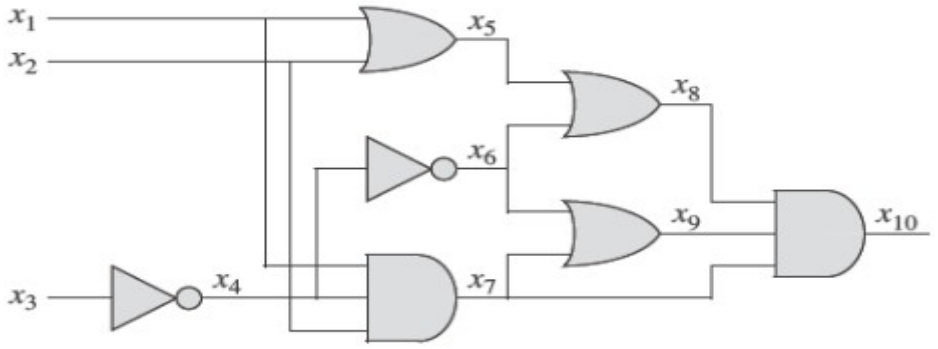
Assim, o algoritmo de redução deve ser um pouco mais inteligente.

Satisfação da fórmula

A Figura abaixo ilustra como superamos esse problema.

- Para cada fio x_i no circuito C, a fórmula Φ tem uma variável xi.
- Podemos expressar como cada porta opera como uma pequena fórmula envolvendo as variáveis de seus fios incidentes.
- Por exemplo, a operação da porta AND de saída é $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$.

Chamamos cada uma dessas pequenas fórmulas de **cláusula**.



Satisfação da fórmula

A fórmula Φ produzida pelo algoritmo de redução é o AND da variável de saída do circuito com a conjunção de cláusulas que descrevem a operação de cada porta. Para o circuito da figura anterior , a fórmula é:

$\Phi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2))$

. $\wedge (x_6 \leftrightarrow \neg x_4) \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$

. $\wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7))$

. $\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$

Dado um circuito C, é simples produzir tal fórmula Φ em tempo polinomial

Satisfação da fórmula

Por que o circuito C é satisfatório exatamente quando a fórmula Φ é satisfatória?

- Se C tem uma atribuição satisfatória, então cada fio do circuito tem um valor bem definido, e a saída do circuito é 1.
- Portanto, quando atribuímos valores de fio a variáveis em Φ, cada cláusula de Φ avalia para 1, e assim a conjunção de todos avalia para 1.
- Por outro lado, se alguma atribuição faz com que Φ avalie para 1, o circuito C é satisfazível por um argumento análogo.

- Assim, mostramos que CIRCUIT-SAT $\leq_P$ SAT, o que completa a prova.

Satisfazibilidade 3-CNF

Definimos a satisfação 3-CNF usando os seguintes termos.

- Um literal em uma fórmula booleana é uma ocorrência de uma variável ou sua negação.
- Uma fórmula booleana está na forma normal conjuntiva, ou CNF, se for expressa como um AND de cláusulas, cada uma das quais é o OR de um ou mais literais.
- Uma fórmula booleana está na forma normal 3-conjuntiva, ou 3-CNF, se cada cláusula tiver exatamente três literais distintos.

- Por exemplo, a fórmula booleana

  (x1 V !x1 V !x2) ^(x3 V x2 V x4)^(!x1 V !x3 V !x4) fórmula 1

  está em 3-CNF.

  - A primeira de suas três cláusulas é (x1 V !x1 V !x2), que contém os três literais x1, !x1, !x2.
- Em 3-CNF-SAT, somos questionados se uma dada fórmula booleana Φ em 3-CNF é satisfatória. O teorema a seguir mostra que é improvável que exista um algoritmo de tempo polinomial que possa determinar a satisfação de fórmulas booleanas, mesmo quando elas são expressas nesta forma normal simples.

Satisfazibilidade 3-CNF

Teorema 2

A satisfazibilidade de fórmulas booleanas na forma normal 3-conjuntiva é NP-completa.

