

Uma argumentação entre *Backtracking* e *Branch and Bound*

Gustavo Lopes Rodrigues¹, Rafael Amauri Diniz Augusto²

¹Instituto de Ciências Exatas e Informática –
Pontifícia Universidade Católica de Minas Gerais(PUC-MG)

Abstract. *Branch and Bound and Backtracking are two strategies in algorithm development. The objective of this project for the discipline of Algorithm Design and Analysis is talk about such strategies indicating, at least, the concepts, when it should be used, in addition to showing examples. We will also present the main similarities and differences between them. Also, if there are similarities or differences with other strategies such as Greedy Approach, Dynamic Programming and Division and Conquer, we will present them in a way to broaden the discussion.*

Resumo. *Branch and Bound e Backtracking são duas estratégias no desenvolvimento de algoritmos. O objetivo deste trabalho feito para a disciplina de Projeto e Análise de Algoritmos é discutir sobre tais estratégias indicando, pelo menos, os conceitos, quando se deve utilizar, além de mostrar exemplos. Apresentaremos também as principais semelhanças e diferenças entre elas. Ainda, se houver semelhanças ou diferenças com outras estratégias como Abordagem Gulosa, Programação Dinâmica e Divisão e Conquista, apresentaremos elas de forma a ampliar a discussão.*

1. Informações gerais

Em discussões sobre desenvolvimento de algoritmos, a escolha da estratégia para resolver um problema é fundamental, visto que existem métodos diferentes, que possuem aplicações diferentes, e com complexidades diferentes. Os principais em questão, são o *Backtracking*, e *Branch and Bound*. Ambos são relativamente parecidos em conceito, porém possuem implementações diferentes, o que resulta em diferentes circunstâncias na qual um é preferível ao outro.

Neste artigo, iremos analisar os dois algoritmos, olhando com cuidado para os conceitos para então construir códigos para então resolver problemas conhecidos da computação. Por fim, iremos dar uma breve olhada em outras estratégias, e comparar com as abordagens mencionadas anteriormente.

Uma observação importante, deixaremos implementações dos problemas apresentados nos artigos no fim deste artigo.

2. Backtracking

Backtracking é uma estratégia para resolver problemas recursivamente e incrementalmente, removendo as soluções parciais que falham em ajudar na solução para o problema. Para aplicar essa técnica, a estratégia tenta encontrar a solução utilizando diversos pequenos *checkpoints*, para os quais o programa pode voltar se a iteração atual para a resolução do problema não ajudar a encontrar a solução final.

Essa estratégia é viável para resolver problemas modulares que requerem muita tentativa e erro, já que ele remove “caminhos” inválidos, e isso salva muito tempo de processamento.

2.1. Problema do labirinto

Considere o labirinto abaixo:

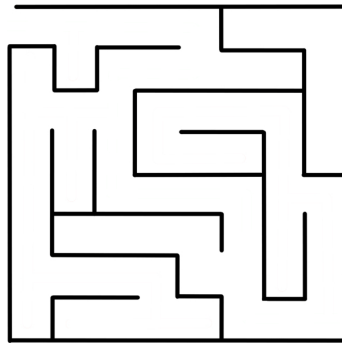


Figura 1. Um exemplo de labirinto

Resolver labirintos é uma aplicação clássica da estratégia de *backtracking*, pois envolve diversas possibilidades de caminhos e é fácil integrar *checkpoints* no problema.

Imaginando que podemos traduzir a imagem acima para uma matriz onde cada ponto de decisão do labirinto é uma célula, que todas células têm ponteiros que apontam para quatro direções: cima, baixo, esquerda e direita, e que esses ponteiros podem levar a outras células ou ser do tipo NULL (indicando que não há uma célula ligada naquela direção), a estratégia de Backtracking pode ser usada para resolver o problema da seguinte forma:

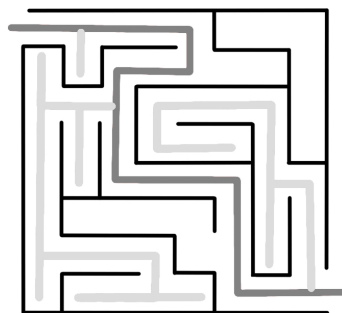


Figura 2. O mesmo labirinto de antes, porém com sua solução destacada

Admitindo que esse algoritmo busca os caminhos disponíveis na ordem [cima, esquerda, baixo, direita], os caminhos destacado em cor leve são caminhos percorridos pelo algoritmo e que foram detectados como caminhos que não contribuem para a solução. A ideia é preservar os checkpoints “certos” que foram percorridos e voltar ao último checkpoint correto ao invés de salvar os caminhos “errados” e recomeçar o percurso desde o início. Dessa forma, apenas os checkpoints “certos” vão ser mantidos e o caminho correto eventualmente será encontrado, sem a necessidade de recalcular e levar em conta os caminhos que já sabemos que são incorretos.

