Branch and Bound e *Backtracking*

Lucas Fonseca Saliba Lucas Santiago de Oliveira

Novembro de 2021

Sumário

1 *Branch and Bound* **3**

1.1 Conceito 3

1.2 Quando usar 3

1.3 Exemplo 4

2 *Backtracking* **9**

2.1 Conceito 9

2.2 Quando usar 9

2.3 Exemplo 10

2.4 Exemplo de algoritmo: 11

3 Abordagem Gulosa **14**

3.1 Conceito 14

3.2 Exemplo de código 14

3.3 Complexidade 15

4 Programação Dinâmica **15**

4.1 Conceito 15

4.2 Exemplo de código 15

4.3 Complexidade 16

5 Divisão e Conquista **16**

5.1 Conceito 16

5.2 Exemplo de algoritmo 18

5.3 Complexidade 20

6 Relação entre os algoritmos **20**

6.1 *Branch and Bound* e *Backtracking* 20

6.2 Abordagem gulosa, Programação Dinâmica e Divisão e Conquista 21

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 necessariamente de tamanho junto. Nesse trabalho iremos apresentar cinco tipos de algotimos para atingir esse objetivo. Com foco no *Branch and Bound* e no *Backtracking* mostraremos seus conceitos separadamente. Assim como, suas semelhanças e diferenças com outros algoritmos que tentam resolver o mesmo problema, mas com abordagens diferentes.

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*[1]) não será resolvido.

Ele é um paradigma de projeto de algoritmo geralmente usado para resolver problemas de otimização combinatória. Esses problemas são tipicamente exponenciais em termos de complexidade de tempo e podem exigir a exploração de todas as permutações possíveis no pior caso. *Branch and Bound* pode resolver esses problemas com relativa rapidez.

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

Vamos usar o problema da mochila abaixo para entender o *Branch and Bound*. "Dados duas matrizes de inteiros $val[0..n-1]$ e $wt[0..n-1]$ que representam valores e pesos associados a n itens, respectivamente. Descubra o subconjunto de valor máximo de $val[]$ de forma que a soma dos pesos deste subconjunto seja menor ou igual à capacidade W da mochila."

Existem diferentes abordagens para resolver o problema acima, porém a solução *Branch and Bound* é o método mais adequado quando os pesos dos itens não são inteiros.

Como encontrar o limite para cada nó da mochila? A ideia é usar o fato de que a abordagem *Greedy* fornece a melhor solução para o problema da mochila fracionária. Para verificar se um determinado nó pode nos dar uma solução melhor ou não, calculamos a solução ótima (por meio do nó) usando a abordagem *Greedy*. Se a solução computada pela abordagem *Greedy* em si é mais do que a melhor até agora, então não podemos obter uma solução melhor por meio do nó.

Passo a passo do algoritmo:

- Classificar todos os itens em ordem decrescente de razão de valor por unidade de peso para que um limite superior possa ser calculado usando o *Greedy Approach*.
- Inicializar o lucro máximo, $maxProfit = 0$.

- Criar uma fila vazia, Q.
- Criar um nó fictício da árvore de decisão e enfileire-o em Q. O lucro e o peso do nó fictício são 0.
- Fazer o seguinte enquanto Q não estiver vazio:
  I. Extrair um item de Q. Deixar o item extraído ser u.
  II. Calcular o lucro do nó do próximo nível. Se o lucro for maior que $maxProfit$, atualizar o $maxProfit$.
  III. Limite de cálculo do nó do próximo nível. Se o limite for maior do que $maxProfit$, adicione o nó do próximo nível a Q.
- Considerar o caso em que o nó do próximo nível não é considerado como parte da solução e adicione um nó à fila com o nível seguinte, mas peso e lucro sem considerar os nós do próximo nível.

