SBVORIN: Organização e Recuperação da Informação

Aula 02: Backtracking e Programação Dinâmica

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Backtracking

- O paradigma de Backtracking, ou "Tentativa e Erro", é relacionado à Busca Completa, ou seja, explorar inteiramente o espaço de soluções, se necessário, em busca da solução desejada;
- É considerado um refinamento da busca completa, uma vez que pode eliminar parte do espaço de soluções sem examiná-lo;
- O princípio é construir uma solução gradativamente na base da tentativa e erro, desmanchando partes da solução quando necessário;
- Normalmente, este paradigma é associado a algoritmos recursivos.

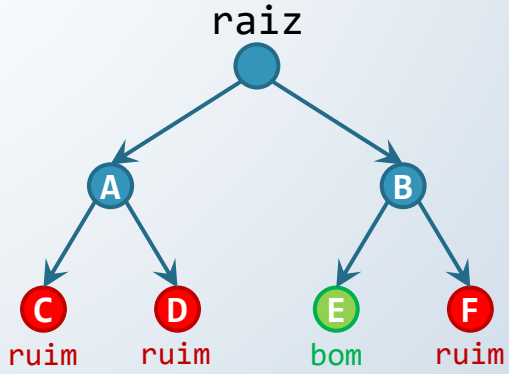
"Backtracking is a type of exhaustive search in which the combinatorial object is constructed recursively, and the recursion tree is pruned, that is, recursive calls are not made when the part of the current object that has been constructed cannot lead to a valid or optimal solution." - Ian Parberry, *Problems on Algorithms*

Backtracking

- O backtracking é aplicado em problemas cuja solução pode ser definida construtivamente, por meio de uma sequência de decisões;
- Outra característica dos problemas passíveis de solução por backtracking é a capacidade de modelagem por uma árvore que representa todas as possíveis sequências de decisão. Essa árvore é chamada de "Árvore de Backtracking".

Backtracking

- **Exemplo:** Suponha um problema genérico em que começamos no nó raiz de uma árvore, que pode ter nós folhas bons e ruins, em qualquer proporção e distribuição;
- **Objetivo:** Alcançar um nó folha bom;
- A partir da raiz, escolhemos sucessivamente nós filhos, até chegarmos a um nó folha:
  - Se o nó folha é bom, paramos;
  - Se o nó folha é ruim, desfazemos nossa última escolha e tomamos outra opção:
    - Caso não haja outra opção, desfazemos a penúltima escolha, e assim sucessivamente;
    - Se voltarmos à raiz da árvore e não houver mais opções, não há folhas boas na árvore.



Backtracking

- Se houver mais do que uma opção disponível para cada uma das n decisões, a busca completa será exponencial;
- A eficiência do backtracking depende da capacidade de limitar esta busca, ou seja, podar a árvore;
- **Podar a árvore:** eliminar as ramificações não promissoras;
- É necessário definir o espaço de busca para o problema:
  - Que inclua a solução ótima;
  - Que possa ser pesquisada de forma organizada, tipicamente, como uma árvore.

Backtracking

- O backtracking é baseado na tentativa e erro;
- De acordo com a característica do problema computacional tratado, construímos uma solução gradativamente, passo a passo:
  - A cada passo, toma-se uma decisão que adicionará um novo elemento à solução parcial;
  - Esta nova solução na verdade é uma tentativa de estender a solução parcial atual, no intuito de termos uma solução completa;
- Se chegarmos em algum ponto em que, por algum motivo, não seja possível estender a solução e não tivermos uma solução completa, ou seja, um erro, pode-se então "voltar atrás", dai o termo backtrack, desfazendo a tentativa.

Backtracking

```
// versão genérica em pseudocódigo
backtracking( v )
entrada: vértice inicial v
se promissor( v ) então
    se éSoluçãoCompleta( v ) então
        armazenarSolução( v );
    fim
fim
para cada filho de v faça
    backtracking( filho );
fim
```

Backtracking

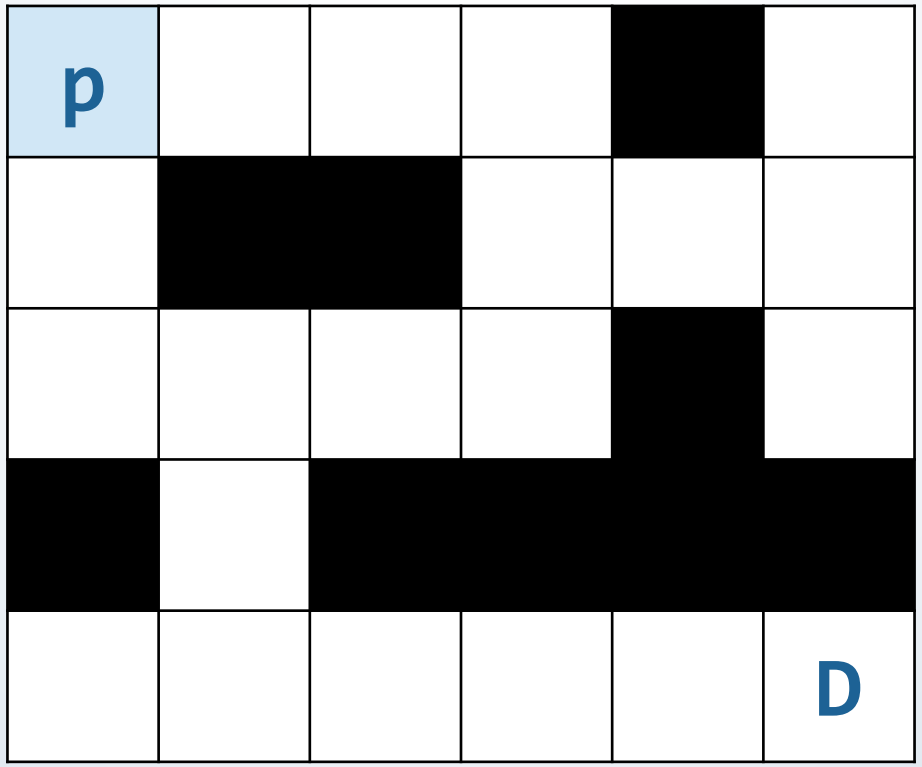
- Ideia central:
  - Programação com retrocesso é mais eficiente para fazer a busca exaustiva;
  - Aplicável quando as soluções candidatas podem ser construídas incrementalmente;
  - Retroceder quando detectar que a solução candidata é inviável;

```
backtrackingRecursivo( soluçãoParcial p ) {
    se ( p é válida ) {
        se ( p é solução ) {
            imprimir p;
        }
        enquanto ( puder estender p para algum s ) {
            backtrackingRecursivo( s );
        }
    }
}
```

- Algoritmos com backtracking em geral utilizam recursão, pois facilita o mecanismo de retrocesso.

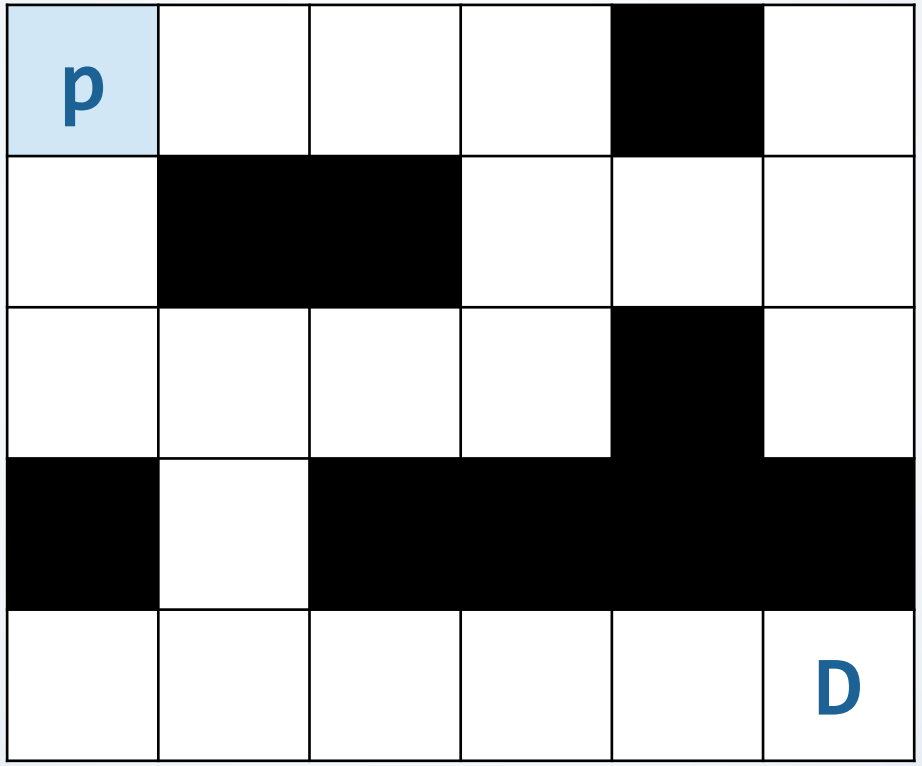
Backtracking
Resolução de Labirinto

- **Problema:** Encontrar um caminho de saída de um labirinto partindo em um ponto **p** e chegando no destino **D**.