Algorithm 1 Backtracking

```
1: procedure FIND_PATH(x,y)
2:   grid[x][y] == visited
3:   for i to 4 do
4:     if UP(x,y+1) then
5:       find_path(x,y+1)
6:     if LEFT(x-1,y) then
7:       find_path(x-1,y)
8:     if DOWN(x,y-1) then
9:       find_path(x,y-1)
10:    if RIGHT(x+1,y) then
11:      find_path(x+1,y)
```

3. Branch and Bound

Branch and Bound é uma técnica para algoritmos que visa melhorar o tempo de processamento ao eliminar as soluções-candidato que claramente não se aplicam e/ou não ajudam a resolver o problema. Esse método geralmente é aplicado em problemas que têm soluções finitas, no qual as soluções podem ser representadas como uma sequência de opções.

A primeira parte da técnica Branch and Bound pede que o algoritmo trate as possíveis soluções como se estivessem em uma estrutura de árvore, onde os nós da árvore são utilizados como partes de possíveis soluções garante que todas soluções sejam consideradas.

3.1. Problema da Mochila

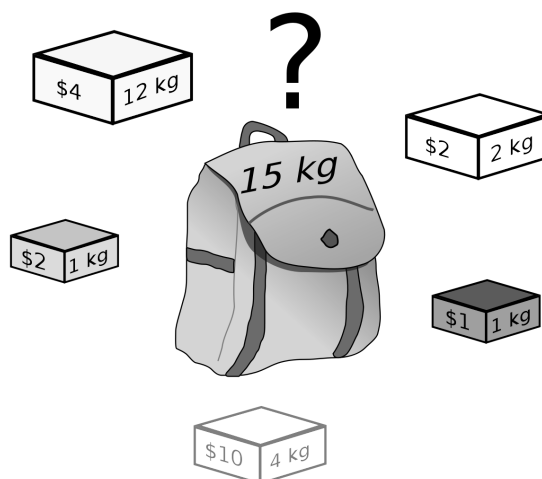


Figura 3. Ilustração do problema da mochila

O problema escolhido para mostrar essa estratégia é o problema do 0/1 Knapsack. A ideia por trás dele é um ladrão que quer roubar uma casa, mas sua mochila só aguenta carregar um peso P_1 . Existem vários itens na casa, cada um com seu valor V e peso P_2 .

Escolha os itens específicos que vão maximizar o valor roubado pelo ladrão de forma que a soma dos pesos dos itens nunca seja maior que P1.

Podemos ilustrar as combinações de itens em uma estrutura de árvore, de forma que cada nó é um item que o ladrão pode roubar, e um caminho percorrendo N nós significa um conjunto de N itens.

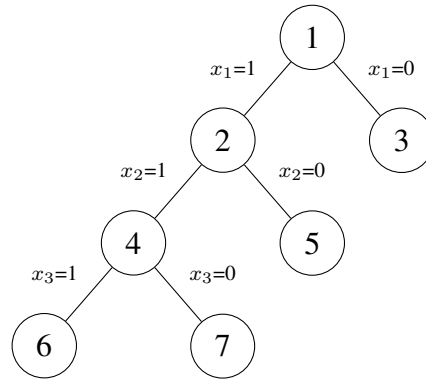


Figura 4. Space State Tree do problema do Knapsack

1	1	1	1	4	5	6	7
c	-38	-38	-32	-38	-36	-32	-38
u	-32	-32	-22	-32	-22	-38	-38

Tabela 1. Valores de C(custo) e U(upperBound)

Algorithm 2 Knapsack

```

1: procedure KNAPSACK(S,F,LB)
2:   if  $LB < p(S)$  then
3:      $LB \leftarrow p(S)$ 
4:    $A \leftarrow UpperBound(F, c - w(S))$ 
5:   if  $A + p(S) \geq LB$  then
6:     return
7:   for  $i \in F$  do
8:      $B \leftarrow UB_i$ 
9:     if  $A + p(S) \geq LB$  then
10:       $F \leftarrow F_i$ 
11:     else
12:      break
  
```

4. Backtracking Vs Branch and Bound

Para terminar esta parte de nossa análise, iremos destacar quatro pontos importantes para comparar ambas estratégias.

- Ambos algoritmos usam uma árvore de estados(state space tree) para encontrar a solução, porém, enquanto Backtracking procura por ela exaustivamente até encontrar a solução, Branch and Bound é forçado a passar pela árvore inteira para encontrar a solução otimizada.
- A pesquisa feita na árvore de estados do Backtracking utiliza um DFS(Depth first search), enquanto que o Branch and Bound pode utilizar DFS, ou até mesmo BFS(Breadth first search)
- Como Backtracking não precisa percorrer por todas as possíveis soluções, esta abordagem é mais eficiente do que Branch and Bound, que precisa percorrer todas as ramificações da árvore de estados.
- O resultado do algoritmo de Backtracking pode ser a solução otimizada, porém existem problemas na implementação de backtracking em que o resultado será uma solução sub-otimizada. Como Branch and Bound varre procurando todas as soluções, este consegue achar a solução otimizada com maior frequência.

5. Outras estratégias

Com as principais estratégias discutidas, fica claro as diferenças e vantagens de ambas estratégias. Agora iremos discutir outras possíveis abordagens para resolução de algoritmos, e comparar com as já discutidas.

