Algoritmos e Complexidade: Princípios

Resolução de Problemas e Busca

Sumário

Algoritmos e Complexidade:

- ♢ Técnicas para Avaliar Algoritmos
- ♢ Análise Assintótica
- ♢ Classes de Problemas

Resolução de Problemas e Busca:

- ♢ Agentes que resolvem problemas
- ♢ Tipos de problemas
- ♢ Formulação de problemas
- ♢ Exemplos de problemas
- ♢ Algoritmos de busca básicos

Técnicas para Avaliar Algoritmos

Como avaliar quão bom ou ruim é um algoritmo?

Duas técnicas:

- Comparação em problemas *benchmark* (padrão). Executo algoritmo a ser testado em um problema-padrão, e comparo seu desempenho com um algoritmo de referência.

  Problema: muito específico. Desempenho pode depender do compilador utilizado, ajuste fino de parâmetros, arquitetura da máquina usada, etc.

- Análise de Algoritmo. Analiso algoritmo em termos de parâmetros característicos, abstraídos de aspectos implementacionais. Evita especificidade.

Análise de Algoritmos: Um Exemplo

```
function SUMMATION( sequence) returns a number
   sum ← 0
   for i ← 1 to LENGTH(sequence)
      sum ← sum + sequence[i]
   end
 return sum
```

Caracterização da entrada (tamanho): comprimento de *sequence* (n).

Caracterização de medida $T(n)$ para tempo de execução: somas, atribuições, etc... No nosso caso, $T(n) = 3n + 1$ (considerando somas e atribuições apenas).

$\diamondsuit$ Em geral, é difícil caracterizar um $T(n)$ único. Frequentemente, calcula-se um pior $T(n)$ (caso-limite). Ainda assim, análise exata pode ser impossível. Me contento com *aproximação assintótica*: para valores de n altos, desprezo termos de ordem baixa. No nosso exemplo, algoritmo é de $O(n)$.

Um algoritmo $O(n)$ é (normalmente) melhor do que um algoritmo $O(n^2)$, mas compare um $T(n^2+1)$ e um $T(100n+1000)$ ... Apesar disso, **análise assintótica é um bom compromisso** entre precisão e facilidade de análise.

Classes de Problemas

Análise assintótica: OK para medir eficiência de algoritmo, mas como descobrir se pode existir um algoritmo melhor para um dado problema? **Análise da Complexidade** estuda problemas ao invés de algoritmos.

Divisão básica:

- Problemas P: Podem ser resolvidos em tempo polinomial. Exemplo: problemas que podem ser resolvidos em tempos $O(logn)$, $O(n)$, $O(n^{1000})$...
- Problemas não-P: Não podem ser resolvidos em tempo polinomial, qualquer que seja o algoritmo usado.
- Problemas NP: aqueles para os quais um algoritmo pode "chutar" solução e *verificá-la* em tempo polinomial. Se meu algoritmo tem sorte e sempre adivinha a solução do problema na primeira tentativa, ou se tenho processamento paralelo que permita tentar todas as possíveis soluções ao mesmo tempo, em que transformo um problema NP?

Isto caracteriza *problemas*. Como achar o algoritmo é problema seu...

P = NP?

É uma das grandes questões abertas em Teoria da Computação. A maioria dos cientistas acha que não. . . .

Classe NP completa: subclasse dos problemas NP mais difíceis. Está provado que:

a) ou todo problema NP-completo é P ou

b) nenhum é.

Interesse prático, pois muitos problemas reais são NP-completo. Enquanto esta questão estiver aberta, um otimista deve acreditar em a) e ter esperança de encontrar algoritmos que resolvam problemas de alta complexidade em tempo polinomial. . .