PARADIGMAS DE PROJETO DE ALGORITMOS

DCE529 - Algoritmos e Estruturas de Dados III

Atualizado em: 21 de agosto de 2024

Iago Carvalho

Departamento de Ciência da Computação

Projeto de algoritmos é um método específico para criar um processo matemático na resolução de problemas.

- Quando aplicado, da-se origem a engenharia de algoritmos

Existem diversas formas de se realizar o projeto de algoritmos

- Cada *forma de pensamento* constitui um diferente paradigma
  - **Recursividade**
  - **Força bruta**
  - **Guloso**
  - Programação dinâmica
  - Divisão e conquista

RECURSIVIDADE

Um algoritmo ou procedimento é dito ser **recursivo** quando ele chama a si mesmo

- Direta ou indiretamente
- Número indefinido de chamadas

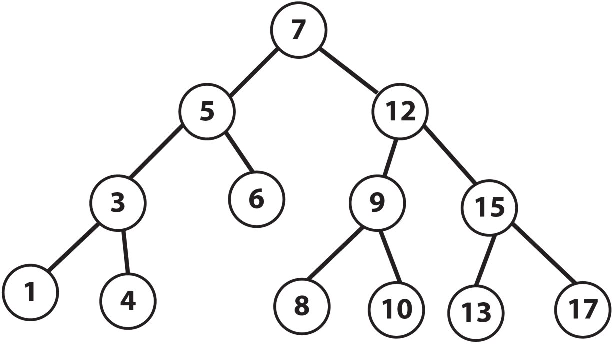
Recursividade permite descrever algoritmos de forma mais clara e concisa

- Existem problemas que são recursivos por natureza

ESTRUTURA RECURSIVA

Árvore binária de pesquisa

- Itens menores estão a esquerda
- Itens maiores estão a direita



BUSCA EM ÁRVORE BINÁRIA - CAMINHAMENTO CENTRAL

```
typedef long TipoChave;

typedef struct {
 TipoChave Chave;
 /* outros componentes */
} Registro;

typedef struct {
 Registro Reg;
 struct Nodo *Esq, *Dir;
} Nodo;
```

```
void Central(Nodo *p)
{
 if (p == NULL)
   return;
 Central(p -> Esq);
 printf("%d\n", p -> Reg.Chave);
 Central(p -> Dir);
}
```

$$T(n) = T\left(\frac{n}{2}\right) + 1 = \Theta(\log n)$$

ALGORITMOS RECURSIVOS

Utilizam *pilhas* para armazenar os dados utilizados em cada etapa recursiva

- Pilha de estados
- Dados não globais são enviados para a pilha como forma de registrar a computação
- Dados são recuperados ao fim das chamadas recursivas

No caso do algoritmo de busca em árvore binária

- Para cada chamada recursiva, são empilhados o valor de p e o local da chamada recursiva
- Quando $p == NULL$, o procedimento retorna
  - Desempilha uma entrada

ALGORITMOS RECURSIVOS

Faz-se necessário uma *condição de parada*

- Alguma condição que, se satisfeita, finaliza as chamadas recorrentes

Um procedimento recursivo P deve estar sujeito a uma condição de parada B

- Deve-se **garantir** que B não seja satisfeita em algum momento
  - $P! = NULL$

Um bom esquema para procedimentos recursivos

$$P \equiv \text{se } B \text{ então } \mathcal{C}[S_i, P]$$

Para demonstrar que uma repetição termina, define-se uma função $f(x)$, sendo x o conjunto de variáveis do programa, tal que

- $f(x) \leq 0$ é a condição de terminação
- $f(x)$ é decrementada em uma unidade a cada chamada recursiva

$$P \equiv \text{se } n > 0 \text{ então } \mathcal{C}[S_i, P(n-1)]$$

QUANDO NÃO UTILIZAR RECORRÊNCIA

Existem casos interessantes de utilizarmos algoritmos recursivos

- O algoritmo de busca binária é um exemplo

Entretanto, existem outros casos onde a utilização de recursão é ruim

- Gasto excessivo de memória com a pilha de estados

Tais casos são caracterizados por chamadas de recorrência como

$$P \equiv \text{se } B \text{ então } \mathcal{C}[S, P]$$

CALCULO DE FIBONACCI

Uma sequência de Fibonacci é uma série matemática na qual cada número da série é dado pela soma dos dois anteriores