5.1. Abordagem gulosa

Abordagem Gulosa é uma estratégia que tenta resolver o problema fazendo a escolha ótima localmente em cada fase com o intuito de encontrar uma ótima global que resolve o problema por completo.

Para o uso dessa abordagem, duas propriedades são de suma importância:

- **Greedy Choice Property:** Uma solução global otimizada pode ser obtida, selecionando uma solução ótima local.
- **Optimal Substructure:** uma solução ótima pode ser construída, usando a solução ótima para os seus subproblemas.

Um exemplo de abordagem gulosa é o Algoritmo de Dijkstra.

5.1.1. Algoritmo de Dijkstra

Dado um grafo qualquer e dois pontos: partida e chegada, encontre o menor caminho possível. Para resolver esse problema, Dijkstra cria o que é chamado de *Shortest path tree(spt)* ou a árvore de caminho mais curto. A cada passo do algoritmo, a **spt** adiciona uma vértice não visitada, e então atualiza a distância das adjacências da vértice que foi adicionada. O algoritmo irá terminar quando a **spt** conter informação de todas as vértices.

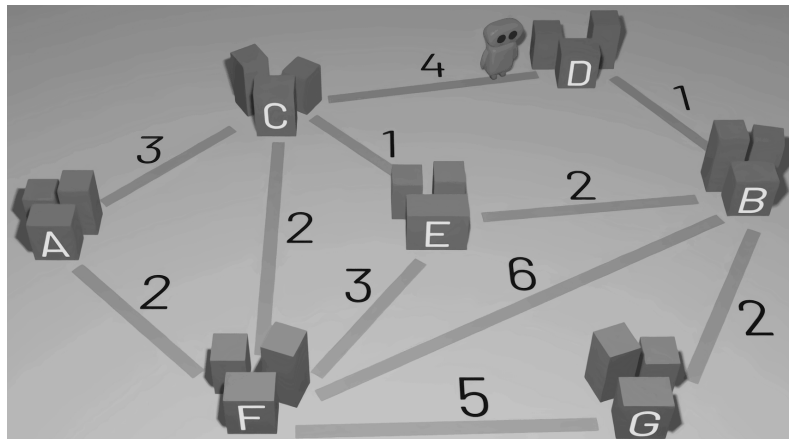


Figura 5. Um caso comum em encontrar o menor caminho entre dois pontos, é encontrar a menor distância entre duas cidades

Algorithm 3 Dijkstra

```

1: procedure DIJKSTRA(origin,destination)
2:   for each  $v \in G.vertices$  do
3:     distance[v] =  $\infty$ 
4:    $A \leftarrow G.vertices$ 
5:   distance[origin] = 0
6:   for  $j = 1$  in  $|G.vertices| - 1$  do
7:      $u := \text{Extract-Min}(A)$ 
8:      $A.pop(u)$ 
9:     for each  $v \in \text{Neighbors}(u)$  do
10:      Update(distance,v)
11:  return distance[destination]

```

5.1.2. Algoritmo Guloso Vs Backtracking

- Por sua natureza, a estratégia Gulosa pode não alcançar imediatamente o resultado global ótimo, um efeito que também está presente nos algoritmos que utilizam *Backtracking*
- Devido a natureza da estratégia de backtracking, tais algoritmos costumam ter implementações recursivas, enquanto que algoritmos gulosos tentam buscar a melhor escolha local de forma iterativa.

5.1.3. Algoritmo Guloso Vs Branch and Bound

- Algoritmos gulosos costumam ter resultados mais rápidos do que técnicas de *Branch and Bound* para problemas com menos *entrada*, mas à medida que a entrada cresce o mesmo pode acabar sendo custoso como o *Branch and Bound*
- *Branch and Bound*, assim como *Backtracking*, consegue voltar a estados anteriores da árvore para encontrar a solução, enquanto isso, algoritmos gulosos tentam buscar a melhor escolha local de forma iterativa e não dividem o problema em "*checkpoints*"

5.2. Divisão e conquista

Divisão e Conquista é uma técnica que pode ser dividida em três partes fundamentais: dividir um problema maior recursivamente em problemas menores (**Dividir**), resolver todos os sub-problemas (**Conquistar**) e, por fim, a solução do problema inicial é dada através da combinação dos resultados de todos os problemas menores computados (**Combine**).

Problemas que usam divisão e conquista são característicos em possuir um princípio chamado **Overlapping subproblems**, ou seja, o problema pode ser dividido em sub-problemas, que são reutilizados várias vezes e que não geram novos subproblemas.

Exemplos de problemas conhecidos que usam desta estratégia são o problema de ordenação interna (usando quicksort ou mergesort) e o problema dos pares de pontos próximos.

5.2.1. Pares de pontos próximos

Dado um conjunto de N pontos em um espaço, é preciso encontrar os dois pontos do conjunto que possuem a menor distância um do outro. Este problema possui mais de uma implementação, porém uma das mais eficientes é utilizando divisão e conquista.

