

UNIVERSIDADE FEDERAL DA PARAÍBA

CAMPUS IV - LITORAL NORTE

DEPARTAMENTO CIÊNCIAS EXATAS - DCX

Estrutura de Dados 2

Trabalho Unidade 03

Nome: Kawhan Laurindo de Lima

Nome: Sidney Jose Gomes Duarte

Nome: Vinicius Teixeira Fernandes

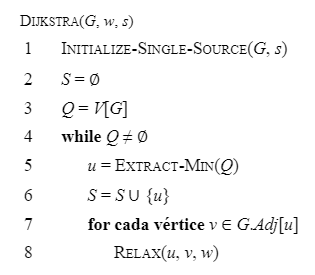
# Caminho-mais-curto - 1 Ponto

1. Escreva os seguintes algoritmos, indique a complexidade deles e quando eles devem ser utilizados:

1. Dijkstra

O algoritmo de Dijkstra resolve o problema de caminhos mínimos de fonte única em um grafo dirigido ponderado G= (V,E) para o caso no qual todos os pesos de arestas são não negativos.

**Complexidade:** O(n² log n) - O(V² log V)



1. Floyd-Warshall

**Complexidade:** O(n³) - O(V³)

Arestas de peso negativo podem estar presentes, mas supomos que não existe nenhum ciclo de peso negativo.

# 

# Agoritmos Gulosos - 1 Ponto

1. **Quais as características dos algoritmos gulosos?**

De modo geral, algoritmos gulosos são usados em problemas de otimização onde é interessante realizar um conjunto de melhores soluções locais, possuindo como objetivo obter uma solução ótima global, tomando como base um determinado parâmetro. Os parâmetros que definem o que será a melhor solução durante a iteração irá variar para cada problema. Podemos então citar algumas das suas principais características:

* As escolhas realizadas são definitivas;
* Não levam em consideração as consequências de suas escolhas;
* Podem fazer cálculos repetitivos.

Essa estratégia nem sempre terá a capacidade de chegar a melhor solução do problema, sendo necessário que coloquemos mais informações nele para conseguirmos chegar a uma solução melhor. Uma das suas principais vantagens é que são algoritmos simples e de fácil implementação. Uma das principais desvantagens é que eles podem entrar em loop, consequentemente desenvolvendo um caminho infinito.

Algoritmos para problemas de otimização, normalmente passam por uma sequência de etapas e cada etapa tem um conjunto de escolhas. Para muitos problemas de otimização é um exagero utilizar programação dinâmica para determinar as melhores escolhas: algoritmos mais simples e mais eficientes servirão. Um algoritmo guloso sempre faz a escolha que parece ser a melhor no momento em questão. Isto é, faz uma escolha localmente ótima, na esperança de que essa escolha leve a uma solução globalmente ótima. Este capítulo explora problemas de otimização para os quais os algoritmos gulosos dão soluções ótimas.

Algoritmos gulosos nem sempre produzem soluções ótimas, mas produzem para muitos problemas. Primeiro, examinaremos um problema simples, mas não trivial: o problema da seleção de atividades, para o qual um algoritmo guloso calcula eficientemente uma solução. Chegaremos ao algoritmo guloso, considerando primeiro uma abordagem de programação dinâmica e depois mostrando que sempre podemos fazer escolhas gulosas para chegar a uma solução ótima.

O método guloso é bastante poderoso e funciona bem para uma ampla faixa de problemas.Muitos são os algoritmos que podem ser vistos como aplicações do método guloso, entre eles os algoritmos de árvore geradora mínima, o algoritmo de Dijkstra para caminhos mais curtos que partem de uma origem única e a heurística gulosa de cobertura de conjuntos de Chvátal .Os algoritmos de árvores geradoras mínimas são um exemplo clássico do método guloso..

1. **Descreva 1 problema que possa ser corretamente resolvido usando a técnica de algoritmos gulosos, descreva o algoritmo que o resolve e sua complexidade.**

Exemplo de situação onde é possível usar um algoritmo guloso: É preciso dar um troco de R$2.89. A melhor solução, isto é, o menor número de moedas possível para obter o valor, consiste em 10 moedas: 2 de valor 100, 3 de valor 25, 1 de valor 10 e 4 de valor 1. De forma geral, agimos tal como um algoritmo guloso: em cada estágio adicionamos a moeda de maior valor possível, de forma a não passar da quantia necessária. Embora seja uma afirmação difícil de provar, é verdade que com os valores dados das moedas, e tendo-se disponível uma quantidade adequada de cada uma, o algoritmo sempre irá fornecer uma solução ótima para o problema. Entretanto, ressalta-se que para diferentes valores de moedas, ou então quando se tem uma quantidade limitada de cada uma, o algoritmo guloso pode vir a não chegar em uma solução ótima, ou até mesmo não chegar a solução nenhuma (mesmo que exista).