A seguir está a implementação em C++ da ideia acima:

```
#include <bits/stdc++.h>
using namespace std;

// Estrutura para o item que armazena o peso e correspondente

// Valor do item
struct Item
{
  float peso;
  int valor;
};

// Estrutura do nó para armazenar informacões de decisão

// Arvore
struct Node
{
  // Nível do nó na árvore de decisão
```

```
19   // Lucro dos nós no caminho da raiz para este nó (incluindo este
       nó)
20   // Limite superior do lucro máximo na subárvore
21   int nivel, lucro, limite;
22   float peso;
23 };
24
25 bool cmp(Item a, Item b)
26 {
27   double r1 = (double)a.valor / a.peso;
28   double r2 = (double)b.valor / b.peso;
29   return r1 > r2;
30 }
31
32 // Retorna o limite do lucro na subárvore enraizada em u.
33 // Esta funcão usa principalmente a solucão Greedy para encontrar
       um limite superior no lucro máximo.
34
35 int limite(Node u, int n, int W, Item arr[])
36 {
37   // se o peso superar a capacidade da mochila, devolva 0 como
       limite esperado
38   if (u.peso >= W)
39     return 0;
40
41   // inicializar vinculado ao lucro pelo lucro atual
42   int limite_lucro = u.lucro;
43
44   // comece a incluir itens do índice 1 mais para o atual
45   int j = u.nivel + 1;
46   int pesoTot = u.peso;
47
48   // verificar a condicão do índice e a capacidade da mochila
49   while ((j < n) && (pesoTot + arr[j].peso <= W))
50   {
51     pesoTot += arr[j].peso;
52     limite_lucro += arr[j].valor;
53     j++;
```

```
54    }
55
56    //Se k não for n, inclua o último item parcialmente para o limite
         superior do lucro
57    if (j < n)
58      limite_lucro += (W - pesoTot) * arr[j].valor /
59                      arr[j].peso;
60
61    return limite_lucro;
62  }
63
64  // Retorna o lucro máximo que podemos obter com a capacidade W
65  int knapsack(int W, Item arr[], int n)
66  {
67    // classificando item com base no valor por unidade
68    sort(arr, arr + n, cmp);
69
70    // faca uma fila para atravessar o nó
71    queue<Node> Q;
72    Node u, v;
73
74    u.nivel = -1;
75    u.lucro = u.peso = 0;
76    Q.push(u);
77
78    //Extraia um por um item da árvore de decisão
79    // calcula o lucro de todos os filhos do item extraído e continue
         salvando maxProfit
80    int maxProfit = 0;
81    while (!Q.empty())
82    {
83      // Desenfileirar um nó
84      u = Q.front();
85      Q.pop();
86
87      // Se for um nó inicial, atribua o nível 0
88      if (u.nivel == -1)
89        v.nivel = 0;
```

```
90
91      // Se não houver nada no próximo nível
92      if (u.nivel == n - 1)
93        continue;
94
95      // Caso contrário, se não o último nó, então incremente o nível
        , calcular o lucro dos nós filhos.
96      v.nivel = u.nivel + 1;
97
98      // Tomando o item do nível atual adicionar atual peso e valor
        do nível para o nó u peso e valor
99      v.peso = u.peso + arr[v.nivel].peso;
100     v.lucro = u.lucro + arr[v.nivel].valor;
101
102     // Se o peso acumulado for menor que W e o lucro é maior do que
         o lucro anterior, atualizar maxprofit
103     if (v.peso <= W && v.lucro > maxProfit)
104       maxProfit = v.lucro;
105
106     // Obtenha o limite superior do lucro para decidir se deve
        adicionar v a Q ou não.
107     v.limite = limite(v, n, W, arr);
108
109     // Se o valor vinculado for maior do que o lucro, então apenas
        empurre na fila para mais consideração
110     if (v.limite > maxProfit)
111       Q.push(v);
112
113     //Faca a mesma coisa, mas sem tirar o item na mochila
114     v.peso = u.peso;
115     v.lucro = u.lucro;
116     v.limite = limite(v, n, W, arr);
117     if (v.limite > maxProfit)
118       Q.push(v);
119   }
120
121   return maxProfit;
122 }
```

```
123
124 // Programa para testar a funcão acima
125 int main()
126 {
127   // Peso da mochila
128   int W = 10;
129   Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100}, {5, 95}, {3,
        30}};
130   int n = sizeof(arr) / sizeof(arr[0]);
131
132   cout << "Lucro máximo possível = "
133        << knapsack(W, arr, n);
134
135   return 0;
136 }
```

2 *Backtracking*

2.1 Conceito

Backtracking é um algoritmo genérico para encontrar alguns problemas computacionais, que vai incrementando os candidatos para a solução e se em algum momento for encontrado que algum dos candidatos não resolve o problema, então vai acontecer um "*backtracking*". Um "*backtracking*" é o processo de retornar e ignorar essa solução, uma vez que ela não é resolve o problema [5].

2.2 Quando usar

O backtracking é aplicado em alguns tipos de problemas específicos [6]. Ele é uma ferramenta importante para resolver problemas de satisfação de restrições, como palavras cruzadas, aritmética verbal, Sudoku e muitos outros quebra-cabeças. Frequentemente, é a técnica mais conveniente para análise, para o problema da mochila e outros problemas de otimização combinatória. Ele depende dos "procedimentos de caixa preta" fornecidos pelo usuário que definem o problema a ser resolvido, a natureza dos candidatos parciais e como eles são

estendidos para candidatos completos. É, portanto, uma metaheurística em vez de um algoritmo específico − embora, ao contrário de muitas outras metaheurísticas, seja garantido que encontrará todas as soluções para um problema finito em um período de tempo limitado [7].