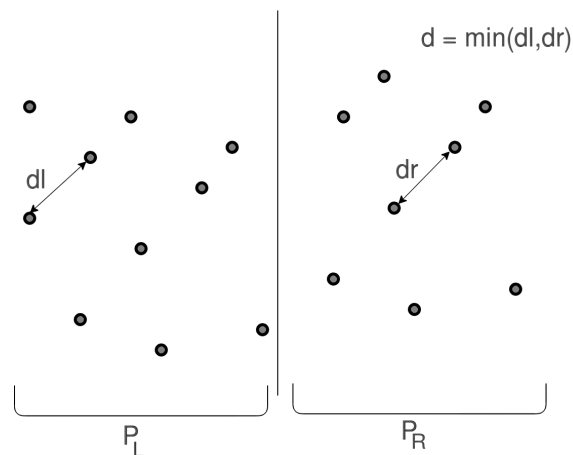


Figura 6. Uma representação visual da separação dos pontos

Supondo que a entrada para o problema é um conjunto de pontos com o eixo y já ordenado, encontramos o ponto do meio, separamos o conjunto ao meio, e então recursivamente descobrimos as menores distâncias entre os conjuntos separados, para então achar a solução geral.

Algorithm 4 Closest pair of points

```
1: procedure DIVIDE-AND-CONQUER(P,Q)
2:   if  $n \geq 3$  then
3:     bruteForce(P,n)
4:      $midPoint \leftarrow P[\frac{n}{2}]$ 
5:      $P_l \leftarrow P[1 \dots \frac{n}{2} - 1]$ 
6:      $Q_l \leftarrow P[1 \dots \frac{n}{2} - 1]$ 
7:      $P_r \leftarrow P[\frac{n}{2} \dots n]$ 
8:      $Q_r \leftarrow P[\frac{n}{2} \dots n]$ 
9:      $d_l \leftarrow DIVIDE - AND - CONQUER(P_l, Q_l)$ 
10:     $d_r \leftarrow DIVIDE - AND - CONQUER(P_r, Q_r)$ 
11:     $d \leftarrow \min(d_l, d_r)$ 
12:     $strip_p \leftarrow []$ 
13:     $strip_q \leftarrow []$ 
14:     $lr \leftarrow P_l + P_r$ 
15:    for  $i$  to  $n$  do
16:      if  $abs(lr[i].x - midPoint.x) < d$  then
17:         $stripP.append(lr[i])$ 
18:      if  $abs(Q[i].x - midPoint.x) < d$  then
19:         $stripQ.append(Q[i])$ 
20:     $sort(stripP, key = y)$ 
21:     $min_p \leftarrow \min(d, stripClosest(stripP), |stripP|, d)$ 
22:     $min_q \leftarrow \min(d, stripClosest(stripQ), |stripQ|, d)$ 
23:    return  $\min(min_p, min_q)$ 
```

5.2.2. Divisão e Conquista Vs Backtracking

- Devido à maneira como ambos são conceitualizados, ambos possuem a natureza de serem recursivos.
- Em divisão e conquista, precisamos analisar a entrada inteira do usuário para resolver o problema, enquanto que Backtracking pode ou não analisar a todas as possibilidades dentro da entrada do usuário

5.2.3. Divisão e Conquista Vs Branch and Bound

- Enquanto divisão e conquista divide a entrada do usuário(Ex: divide o vetor de números para ordená-los) para resolver os sub-problemas e então resolver a combinação desses problemas, a outra estratégia divide o espaço da solução para o problema.
- Tanto Branch and Bound e Divide and Conquer são estratégias que partem dos mesmos princípios: ambas querem dividir o problema em partes menores para resolver o problema todo.

5.3. Programação Dinâmica

Programação dinâmica é uma estratégia de construção de algoritmos que possui uma grande eficácia na resolução de problemas de otimização combinatória. Em outras palavras, essa estratégia é ideal quando o problema original é dividido em subproblemas, que se repetem, logo podem ser memorizados para evitar recálculos.

Antes de aplicar essa técnica, o problema precisa atender duas propriedades: um problema com subproblemas (**Overlapping subproblems**), propriedade já vista em Divisão e conquista e uma subestrutura ideal (**Optimal substructure**), propriedade já vista em algoritmos gulosos.

Para melhor exemplificar esses conceitos, vamos analisar um clássico problema da computação, e também um caso perfeito para ilustrar a estratégia de programação dinâmica.

5.3.1. Fibonacci

Descoberto pelo matemático italiano do século 12, Leonardo Fibonacci, a sequência Fibonacci é uma sucessão de números que aparece codificada nos mais diversos fenômenos da natureza (espiral de galáxias, ciclones, conchas e vários outros). Na computação, temos o problema de criar um algoritmo capaz de calcular o número da sequência de Fibonacci.

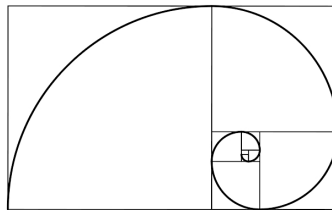


Figura 7. Uma imagem ilustrativa da sequência Fibonacci

A ideia para resolução desse problema é bem simples: sabendo que a sequência inicia com 0 e 1, os próximos números serão sempre a soma dos dois números anteriores. A seguir temos um exemplo de uma implementação ingênua do algoritmo da sequência de Fibonacci.