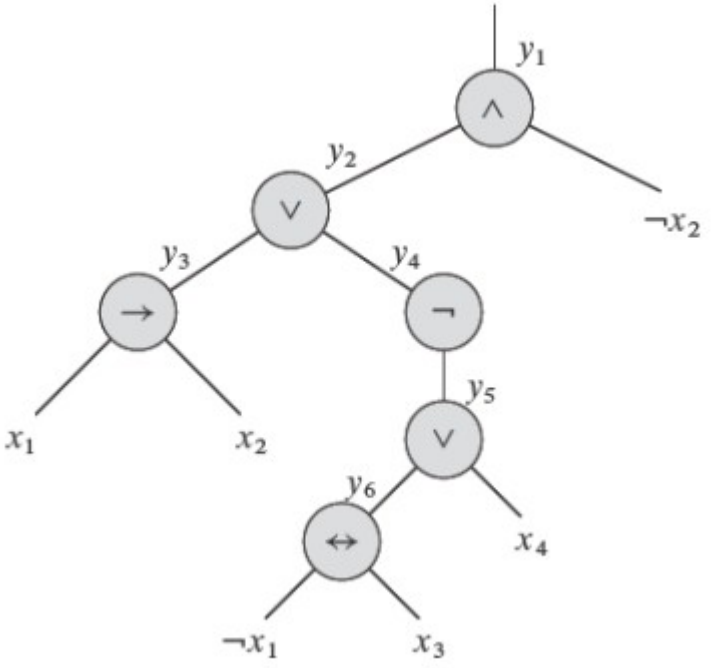
Prova: O argumento que usamos na prova do Teorema 1 para mostrar que SAT Є NP se aplica igualmente bem aqui para mostrar que 3-CNF-SAT Є NP. Pelo Lema 1, portanto, precisamos apenas mostrar que SAT $\leq_P$ 3-CNF-SAT. Dividimos o algoritmo de redução em três etapas básicas. Cada etapa transforma progressivamente a fórmula de entrada Φ mais próxima da forma normal 3-conjuntiva desejada.

- O primeiro passo é similar ao usado para provar CIRCUIT-SAT $\leq_P$ SAT no Teorema 1. Primeiro, construímos uma árvore de “análise sintática” binária para a fórmula de entrada Φ, com literais como folhas e conectivos como nós internos. A Figura a seguir mostra tal árvore de análise sintática para a fórmula

$$\Phi = ((x_1 \rightarrow x_2) \text{ V } !((!x_1 \leftrightarrow x_3) \text{ V } x_4) \text{ ^ } !x_2 \qquad \text{Equação 2.1}$$

Satisfazibilidade 3-CNF

Caso a fórmula de entrada contenha uma cláusula como o OR de vários literais, usamos associatividade para colocar a expressão entre parênteses completamente, de modo que cada nó interno na árvore resultante tenha 1 ou 2 filhos. Agora podemos pensar na árvore de análise binária como um circuito para calcular a função.



Satisfazibilidade 3-CNF

Imitando a redução na prova do Teorema 1, introduzimos uma variável y_i para a saída de cada nó interno. Então, reescrevemos a fórmula original Φ como o AND da variável raiz e uma conjunção de cláusulas descrevendo a operação de cada nó. Para a fórmula 1, a expressão resultante é

$$\Phi' = (y_1 \leftrightarrow (y_2 \wedge !x_2)) \wedge (y_2 \leftrightarrow (y_3 \vee y_4))$$
$$\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \wedge (y_4 \leftrightarrow !y_5)$$
$$\wedge (y_6 \leftrightarrow (!x_1 \leftrightarrow x_3))$$

Satisfazibilidade 3-CNF

O segundo passo da redução converte cada cláusula Φ' , em forma normal conjuntiva.

- Construímos uma tabela verdade de Φ' avaliando todas as atribuições possíveis para suas variáveis.
- Cada linha da tabela verdade consiste em possíveis atribuições das variáveis da cláusula, juntamente com o valor da cláusula sob essa atribuição.
- Usando as entradas da tabela verdade que avaliam para 0, construímos uma fórmula em forma normal disjuntiva (ou DNF)-um OR de ANDs- que é equivalente a !Φ'.
- Então, negamos essa fórmula e a convertemos em uma fórmula CNF Φ''_i , usando as leis de DeMorgan para lógica proporcional,

  !(a ^ b) = !a V !b

  !(a V b) = !a ^ !b
- para complementar todas os literais, trocando ORs por ANDs, e ANDs por Ors.

Satisfazibilidade 3-CNF

Em nosso exemplo,convertemos a cláusula $\Phi'_1=(y_1 \leftrightarrow (y_2 \wedge !x_2))$ em CNF como segue, sua tabela verdade está a seguir.