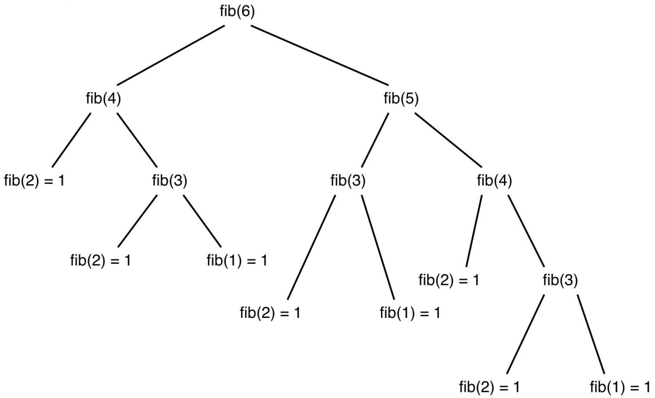
- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \ldots$

Pode-se computar esta sequência de Fibonacci acima utilizando as seguintes regras

- $f_0 = 0$
- $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}, \forall n \geq 2$

ALGORITMO RECURSIVO PARA FIBONACCI

```
int Fib(int n){
 if (n < 2)
   return n;
 else
   return Fib(n-1) + Fib(n-2);
}
```



ALGORITMO RECURSIVO PARA FIBONACCI

O algoritmo recursivo para Fibonacci é extremamente inificiente

- Espaço: $f(n) = \mathcal{O}(\phi^n)$, onde $\phi \approx 1,618$ é a razão áurea
- Tempo: Como cada computo leva $\Theta(1)$, então a complexidade de tempo é igual a de espaço

ALGORITMO ITERATIVO PARA FIBONACCI

```
int FibIte (int n){
  int i = 1, k, F = 0;
  for (k = 1; k <= n; k++){
    F += i;
    i = F - i;
  }
  return F;
}
```

- Complexidade de tempo: $f(n) = \mathcal{O}(n)$
- Complexidade de espaço: $f(n) = \mathcal{O}(1)$

ALGORITMOS PARA FIBONACCI

n	10	20	30	50	100
Recursiva	8 ms	1 s	2 min	21 dias	10^9 anos
Iterativa	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

FORÇA BRUTA (TENTATIVA E ERRO)

FORÇA BRUTA

São aqueles algoritmos que testam **todas** as soluções possíveis de um problema

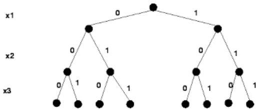
- Todos os caminhos entre dois pontos
- Todas as senhas possíveis em um sistema
- ...

Esta amostragem é feita de forma indiscriminada

- Não existe nenhum mecanismo que diz se a busca pode ser encerrada

Soluções são enumeradas de forma semelhante a uma árvore

- Na maioria das vezes, esta árvore cresce exponencialmente



FORÇA BRUTA

Algoritmos de força bruta não seguem regras fixas de computação

1. Tenta e registra um passo em direção a solução final
2. Caso o passo leve a solução
   - Registra-se o passo e retorna a solução encontrada
3. Caso o passo não leve a uma solução
   - Ele é apagado e retirado do registro

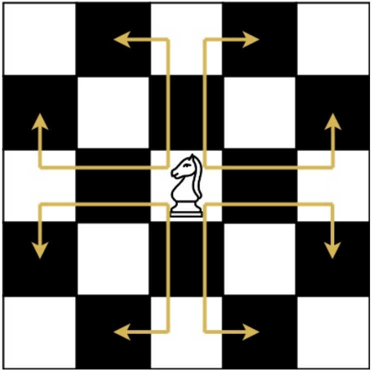
Algoritmos de força bruta são simples e fáceis de implementar

- Entretanto, não são úteis para a resolução de problemas de médio ou grande porte