Algorithm 5 Naive Fibonacci

```
1: procedure FIB(n)
2:   if  $n \geq 2$  then
3:     return n
4:   return fib(n-1) + fib(n-2)
```

Esta implementação a primeira vista não parece ter nenhum problema. Porém, é quando tentamos executar esse código que o problema se mostra. O código consegue calcular o 6º, 7º e 8º número da sequência, porém o código não consegue calcular o 50º número.

Para entender a natureza desse problema precisamos entender a complexidade desse código, logo, precisamos da árvore de recursão gerada pelo código:

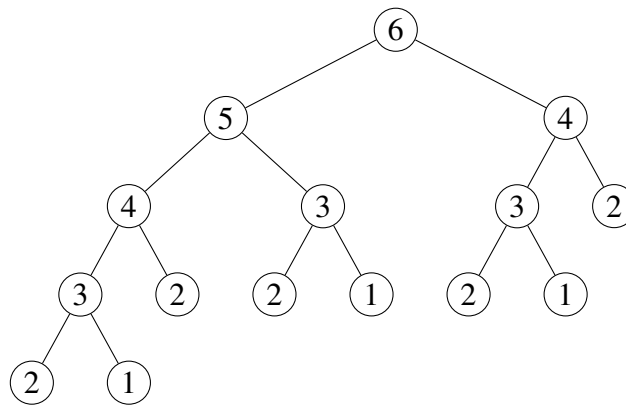


Figura 8. encontrando o 6º número da sequência Fibonacci

A primeira coisa a se notar é a complexidade do algoritmo. A árvore formada possui uma altura N (o número que queremos achar da sequência), e a cada novo nível da árvore, formamos no máximo, dois novos nós, logo podemos concluir que a complexidade é de $O(2^n)$, justificando também, porque não conseguimos calcular o 50º número, já que este levaria um tempo muito alto.

Então agora a pergunta fica: como podemos otimizar essa árvore, de forma que o algoritmo consiga ter um tempo linear? Voltando à árvore, podemos observar um padrão recorrente.

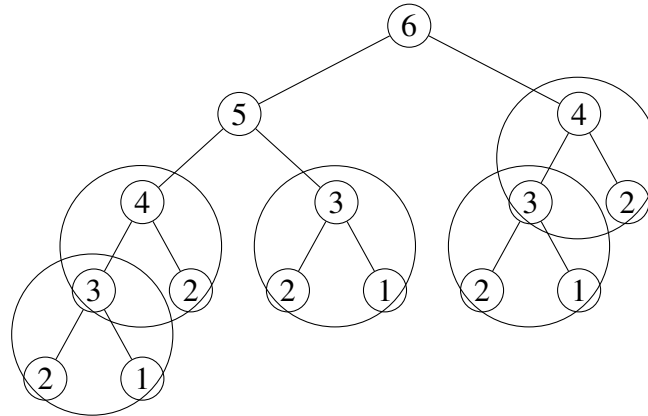


Figura 9. Árvore de recursão, com os sub-problemas repetidos circutados

Com esse segundo desenho, fica mais evidente que existe passos da recursão que estão repetidos, ou seja, o programa acaba resolvendo mais de uma vez os mesmos problemas de forma desnecessária. Em conclusão, para resolver esses problemas, o ideal seria uma estrutura que possa guardar resultados de recursões já feitas para usar depois quando o mesmo problema for encontrado, resultando em uma árvore de recursão da seguinte forma:

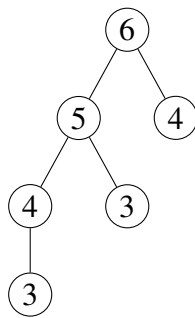


Figura 10. Árvore de recursão otimizada

Com a otimização percebemos que agora a complexidade do código melhorou drasticamente, tendo um resultado de $O(2n)$, visto que é resultado da altura da árvore multiplicado por 2, que pode ser simplificado para apenas $O(n)$.

Eis então agora o código otimizado:

Algorithm 6 Optimized Fibonacci

```

1: procedure FIB(n,memo)
2:   if  $n \geq 2$  then
3:     return n
4:   if n in memo then
5:     return memo[n]
6:   memo  $\leftarrow$  fib(n-1) + fib(n-2)
7:   return memo[n]

```

5.3.2. Programação dinâmica Vs Backtracking

- Programação dinâmica procura a solução ótima para o problema, enquanto que backtracking tenta sistematicamente satisfazer um critério, podendo ou não varrer por toda a árvore, assim como pode ou não encontrar a solução ótima.
- Ambos algoritmos possuem conceitos que tem natureza recursiva. Entretanto programação dinâmica ainda pode ser implementado de forma iterativa, usando bottom-up (tabulation).

5.3.3. Programação dinâmica Vs Branch and Bound

- Programação dinâmica tenta minimizar os passos para resolver o problema, evitando subproblemas já resolvidos. Enquanto isso, branch-and-bound tenta sistematicamente enumerar todos os candidatos possíveis na árvore de soluções possíveis,
- Branch and Bound permite backtracking, para selecionar as melhores soluções dentro da árvore de estados, enquanto programação dinâmica não utiliza esse tipo de recursão. Como resultado, Branch and Bound pode se demonstrar mais lento que programação dinâmica em alguns casos.