Segue um algoritmo que dá o troco para n unidades usando o menor número possível de moedas. **C** é o conjunto de moedas,  **S** é o conjunto que irá conter a solução, **s** é a soma dos itens em **S**.

**A complexidade é O(n).**

* **função** Troco(n)

**const** C ← {100, 25, 10, 5, 1}

S ← ∅

s←0

**enquanto** s ≠ n **faça**

x←o maior item em C tal que s + x ≤ n

**se** este item não existe **então**

**retorne** “Não foi encontrada uma solução!"

**fim se**

S ←A ∪ {uma moeda de valor x}

* s ← s + x

**fim enquanto**

**retorne** S

* **fim função**

# Programação Dinâmica - 1 Ponto

1. **Quais as características da programação dinâmica?**

A programação dinâmica, assim como o método de divisão e conquista, resolve problemas combinando as soluções para subproblemas.(Nesse contexto, “programação” se refere a um método tabular, não ao processo de escrever código de computador). A programação dinâmica se aplica quando os subproblemas se sobrepõem, isto é, quando os subproblemas compartilham subproblemas. Um algoritmo de programação dinâmica resolve cada subproblema só uma vez e depois grava sua resposta em uma tabela, evitando assim, o trabalho de recalcular a resposta toda vez que resolver cada subproblema. Em geral, aplicamos a programação dinâmica a problemas de otimização.

O espaço dos subproblemas deve ser pequeno. Em outras palavras, qualquer algoritmo recursivo que resolva um problema deve resolver os mesmos subproblemas repetidamente, em vez de gerar novos subproblemas.

Programação dinâmica é um paradigma algorítmico muito poderoso no qual um pro- blema é resolvido identificando-se uma coleção de subproblemas e lidando com eles um por um, primeiro os menores, usando as respostas aos problemas menores para ajudar a descobrir as respostas aos maiores, até que toda a coleção esteja solucionada. Em programação dinâmica não é dado um dag: o dag está implícito. Seus nós são os subproblemas que definimos e suas arestas são as dependências entre subproblemas: se para resolver o subproblema B precisamos da resposta ao subproblema A, então existe uma aresta (conceitual) de A para B. Nesse caso, A é considerado um subproblema menor do que B — e será sempre menor, em um sentido óbvio.

Características que um problemas deve apresentar:

• Subestrutura ótima

• Subproblemas coincidentes

* Possui similaridade com dividir e conquistar
  + Técnica de calcular o valor de equações recorrentes
  + Divide o problema em instâncias menores do mesmo problema
  + Resolve os subproblemas e combina as soluções para obter a solução do problema original
* Diferenças
  + Calcular o valor de equações recorrentes de forma eficiente
    - Armazenar resultados parciais em uma tabela
    - Evitar recursões desnecessárias
* Só funciona quando o princípio da otimalidade é válido
* Princípio da otimalidade(Subestrutura ótima ):
  + “A solução ótima do problema original contém soluções ótimas dos subproblemas e, vice-versa.”

**O desenvolvimento de um algoritmo de programação dinâmica segue uma sequência de quatro etapas:**

1.Caracterizar a estrutura de uma solução ótima.

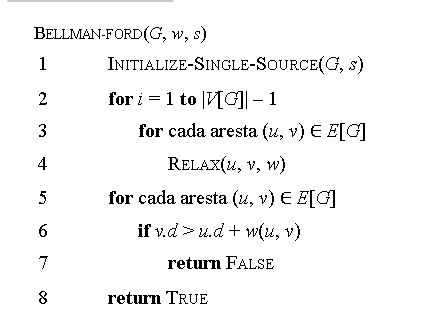
2.Definir recursivamente o valor de uma solução ótima.

3.Calcular o valor de uma solução ótima, normalmente de baixo para cima.