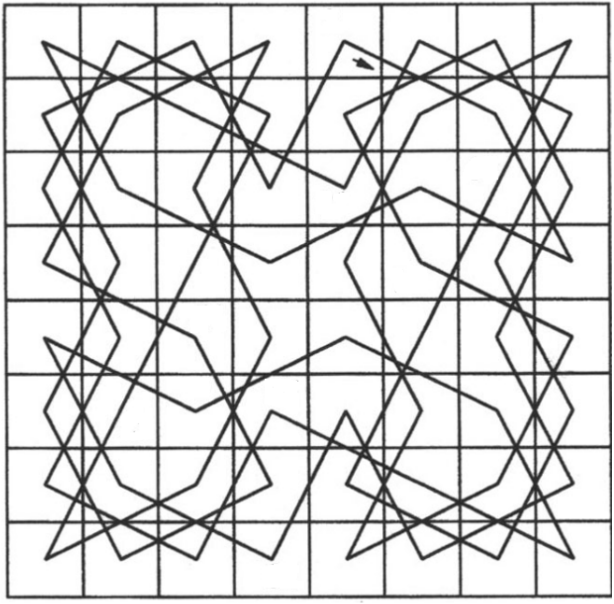
PASSEIO DO CAVALO

Um problema interessante é o *Problema do Passeio do Cavalo*

- Tabuleiro de xadrez com n por m posições
- Cavalo inicialmente posicionado na casa x_0, y_0
- Cavalo tem que passar por todas as casas sem repetição



PASSEIO DO CAVALO



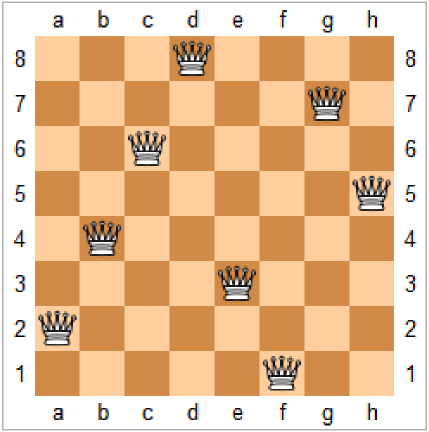
PASSEIO DO CAVALO

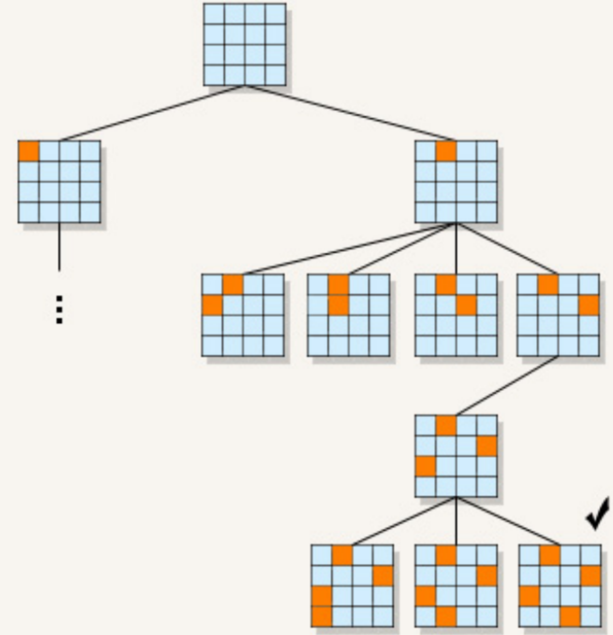
```
void tenta(){
  inicialização da seleção de movimentos;

  do{
    seleciona próximo candidato ao movimento
    if(aceitável){
      registra movimento
      if(tabuleiro não está cheio){
        tenta novo movimento //chamada recursiva de tenta
        if(não sucedido) apaga registro anterior;
      }
    }
  }while(movimento não sucedido e não acabaram candidatos)
}
```

N-RAINHAS

Outro problema interessante é o *Problema das N-Rainhas*

- Tabuleiro de xadrez com *n* por *n* posições
- Deve-se posicionar *n* rainhas neste tabuleiro
- Nenhuma rainha deve ser capaz de atacar a outra





COMO MELHORAR O FORÇA BRUTA

Existem algumas maneiras de se implementar um algoritmo de força bruta de maneira mais eficiente

Backtracking

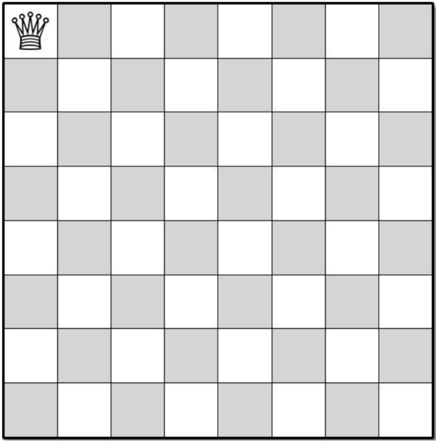
- Monta uma árvore de possíveis estados do problema
- Percorre a árvore de maneira ordenada
  - Busca em largura
  - Busca em profundidade
  - ...
- Possui mecanismo para poda de estados ineficientes

Branch-and-bound

- Quebra o problema em subproblemas menores
- Utiliza algoritmos de corte para eliminar subproblemas que não levam a solução

Vamos fazer um algoritmo backtracking para o problema das N-Rainhas

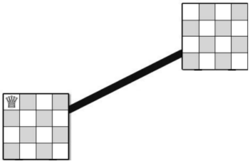
Inicialmente, vamos inserir uma rainha na primeira posição



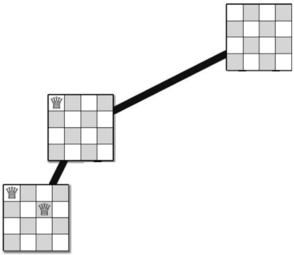
Quantos possíveis estados existem?

