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

Gustavo Lopes Rodrigues[1], **Rafael Amauri Diniz Augusto**[2]

[1]Instituto 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 discorrer 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 existe métodos diferentes, que possuem aplicações diferentes, e com complexidades diferentes. Os principais em questão, são o de *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 usado no lugar do outro.

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

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, o algoritmo 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:

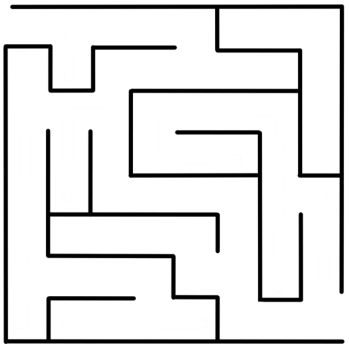

Figure 1. Um exemplo de labirinto

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.

3. Branch and Bound

O método de Ramificar e limitar (em inglês, Branch and bound) é um algoritmo para encontrar soluções ótimas para vários problemas de otimização, especialmente em otimização combinatória. Consiste em uma enumeração sistemática de todos os candidatos a solução, através da qual grandes subconjuntos de candidatos infrutíferos são descartados em massa utilizando os limites superior e inferior da quantia otimizada.

O método foi proposto por A. H. Land e A. G. Doig em 1960 para programação discreta. É utilizado para vários problemas NP-completos como o problema do caixeiro viajante e o problema da mochila.

3.1. Atribuição de trabalho

	Job 1	**Job 2**	**Job 3**
A	9	3	4
B	7	8	4
C	10	5	2

4. 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.

4.1. Abordagem gulosa

Algoritmo guloso é a estratégia que tenta resolver o problema fazendo a escolha localmente ótima em cada fase com a esperança de encontrar um ótimo global que resolve todo o problema.

Na solução de alguns problemas combinatórios a estratégia gulosa pode assegurar a obtenção de soluções ótimas, o que não é muito comum. No entanto, quando o problema a ser resolvido pertencer à classe NP-completo ou NP-difícil, a estratégia gulosa torna-se atrativa para a obtenção de solução aproximada em tempo polinomial.

Um exemplo de abordagem gulosa é o Algoritmo de Dijkstra.

4.1.1. Algoritmo de Dikstra

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.

4.2. Divisão e conquista

Divisão e Conquista é uma estratégia para projeto de algoritmos utilizada pela primeira vez por Anatolii Karatsuba em 1960. Esta técnica consiste em dividir um problema maior recursivamente em problemas menores, até que o problema possa ser resolvido diretamente. Então a solução do problema inicial é dada através da combinação dos resultados de todos os problemas menores computados. Um exemplo famoso de problemas que usam desta estratégia, é o problema de ordenação interna(quicksort e mergesort), e o problema dos pares de pontos próximos.

4.2.1. Pares de pontos próximos

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

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

4.3. 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 divido em subproblemas, que se repetem, logo podem ser memorizados para evitar recálculo.

Antes de aplicar essa técnica, o problema precisa atender duas propriedades: um problema com subproblemas(Overlapping subproblems) e uma subestrutura ideal (Optimal substructure).

- **Overlapping subproblems**: Uma característica também presente em divisão e conquista, essa propriedade é presente em problemas, onde este pode ser divido em subproblemas que são reutilizados várias vezes.
- **Optimal substructure**: Um problema possui uma subestrutura ideal, quando uma solução ótima pode ser construída, usando solução ótima para os seus subproblemas.

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

4.3.1. Fibonnaci

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

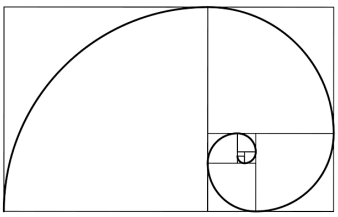

Figure 2. Uma imagem ilustrativa da sequência Fibonnaci

A ideia da 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 fibonnaci.

```
1
2 # Implementacao ingenua de um
3 # algoritmo para resolver fibonnaci
4 # de forma recursiva
5 def fib(n):
6     if n <= 2:
7         return 1
8
9     return (fib(n-1) + fib(n-2))
10
11 print(fib(50))
```

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 6,7 e 8 número da sequência, porém, o código não consegue calcular o 50° número.