- A fórmula DNF equivalente a $!\Phi'_1$ é:

  (y1^ y2 ^ x2) V (y1 ^ !y2 ^ x2) V (y1 ^ !y2 ^ !x2) V(!y1 ^ y2 ^ !x2)

- Negando e aplicando a Lei DeMorgan, nós temos a fórmula CNF:

  Φ''_1=(!y1 V !y2 V x2)^(!y1 V y2 V !x2) ^ (!y1 V y2 V x2) ^(y1 ^ !y2 ^ x2)

  que é equivalente a cláusula original Φ'_1.

- Neste ponto, convertemos cada cláusula Φ'_i da fórmula Φ' em uma fórmula CNF Φ''_i , e assim Φ'_i é equivalente à fórmula CNF Φ''_i consistindo da conjunção de Φ''_i. Além disso, cada cláusula de Φ''_i tem no máximo 3 literais.

Satisfazibilidade 3-CNF

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Satisfazibilidade 3-CNF

O terceiro e último passo da redução transforma ainda mais a fórmula para que cada cláusula tenha exatamente 3 literais distintos.

- Construímos a fórmula 3-CNF final Φ''' a partir das cláusulas da fórmula CNF Φ''. A fórmula Φ''' também usa duas variáveis auxiliares que chamaremos de p e q. Para cada cláusula Ci de Φ'', incluímos as seguintes cláusulas em Φ''' :
  - Se C_i tiver 3 literais distintos, então simplesmente inclua C_i como uma cláusula de Φ'''.
  - Se C_i tiver 2 literais distintos, isto é, se $C_i = (l_1 \vee l_2)$, onde l_1 e l_2 são literais, então inclua $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ como cláusulas de Φ''' . Os literais p e !p meramente cumprem o requisito sintático de que cada cláusula de Φ''' tenha exatamente 3 literais distintos. Seja p=0 ou p=1, uma das cláusulas é equivalente a $l_1 \vee l_2$, e a outra é avaliada como 1, que é a identidade para AND.
  - Se Ci tiver apenas 1 literal distinto l, então inclua $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ como cláusulas de Φ''' . Independentemente dos valores de p e q, uma das quatro cláusulas é equivalente a l, e as outras 3 são avaliadas como 1.

Satisfazibilidade 3-CNF

Podemos ver que a fórmula 3-CNF Φ''' é satisfatória se e somente se Φ for satisfatória inspecionando cada uma das três etapas.

- Como a redução de CIRCUIT-SAT para SAT, a construção de Φ' a partir de Φ na primeira etapa preserva a satisfação.
- A segunda etapa produz uma fórmula CNF Φ'' que é algebricamente equivalente a Φ'.
- A terceira etapa produz uma fórmula 3-CNF Φ''' que é efetivamente equivalente a Φ'', já que qualquer atribuição às variáveis p e q produz uma fórmula que é algebricamente equivalente a Φ'' .

Também devemos mostrar que a redução pode ser computada em tempo polinomial. Construir Φ’ a partir de Φ introduz no máximo 1 variável e 1 cláusula por conectivo em Φ. Construir Φ'' a partir de Φ' pode introduzir no máximo 8 cláusulas em Φ'' para cada cláusula de Φ', já que cada cláusula de Φ' tem no máximo 3 variáveis, e a tabela verdade para cada cláusula tem no máximo $2^3 = 8$ linhas. A construção de Φ''' a partir de Φ'' introduz no máximo 4 cláusulas em Φ''' para cada cláusula de Φ''. Assim, o tamanho da fórmula resultante Φ''' é polinomial no comprimento da fórmula original. Cada uma das construções pode ser facilmente realizada em tempo polinomial.

Satisfazibilidade 3-CNF

Também devemos mostrar que a redução pode ser computada em tempo polinomial.

- Construir Φ’ a partir de Φ introduz no máximo 1 variável e 1 cláusula por conectivo em Φ.
- Construir Φ'' a partir de Φ' pode introduzir no máximo 8 cláusulas em Φ'' para cada cláusula de Φ', já que cada cláusula de Φ' tem no máximo 3 variáveis, e a tabela verdade para cada cláusula tem no máximo $2^3 = 8$ linhas.
- A construção de Φ''' a partir de Φ'' introduz no máximo 4 cláusulas em Φ''' para cada cláusula de Φ''.

- Assim, o tamanho da fórmula resultante Φ''' é polinomial no comprimento da fórmula original. Cada uma das construções pode ser facilmente realizada em tempo polinomial.