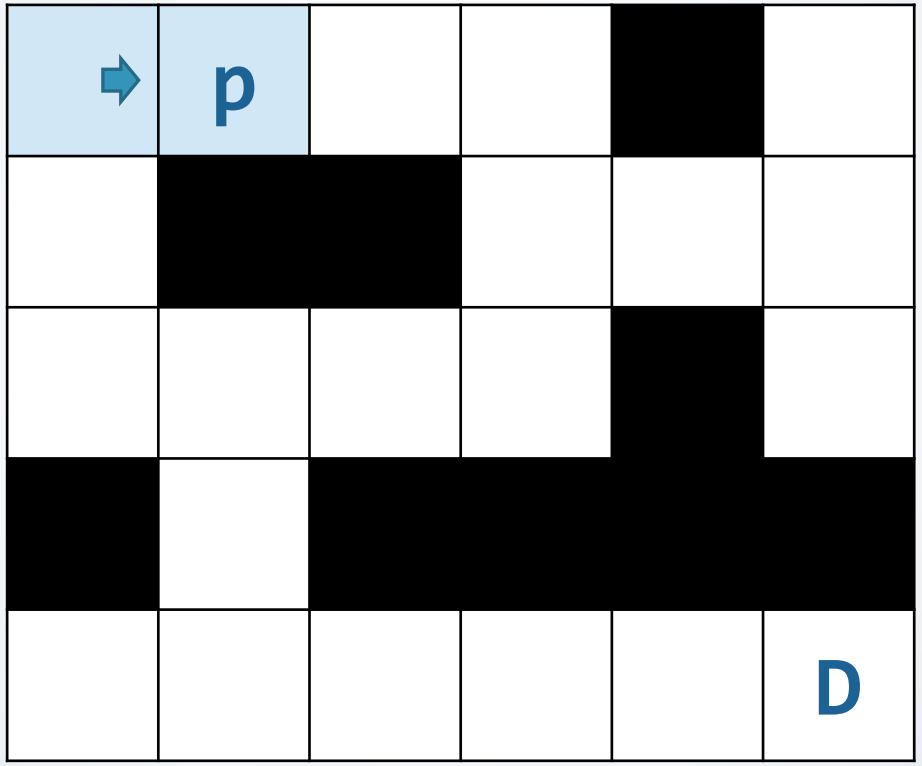
Backtracking

Resolução de Labirinto



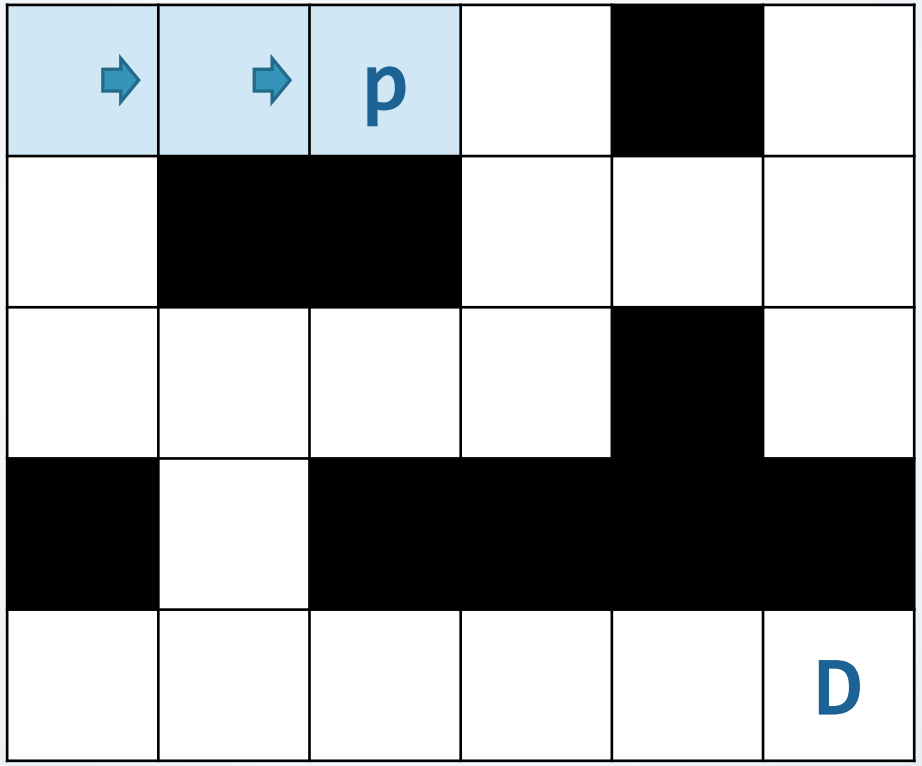
Backtracking

Resolução de Labirinto



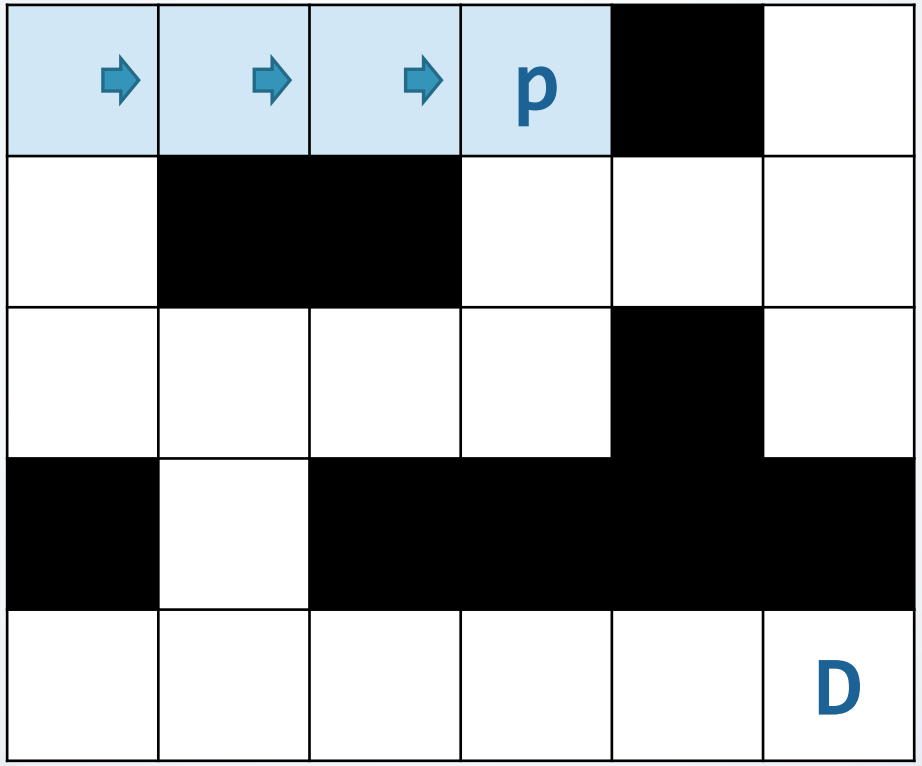
Backtracking

Resolução de Labirinto



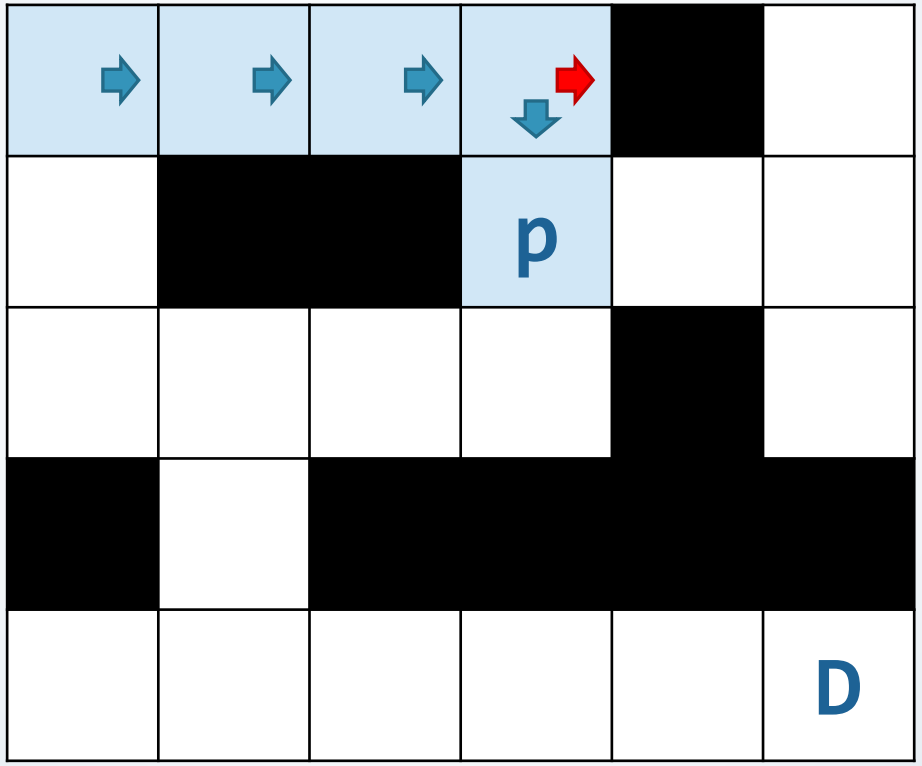
Backtracking

Resolução de Labirinto

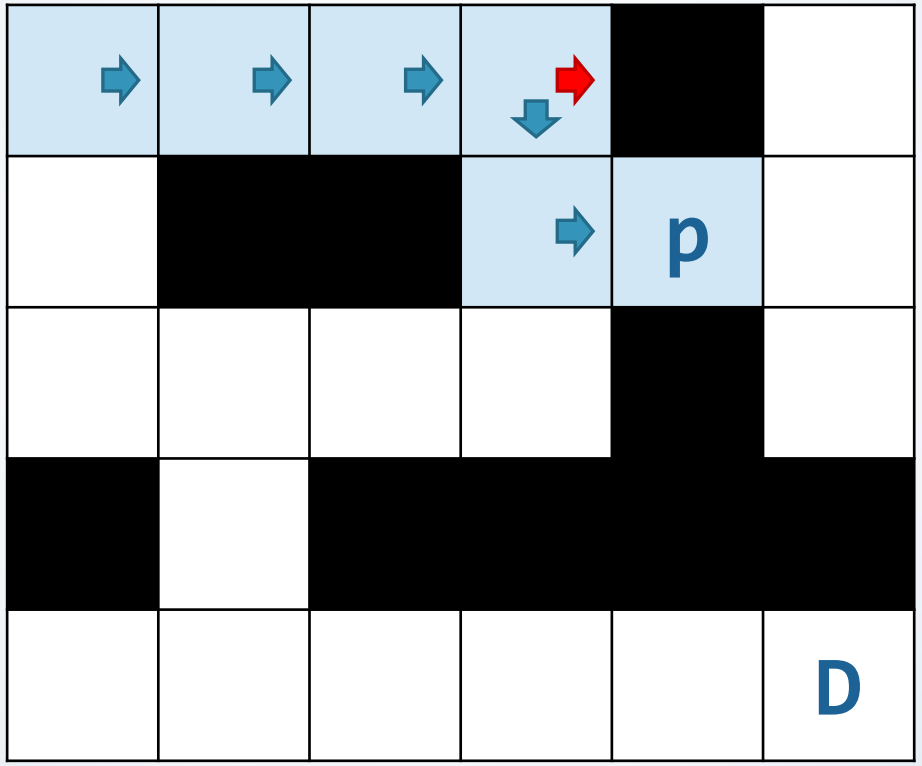


Backtracking

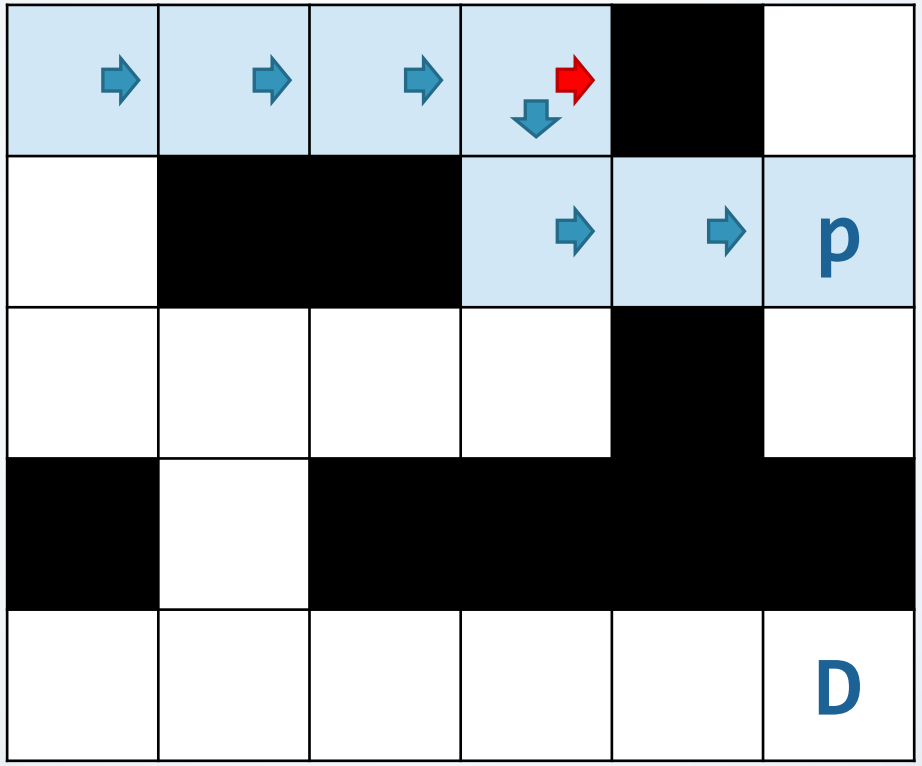
Resolução de Labirinto



Backtracking
