Análise e Projeto de Algoritmos

Prof. Eduardo Barrére

eduardo.barrere@ice.ufjf.br
www.barrere.ufjf.br

www.ufjf.br/pgcc
www.dcc.ufjf.br

A Classe de Problemas P

A classe de algoritmos **P** é formada pelos procedimentos para os quais existe um polinômio *p(n)* que limita o número de passos do processamento se este for iniciado com uma entrada de tamanho *n*.

- Um problema diz-se limitado polinomialmente se existe um algoritmo limitado polinomialmente que o resolve.
- A Classe P e constituída pelos problemas de decisão limitados polinomialmente.
- A classe P inclui todos os problemas razoáveis mas também problemas de difícil resolução (polinômios de crescimento muito rápido!)
- No entanto, e certo que um problema que não pertença a P será de resolução praticamente impossível ☹

A Classe de Problemas NP

Tipicamente um problema de decisão corresponde a obtenção de uma resposta para um problema de existência de um objeto, e uma solução para o problema corresponde a um tal objeto que justifica uma resposta verdadeira:

- Uma coloração de um grafo com k cores no máximo;
- Um circuito de Hamilton com peso ≤ k . . .

Uma solução proposta para um problema de decisão e um objeto do tipo procurado, mas sobre o qual não se sabe se e uma solução:

- Uma coloração qualquer do grafo;
- Um circuito qualquer de Hamilton no grafo.

A Classe de Problemas NP

Para cada problema faz sentido que exista um algoritmo que, dada uma solução proposta, verifica se ela é ou não solução do problema.

Uma solução proposta, por exemplo, será descrita por uma string de algum conjunto finito arbitrário de caracteres. A verificação da solução implica:

1. Verificar se a string obedece ao formato utilizado para descrever as soluções, ou seja, que e sintaticamente correta.
2. Verificar se a solução proposta descrita pela string verifica o critério do problema, ou seja é de fato uma solução.

Informalmente, a Classe NP é constituída pelos problemas de decisão em que a verificação de soluções pode ser feita em tempo polinomial (certificado em tempo polinomial).

Algoritmos Não-determinísticos

São algoritmos de aplicação teórica, utilizados para a classificação de problemas. Um algoritmo não-determinístico tem duas fases:

1. **Fase não-determinística**: é escrita em memória uma string arbitraria s. Em cada execução do algoritmo esta string pode ser diferente.

2. **Fase determinística**: o algoritmo agora lê s e processa-a, após pode suceder uma de três situações:

   (a) algoritmo para com resposta sim

   (b) algoritmo para com resposta não

   (c) algoritmo não para

- A primeira fase pode ser vista como uma “tentativa de adivinhar" uma solução.
- A segunda fase verifica se este “palpite" obtido na primeira fase corresponde ou não a uma solução para o problema.

Algoritmos Não-determinísticos

- A resposta de um algoritmo A não-determinístico a um input x define-se como sendo “sim” se existe uma execução de A sobre x que produz output “sim".

A resposta “sim" de A a x corresponde então a existência de uma solução para o problema de decisão com input x.

- O numero de passos de execução de um algoritmo ND corresponde a soma dos passos das duas fases.

A Classe de Problemas NP

- Um algoritmo ND A diz-se limitado polinomialmente se existe um polinômio P(x) tal que para cada input (de dimensão n) para o qual a resposta de A seja “sim", exista uma execução do algoritmo que produz output “sim" em tempo O(P(x)).

- A Classe NP e constituída por todos os problemas de decisão para os quais existe um algoritmo não-determinístico limitado polinomialmente.

[NP = Nondeterministic Polynomial-bounded]

- **Teorema:** Os problemas de Coloracão de Grafos, Bin Packing, Mochila, Caminhos e Ciclos de Hamilton, e Caixeiro Viajante são todos NP.

Exemplo de Problema de Decisão

Coloração de Grafos: existira uma coloração de G = (V,E) com k ou menos cores?