Para alguns casos, o *backtracking* é usado para o problema de enumeração, a fim de encontrar o conjunto de todas as soluções viáveis para o problema. Por outro lado, o *backtracking* não é considerado uma técnica otimizada para resolver um problema. Ele encontra sua aplicação quando a solução necessária para um problema não tem limite de tempo.

2.3 Exemplo

Explicação de *backtracking* atravéz do problema do 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.

2.4 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      # Funcão 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 solucão
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.
        Se houver um ciclo hamiltoniano então o caminho pode ser
```

```
    iniciado de qualquer ponto do ciclo, pois o gráfico não é
    direcionado
57          caminho[0] = 0
58
59          if self.hamCycleUtil(caminho,1) == False:
60              print ("A solucao nao existe \n")
61              return False
62
63          self.printSolution(caminho)
64          return True
65
66      def printSolution(self, caminho):
67          print ("A Solucão Existe: Seguindo ",
68                     "é um ciclo hamiltoniano")
69          for vertex in caminho:
70              print (vertex, end = " ")
71          print (caminho[0], "\n")
72
73  # Codigo do condutor
74
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 Abordagem Gulosa

3.1 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. Discussão sobre os tipos de formas de implementar o algoritmo de Fibonacci [11].

3.2 Exemplo de código

```
# Código de Fibonacci usando abordagem gulosa.
# Muitos elementos dentro da árvore de recursão
# serão recalculados inúmeras vezes
import time

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

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


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 15: {timer(fib, 15)}')
print(f'Fibonnaci de 20: {timer(fib, 20)}')
print(f'Fibonnaci de 30: {timer(fib, 30)}')
```

3.3 Complexidade

Devido ao fato desse algoritmo criar uma árvore e deslocar dentro dela vários nós são iguais e mesmo assim recalculados. Com isso o custo é $\mathcal{O}(n^2)$.

4 Programação Dinâmica

4.1 Conceito

A ideia inicial da progamação dinâmica está baseada em armazenar todas as operações que serão repetidas durante uma recursão. Dessa forma, se durante a recursão tiver um nó igual que já foi previamente calculado, não será necessário que o cálculo seja feito novamente. Em árvores criadas como o algoritmo de fibonacci em que muitos nós são iguais, todos eles serão calculados apenas uma vez [12].

4.2 Exemplo de código

```
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,
        memo)
21      return memo[numero]
22
23  def timer(fib, num):
24      start = time.time()
25      fib(num)
26      end = time.time()
27      return end - start
28
29  # Escrevendo os tempos na tela
30  print(f'Fibonnaci de 10: {timer(fib, 10)}')
31  print(f'Fibonnaci de 10 memoization: {timer(fibMemo, 10)}')
32
33  print(f'Fibonnaci de 15: {timer(fib, 15)}')
34  print(f'Fibonnaci de 15 memoization: {timer(fibMemo, 15)}')
35
36  print(f'Fibonnaci de 20: {timer(fib, 20)}')
37  print(f'Fibonnaci de 20 memoization: {timer(fibMemo, 20)}')
38
39  print(f'Fibonnaci de 30: {timer(fib, 30)}')
40  print(f'Fibonnaci de 500 memoization: {timer(fibMemo, 500)}')
```

4.3 Complexidade

Como o algoritmo armazena todas as equações já feitas, todas as operações são transformar de $\mathcal{O}(n^2)$ para $\mathcal{O}(n)$. Precisando apenas do tempo de encontrar o elemento dentro do dicionário do *Python* e retornando ele [13].

5 Divisão e Conquista

5.1 Conceito

Em ciência da computação, *divide and conquer* é um paradigma de projeto de algoritmo. Um algoritmo de divisão e conquista divide recursivamente um problema em dois ou mais subproblemas do mesmo tipo ou de tipo relacionado,

até que se tornem simples o suficiente para serem resolvidos diretamente. As soluções para os subproblemas são então combinadas para fornecer uma solução para o problema original.

A técnica de dividir e conquistar é a base de algoritmos eficientes para muitos problemas, como classificação, multiplicação de grandes números, localização do par mais próximo de pontos e análise sintática.

Esta técnica pode ser dividida nas seguintes três partes:

- Dividir: Envolve a divisão do problema em subproblemas menores.
- Conquistar: Resolva subproblemas chamando recursivamente até que seja resolvido.
- Combinar: Combine os subproblemas para obter a solução final de todo o problema.