Resolução de Labirinto

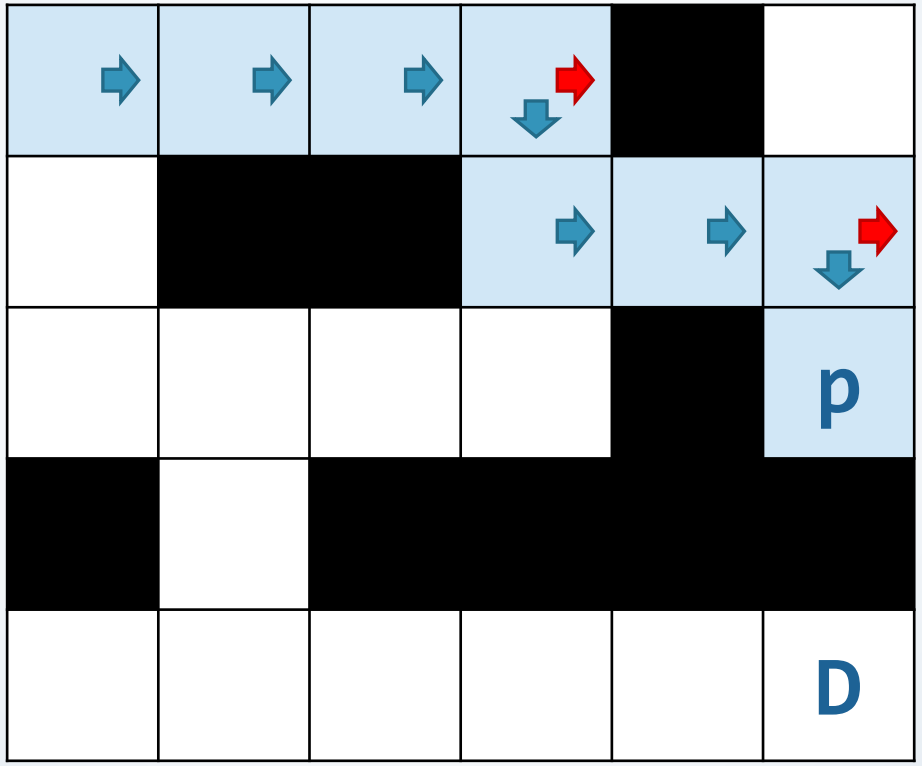


Backtracking
Resolução de Labirinto



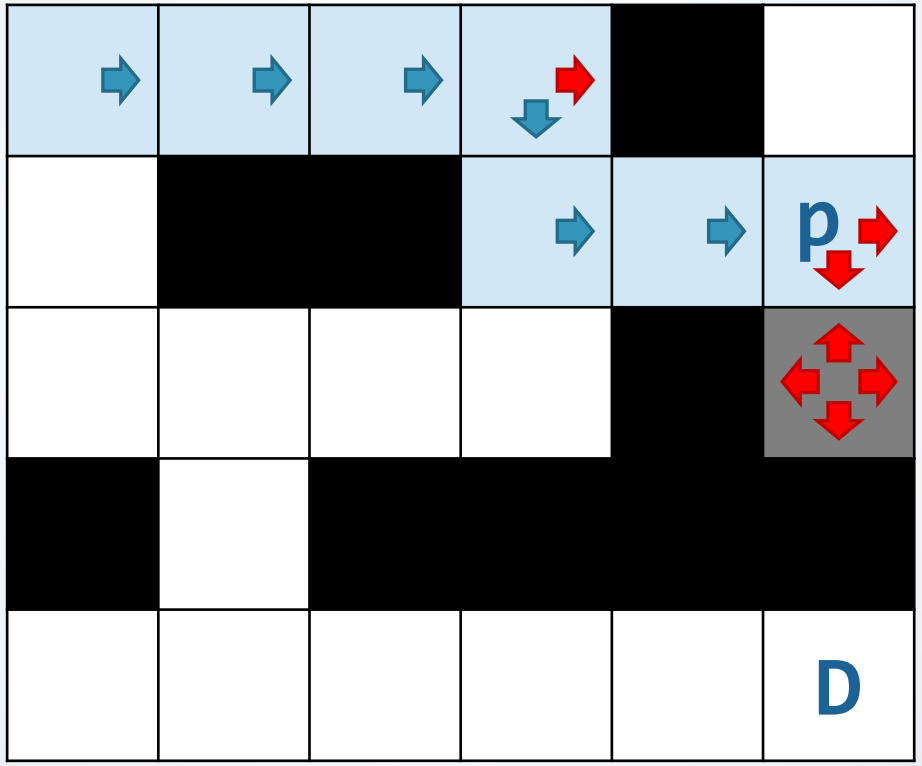
Backtracking

Resolução de Labirinto



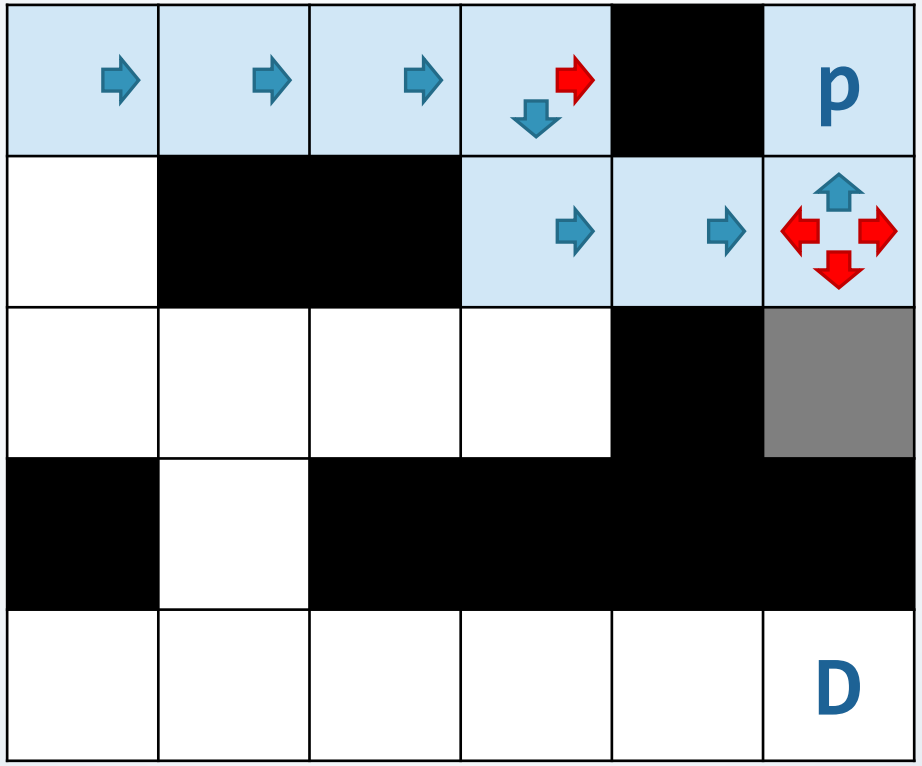
Backtracking

Resolução de Labirinto



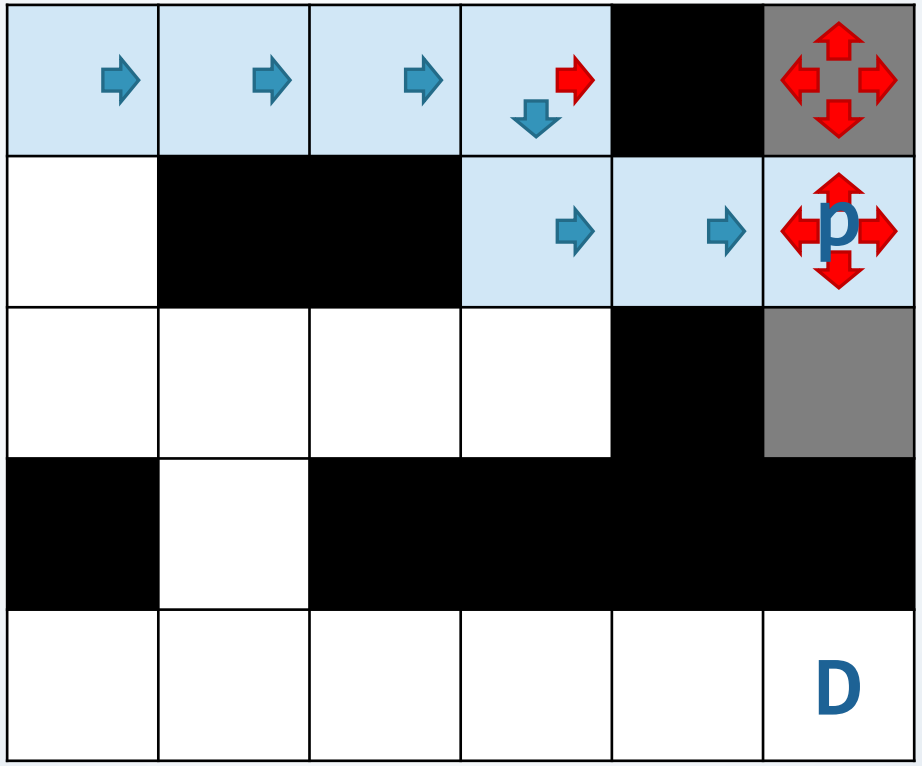
Backtracking

Resolução de Labirinto



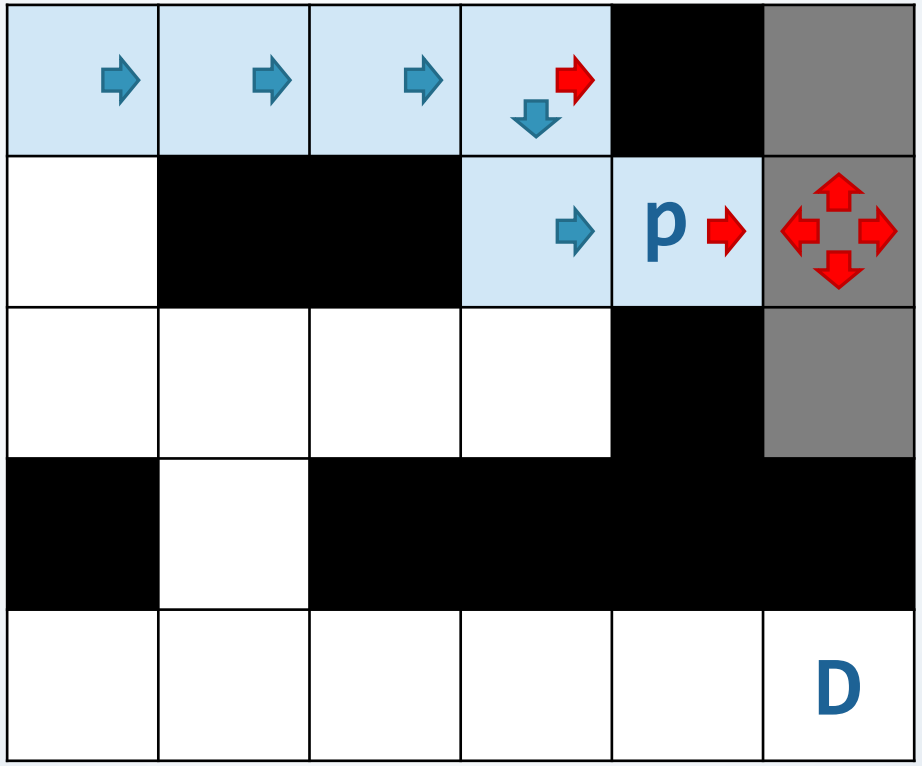
Backtracking

Resolução de Labirinto



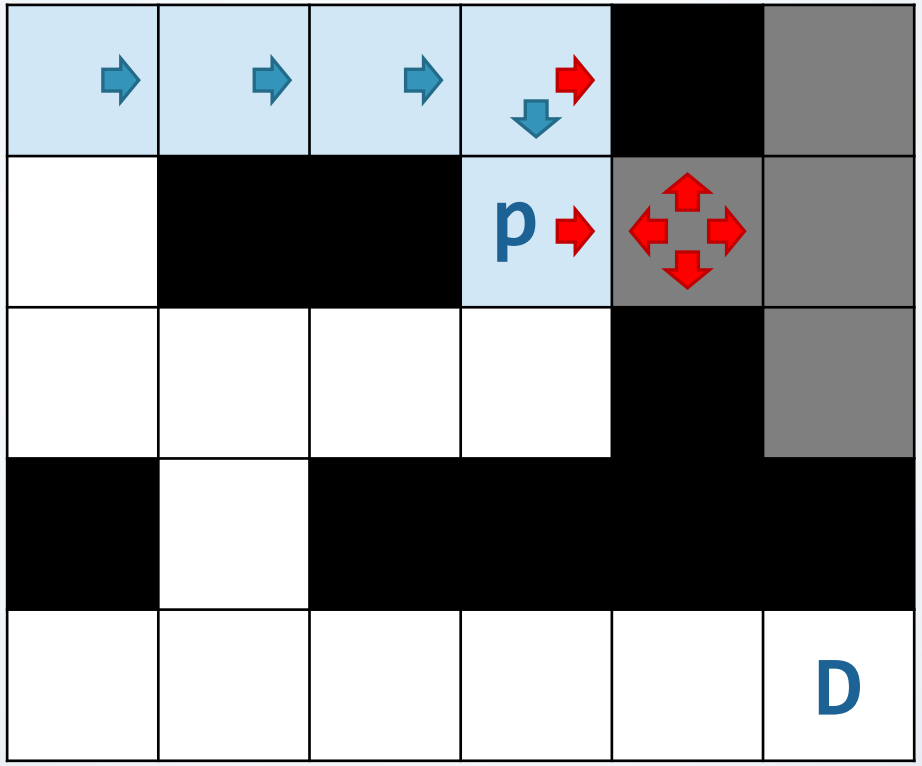
