Técnicas algoritmicas

Carlos Tavares

9 de Dezembro de 2024

Algoritmos... i

A noção de algoritmo: um processo muito bem definido de modo a que uma máquina pode compreender e executar.

É uma noção que temos vindo repetidamente a revisitar ao longo do curso: cada solução de um problema é um algoritmo.

A noção de algoritmo é muito vasta e quase toda as àreas da ciência têm algoritmos para alguma coisa.

Algoritmos... ii

Há no entanto, um conjunto de algoritmos que são considerados "cultura geral"em ciência de computadores

Referência: The Art of Computer Programming- Donald Knuth, 12 capítulos, 7 Volumes

- Geometria computacional;
- Processamento de texto;
- Métodos numéricos;
- **Pesquisa em espaços ordenados**;
- **Algoritmos combinatórios, pesquisa em espaços combinatórios**
    - **Permutações e combinações**;
    - Grafos;

Algoritmos... iii

É sempre possível reduzir um problema de computação a outro problema computacional mais complexo;

Pesquisa **força-bruta** é uma solução universal para todos os problemas e surpreendentemente fácil de implementar em muitas situações.

Busca, Otimização e Contagem i

Problemas de busca e otimização estão em toda parte na ciência da computação:

Busca (B): qual elemento do espaço de busca constitui a solução.

Otimização (O): quais elementos do espaço de busca constituem a **melhor** solução.

Contagem (C): Quantos elementos do espaço de busca são uma solução para o problema.

Busca, Otimização e Contagem ii

Tipicamente, podem ser formulados como instâncias uns dos outros, mas os problemas de otimização são geralmente mais difíceis do que os de busca, e os de contagem são mais difíceis do que os de otimização.

$$B \subseteq O \subseteq C$$

Exemplo: encontrar um elemento numa lista.

Busca, Otimização e Contagem iii

Meta-algoritmos para problemas de busca, otimização e contagem: **força bruta**, percorra todo o (espaço de busca) até encontrar (uma solução/a melhor solução/todas as soluções).

Problema de Busca:

```
def is_solution (possible_solution):
    # retorna verdadeiro se possible_solution é a solução do problema.

def search ():
    for i in solution_space:
        if is_solution(i) == True:
            return i
```

Busca, Otimização e Contagem iv

Problema de Otimização

```
def evaluate (possible_solution):
    # retorna uma pontuação para possible_solution.

def search():
    for i in solution_space:
        best = 0
        if evaluate(i) > best:
            best = i
    return best
```

Contagem

```
def is_solution (possible_solution):
    # retorna verdadeiro se possible_solution é solução do problema.

def count ():
    count = 0
    for i in solution_space:
        if is_solution(i):
            count = count + 1
    return count
```

Explorando Espaços de Busca i

O primeiro passo é identificar o espaço de busca e encontrar uma forma de o percorrer.

Busca Linear: O(N)

Exemplo: encontrar um item numa coleção de elementos.

Busca Quadrática: O(N^2)

Exemplo: encontrar o par de números que resulta numa certa soma, ou encontrar um par de pontos que tenha uma certa distância.

Explorando Espaços de Busca ii

Solução:

```
for i in collection:
    for j in collection:
        verify_solution(i, j)
```

Busca Cúbica: O(N^3)

Exemplo: Encontrar triângulos.

Solução:

```
for i in collection:
    for j in collection:
        for h in collection:
            verify_solution(i, j, h)
```

Explorando Espaços de Busca iii

Pode haver ordens ainda mais altas, como N^4, N^5, ...

Há exemplos para todas as ordens, mas ciclos aninhados não são muito interessantes.

Como podemos construir espaços de busca de forma elegante? Use Python **itertools**!

import itertools as it

Similar a ciclos aninhados:
it.product (Collection, repeat=N)

Explorando Espaços de Busca iv

Como usar isto para uma *busca força bruta* (para *N* dimensões)?

```
for i in itertools.product (collection, repeat=N):
    verify_solution(i)
```

Exemplo: encontrar pares de números que resultam numa certa soma:

```
for i in itertools.product (collection, repeat=2):
    if i[0] + i[1] == sum:
        return i
```

itertools.product também é um construtor preguiçoso!

