

Branch and Bound e Backtracking

Lucas Saliba Lucas Santiago de Oliveira

Novembro de 2021

Sumário

1	<i>Branch and Bound</i>	2
1.1	Conceito	2
1.2	Quando usar	2
1.3	Exemplo	2
2	<i>Backtracking</i>	2
2.1	Conceito	2
2.2	Quando usar	2
2.3	Exemplo	2
3	Relação entre <i>Branch and Bound</i> e <i>Backtracking</i>	6
4	Relação com a Abordagem Gulosa	6
5	Relação com Programação Dinâmica	6
6	Relação com Divisão e Conquista	7

Resumo

Para começar falando sobre os algoritmos de otimização de problemas, começarei falando sobre o que é um problema linear na programação. Se considerarmos que exista um problema X que leva Y de tempo para ser resolvido e que Y cresça diretamente proporcional ao crescimento de X , então temos um problema linear. A ideia de todos os algoritmos que serão citados nesse documento é fazer Y crescer mais lentamente. Dessa forma, se o problema X dobrar de tamanho Y não dobrará de tamanho junto.

1 *Branch and Bound*

1.1 Conceito

Branch and Bound ou ramificar e limitar é um estilo de algoritmo que aplica otimizações para tornar problemas lentos em problemas rápidos. A ideia desse estilo de algoritmo é separar o problema em problemas menores e

1.2 Quando usar

1.3 Exemplo

2 *Backtracking*

2.1 Conceito

2.2 Quando usar

2.3 Exemplo

Explicação do problema Ciclo Hamiltoniano

O caminho hamiltoniano em um gráfico não direcionado é um caminho que visita cada vértice exatamente uma vez. Um ciclo hamiltoniano é um caminho hamiltoniano tal que existe uma aresta (no gráfico) do último vértice ao primeiro

vértice do caminho hamiltoniano. Determine se um determinado gráfico contém Ciclo Hamiltoniano ou não. Se contiver, imprime o caminho. A seguir estão as entradas e saídas da função necessária.

Entrada do problema

Um gráfico de matriz 2D $[V][V]$ onde V é o número de vértices no gráfico e o gráfico $[V][V]$ é a representação da matriz de adjacência do gráfico. Um gráfico de valor $[i][j]$ é 1 se houver uma borda direta de i para j , caso contrário, gráfico $[i][j]$ é 0.

Saida do problema

Um caminho de matriz $[V]$ que deve conter o Caminho Hamiltoniano. o caminho $[i]$ deve representar o i^o vértice no caminho hamiltoniano. O código também deve retornar falso se não houver um ciclo hamiltoniano no gráfico.

Algoritmo Backtracking

Cria uma matriz de caminho vazia e adiciona o vértice 0 a ela. Adiciona outros vértices, começando do vértice 1. Antes de adicionar um vértice, verifica se ele é adjacente ao vértice adicionado anteriormente e se já não foi adicionado. Se encontrarmos tal vértice, adicionamos o vértice como parte da solução. Se não encontrarmos um vértice, retornamos falso.

Exemplo de algoritmo

```
1
2 # Algoritmo em python para resolver o problema do ciclo
   hamiltoniano com Backtracking
3
4 class Grafo():
5     def __init__(self, vertices):
6         self.grafo = [[0 for column in range(vertices)]
7                        for row in range(vertices)]
8         self.V = vertices
```

```

9
10 # Verifica se este vértice é um vértice adjacente do vértice
    adicionado anteriormente e não é incluído no caminho anterior
11
12 def isSafe(self, v, pos, caminho):
13
14     # Verifica se o vértice atual e o último vértice no caminho
    são adjacentes
15     if self.grafo[ caminho[pos-1] ][v] == 0:
16         return False
17
18     # Verifica se o vértice atual ainda não está no caminho
19     for vertex in caminho:
20         if vertex == v:
21             return False
22
23     return True
24
25 # Função recursiva para resolver o problema do ciclo
    hamiltoniano
26 def hamCycleUtil(self, caminho, pos):
27
28     # Caso base: se todos os vértices forem incluídos no
    caminho
29     if pos == self.V:
30
31         # O último vértice deve ser adjacente ao primeiro vé
    rtice no caminho para fazer um ciclo
32         if self.grafo[ caminho[pos-1] ][ caminho[0] ] == 1:
33             return True
34         else:
35             return False
36
37     # Tenta vértices diferentes como um próximo candidato no
    ciclo hamiltoniano.
38     # Não tentamos 0 como incluímos 0 como ponto de partida em
    hamCycle ()
39     for v in range(1,self.V):