4.Construir uma solução ótima com as informações calculadas

1. **Descreva 1 problema que possa ser corretamente resolvido usando a técnica programação dinâmica, descreva o algoritmo que o resolve e sua complexidade.**

* O algoritmo de Bellman –Ford resolve o problema de caminhos mínimos de fonte única no caso geral no qual os pesos das arestas podem ser negativos. Dado um grafo dirigido ponderado G = (V, E) com fonte s e função peso w : E → ℝ, o algoritmo de Bellman–Ford devolve um valor booleano que indica se existe ou não um ciclo de peso negativo que pode ser alcançado da fonte. Se tal ciclo existe, o algoritmo indica que não há nenhuma solução. Se tal ciclo não existe, o algoritmo produz os caminhos mínimos e seus pesos.
* O algoritmo relaxa arestas diminuindo progressivamente uma estimativa v.d do peso de um caminho mínimo da fonte s a cada vértice v ∈ V até chegar ao peso propriamente dito do caminho mínimo δ(s, v). O algoritmo retorna TRUE se e somente se o grafo não contém nenhum ciclo de peso negativo que possa ser alcançado da fonte.



**Complexidade = 0(V E) = O(n³)**

1. Quais dos seguintes algoritmos são gulosos e quais são dinâmicos?

Kruskal - Programação guloso

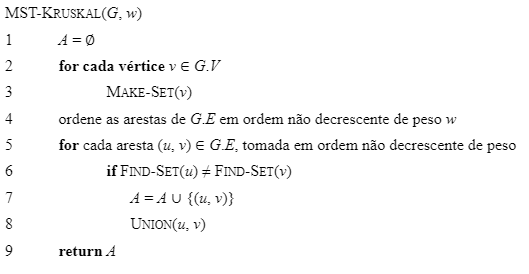
Ford-Bellman - Programação dinâmica

Dijkstra - Programação guloso

Prim - Programação guloso

Floyd-Warshall - Programação dinâmica

# Árvore Geradora e Conjuntos Disjuntos - 1 Ponto

1. **Descreva um algoritmo para resolver o problema da árvore geradora mínima e indique sua complexidade.**

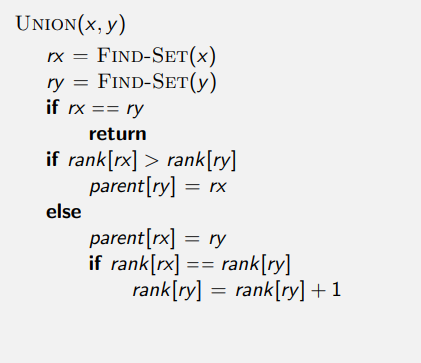
O tempo de execução do algoritmo de Kruskal para um grafo *G* = (*V*, *E*) depende da implementação da estrutura de dados de conjuntos disjuntos. Suponhamos que usamos implementação de floresta de conjuntos disjuntos com as heurísticas de união por ordenação e compressão de caminho, já que essa é a implementação assintoticamente mais rápida conhecida. A Inicialização do conjunto A na linha 1 demora o tempo O(1), e o tempo para ordenar as arestas na linha 4 é O(E lg E). (Consideraremos em breve o custo das |V| operações MAKE-SET no laço for das linhas 2–3.) O laço for das linhas 5–8 executa O(E) operações FIND-SET e UNION na floresta de conjuntos disjuntos. Essas operações, mais as |V| operações MAKE-SET, demoram o tempo total O((V + E) α(V)), onde α é a função de crescimento muito lento. Temos |E| ≥ |V| – 1 e, assim, as operações de conjuntos disjuntos demoram o tempo O(E α(V)). Além disso, visto que α(|V|) = O(lg V) = O(lg E), o tempo de execução total do algoritmo de Kruskal é O(E lg E). Observando que |E| < |V|2, temos lg |E| = O(lg V) e, portanto, podemos reescrever o tempo de execução do algoritmo de Kruskal como O(E lg V).

1. **Descreva a estrutura de conjuntos disjuntos, sua função de união e de achar a que conjunto um elemento pertence.**

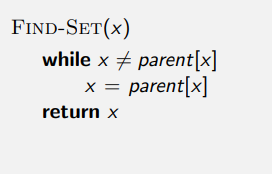
Uma estrutura de dados conjuntos-disjuntos é uma coleção S = {S1,..., Sk} de conjuntos dinâmicos disjuntos. Cada conjunto Sk é identificado por um representante, que é um membro do conjunto. Tipicamente não importa quem é o representante, apenas que ele seja consistente.

Union(x,y): executa a união dos conjuntos que contêm x e y, digamos Sx e Sy, em um conjunto único.

Sx T Sy = 0.

