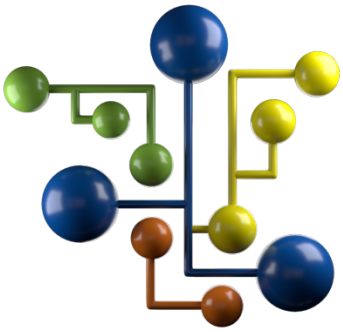

Data Science Academy

www.datascienceacademy.com.br

Introdução à Inteligência Artificial

Análise de Complexidade

Os cientistas de computação frequentemente se encontram diante da tarefa de comparar algoritmos para ver o quanto eles são rápidos ou a quantidade de memória que eles exigem. Existem duas abordagens para desempenhar essa tarefa. A primeira é o benchmarking — a execução dos algoritmos em um computador e a medição da velocidade em segundos e do consumo de memória em bytes. Em última instância, é isso o que realmente importa, mas um benchmarking pode ser insatisfatório devido ao fato de ser muito específico: ele mede o desempenho de um programa específico escrito em uma linguagem específica, funcionando em um computador específico, com um compilador específico e com dados de entrada específicos. A partir do único resultado que o benchmarking fornece, talvez seja difícil prever o quanto o algoritmo se comportaria bem em um compilador, computador ou conjunto de dados diferente. A segunda abordagem baseia-se em uma análise de algoritmos matemáticos, independentemente da execução específica e da entrada, como discutido a seguir.

Examinaremos a abordagem por meio do exemplo a seguir, um programa para calcular a soma de uma sequência de números:

```
função SOMATÓRIO(sequência) retorna um número
  soma ← 0
  para i = 1 até COMPRIMENTO(sequência) faça
      soma ← soma + sequência[i]
  retornar soma
```

A primeira etapa da análise consiste em realizar uma abstração sobre a entrada, a fim de encontrar algum parâmetro ou parâmetros que caracterizem o tamanho da entrada. Nesse exemplo, a entrada pode ser caracterizada pelo comprimento da sequência, que chamaremos n. A segunda etapa consiste em realizar uma abstração sobre a implementação, com o objetivo de encontrar alguma medida que reflita o tempo de execução do algoritmo, mas que não esteja ligada a um compilador ou computador específico. No caso do programa SOMATÓRIO, isso poderia ser apenas o número de linhas de código executadas ou talvez pudesse ser mais detalhado, medindo o número de adições, atribuições, referências a vetores e desvios executados pelo algoritmo. De qualquer modo, isso nos dá uma caracterização do número total de passos executados pelo algoritmo como uma função do tamanho da entrada. Chamaremos essa caracterização de T(n). Se contarmos as linhas de código, teremos T(n) = 2n + 2 em nosso exemplo.

Se todos os programas fossem tão simples quanto o SOMATÓRIO, a análise de algoritmos seria um campo trivial. Porém, dois problemas a tornam mais complicada. Primeiro, é raro encontrar um parâmetro como n que caracterize completamente o número de passos executados por um algoritmo. Em vez disso, em geral o melhor que podemos fazer é calcular o Tpior(n) do pior caso ou o Tmédio(n) do caso médio. Calcular uma média significa que a análise deve pressupor alguma distribuição nas entradas.

O segundo problema é que os algoritmos tendem a resistir à análise exata. Nesse caso, é necessário recuar até uma aproximação. Dizemos que o algoritmo SOMATÓRIO é O (n), o que significa que sua medida é, no máximo, uma constante vezes n, com a possível exceção de alguns valores pequenos de n. Mais formalmente,

$$T(n) \text{ é } O(f(n)) \text{ se } T(n) \leq kf(n) \text{ para algum } k, \text{ para todo } n > n_0$$

A notação O( ) nos fornece aquilo que se denomina análise assintótica. Podemos afirmar sem dúvida que, à medida que n se aproxima assintoticamente de infinito, um algoritmo O(n) é melhor que um algoritmo O(n2). Um único valor de benchmarking poderia não substanciar tal afirmação. A notação O( ) realiza uma abstração sobre fatores constantes, o que a torna mais fácil de usar, embora menos precisa que a notação T( ). Por exemplo, um algoritmo O(n2) sempre será pior que um algoritmo O(n) em longo prazo, mas, se os dois algoritmos forem T(n2 + 1) e T(100n + 1000), o algoritmo O(n2) será de fato melhor para n < 110.

Apesar dessa desvantagem, a análise assintótica é a ferramenta mais amplamente utilizada para análise de algoritmos. É precisamente porque a análise realiza a abstração sobre o número exato de operações (ignorando o fator constante k) e sobre o conteúdo exato da entrada (considerando apenas seu tamanho n) que a análise se torna matematicamente possível. A notação O( ) é um bom compromisso entre precisão e facilidade de análise.

Problemas NP e inerentemente difíceis

A análise de algoritmos e a notação O( ) nos permitem abordar a eficiência de um algoritmo específico. Porém, elas não têm nenhuma relação com o fato de ser ou não possível existir um algoritmo melhor para determinado problema. O campo da análise de complexidade analisa problemas em vez de algoritmos. A primeira divisão bruta se dá entre problemas que podem ser resolvidos em tempo polinomial e problemas que não podem ser resolvidos em tempo polinomial, não importando que algoritmo seja usado. A classe de problemas polinomiais — aqueles que podem ser resolvidos no tempo O(nk) para algum k constante — é chamada P. Às vezes, esses problemas se denominam problemas “fáceis” porque a classe contém os problemas com tempos de execução semelhantes a O(log n) e O(n). Porém, ela também contém os problemas com tempo O(n1000) e, assim, o adjetivo “fácil” não deve ser considerado de forma muito literal.

Outra classe importante de problemas é a classe NP, de problemas polinomiais não determinísticos. Um problema está nessa classe se existe algum algoritmo que possa pressupor uma solução e depois verificar se o palpite está correto em tempo polinomial. A ideia é que, se você tiver um número arbitrariamente grande de processadores, de forma que possa experimentar todos os palpites ao mesmo tempo ou, se tiver muita sorte e sempre acertar o palpite na primeira vez, os problemas NP se tornarão problemas P. Uma das maiores questões em aberto em ciência da computação é se a classe NP é equivalente à classe P quando não se

tem o luxo de um número infinito de processadores ou conjetura onisciente. A maioria dos cientistas da computação está convencida de que P ≠ NP — de que os problemas NP são inerentemente difíceis e não possuem nenhum algoritmo de tempo polinomial, embora isso nunca tenha sido demonstrado.

As pessoas interessadas em decidir se P = NP examinam uma subclasse de NP chamada problemas NP-completos. A palavra “completos” é usada nesse caso no sentido de “mais extremo” e, portanto, se refere aos problemas mais difíceis da classe NP. Foi demonstrado que todos os problemas NPcompletos estão em P ou nenhum deles está. Isso torna a classe teoricamente interessante, mas a classe também tem interesse prático porque muitos problemas importantes são reconhecidos como NP-completos. Um exemplo é o problema de satisfatibilidade: dada uma sentença de lógica proposicional, existe uma atribuição de valores-verdade para os símbolos de proposições da sentença que a torne verdadeira? A menos que ocorra um milagre e P = NP, não pode haver nenhum algoritmo que resolva todos os problemas de satisfabilidade em tempo polinomial. No entanto, a IA está mais interessada em descobrir se existem algoritmos que funcionam com eficiência em problemas típicos extraídos de uma distribuição predeterminada.

Referências:

Livro: Inteligência Artificial
Autor: Peter Norvig