```

```

40
41         if self.isSafe(v, pos, caminho) == True:
42
43             caminho[pos] = v
44
45             if self.hamCycleUtil(caminho, pos+1) == True:
46                 return True
47
48             #Remove o vértice atual se não leva a uma solução
49             caminho[pos] = -1
50
51         return False
52
53     def hamCycle(self):
54         caminho = [-1] * self.V
55
56         # Vamos colocar o vértice 0 como o primeiro vértice no caminho.
57         # Se houver um ciclo hamiltoniano então o caminho pode ser
58         # iniciado de qualquer ponto do ciclo, pois o gráfico não é
59         # direcionado
60         caminho[0] = 0
61
62         if self.hamCycleUtil(caminho,1) == False:
63             print ("A solucao nao existe \n")
64             return False
65
66         self.printSolution(caminho)
67         return True
68
69     def printSolution(self, caminho):
70         print ("A Solução Existe: Seguindo ",
71               "é um ciclo hamiltoniano")
72         for vertex in caminho:
73             print (vertex, end = " ")
74         print (caminho[0], "\n")
75
76     # Codigo do condutor

```

```

75 # Criacao do grafo 1 de teste
76 g1 = Grafo(5)
77 g1.grafo = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
78              [0, 1, 0, 0, 1],[1, 1, 0, 0, 1],
79              [0, 1, 1, 1, 0], ]
80
81 # Printa solucao
82 g1.hamCycle();
83
84 # Criacao do grafo 2 de teste
85 g2 = Grafo(5)
86 g2.grafo = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
87              [0, 1, 0, 0, 1], [1, 1, 0, 0, 0],
88              [0, 1, 1, 0, 0], ]
89
90 # Printa solucao
91 g2.hamCycle();

```

3 Relação entre *Branch and Bound* e *Backtracking*

4 Relação com a Abordagem Gulosa

5 Relação com Programação Dinâmica

Exemplo

```

1 # Código baseado na implementação da free code camp
2 # Referência: https://youtu.be/oBt53YbR9Kk?t=210
3 import time
4
5 # Código do fibonacci
6 def fib(numero):
7     if (numero <= 2):
8         return 1
9

```

```

10     return fib(numero-1) + fib(numero-2)
11
12 # Versão do código usando programação dinâmica com memoization
13 def fibMemo(numero, memo={}):
14     if (numero in memo):
15         return memo[numero]
16
17     if (numero <= 2):
18         return 1
19
20     memo[numero] = fibMemo(numero - 1, memo) + fibMemo(numero - 2,
21                                                         memo)
22     return memo[numero]
23
24 def timer(fib, num):
25     start = time.time()
26     fib(num)
27     end = time.time()
28     return end - start
29
30 # Escrevendo os tempos na tela
31 print(f'Fibonnaci de 10: {timer(fib, 10)}')
32 print(f'Fibonnaci de 10 memoization: {timer(fibMemo, 10)}')
33
34 print(f'Fibonnaci de 15: {timer(fib, 15)}')
35 print(f'Fibonnaci de 15 memoization: {timer(fibMemo, 15)}')
36
37 print(f'Fibonnaci de 20: {timer(fib, 20)}')
38 print(f'Fibonnaci de 20 memoization: {timer(fibMemo, 20)}')
39
40 print(f'Fibonnaci de 30: {timer(fib, 30)}')
41 print(f'Fibonnaci de 500 memoization: {timer(fibMemo, 500)}')

```

6 Relação com Divisão e Conquista