Outros Espaços de Busca i

Combinações:

itertools.combinations (collection, length)

Busca força bruta sobre todas as combinações:

```
for i in itertools.combinations (collection, 2):
    verify_solution(i)
```

Outros Espaços de Busca ii

Permutações:

itertools.permutations (collection)

```
for i in itertools.permutations(collection):
    verify_solution(i)
```

Alguns pequenos exercícios.

Exercício 1. Encontre uma linha com uma certa norma a partir de um conjunto de pontos.

Exercício 2. Encontre todos os subconjuntos de um conjunto.

Exercício 3. Quebre uma palavra-passe de n caracteres ASCII.

Exercício 4. Dado um conjunto de ações com certo risco e vantagem associados, determine qual é o melhor conjunto de n ações com a maior vantagem e risco não superior a x.

Exercício 5. Dado um conjunto de cidades, verifique qual é o trajeto que passa por todas as cidades com o menor custo (Problema do Caixeiro Viajante).

Também e possivel explorar o espaço de soluções, partindo a solução em partes e ir compondo a solução final à medida que o progresso é feito. Quando uma parte da solução não se revela boa, podemos voltar atrás e voltar a tentar por outro caminho (backtracking).

Exercicio. Implemente um programa para resolver um labirinto.

Ordenação

Ordenação i

Às vezes, o espaço de busca possui uma estrutura. Sabemos que a solução não está em certas partes do espaço de solução.

Uma dessas possíveis estruturas ocorre se o espaço de busca está ordenado. Exemplo: encontrar a soma máxima de três elementos numa lista.

l = [2, 7, 5, 8, 9, 3, 4, 1]

Busca força bruta: N^3

Se estiver ordenada: (?)

l = [1, 2, 3, 4, 5, 7, 8, 9]

Ordenação ii

A soma mais alta corresponde à soma dos três últimos elementos (24):

l = [1, 2, 3, 4, 5, **7, 8, 9**]

Isto é equivalente ao problema de encontrar os **k** elementos mínimos (ou máximos).

Sabemos que podemos encontrar o mínimo em N, e podemos encontrar os k mínimos em $k \cdot N$ tempo.

Quando devemos usar ordenação?

Qual é o custo da ordenação?

Há muitos algoritmos de ordenação:

- Bubble sort (Custo: N^2);
- Quicksort, Mergesort (Custo: $N \log N$);
- Muitos outros... com pequenas vantagens.

Então, assumindo quicksort, quando é vantajoso usá-lo em vez de busca linear para k-mínimos?

Ordenação no Python:

l.sort() # onde l é uma lista.
l.sort(reverse=True|False, key=my_function)

Busca em Espaços Ordenados i

A busca em espaços ordenados é geralmente mais fácil do que em espaços não ordenados: podemos encontrar um elemento em $O(\log N)$. Exemplo:

Encontrar 2 em $\{1, 2, 3, 4, 5\}$:
Escolha o elemento do meio: 3.
Elemento é menor que 3, então reinicie o algoritmo com a parte esquerda de 3.

Encontrar 2 em $\{1, 2\}$:
Elemento do meio: 1.
...

Encontrar 2 em $\{2\}$:
Elemento encontrado!

Busca em Espaços Ordenados ii

Este tipo de implementação pode ser feito usando busca binária!

Busca em Espaços Ordenados iii

```
def binary_search_recursive(lst, target):
    if len(lst) == 0:
        return -1

    mid = len(lst) // 2

    # Verifica se o elemento está no meio
    if lst[mid] == target:
        return mid

    # Se o elemento for menor que o meio, ele está no subarray esquerdo
    elif lst[mid] > target:
        return binary_search_recursive(lst[:mid], target)
```

Busca em Espaços Ordenados iv

```
# Caso contrário, o elemento está no subarray direito
else:
    return binary_search_recursive(lst[mid + 1:], target)
```

Exercício. Use o algoritmo acima para fazer uma busca binária numa lista de números.

Exercício. Usando o facto de que um conjunto de pontos pode ser ordenado por uma das suas coordenadas, forneça um algoritmo para encontrar um segmento de linha num conjunto de pontos.

Perguntas?