6. Conclusão

No fim das contas, *Backtracking* e *Branch and Bound* são estratégias bem semelhantes, porém, como foi demonstrado neste artigo, eles não substituem um ao outro. Devido a isso, cabe ao programador analisar na qual este quer fazer a implementação de um algoritmo de otimização, pensando qual são as necessidades do contexto atual.

Isso também vale para as outras estratégias mencionadas, todas elas possuem suas aplicações específicas, por mais que possuem propriedades semelhantes umas as outras.

Referências

- Datta, S. (2020). Branch and bound algorithm. <https://www.baeldung.com/cs/branch-and-bound>. Acessado em 28 de Novembro de 2021.
- Desconhecido (2020). Divide and conquer algorithm. <https://www.programiz.com/dsa/divide-and-conquer>. Acessado em 28 de Novembro de 2021.
- Desconhecido (2021). Closest pair of points using divide and conquer algorithm. <https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/?ref=lbp>. Acessado em 28 de Novembro de 2021.
- freeCodeCamp.org (2020). Dynamic Programming - Learn to Solve Algorithmic Problems & Coding Challenges. <https://www.youtube.com/watch?v=oBt53YbR9Kk>. Acessado em 28 de Novembro de 2021.
- Toffolo, T. and Carvalho, M. A. (2021). Backtracking. http://www3.decom.ufop.br/toffolo/site_media/uploads/2011-1/bcc402/slides/10._backtracking.pdf. Acessado em 28 de Novembro de 2021.
- Walker, A. (2021). Greedy Algorithm with Example: What is, Method and Approach. <https://www.guru99.com/greedy-algorithm.html>. Acessado em 28 de Novembro de 2021.
- Yoshioka, H. (2018). Knapsacker (python version). <https://github.com/irohiroki/knapsacker-py>. Acessado em 28 de Novembro de 2021.

Abstract

Backtracking

```
1 import Celula
2
3 class Labirinto:
4     def __init__(self, num_rows, num_columns, order_to_check):
5
6         # Indica a ordem que vai os vizinhos vao ser checados
7         self.order_to_check = order_to_check
8
9         # Numero de linhas no grid
10        self.num_rows = num_rows
11        # Numero de colunas no grid
12        self.num_columns = num_columns
13
14        self.grid = []
15
16        # Preenche o grid
17        tmp_cell = Celula.Celula(0)
18        for i in range(self.num_columns):
19            self.grid.append([tmp_cell for x in range(self.num_rows)])
20
21
22        # Printar o grid
23        def __str__(self):
24            grid_as_string = ""
25
26            for i in range(self.num_columns):
27                for j in range(self.num_rows):
28                    grid_as_string += f"{self.grid[i][j].get_value()} "
29
30                grid_as_string += "\n"
31
32            return grid_as_string
33
34
35        # Adiciona a celula cell em [pos_y][pos_x]
36        def insert(self, cell_value, pos_y, pos_x):
37            self.grid[pos_y][pos_x] = Celula.Celula(cell_value)
38
39
40        # Jeito rapido de resolver IndexError porque nao quero gastar
41        # muito tempo nesse codigo
42        try:
43            # Verificar se existe uma celula em cima
44            if self.grid[pos_y-1][pos_x].get_value() != 0:
45                self.grid[pos_y][pos_x].set_up(self.grid[pos_y-1][pos_x])
46
47                self.grid[pos_y-1][pos_x].set_down(self.grid[pos_y][pos_x])
48
49        except IndexError:
50            pass
51
52        try:
```

```

51         # Verificar se existe uma celula embaixo
52         if self.grid[pos_y+1][pos_x].get_value() != 0:
53             self.grid[pos_y][pos_x].set_down(self.grid[pos_y+1][
pos_x])
54             self.grid[pos_y+1][pos_x].set_up(self.grid[pos_y][pos_x
])
55
56     except IndexError:
57         pass
58
59     try:
60         # Verificar se existe uma celula na esquerda
61         if self.grid[pos_y][pos_x-1].get_value() != 0:
62             self.grid[pos_y][pos_x].set_left(self.grid[pos_y][pos_x
])
63             self.grid[pos_y][pos_x-1].set_right(self.grid[pos_y][
pos_x])
64
65     except IndexError:
66         pass
67
68     try:
69         # Verificar se existe uma celula na direita
70         if self.grid[pos_y+1][pos_x].get_value() != 0:
71             self.grid[pos_y][pos_x].set_right(self.grid[pos_y+1][
pos_x])
72             self.grid[pos_y+1][pos_x].set_left(self.grid[pos_y][
pos_x])
73
74     except IndexError:
75         pass
76
77
78     def find_path(self, pos_x, pos_y):
79
80         self.grid[pos_y][pos_x].visited = True
81
82         # Se for a saida, printar a posicao dela!
83         if self.grid[pos_y][pos_x].value == 2:
84             print(f"Saida encontrada na posicao [{pos_x}][{pos_y}]!")
85
86
87         # Verificar na ordem que foi recebida pela funcao
88         for i in self.order_to_check:
89             # Se existe alguem em cima, se esse alguem for diferente de
None e de Zero, abrir uma recursao naquela posicao
90             # pois eh um caminho!
91             if i == "up" and self.grid[pos_y][pos_x].up != None and
self.grid[pos_y][pos_x].up != 0:
92                 if not self.grid[pos_y][pos_x].up.visited:
93                     self.find_path(pos_x, pos_y-1)
94
95             # Se existe alguem na esquerda, se esse alguem for
diferente de None e de Zero, abrir uma recursao naquela posicao
96             # pois eh um caminho!
97             if i == "left" and self.grid[pos_y][pos_x].left != None and