Backtracking

Resolução de Labirinto



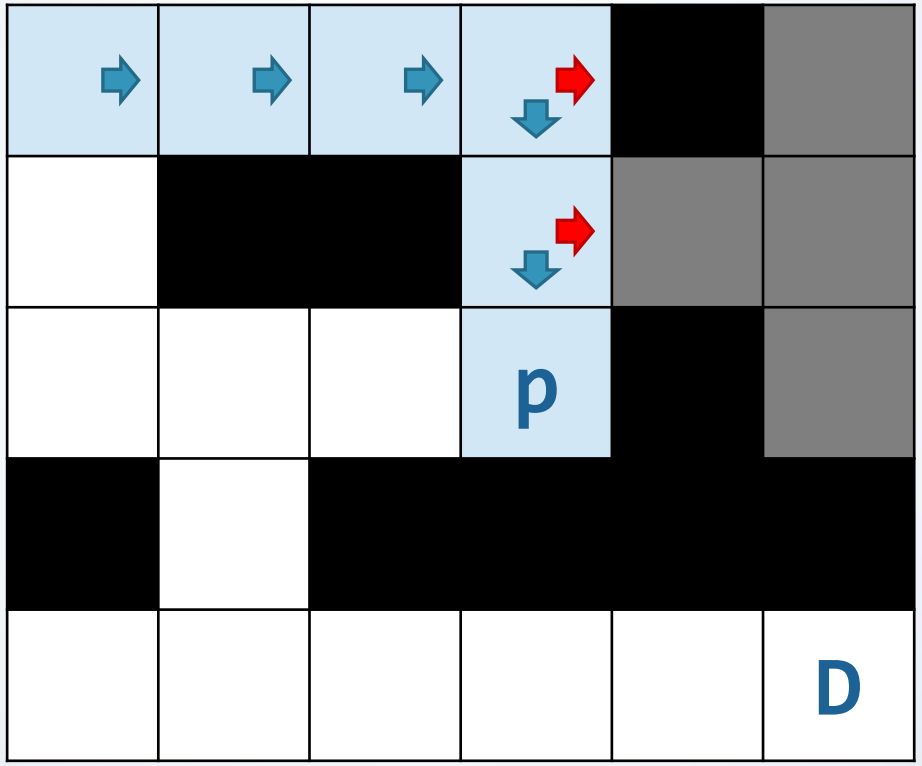
Backtracking

Resolução de Labirinto



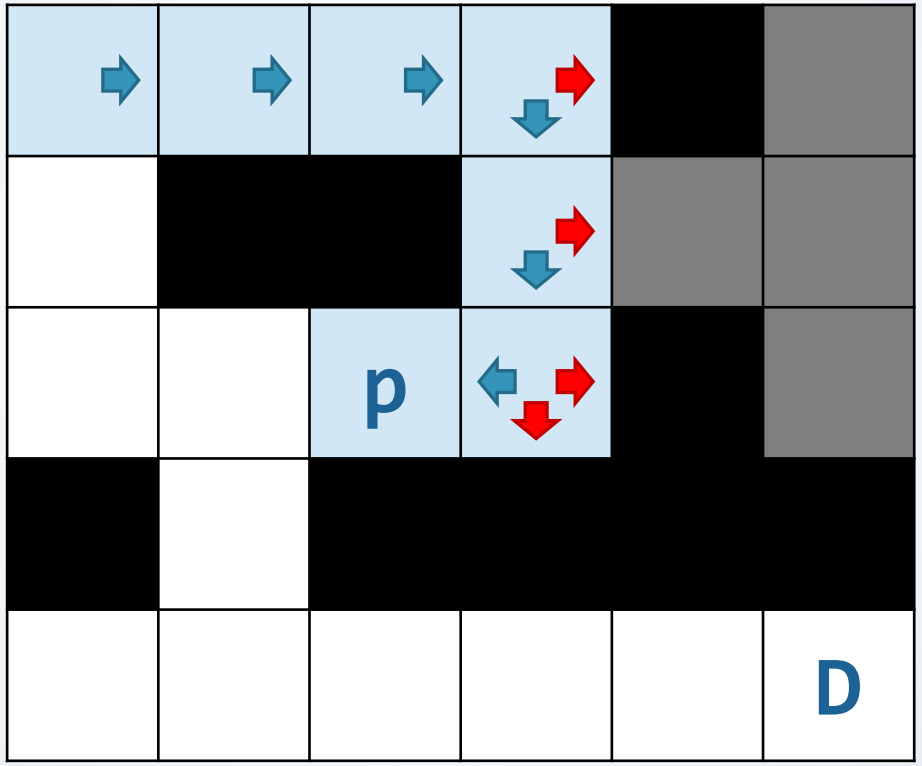
Backtracking

Resolução de Labirinto



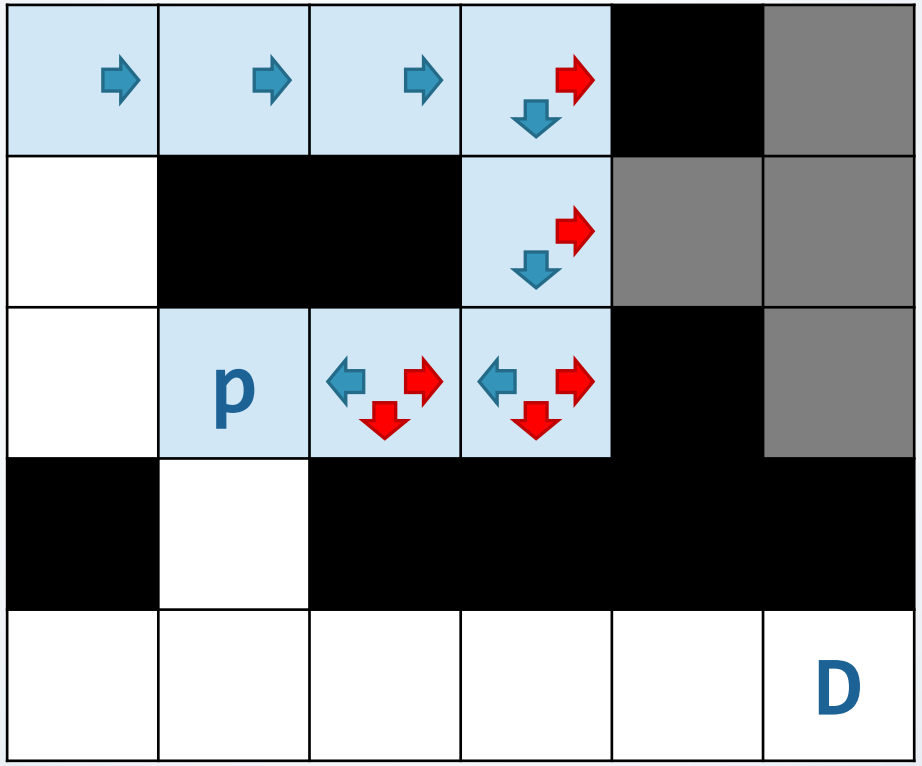
Backtracking

Resolução de Labirinto

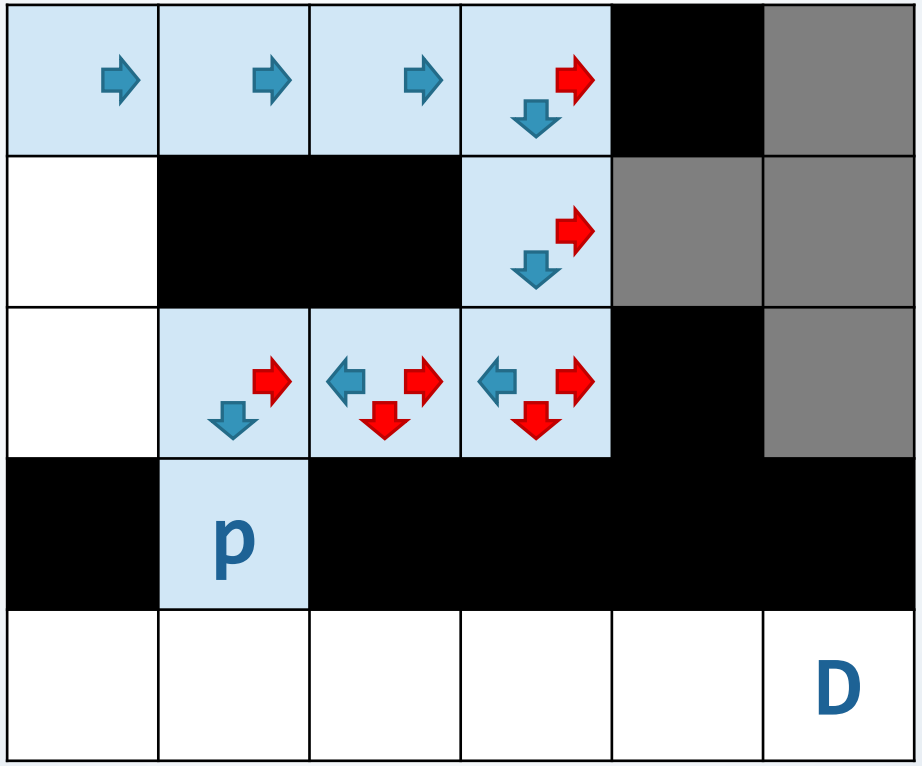


Backtracking

Resolução de Labirinto

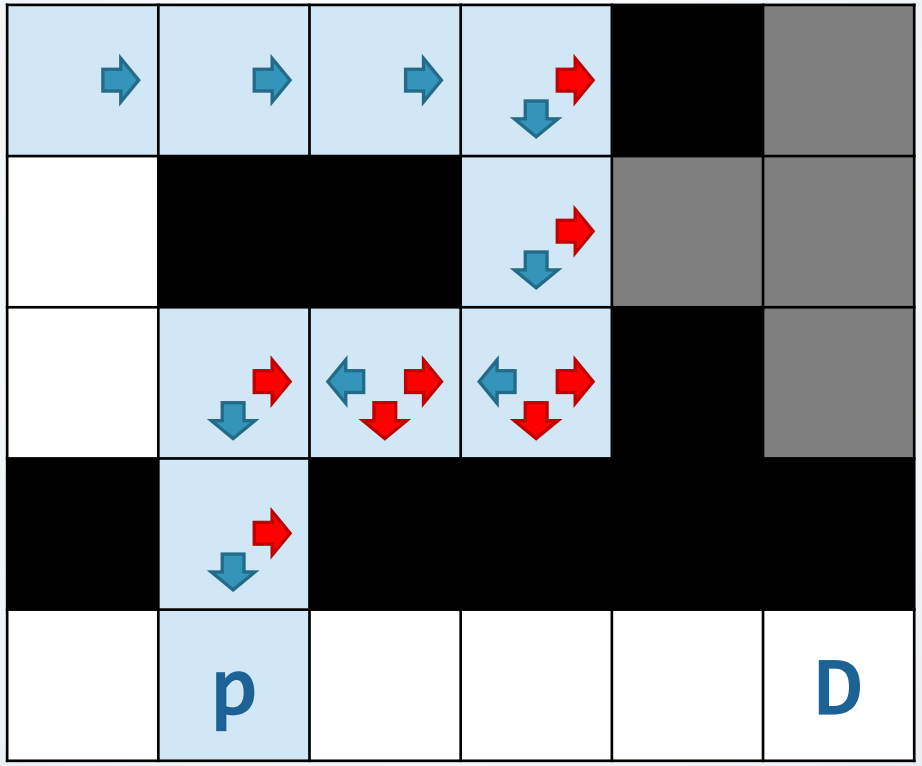


Backtracking
Resolução de Labirinto