- formato das soluções propostas: strings contendo caracteres correspondendo a cores (R = vermelho, G = verde, . . .), encontrando-se na posição i da string a cor atribuída ao vértice de índice i no grafo.

Verificação de uma solução:

1. verificar que a string tem comprimento |V| e todos os caracteres correspondem a cores (verificação sintática);
2. percorrer as listas (ou matriz) de adjacências do grafo e verificar que todos os pares de vértices adjacentes têm cores diferentes;
3. verificar se a string contem no máximo k cores diferentes.

Exemplo : Coloração de Grafos

Seja G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 4), (2, 4), (2, 3), (3, 5), (2, 5), (3, 4), (4, 5)}).

Poderá G ser colorido com 4 cores (RGBY)?

- Considere um algoritmo não-determinístico que gera sucessivamente as seguintes soluções propostas. Quais são de fato soluções (i.e., qual o output do algoritmo em cada caso?)

RGRBG

RGRB

RBYGO

RGRBY

R%*,G@

P e NP

Evidentemente **P** está contido em **NP !!!!????**

Prova: Qualquer algoritmo determinístico para um problema de decisão é um caso particular de um algoritmo ND. Seja *A* um algoritmo determinístico para um problema p_0 pertencente a P. Então podemos construir um algoritmo ND *A'* a partir de *A*:

1. a fase não-determinística escreve s ="" em zero passos;
2. a fase determinística e constituída por *A* (ignorando a string s escrita pela primeira fase).

Sendo assim:

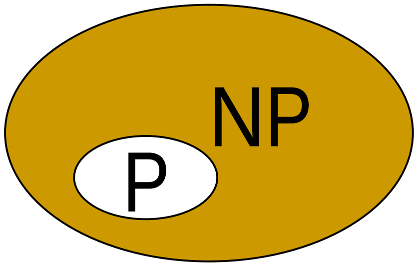
- *A'* dá a resposta sim ou não correta;
- *A* executa em tempo polinomial, logo *A'* executa também em tempo polinomial.
- Fica assim provado que p_0 pertence a NP.

P e NP

Será que NP está contido em P, ou seja P = NP ?

- Será que o não-determinismo é mais poderoso do que o determinismo?
- Em outras palavras: será que alguns problemas que não podem ser resolvidos em tempo polinomial por um algoritmo simples podem ser resolvidos em tempo polinomial por algoritmos contendo um “gerador de palpites" não-determinístico?

Imagina-se que NP seja muito maior do que P.



- Para todos os problemas NP apresentados anteriormente não são conhecidos algoritmos determinísticos limitados polinomialmente, no entanto para nenhum deles foi provado um limite $\Omega(f(n))$ com f(n) assintoticamente superior a qualquer polinômio.

A questão [P = NP ?] permanece aberta (e muito valiosa!)

P e NP

Resta-nos a estratégia **“força bruta"**: para qualquer problema NP, a resposta (sim / não) correta para o problema pode ser obtida deterministicamente:

1. Seja A um algoritmo ND para o problema, e p(n) o polinômio que o limita;
2. Cada string gerada pela primeira fase de A tem comprimento máximo p(n);
3. Se o conjunto de caracteres utilizado contiver c caracteres, existirão $c^{p(n)}$ strings possíveis - um numero exponencial em n.
4. Para resolver o problema, basta executar a segunda fase de A sucessivamente para cada string gerada na primeira fase, parando se se obtiver output “sim".

Assim, a estratégia “força bruta" resolve os problemas NP em tempo exponencial

O tamanho de um problema

Problema: Dado um inteiro positivo n, haverá dois inteiros j, k > 1 tais que n=jk ? (i.e, será n não-primo?)

Considere-se o seguinte algoritmo do tipo “forca bruta":

```
found = 0;
j = 2;
while ((!found) && j < n) {
if (n mod j == 0) found = 1;
else j++;
}
```

- Este algoritmo executa em tempo O(n), no entanto trata-se de um problema famoso pela sua dificuldade (utilizado em muitos algoritmos de criptografia).
- De fato é importante identificar corretamente o tamanho de n, uma vez que disso depende a classificação do algoritmo como polinomial ou exponencial.