```

```

self.grid[pos_y][pos_x].left != 0:
98     if not self.grid[pos_y][pos_x].left.visited:
99         self.find_path(pos_x-1, pos_y)
100
101     # Se existe alguem embaixo, se esse alguem for diferente de
None e de Zero, abrir uma recursao naquela posicao
102     # pois eh um caminho!
103     if i == "down" and self.grid[pos_y][pos_x].down != None and
self.grid[pos_y][pos_x].down != 0:
104         if not self.grid[pos_y][pos_x].down.visited:
105             self.find_path(pos_x, pos_y+1)
106
107     # Se existe alguem na direita, se esse alguem for diferente
de None e de Zero, abrir uma recursao naquela posicao
108     # pois eh um caminho!
109     if i == "right" and self.grid[pos_y][pos_x].right != None
and self.grid[pos_y][pos_x].right != 0:
110         if not self.grid[pos_y][pos_x].right.visited:
111             self.find_path(pos_x+1, pos_y)

```

Branch and Bound

```

1 from functools import reduce
2 from sortedcontainers import SortedSet
3 from node import Node
4
5 class Knapsacker:
6     def __init__(self, items, capacity):
7         # Items in the sack
8         self.items = sorted(items, key=lambda item: float(item.value) /
item.cost, reverse=True)
9         # Total weight the sack holds
10        self.capacity = capacity
11        # Temporary list
12        self.candidates = SortedSet(key=lambda node: -(node.upper_bound * 2
** 16 + node.item_index))
13
14
15    def pack(self):
16        self.make_initial_node()
17
18        while True:
19            node = self.select_node_to_grow()
20            if node:
21                self.grow(node, node.item_index + 1)
22            else:
23                return self.node_with_highest_value().items()
24
25
26    def grow(self, node, next_index):
27        next_item = self.items[next_index] if next_index < len(self.items)
else None
28
29        if node.positive_child_growable():
30            if next_item and node.cost() + next_item.cost <= self.capacity:
31                upper_bound = node.value() + next_item.value + self.

```

```

upper_bound_beyond(next_index, capacity=self.capacity - node.cost()
- next_item.cost)
32     node.positive_child = Node(next_index, next_item, upper_bound,
node)
33     self.candidates.add(node.positive_child)
34     else:
35         node.cap_positive_child()
36     elif node.negative_child_growable():
37         if next_item:
38             upper_bound = node.value() + self.upper_bound_beyond(next_index
, capacity=self.capacity - node.cost())
39             node.negative_child = Node(next_index, None, upper_bound, node)
40             self.candidates.add(node.negative_child)
41             self.candidates.remove(node)
42         else:
43             node.cap_negative_child()
44
45
46 def upper_bound_beyond(self, prev_index, capacity=0):
47     index = prev_index + 1
48
49     if index >= len(self.items):
50         return 0
51     item = self.items[index]
52
53     if item.cost > capacity:
54         return float(item.value) * capacity / item.cost
55     else:
56         return item.value + self.upper_bound_beyond(index, capacity=
capacity - item.cost)
57
58
59 def make_initial_node(self):
60     self.candidates.add(Node(-1, None, self.upper_bound_beyond(-1,
capacity=self.capacity), None))
61
62
63 def select_node_to_grow(self):
64     hopeful_node = self.candidates[0]
65
66     if not hopeful_node.leaf():
67         return hopeful_node
68
69
70 def node_with_highest_value(self):
71     return reduce(lambda provisional, candidate: provisional if
provisional.value() >= candidate.value() else candidate, self.
candidates)

```

Divisão e conquista

```

1 import copy
2 import math
3
4 class Point():
5     def __init__(self, x, y):

```



```

6         self.x = x
7         self.y = y
8
9     def __str__(self):
10         return f" x: {self.x}\n y: {self.y}"
11
12     """
13     Algoritmo de Brute Force
14     Retorna a menor distancia dentre os pontos
15     P : array de pontos
16     n : tamanho do array
17     """
18
19     def bruteForce(P, n):
20         min_val = float('inf')
21         for i in range(n):
22             for j in range(i + 1, n):
23                 if dist(P[i], P[j]) < min_val:
24                     min_val = dist(P[i], P[j])
25
26         return min_val
27
28     # Calcula a distancia entre dois ponto
29     def dist(p1, p2):
30         return math.sqrt((p1.x - p2.x) *
31                          (p1.x - p2.x) +
32                          (p1.y - p2.y) *
33                          (p1.y - p2.y))
34
35     """çã
36     Funo para encontrar a menor distancia entre um dado
37     grupo de pontos
38     strip: pontos a serem avaliados
39     size: tamanho do array
40     d: distancia inicial
41     """
42     def stripClosest(strip, size, d):
43
44         min_val = d
45
46         for i in range(size):
47             j = i + 1
48             while j < size and (strip[j].y -
49                                strip[i].y) < min_val:
50                 min_val = dist(strip[i], strip[j])
51                 j += 1
52
53         return min_val
54
55     """
56     Ordem o vector o vector de pontos mais proximos do eixo X
57     P : array de pontos
58     Q : inico do array
59     n : fim do array
60     """
61     def closestUtil(P, Q, n):