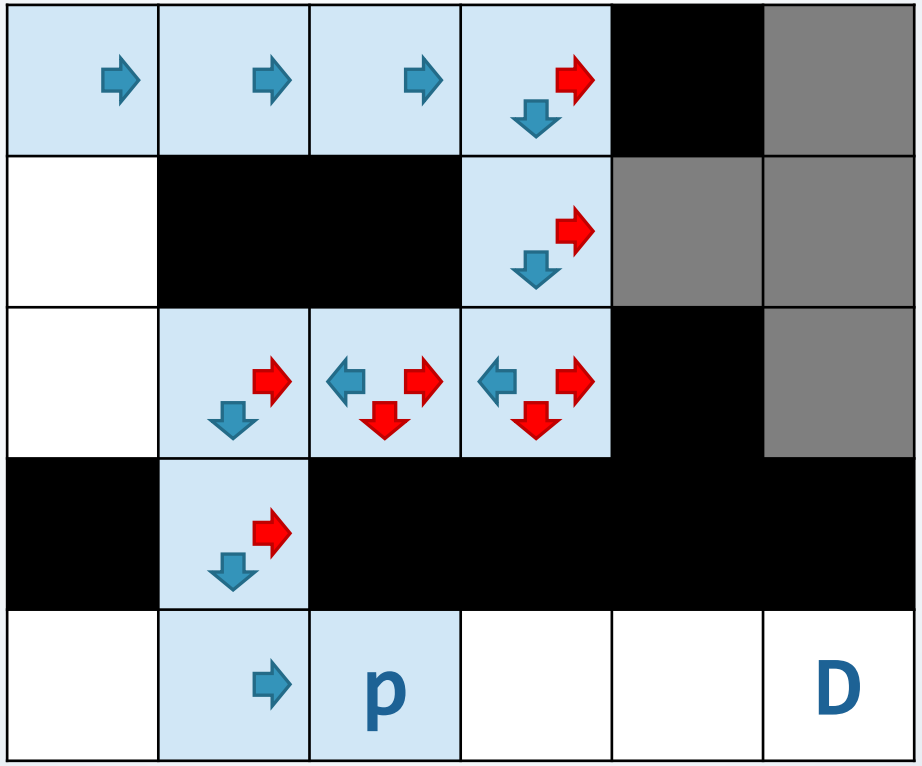
Backtracking

Resolução de Labirinto

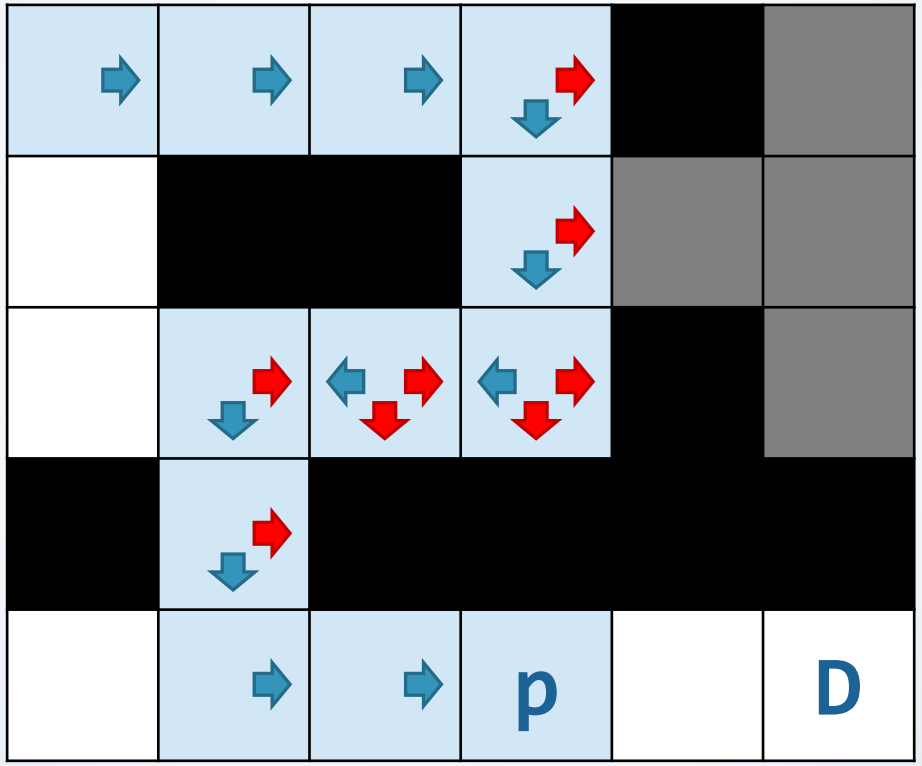


Backtracking

Resolução de Labirinto

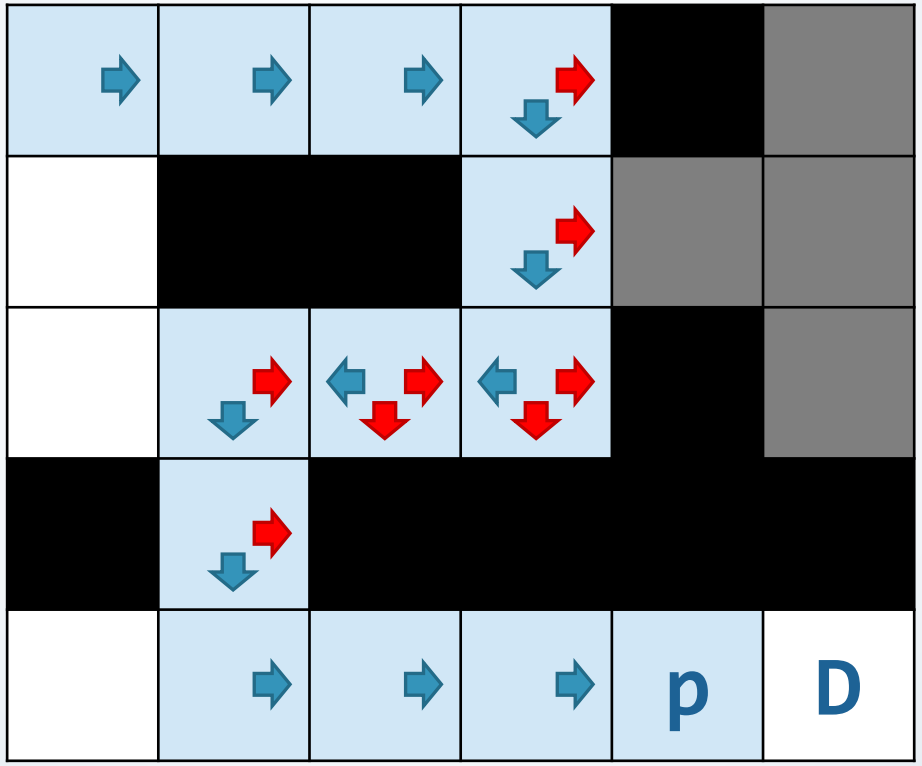


Backtracking
Resolução de Labirinto

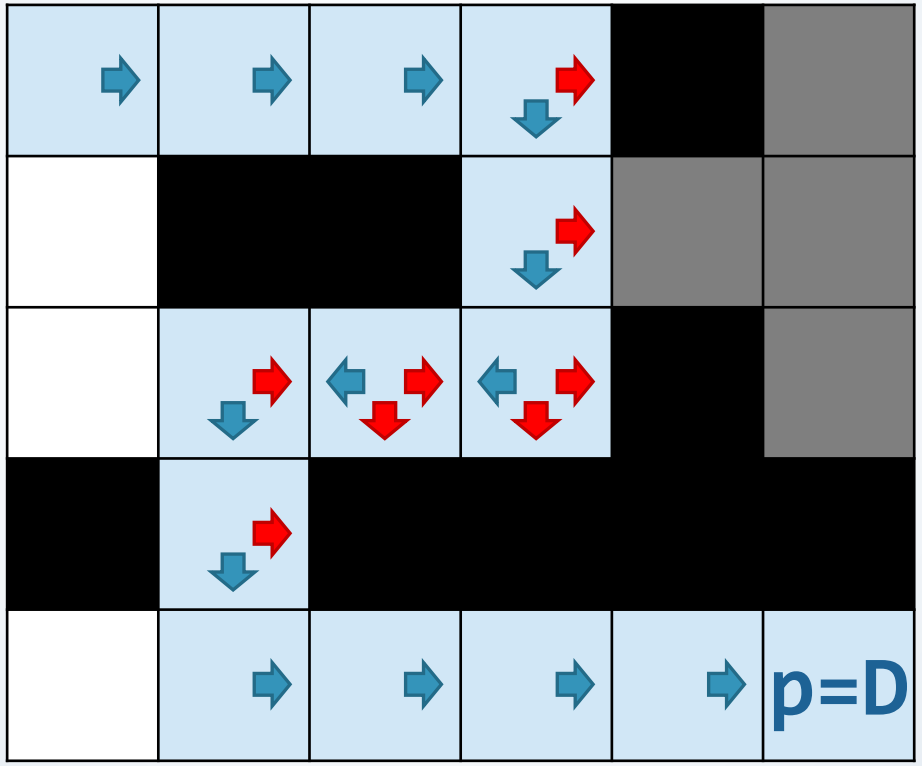


Backtracking

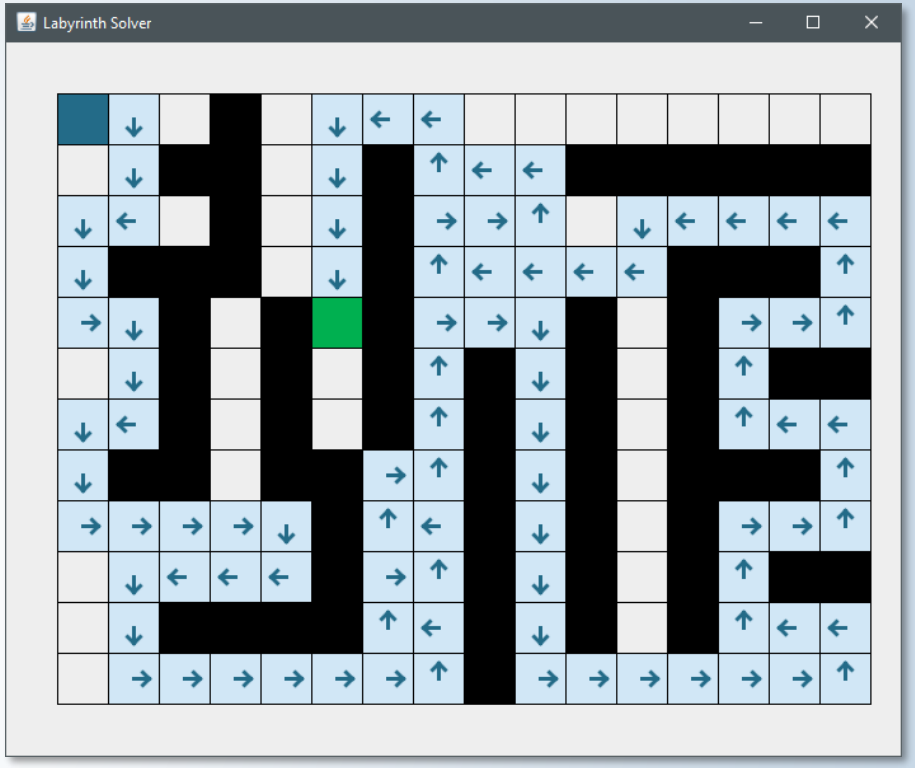
Resolução de Labirinto



Backtracking
Resolução de Labirinto



Backtracking
Resolução de Labirinto