A grande relação entre o *Divide and Conquer*, o *Branch and Bound* e o *Backtracking* é o fato de todos serem de natureza recursiva devido a maneira em que suas estrategias são conceptualizadas. O *Backtracking* é uma otimização do *brute-force*, onde o algoritmo retorna quando percebe que não tem como achar a resposta pelo caminho atual, por isso ele na maioria das vezes é recursivo. O divisão e conquista divide o problema em subproblemas, usando a recursão para esse processo. Eles possuem a diferença de: O *backtracking* tenta fazer apenas parte da solução, em busca da solução ótima, já o *divide and conquer* sempre vai dividir o problema inteiro para a sua resolução.

A seguir estão alguns algoritmos padrão que seguem o algoritmo *Divide and Conquer*:

- Quicksort
- MergeSort
- Par de pontos mais próximo

Exemplo: Encontrar o elemento máximo e mínimo de um array:

Entrada: 70, 250, 50, 80, 140, 12, 14

Saída: o número mínimo em uma determinada matriz é: 12

O número máximo em uma determinada matriz é: 250

Abordagem: encontrar o elemento máximo e mínimo de um determinado array é um aplicativo para dividir para conquistar. Neste problema, encontraremos os elementos máximo e mínimo em um determinado array. Estamos usando uma abordagem de dividir e conquistar (DAC [14]) que tem três etapas: dividir, conquistar e combinar.

Neste problema, estamos usando a abordagem recursiva para encontrar o máximo, onde veremos que apenas dois elementos sobraram e então podemos facilmente usar a condição, ou seja, $if(a[indice] > a[indice + 1])$. Na condição acima, verificamos a condição do lado esquerdo para descobrir o máximo. Agora, veremos a condição do lado direito para encontrar o máximo. Função recursiva para verificar o lado direito do índice atual de uma matriz. Agora, vamos comparar a condição e verificar o lado direito do índice atual de um determinado array. No programa fornecido, vamos implementar essa lógica para verificar a condição do lado direito no índice atual.

Para encontrar o mínimo, vamos implementar uma logica recursiva igual fizemos para achar o maximo.

5.2 Exemplo de algoritmo

```
1
2 # Funcão para encontrar o no máximo em um array.
3 def DAC_Max(a, index, l):
4     max = -1;
5
6     if (index >= l - 2):
7         if (a[index] > a[index + 1]):
8             return a[index];
9         else:
10            return a[index + 1];
```

```
11
12      # Lógica para encontrar o elemento máximo no array
13      max = DAC_Max(a, index + 1, l);
14
15      if (a[index] > max):
16          return a[index];
17      else:
18          return max;
19
20  # Funcão para encontrar o no minimo em um array.
21  def DAC_Min(a, index, l):
22      min = 0;
23      if (index >= l - 2):
24          if (a[index] < a[index + 1]):
25              return a[index];
26          else:
27              return a[index + 1];
28
29      # Lógica para encontrar o elemento minimo no array
30      min = DAC_Min(a, index + 1, l);
31
32      if (a[index] < min):
33          return a[index];
34      else:
35          return min;
36
37  # Codigo teste
38  if __name__ == '__main__':
39
40      # Definindo as variáveis
41      min, max = 0, -1;
42
43      # Inicializando o array
44      a = [70, 250, 50, 80, 140, 12, 14];
45
46      # Recursão - funcão DAC_Max chamada
47      max = DAC_Max(a, 0, 7);
48
```

```
49      # Recursão - funcão DAC_Max chamada
50      min = DAC_Min(a, 0, 7);
51      print("O número mínimo em um determinado array é : ", min);
52      print("O número maximo em um determinado array é : ", max);
```

[15]

5.3 Complexidade

A complexidade dos algoritmos de divisão e conquista, por conta de sua abordagem, podem ir de $\mathcal{O}n^2$ como um *bubblesort* para $n \log_2 n$ como um *quicksort* [16].

6 Relação entre os algoritmos

6.1 *Branch and Bound* e *Backtracking*

A principal semelhança entre ambos os algoritmos é tentar resolver problemas mais complexos de formas mais simples, divindo em subconjuntos menores e os resolvendo separadamente. Com isso, ambos são algoritmos inventados para otimizar problemas normalmente combinatórios que exigiriam que fossem testados todos os casos para se obter a resposta. Se *Backtracking* conseguir encontrar uma solução mais rápida ele vai escolhê-la, se não descartá-la e não resolverá aquele subconjunto. *Branch and Bound* vai um pouco além, não se limitando apenas a tentar resolver o problema, mas a resolver de forma mais eficiente. Então aqui encontramos a primeira diferença. O primeiro algoritmo tentará resolver o problema de qualquer forma, enquanto o segundo só resolverá se for mais rápido do que um simples *bruteforce*.

