

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 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 um tipo de algoritmo que representa um refinamento da busca por força bruta, em que múltiplas soluções podem ser eliminadas sem serem explicitamente examinadas. O termo foi cunhado pelo matemático estado-unidense D. H. Lehmer na década de 1950.

O procedimento é usado em linguagens de programação como Prolog. Uma busca inicial em um programa nessa linguagem segue o padrão busca em profundidade, ou seja, a árvore é percorrida sistematicamente de cima para baixo e da esquerda para direita.

Quando essa pesquisa falha, ou é encontrado um nodo terminal da árvore, entra em funcionamento o mecanismo de backtracking. Esse procedimento faz com que o sistema retorne pelo mesmo caminho percorrido com a finalidade de encontrar soluções alternativas.

2.1. Problema do labirinto

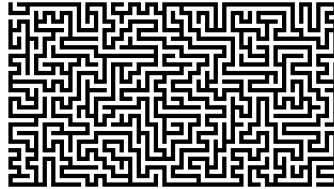


Figure 1. Um exemplo de labirinto

Problema do labirinto

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

Section titles must be in boldface, 13pt, flush left. There should be an extra 12 pt of space before each title. Section numbering is optional. The first paragraph of each section should not be indented, while the first lines of subsequent paragraphs should be indented by 1.27 cm.

4.1. Abordagem gulosa

Algoritmo guloso ou míope é técnica de projeto de algoritmos que tenta resolver o problema fazendo a escolha localmente ótima em cada fase com a esperança de encontrar um ótimo global.

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.

4.2. Divisão e conquista

Divisão e Conquista (do inglês Divide and Conquer) em computação é uma técnica de projeto de algoritmos utilizada pela primeira vez por Anatolii Karatsuba em 1960 no algoritmo de Karatsuba.

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. Vários problemas podem ser solucionados através desta técnica, como a ordenação de números através do algoritmo merge sort e a transformação discreta de Fourier através da transformada rápida de Fourier. Outro problema clássico que pode ser resolvido através desta técnica é a Torre de Hanoi.

4.2.1. Pares de pontos próximos

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 dividido 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 dividido 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

Descoberto 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.

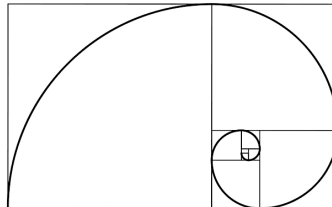


Figure 2. Uma imagem ilustrativa da sequência Fibonnaci

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 50º número.

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

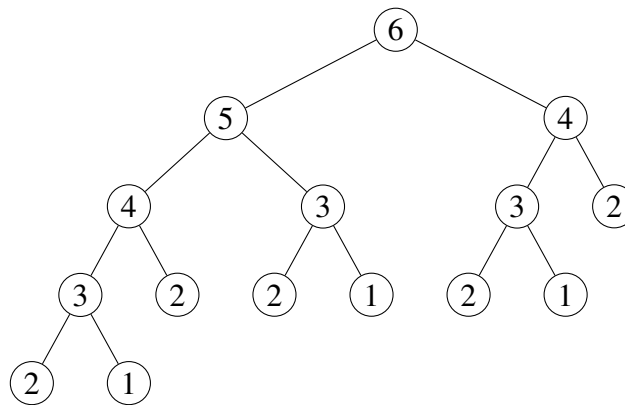


Figure 3. encontrando o 6º número da sequência fibonnaci

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 50º 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.

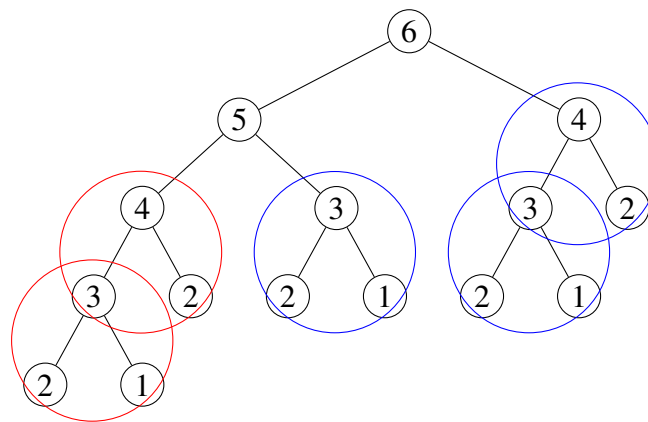


Figure 4. Vermelho são os passos que são calculados, e o que está em azul é repetido

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.

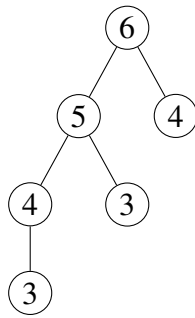


Figure 5. Árvore de recursão otimizada

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:

```

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))

1
2 # Implementacao do algoritmo para resolver o problema de
3 # fibonnaci a partir de recursao, usando programacao
4 # dinamica, com a tecnica de tabulation
5 def fib(n):
6     table = [0 for i in range(n+1)]
7     table[1] = 1
8
9     for i in range(n-1):
10         table[i + 1] += table[i]
11         table[i + 2] += table[i]
12
13     return table[n-1] + table[n-2]
14
15 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 T_EX 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.