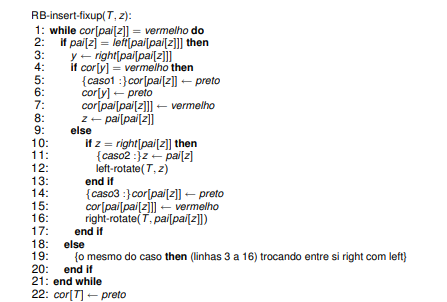


Find-Set(x): Retorna um ponteiro para o representante (único) do conjunto que contém x.



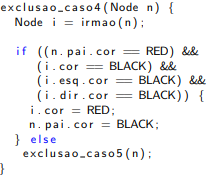
# Árvore Rubro-Negra - 1 Ponto

1. Escreva o pseudocódigo que corrige as eventuais violações da árvore rubro negra conhecidas como casos 1, 2 e 3.

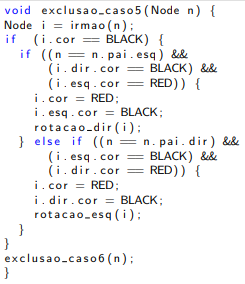


1. Escreva o pseudocódigo que corrige as eventuais violações da árvore rubro negra conhecidas como casos 4, 5 e 6.

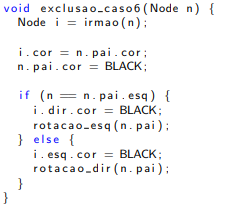
Casi 4:



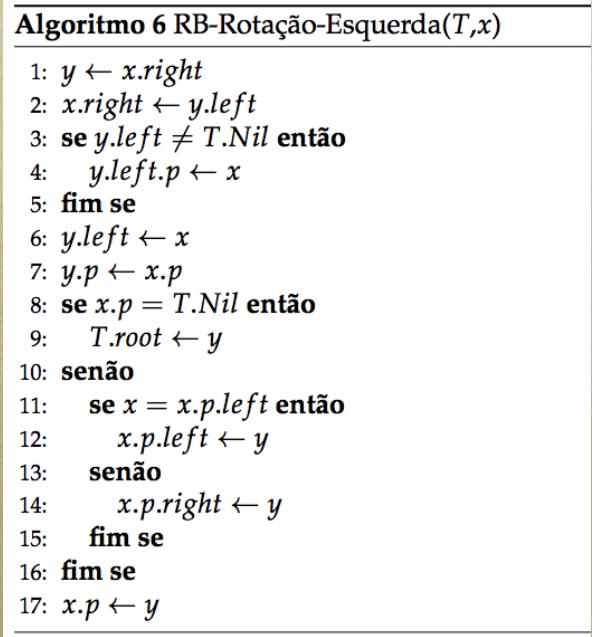
Caso 5:



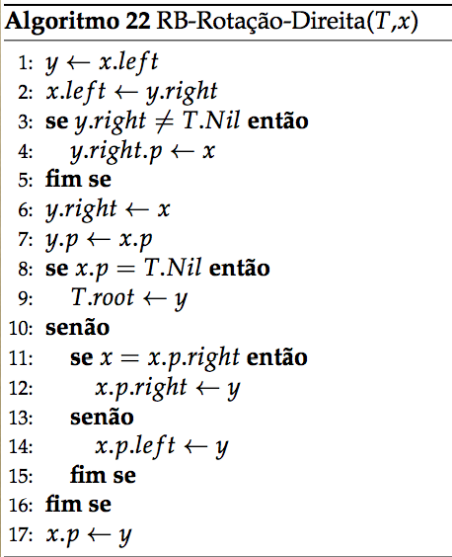
Caso 6:



1. Escreva o Código de Rotação à Esquerda de um nó da árvore.

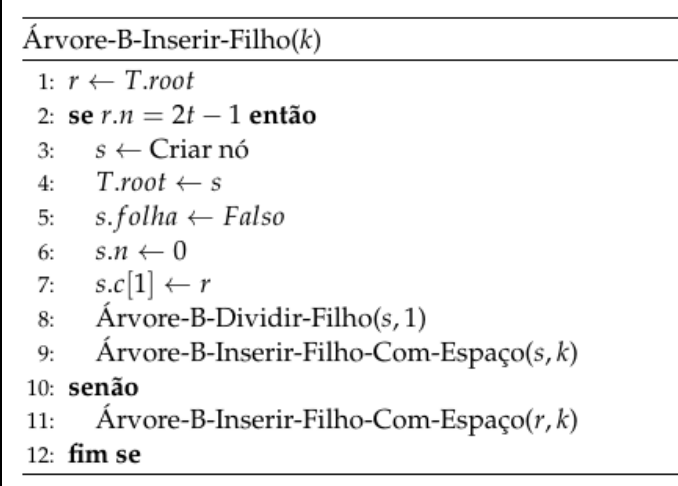


1. Escreva o Código de Rotação à Direita de um nó da árvore

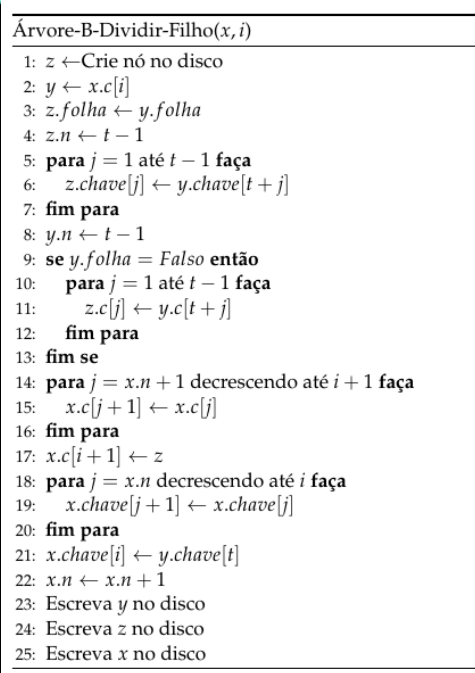


# Árvore B - 1 Ponto

1. Escreva o algoritmo de inserção em uma Árvore B.



1. Escreva o algoritmo que divide um filho em uma Árvore B.



# NP-Completude - 2 Pontos

1. De na os seguintes conjuntos:

**P;**

* Problemas de decisão para os quais conhecemos um algoritmo de tempo polinomial para resolvê lo
* Problemas de decisão nos quais conhecemos um algoritmo de tempo polinomial para resolvê-lo
* Um algoritmo é dito polinomial se é limitado por O(n^k) onde k é uma constante inteira
* A classe P consiste nos problemas que podem ser resolvidos em tempo polinomial. Mais especificamente, são problemas que podem ser resolvidos no tempo O(nk) para alguma constante k, onde n é o tamanho da entrada para o problema. A maioria dos problemas examinados em capítulos anteriores pertence à classe P.

**NP;**

* Problemas de decisão para os quais se uma máquina de Turing não determinística de tempo polinomial nos fornecer uma solução então podemos verificar se a solução é correta em tempo polinomial
* A atividade de verificar se uma candidata a solução do problema está certa ou não possa ser feito em tempo polinomial
* A classe NP consiste nos problemas que são “verificáveis” em tempo polinomial. O que significa um problema ser verificável? Se tivéssemos algum tipo de “certificado” de uma solução, poderíamos verificar se o certificado é correto em tempo polinomial para o tamanho da entrada para o problema. Por exemplo, no problema do ciclo hamiltoniano, dado um grafo dirigido G = (V, E), um certificado seria uma sequência 〈v1,v2,v3,...,v|V| de |V| vértices. É fácil verificar em tempo polinomial que (vi, vi+1) ∈ E para i = 1, 2, 3, ..., |V| – 1 e também que (v|V|, v1) ∈ E. Como outro exemplo; para satisfazibilidade 3-CNF um certificado seria uma atribuição de valores a variáveis. Poderíamos verificar em tempo polinomial que essa atribuição satisfaz a fórmula booleana.

**NP-Complete (NP-Completo);**

* Um problema Π é dito NP-Completo se:
  + Quaisquer instância de um problema da classe NP possa ser reduzida, transformada, em tempo polinomial, numa instância de Π
  + Contrapartida
    - Se for descoberto algum algoritmo em tempo polinomial que resolva Π então é possível resolver qualquer problema em NP em tempo polinomial
* NP-completo é um subconjunto de NP, o conjunto de todos os problemas de decisão cujas soluções podem ser verificadas em tempo polinomial; NP pode ser equivalentemente definida como o conjunto de problemas de decisão que podem ser solucionados em tempo polinomial em uma Máquina de Turing não determinística. Um problema p em NP também está em NPC Se e somente se todos os outros problemas em NP podem ser transformados em p em tempo polinomial.