Para ficar mais simples o entendimento:

Parâmetro	*Backtracking*	*Branch and Bound*
Abordagem	Usado para encontrar todas as soluções possíveis disponíveis para um problema. Quando ele percebe que fez uma escolha errada, ele desfaz a última escolha, fazendo o backup dela. Ele pesquisa a árvore do espaço de estados até encontrar uma solução para o problema.	Usado para resolver problemas de otimização. Quando ele percebe que já tem uma solução ótima melhor para a qual a pré-solução o conduz, ele abandona essa pré-solução. Ele pesquisa completamente a árvore do espaço de estado para obter a solução ideal.
Função	Envolve a função de viabilidade.	Envolve uma função delimitadora.
Problemas	Usado para resolver o problema de decisão.	Usado para resolver o problema de otimização.
Busca	O estado da árvore atual e buscado até que a solução seja obtida.	A solução ideal pode estar presente em qualquer lugar na árvore do espaço de estado, a árvore precisa ser pesquisada completamente.
Eficiência	Mais eficiente.	Menos eficiente.

6.2 Abordagem gulosa, Programação Dinâmica e Divisão e Conquista

Abordagem gulosa, programação dinâmica e divisão e conquista são formas em que os dois algoritmos previamente citados utilizam-se para conseguir resolver os problemas. Dessa forma, os três algoritmos são como ferramentas para

resolver os problemas daqueles dois.

Explicando de uma forma mais simples:

Primeiramente para se resolver um problema, todos os cientistas tentam uma abordagem gulosa. Para começar é sempre importante encontrar um algoritmo que resolva o problema, mesmo que seja lento. Não adianta otimizar um algoritmo que tenha erros na resolução dos problemas. Quando passa-se dessa fase, começamos a tentar otimizar todos os passos que forem possíveis. Com isso, podemos usar tanto programação dinâmica para armazenar soluções já conhecidas, quanto divisão e conquista para resolver problemas semelhantes de cada vez.

Tanto no *backtracking* quanto em *branch and bound* várias recursões são abertas todas as vezes que é encontrado dois ou mais possíveis caminhos para solucionar o problema. Encontrando algum que não possua a resposta, é só eliminá-lo e continuar a busca − divisão e conquista −. Ambos possuem uma parte de programação dinâmica ao considerarmos que quando é encontrado um caminho ruim para resolver o problema todos os caminhos subsequentes serão eliminados junto. Não sendo necessário despender tempo calculando caminhos que não trarão a solução do problema.

A grande diferença entre os dois se dá no momento do descarte dos caminhos. Para o *backtracking* será descartado apenas se não trouxer a resposta. Enquanto em *branch and bound* será descartado no momento que for percebido que um simples *bruteforce* nesse subconjunto - que estamos chamando de caminhos - seria o bastante para resolver o problema. Dessa forma, *branch and bound* é focado apenas em reduzir a dificuldade de problemas para posteriormente serem resolvidos.

Referências

[1] Geeks for Geeks, Disjoint Sets, acessado em 27/11/2021.

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

[3] Geeks for Geeks, Knapsack using Branch and Bound acessado em 27/11/2021.

[4] Geeks for Geeks, Implementation of 0/1 Knapsack using Branch and Bound, acessado em 27/11/2021.

[5] Free Code Camp, Backtracking Algorithms Explained, acessado em 28/11/2021.

[6] Wikipedia, Backtracking, acessado em 28/11/2021.

[7] Baeldung, Backtracking Algorithms acessado em 28/11/2021.

[8] Wikipedia, Knight's tour, acessado em 28/11/2021.

[9] Geeks for Geeks, Backtracking Algorithms, acessado em 27/11/2021.

[10] Geeks for Geeks, Hamiltonian Cycle — Backtracking-6, acessado em 27/11/2021.

[11] Stack Overflow, Discussion about Fibonacci's *Greedy algorithm*, acessado em 28/11/2021.

[12] Geeks For Geeks, Dynamic Programing, acessado em 28/11/2021.

[13] FreeCodeCamp, Dynamic Programing, acessado em 27/11/2021.

[14] Wikipedia, Divide and Conquer Algorithm, acessado em 28/11/2021.

[15] Geeks for Geeks, Divide and Conquer Algorithm Introduction, acessado em 28/11/2021.

[16] Free Code Camp, Divide and Conquer Algorithm Meaning: Explained with Examples, acessado em 28/11/2021.