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

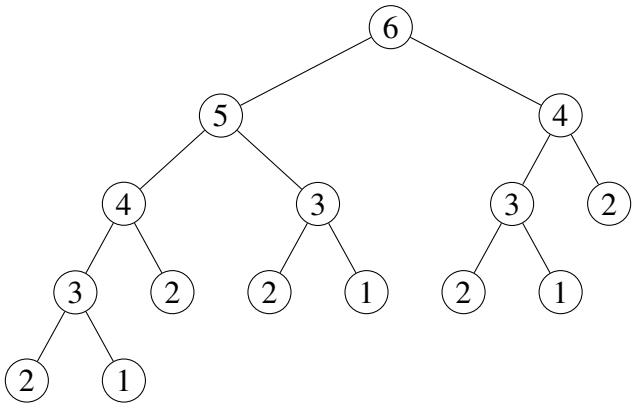

Figure 3. encontrando o 6° número da sequência fibonnaci

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 50° número, já que este levaria um tempo equivalente a 2^{50}.

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

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 assim.

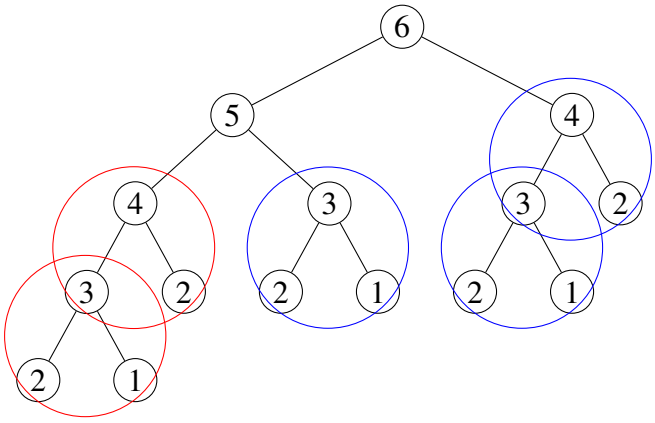

Figure 4. Vermelho são os passos que são calculados, e o que está em azul é repetido

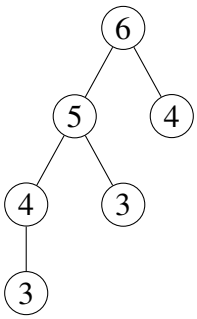

Figure 5. Árvore de recursão otimizada

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:

```

# Implementacao do algoritmo para resolver o problema de
# fibonnaci a partir de recursao, usando programacao
# dinamica, com a tecnica de memoization
def fib(n, memo = {}):
    if n in memo:
        return memo[n]
    if n <= 2:
        return 1
    memo[n] = fib(n-1,memo) + fib(n-2,memo)
    return memo[n]

print(fib(50))
```

```

# Implementacaoo do algoritmo para resolver o problema de
# fibonnaci a partir de recursao, usando programacao
# dinamica, com a tecnica de tabulation
def fib(n):
    table = [0 for i in range(n+1)]
```

```
    table[1] = 1

    for i in range(n-1):
        table[i + 1] += table[i]
        table[i + 2] += table[i]

    return table[n-1] + table[n-2]

print(fib(50))
```

5. References

Bibliographic references must be unambiguous and uniform. We recommend giving the author names references in brackets, e.g. [Knuth 1984], [Boulic and Renault 1991], and [Smith and Jones 1999].

The references must be listed using 12 point font size, with 6 points of space before each reference. The first line of each reference should not be indented, while the subsequent should be indented by 0.5 cm.

References

Boulic, R. and Renault, O. (1991). 3d hierarchies for animation. In Magnenat-Thalmann, N. and Thalmann, D., editors, *New Trends in Animation and Visualization*. John Wiley & Sons ltd.

Knuth, D. E. (1984). *The TEX Book*. Addison-Wesley, 15th edition.

Smith, A. and Jones, B. (1999). On the complexity of computing. In Smith-Jones, A. B., editor, *Advances in Computer Science*, pages 555–566. Publishing Press.