**NP-Hard (NP-Dif cil ou N-Árduo)**

* NP-hard é a classe de problemas de decisão aos quais todos os problemas no NP podem ser reduzidos no tempo polinomial por uma máquina determinística de Turing. NP-complete é a interseção de NP-hard e NP.
* NP-difícil (ou NP-hard, ou NP-complexo) na teoria da complexidade computacional, é uma classe de problemas que são, informalmente, "Pelo menos tão difíceis quanto os problemas mais difíceis em NP". Um problema H é NP-difícil se e somente se (sse) existe um problema NP-completo L que é Turing-redutível em tempo polinomial para H (i.e., L?=?TH). Em outras palavras, L pode ser resolvido em tempo polinomial por uma Máquina de Turing não determinística com um oráculo para H. Informalmente, podemos pensar em um algoritmo que pode chamar tal Máquina de Turing Não-Determinística como uma sub-rotina para resolver H, e resolver L em tempo polinomial, se a chamada da sub-rotina leva apenas um passo para computar. Problemas NP-difíceis podem ser de qualquer tipo: problemas de decisão, problemas de pesquisa ou problemas de otimização.

1. Descreva três problemas NP-Completos.

**Problema do clique**

Em ciência da computação, o problema do clique refere-se a qualquer problema que possui como objetivo encontrar subgrafos completos ("cliques") em um grafo. Como exemplo, o problema de encontrar conjuntos de nós em que todos os elementos estão conectados entre si. Por exemplo, o problema clique surge no seguinte cenário. Considere uma rede social, onde os vértices do grafo representam pessoas, e as arestas representam o conhecimento mútuo. Para encontrar um maior subconjunto de pessoas, em que todas conhecem umas às outras, pode-se sistematicamente inspecionar todos os subconjuntos, um processo que é muito demorado para ser prático para as redes sociais, mesmo que pequenas. Embora a pesquisa por força bruta possa ser melhorada através de algoritmos mais eficientes, todos estes algoritmos levam tempo exponencial para resolver o problema. Portanto, grande parte da teoria sobre o problema do clique é dedicada à identificação de tipos especiais de grafo que admitem algoritmos mais eficientes, ou a definição da dificuldade computacional do problema geral em vários modelos de computação. Junto com seus aplicativos em redes sociais , o clique também tem muitas aplicações em bioinformática e química computacional. Problemas que envolvem o clique: encontrar o clique máximo (um clique com o maior número de vértices); encontrar o clique com maior valor em um grafo valorado; listar todos os cliques máximos (cliques que não podem ser ampliados); resolver o problema de decisão de testar se um grafo contém um clique maior que um tamanho determinado. Esses problemas são todos difíceis: o problema de decisão clique é NP-completo (um dos 21 problemas NP-Completo de Karp), e listar todos os cliques máximos pode exigir tempo exponencial. No entanto, existem algoritmos para esses problemas que são executados em tempo exponencial ou que lidam com grafos de entrada mais especializados em tempo polinomial.

**Problema da mochila**

O problema da mochila (em inglês, Knapsack problem) é um problema de optimização combinatória. O nome dá-se devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo. O problema da mochila é um dos 21 problemas NP-completos de Richard Karp, exposto em 1972. A formulação do problema é extremamente simples, porém sua solução é mais complexa. Este problema é a base do primeiro algoritmo de chave pública (chaves assimétricas).Normalmente este problema é resolvido com programação dinâmica, obtendo então a resolução exata do problema, mas também sendo possível usar PSE (procedimento de separação e evolução). Existem também outras técnicas, como usar algoritmo guloso, meta-heurística (algoritmos genéticos) para soluções aproximadas.

**Satisfazibilidade 2-CNF e satisfazibilidade 3-CNF**

Uma fórmula booleana contém variáveis cujos valores são 0 ou 1; conectivos booleanos como ∧ (AND), ∨ (OR) e ¬ (NOT); e parênteses. Uma fórmula booleana é satisfazível se existe alguma atribuição dos valores 0 e 1 às suas variáveis que faça com que ela seja avaliada como 1. Definiremos os termos em linguagem mais formal mais adiante neste capítulo; porém, informalmente, uma fórmula booleana está em forma normal k-conjuntiva, ou k-CNF, se for o AND de cláusulas OR de exatamente k variáveis ou de suas negações. Por exemplo, a fórmula booleana (x1 ∨ ¬x2) ∧ (¬x1 ∨ ¬x3) ∧ (¬x2 ∨ ¬x3) está em 2-CNF. (Ela tem a atribuição que satisfaz x1 = 1, x2 = 0, x3 = 1.) Embora possamos determinar em tempo polinomial se uma fórmula 2-CNF é satisfazível, veremos mais adiante neste capítulo que determinar se uma fórmula 3-CNF é satisfazível é NP-completo.