```

```

62
63
64     # Caso a quantidade de pontos seja menor que 3
65     # usa-se o metodo de çfora bruta
66     if n <= 3:
67         return bruteForce(P, n)
68
69     # pegar a metade de n
70     mid = n // 2
71     midPoint = P[mid]
72
73     Pl = P[:mid]
74     Pr = P[mid:]
75
76     dl = closestUtil(Pl, Q, mid)
77     dr = closestUtil(Pr, Q, n - mid)
78
79     # Acha a menor distancia
80     d = min(dl, dr)
81
82     stripP = []
83     stripQ = []
84     lr = Pl + Pr
85     for i in range(n):
86         if abs(lr[i].x - midPoint.x) < d:
87             stripP.append(lr[i])
88         if abs(Q[i].x - midPoint.x) < d:
89             stripQ.append(Q[i])
90
91     stripP.sort(key = lambda point: point.y) #<-- REQUIRED
92     min_a = min(d, stripClosest(stripP, len(stripP), d))
93     min_b = min(d, stripClosest(stripQ, len(stripQ), d))
94
95     return min(min_a, min_b)
96
97
98
99 """çã
100 Funo principal da "classe"(orquestradora)
101 """
102 def closest(P, n):
103     P.sort(key = lambda point: point.x)
104     Q = copy.deepcopy(P)
105     Q.sort(key = lambda point: point.y)
106
107
108     return closestUtil(P, Q, n)
109
110
111 n = 5
112
113 P = [Point(0, 2), Point(6, 67),
114       Point(43, 71), Point(39, 107),
115       Point(189, 140)]
116
117 print(f'Smallest distance: {closest(P,n)}')
```

Algoritmo guloso

```
1 INF = 0x3f3f3f3f
2
3 class Graph:
4
5     def __init__(self, vertices) -> None:
6         self.vertices = vertices
7         self.adj_matrix = [[INF for x in range(vertices)] for y in
8                               range(vertices)]
9
10    def add_edge(self, v1, v2, weight) -> None:
11
12        if self.adj_matrix[v2][v1] != INF:
13            self.adj_matrix[v1][v2] = 0
14            self.adj_matrix[v2][v1] = 0
15
16        else:
17            self.adj_matrix[v1][v2] = weight
18
19    def dijkstra(self, origin, destination) -> int:
20
21        spt = [INF for i in range(self.vertices)]
22        spt[origin] = 0
23
24        visited_vertices = set()
25
26        while len(visited_vertices) != self.vertices:
27
28            min_cost_index = -1
29            min_cost_weight = INF
30
31            for i in range(self.vertices):
32
33                if i not in visited_vertices and min_cost_weight > spt[
34                    i]:
35                    min_cost_index = i
36                    min_cost_weight = spt[min_cost_index]
37
38            if min_cost_weight == INF:
39                break
40
41            visited_vertices.add(min_cost_index)
42
43            for i in range(self.vertices):
44
45                travel_weight = spt[min_cost_index] + self.adj_matrix[
46                    min_cost_index][i]
47
48                if spt[i] > travel_weight:
49                    spt[i] = travel_weight
50
51            return spt[destination]
52
53 while 1:
```

```

53
54     graph_input = input().split(' ')
55
56     vertices = int(graph_input[0])
57     edges = int(graph_input[1])
58
59     if vertices == 0 and edges == 0:
60         break
61
62     g = Graph(vertices)
63
64     for i in range(edges):
65
66         edge_input = input().split(' ')
67
68         v1 = int(edge_input[0])
69         v2 = int(edge_input[1])
70         weight = int(edge_input[2])
71
72         g.add_edge(v1 - 1, v2 - 1, weight)
73
74     inquiries = int(input())
75
76     for i in range(inquiries):
77
78         travel_input = input().split(' ')
79
80         origin = int(travel_input[0])
81         destination = int(travel_input[1])
82
83         hours_to_traverse = g.dijkstra(origin - 1, destination - 1)
84
85         if hours_to_traverse < INF:
86             print(f'{hours_to_traverse}')
87         else:
88             print('Impossible')

```

Programação dinâmica

```

1
2 # Implementacao do algoritmo para resolver o problema de
3 # fibonnaci a partir de recursao, usando programacao
4 # dinamica, com a tecnica de memoization
5 def fib(n, memo = {}):
6     if n in memo:
7         return memo[n]
8     if n <= 2:
9         return 1
10    memo[n] = fib(n-1, memo) + fib(n-2, memo)
11    return memo[n]
12
13 print(fib(50))

```