Tamanho e Representação

- O tamanho de um número é o número de caracteres necessários para o escrever: o tamanho de 3500 é 4.

- Um inteiro n em notação decimal ocupa aproximadamente $\log_{10} n$ dígitos. Em notação binária ocupa $\log_2 n$ dígitos.

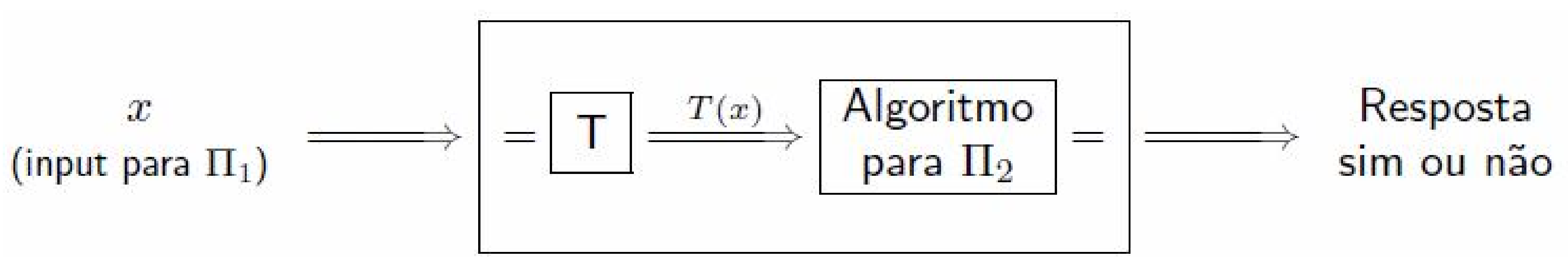
- Observe: se um algoritmo executa no pior caso em tempo linear n, e n tem tamanho $s = \log_k n$, então o algoritmo executa em tempo k^s, ou seja $T(s) = O(k^s)$.

Um algoritmo de tempo aparentemente linear e de fato de tempo exponencial! ☹

Redução de Problemas

Considere dois problemas π1 e π2.

- Dispomos por hipótese de um algoritmo para resolver π2, e de uma função de transformação de inputs T(), que transforma um input x para π1 num input T(x) para π2, obedecendo a seguinte restrição:
- A resposta correta para π1 com input x é *sim* se a resposta correta para π2 com input T(x) é *sim*.
- Por composição obtêm-se facilmente um algoritmo para π1:



- Nestas circunstâncias reduzimos π1 a π2.

Exemplo de Redução de Problemas

- π1: Dadas n variáveis booleanas, será que pelo menos uma delas tem valor verdadeiro?
- π2: Dados n inteiros, será positivo o maior deles?

T(): Seja $T(x_1, ... ,x_n) = y_1, ... ,y_n$ com $y_i = 1$ se x_i for verdadeiro, e $y_i = 0$ se x_i for falso.

- qualquer algoritmo para π2 resolve π1 quando aplicado a $T(x_1, ... ,x_n)$.

Redução Polinomiais

Uma função T() do conjunto de inputs de um problema de decisão π1 para o conjunto de inputs de um problema de decisão π2 diz-se uma redução polinomial de π1 em π2 se:

1. T() pode ser calculada em tempo polinomial; e

2. Para cada input x para π1, a resposta correta para π2 sobre T(x) e igual a resposta correta para π1 sobre x.

π1 diz-se polinomialmente redutível a π2 se existe uma redução polinomial de π1 em π2.

Redução Polinomiais

A ideia que se pretende exprimir e que π2 e pelo menos tão difícil de resolver como π1.

Teorema. Se π1 é polinomialmente redutível a π2 e π2 esta em P, então π1 esta em P.