- $8^8 = 16.777.216$
- 92 estados corretos

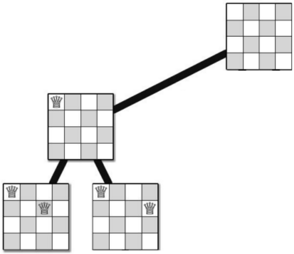
BACKTRACKING PARA N-RAINHAS



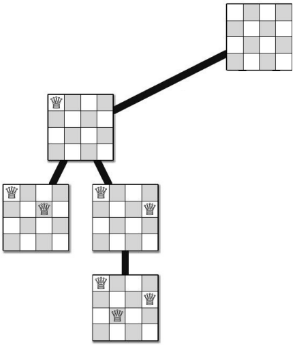
BACKTRACKING PARA N-RAINHAS



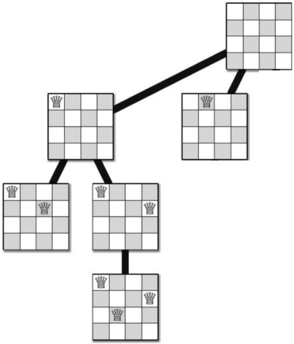
BACKTRACKING PARA N-RAINHAS



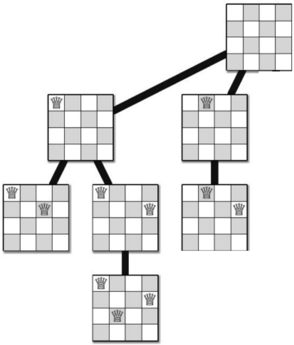
BACKTRACKING PARA N-RAINHAS



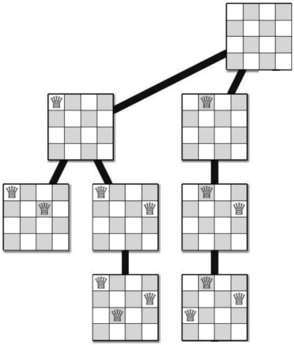
BACKTRACKING PARA N-RAINHAS



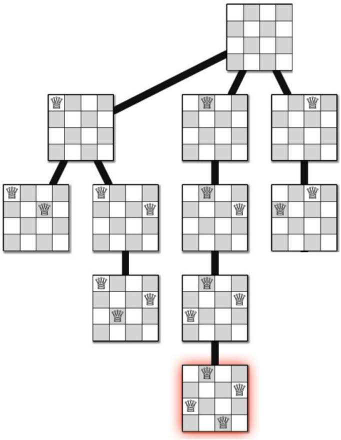
BACKTRACKING PARA N-RAINHAS



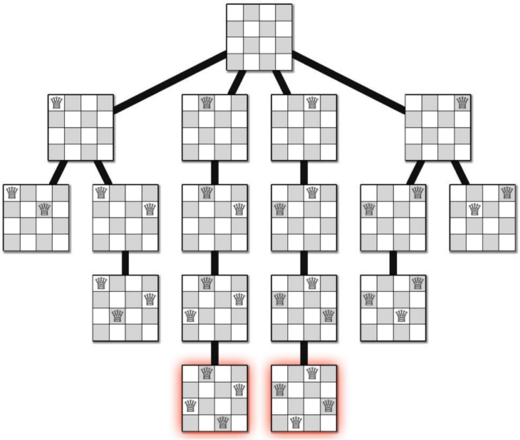
BACKTRACKING PARA N-RAINHAS



BACKTRACKING PARA N-RAINHAS



BACKTRACKING PARA N-RAINHAS



Algoritmo de Gauss

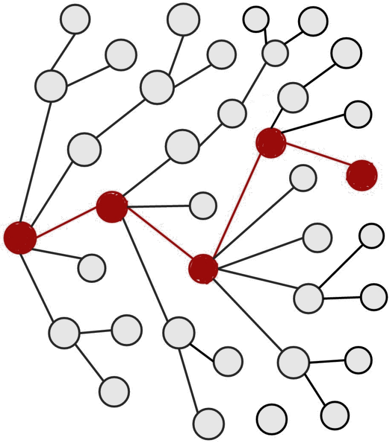
```
funcao posicionarRainhas (t, n, r) {
    entrada: vetor t representando um tabuleiro indexado por[1..n], e a linha atual r
    saída: soluções para o problema das n rainhas são impressas

    se r = n + 1 {
        imprimaSolução();
        retorne
    }
    para j = 1 até n {
        legal = verdadeiro
        para i = 1 até r - 1 {
            se (t[i] = j ou t[i] = j + r - i ou t[i] = j - r + i) {
                legal = falso
            }
        }
        se legal == verdadeiro {
            t[r] = j
            posicionarRainhas (t, n, r + 1)
        }
    }
    retorne
}
```

ALGORITMOS GULOSOS

ALGORITMO GULOSO

Uma simplificação do algoritmo de força bruta

- Parecido com uma busca única em uma árvore de força bruta
- Escolhe uma opção e nunca volta atrás



ALGORITMO GULOSO

Todas as decisões tomadas são *míopes*

- Isto é, elas não enxergam a frente
- Não consideram passos futuros
- Consideram unicamente a melhor decisão local