- Note que a solução encontrada nem sempre é a melhor;
- É importante destacar que o caminho que as soluções parciais percorrem nem sempre corresponde a um caminho no problema sendo resolvido, mas é um caminho lógico na árvore de decisão sendo percorrida;
- O número de soluções cresce exponencialmente em relação ao número de posições do labirinto na ordem de $O(3^{linhas \times colunas})$.

Backtracking
Resolução de Labirinto

- Implementação e Testes:
  - LabyrinthSolver.java
  - LabyrinthSolverTest.java
  - LabyrinthSolverTestGUI.java

Backtracking
Quadrado Latino

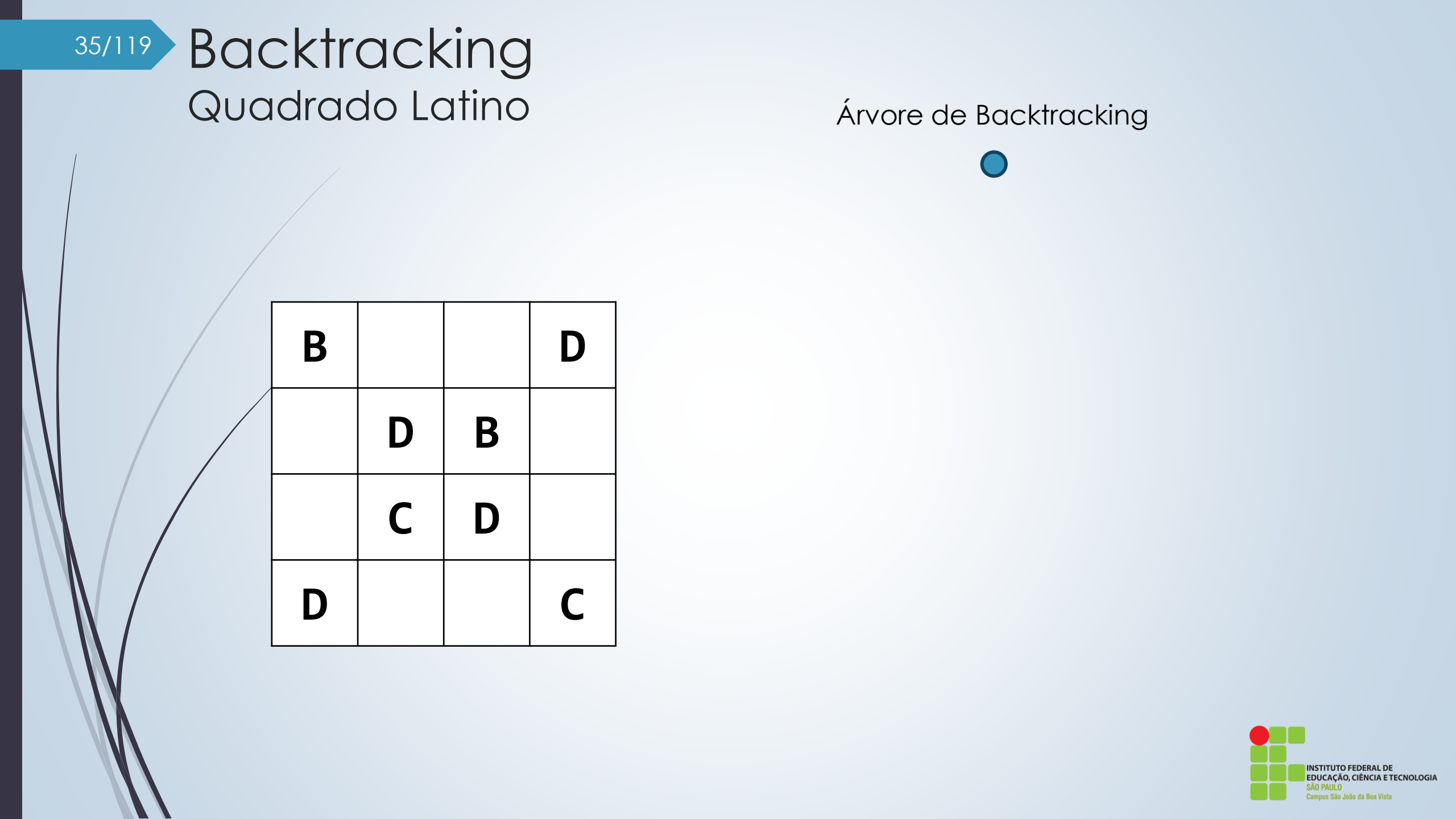
- **Problema:** Dada uma matriz previamente preenchida com as letras A, B, C e D, deve-se completar seu preenchimento, de tal forma que não possa ter na mesma linha e na mesma coluna as letras previamente já inseridas, ou seja, cada linha e cada coluna deve ter apenas uma ocorrência de cada letra.