Prova: Sejam p() o polinomio que limita o comportamento da função de redução T(), e q() o que limita o comportamento de um algoritmo para π2.

Se x for um input para π1 de tamanho n, então T(x) tem tamanho p(n) no máximo. Então o algoritmo para π2 executará no máximo q(p(n)) passos.

O tempo total dispendido é limitado polinomialmente por p(n)+q(p(n)).

Problemas NP-completos

Um problema π diz-se NP-completo se está em NP, e para qualquer outro problema π1 em NP se tem π1 polinomialmente redutível a π.

- Conclusões:

1. Para provar P=NP (o que e altamente improvável) bastaria provar que um problema NP-completo pode ser resolvido por um algoritmo limitado polinomialmente.
2. Como e improvável que seja P=NP, e também improvável que tal algoritmo exista!

Os problemas *intratáveis pertencem à classe NP (não-polinomiais). Dentro da classe NP estão os problemas NP-completo que são os mais interessantes, pois não se conhece uma solução determinística capaz de ser executada em tempo polinomial. Mais do que isso, um problema NP-completo pode ser reduzido (transformado) a outro problema NP-completo qua*lquer.

Problemas NP-completos

Para provar que um problema π é NP-completo basta mostrar que qualquer problema em NP é polinomialmente redutível a π.

Por exemplo, para mostrar que o problema de coloração de grafos com 4 cores é NP-completo, podemos mostrar que para qualquer outro problema π1 em NP, existe uma função T() tal que:

- T() transforma em tempo polinomial um input x de π1 num grafo G;
- Este grafo descreve (num sentido informal) a computação de um algoritmo ND para π1, atuando sobre o input x;
- G poderá ser colorido com 4 cores se a computação acima resultar em *sim*.

Problemas NP-completos

No entanto, depois de provado que um qualquer problema e NP-completo (pelo método anterior), surge outro método:

Teorema: Para provar que um problema π em NP é NP-completo basta mostrar, para outro problema NP-completo π1 conhecido, que π1 é polinomialmente redutível a π .

Prova: Como π1 é NP-completo, todos os problemas de NP são polinomialmente redutíveis a π1. Se provarmos π1 é polinomialmente redutível a π, teremos por transitividade que todos os problemas NP são polinomialmente redutíveis a π, logo π é NP-completo.

Teorema: Todos os problemas apresentados neste capitulo são NP-completos.

Exemplo

Baseado no problema dos Circuitos de Hamilton para grafos orientados (HO) e NP-completo, desejamos agora provar que o mesmo problema, mas para grafos não-orientados (HNO), é tambem NP-completo. Basta provar que HO é polinomialmente redutível a HNO.

Seja G = (V,E) orientado, e G' = (V',E') com

- $V' = \{v^i \mid v \in V, i = 1, 2, 3\}$
- $E' = \{(v^1, v^2), (v^2, v^3) \mid v \in V\}$

A função T : G -> G' constrói um grafo com 3|V| vértices e 2|V| + |E| arcos, executando em tempo polinomial.

É fácil provar que G tem um circuito de Hamilton (orientado) se G' tem um circuito de Hamilton (não-orientado). Logo T() é uma redução polinomial de HO em HNO.

Restrições sobre os Problemas

A introdução de restrições pode simplificar muito (ou não!) um problema.

Por exemplo: se restringirmos os problemas sobre grafos a grafos de Grau máximo 2 (nenhum vértice tem mais de dois arcos), os problemas dos circuitos de Hamilton e da coloração são resolúveis em tempo polinomial.

- Para o grau máximo 3 o primeiro torna-se NP-completo; para o grau máximo 4 o segundo torna-se também NP-completo.
- O problema de decisão de coloração só é NP-completo para $k \geq 3$ cores. Para $k \leq 2$ cores, a resolução e fácil.