Existem problemas que são solucionados, em sua otimalidade, por algoritmos gulosos

- Computar a árvore geradora mínima de um grafo
- Encontrar o menor caminho entre dois pontos em um grafo

Entretanto, existem outros problemas para os quais algoritmos gulosos encontram somente soluções aproximadas

- Problema da mochila binária
- Problema do caixeiro viajante
- Problema da coloração de grafos

SUBESTRUTURA ÓTIMA

Algoritmos gulosos funcionam muito bem em problemas que contenham uma subestrutura ótima

- Qualquer sub-parte de uma solução ótima do problema também é ótima

No geral, problemas que contém uma subestrutura ótima pertencem a P

- Podem ser resolvidos em tempo polinomial com um algoritmo determinístico

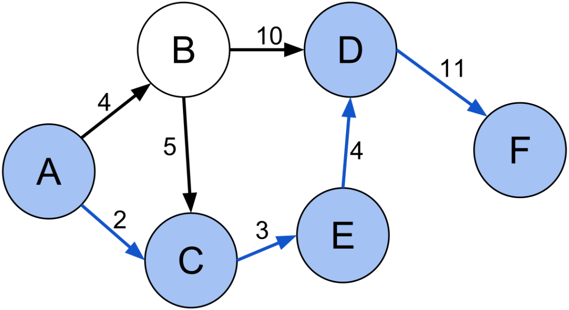
De forma complementar, se um problema não possui subesturutra ótima, provavelmente ele pertence a NP

- Não podem ser resolvidos em tempo polinomial com um algoritmo determinístico

SUBESTRUTURA ÓTIMA - CAMINHO MAIS CURTO

O caminho mais curto de A para F tem custo 20

O caminho mais curto de A para C, D e E está incluído no caminho de A para F

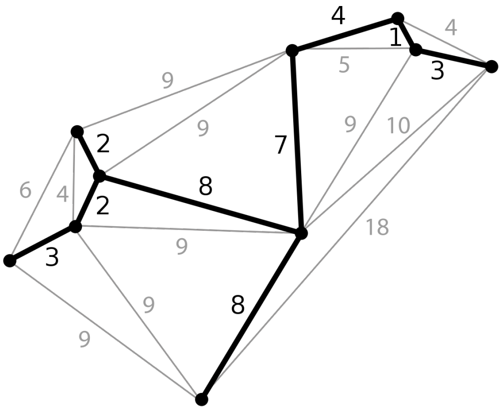


SUBESTRUTURA ÓTIMA - ÁRVORE GERADORA MÍNIMA

A Árvore Geradora Mínima deste grafo tem peso 30

Qualquer sub-árvore escolhida dentro da árvore geradora mínima também é mínima

- Qualquer subconjunto de vértices



PASSOS DE UM ALGORITMO GULOSO

1. Seleciona um elemento conforme uma função gulosa
2. Marca o elemento como escolhido
    - Evita selecionar o mesmo elemento por duas vezes ou mais
3. Atualiza a entrada
4. Examina o elemento quanto a sua viabilidade
5. Decide se o elemento participa ou não da solução