# Branch and Bound e Backtracking

Lucas Saliba    Lucas Santiago de Oliveira

Novembro de 2021

## Sumário

**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 $Y$ cresce diretamente proporcinal 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 separa o problema em problemas menores e tenta resolver esse subproblema da forma mais otimizada que conseguir. Caso não seja encontrado um algoritmo eficiente para resolver o problema, esse subconjunto (*disjoint set*) não será resolvido.

## 1.2 Quando usar

Ótimo uso para esse estilo de programação é em problemas de NP-Difícil [2] como:

- Problema de satisfatibilidade máxima
- Problema do caixeiro-viajante
- Filogenética computacional

Os três problemas tem o mesmo grande problema de não existir um algoritmo eficiente que o resolva diretamente. Dessa forma, *Branch and Bound* separaria esses problemas em problemas menores e tentaria resolver cada subconjunto de forma independente.

### 1.3 Exemplo

## 2 *Backtracking*

### 2.1 Conceito

### 2.2 Quando usar

### 2.3 Exemplo

#### Explicacao 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 iº 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

```python

# Algoritmo em python para resolver o problema do ciclo hamiltoniano com Backtracking

class Grafo():
    def __init__(self, vertices):
        self.grafo = [[0 for column in range(vertices)]
                            for row in range(vertices)]
        self.V = vertices

# Verifica se este vértice é um vértice adjacente do vértice adicionado anteriormente e não é incluído no caminho anterior

    def isSafe(self, v, pos, caminho):

        # Verifica se o vértice atual e o último vértice no caminho são adjacentes
        if self.grafo[ caminho[pos-1] ][v] == 0:
            return False

        # Verifica se o vértice atual ainda não está no caminho
        for vertex in caminho:
            if vertex == v:
                return False

        return True

    # Funcão recursiva para resolver o problema do ciclo hamiltoniano
    def hamCycleUtil(self, caminho, pos):

```

```python
        # Caso base: se todos os vértices forem incluídos no caminho
        if pos == self.V:

            # O último vértice deve ser adjacente ao primeiro vértice no caminho para fazer um ciclo
            if self.grafo[ caminho[pos-1] ][ caminho[0] ] == 1:
                return True
            else:
                return False

        # Tenta vértices diferentes como um próximo candidato no ciclo hamiltoniano.
        # Não tentamos 0 como incluímos 0 como ponto de partida em hamCycle ()
        for v in range(1,self.V):

            if self.isSafe(v, pos, caminho) == True:

                caminho[pos] = v

                if self.hamCycleUtil(caminho, pos+1) == True:
                    return True

                #Remove o vértice atual se não leva a uma solucão
                caminho[pos] = -1

        return False

    def hamCycle(self):
        caminho = [-1] * self.V

   # Vamos colocar o vértice 0 como o primeiro vértice no caminho. Se houver um ciclo hamiltoniano então o caminho pode ser iniciado de qualquer ponto do ciclo, pois o gráfico não é direcionado
        caminho[0] = 0

```

```
        if self.hamCycleUtil(caminho,1) == False:
            print ("A solucao nao existe \n")
            return False

        self.printSolution(caminho)
        return True

    def printSolution(self, caminho):
        print ("A Solução Existe: Seguindo ",
                  "é um ciclo hamiltoniano")
        for vertex in caminho:
            print (vertex, end = " ")
        print (caminho[0], "\n")

# Codigo do condutor

# Criacao do grafo 1 de teste
g1 = Grafo(5)
g1.grafo = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
            [0, 1, 0, 0, 1,],[1, 1, 0, 0, 1],
            [0, 1, 1, 1, 0], ]

# Printa solucao
g1.hamCycle();

# Criacao do grafo 2 de teste
g2 = Grafo(5)
g2.grafo = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
        [0, 1, 0, 0, 1,], [1, 1, 0, 0, 0],
        [0, 1, 1, 0, 0], ]

# Printa solucao
g2.hamCycle();
```

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

# 4 Relação com a Abordagem Gulosa

## Conceito

As abordagens gulosas são baseadas em tentar resolver um problema sem se importar com otimizações para que o resultado seja obtido rapidamente. O primeiro passo para conseguir uma solução eficiente para um problema é conseguir apenas uma solução para o problema, mesmo que não consiga rapidamente.

# 5 Relação com Programação Dinâmica

## Exemplo

Código de exemplo: [1]

```python
# Código baseado na implementacão da free code camp
# Referência: https://youtu.be/oBt53YbR9Kk?t=210
import time

# Código do fibonacci
def fib(numero):
    if (numero <= 2):
        return 1

    return fib(numero-1) + fib(numero-2)

# Versão do código usando programacão dinâmica com memoization
def fibMemo(numero, memo={}):
    if (numero in memo):
        return memo[numero]

    if (numero <= 2):
```

```
        return 1

    memo[numero] = fibMemo(numero - 1, memo) + fibMemo(numero - 2, memo)
    return memo[numero]

def timer(fib, num):
    start = time.time()
    fib(num)
    end = time.time()
    return end - start

# Escrevendo os tempos na tela
print(f'Fibonnaci de 10: {timer(fib, 10)}')
print(f'Fibonnaci de 10 memoization: {timer(fibMemo, 10)}')

print(f'Fibonnaci de 15: {timer(fib, 15)}')
print(f'Fibonnaci de 15 memoization: {timer(fibMemo, 15)}')

print(f'Fibonnaci de 20: {timer(fib, 20)}')
print(f'Fibonnaci de 20 memoization: {timer(fibMemo, 20)}')

print(f'Fibonnaci de 30: {timer(fib, 30)}')
print(f'Fibonnaci de 500 memoization: {timer(fibMemo, 500)}')
```

# 6 Relação com Divisão e Conquista

# Referências


[1] FreeCodeCamp, Dynamic Programing, acessado dia 27/11/2021.

[2] Wikipedia, NP-Hardness, acessado dia 27/11/2021.