Semelhanças Aparentes

Alguns problemas aparentemente parecidos podem ter complexidades muito diferentes:

- O problema do caminho mais curto entre dois vértices esta em P; no entanto o do caminho mais longo é NP-completo.
- Circuito de Euler: ciclo que atravessa cada arco de um grafo orientado e ligado exatamente uma vez. Pode ser determinado (ou provada a sua não existência) em tempo linear em |E|. Mas a determinação de circuitos de Hamilton é NP-completa.

Problemas de Decisão vs. de Otimização

Todo o estudo dos problemas NP e NP-completos foi efetuado para problemas de decisão, para os quais a formalização é mais fácil.

Claramente o problema de otimização é de resolução mais difícil do que o correspondente problema de decisão (basta observar que a partir da solução do primeiro se obtém facilmente a solução do segundo).

- Por exemplo, conhecendo o número mínimo de cores com que se pode colorir um grafo G, é trivial responder a questão “poderá G ser colorido com k cores?" para qualquer k.
- Se P=NP, ou seja se existissem algoritmos polinomiais para os problemas NP de decisão, poder-se-ia em muitos casos obter algoritmos polinomiais para os correspondentes problemas de otimizacão.

- Como?

Resolução de Problemas NP-completos

Estratégias possíveis:

- Escolher o mais eficiente dos algoritmos exponenciais ...
- Concentrar a escolha na analise de caso médio em vez de pior caso.
- Em particular, um estudo dos padrões de inputs que ocorrem com mais frequência pode levar a escolha de um algoritmo que se comporte melhor para esses inputs.
- Escolha pode depender mais de resultados empíricos do que de uma análise rigorosa.
- Estratégia alternativa: Algoritmos de Aproximação.

Algoritmos de Aproximação ou Heurísticos

Princípio: utilizar algoritmos rápidos (da classe P) que não produzem garantidamente uma solução ótima, mas sim próxima da solução ótima.

- Muitas heurísticas utilizadas são simples e eficientes, resultando apesar disso em soluções muito próximas da ótima.
- A definição de “proximidade à solução ótima" depende do problema.
- Uma solução aproximada para, por exemplo, o problema do caixeiro viajante, não e uma solução que passa por “quase todos" os vértices do grafo, mas sim uma solução que passa por todos, e cujo peso é próximo do mínimo possível.
- Por outras palavras, as soluções ótimas devem sempre ser soluções propostas por alguma execução de um algoritmo não-determinístico para o problema.

Exemplo de Algoritmo Heurístico para “Mochila"

Distribuir n objetos de dimensões s_1, ..., s_n, com $0 < s_i \leq 1$ pelo número mínimo de gavetas de dimensão 1.

- **Estratégia ótima**: considerar todas as distribuições possíveis (número máximo de gavetas = n). O número de soluções é naturalmente exponencial em n.
- **Estrategia First Fit**: colocar cada objeto na primeira gaveta em que ele couber.
- **Estratégia First Fit Decreasing**: ordenar os objetos por dimensão decrescente antes de aplicar a estratégia “First Fit".

“First Fit": Algoritmo Detalhado

```
int FF (float S[], int n, int bin[])
{
  float used[n];
  int i; /* objetos */
  int j; /* gavetas */
  int m = 0; /* número necessário de gavetas
     */
  for (j=1 ; j<=n ; j++) used[j] = 0;
  for (i=1 ; i<=n ; i++)
  {
     j = 1;
     while (used[j]+S[i] > 1) j++;
     bin[i] = j;
     if (j>m) m=j;
     used[j] = used[j]+S[i];
   }
 return m;
}
```

Observações

- Tempo de execução de “First Fit": $\theta(n^2)$.
- Qual o tempo de execução de FFD?

- **Teorema [limite superior]**: o número de gavetas usadas por FFD nunca excede em mais de 22.

- No entanto o algoritmo comporta-se geralmente muito melhor do que indicado por este limite. Estudos empíricos sobre inputs grandes (com uma distribuição uniforme dos tamanhos) mostram que o número de gavetas extra é aproximadamente $0,3n^{½}$