B			D
	D	B	
	C	D	
D			C

Backtracking
Quadrado Latino

B			D
	D	B	
	C	D	
D			C

Árvore de Backtracking



Backtracking
Quadrado Latino

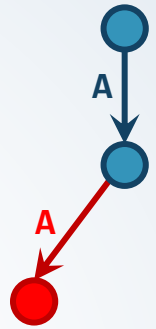
Árvore de Backtracking

B	A		D
	D	B	
	C	D	
D			C

Backtracking
Quadrado Latino

B	A	A	D
	D	B	
	C	D	
D			C

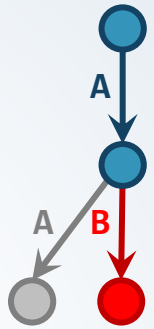
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	B	D
	D	B	
	C	D	
D			C

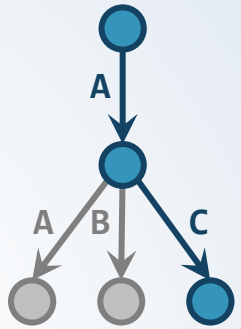
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
	D	B	
	C	D	
D			C

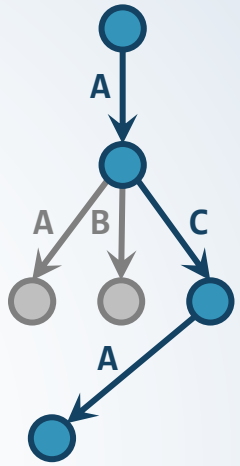
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
A	D	B	
	C	D	
D			C

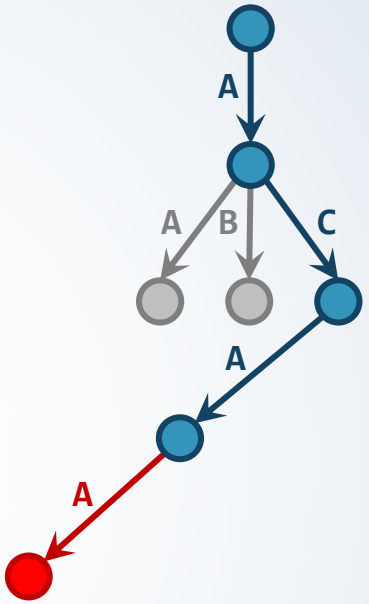
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
A	D	B	A
	C	D	
D			C

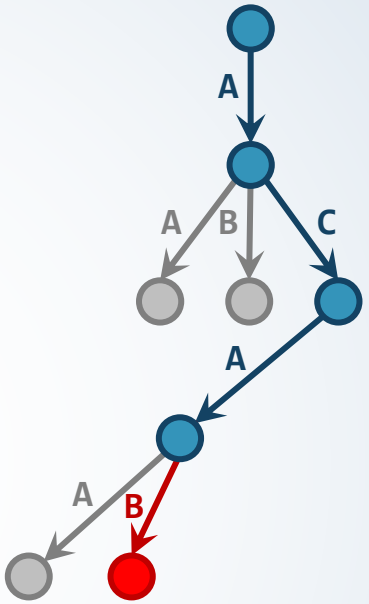
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
A	D	B	B
	C	D	
D			C

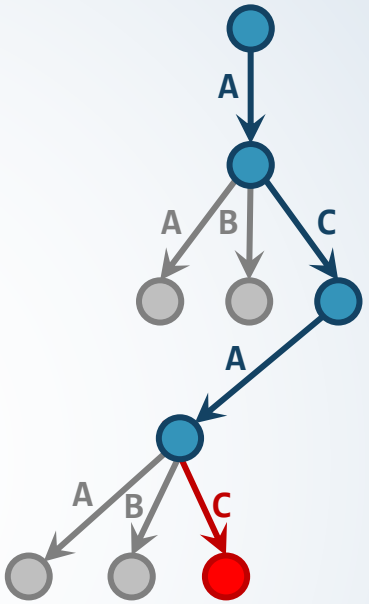
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
A	D	B	C
	C	D	
D			C

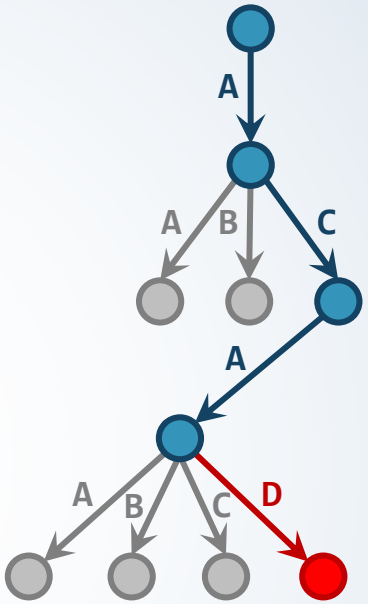
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
A	D	B	D
	C	D	
D			C

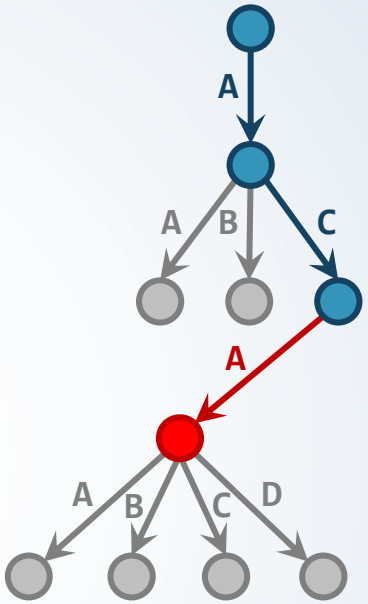
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
A	D	B	
	C	D	
D			C

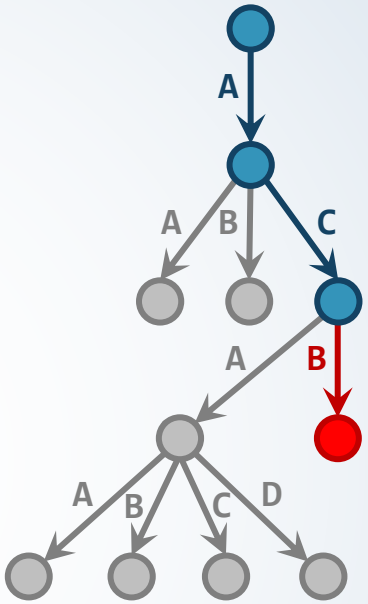
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
B	D	B	
	C	D	
D			C

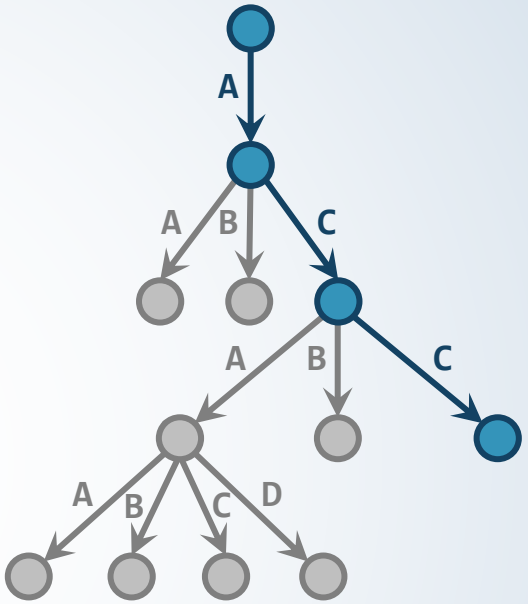
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
C	D	B	
	C	D	
D			C

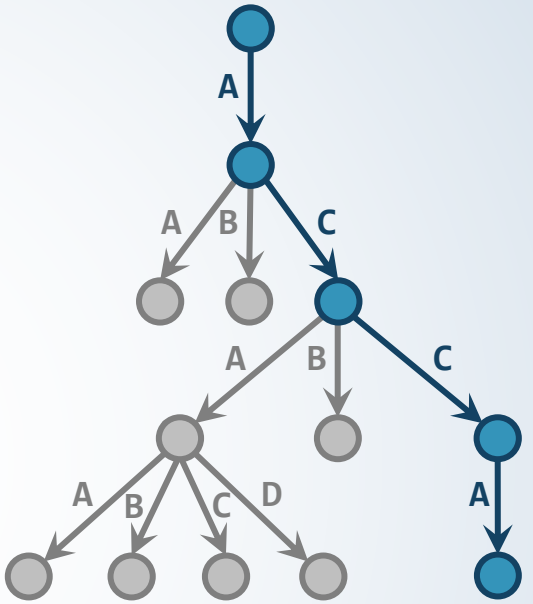
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
C	D	B	A
	C	D	
D			C

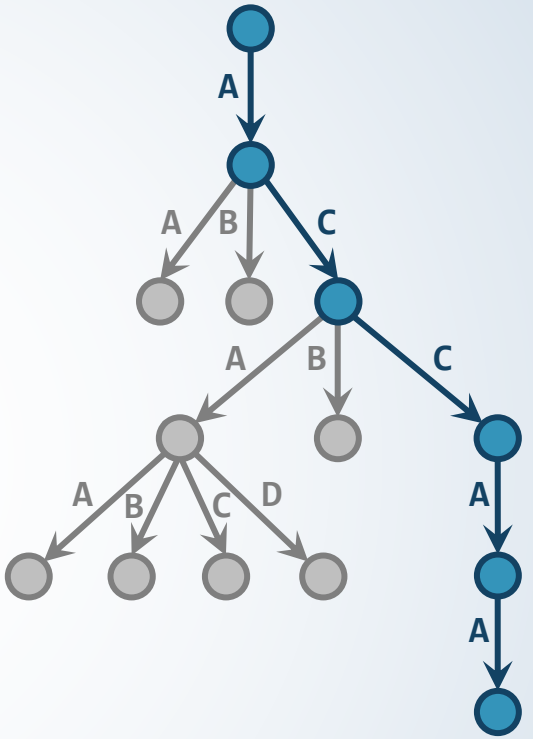
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
C	D	B	A
A	C	D	
D			C

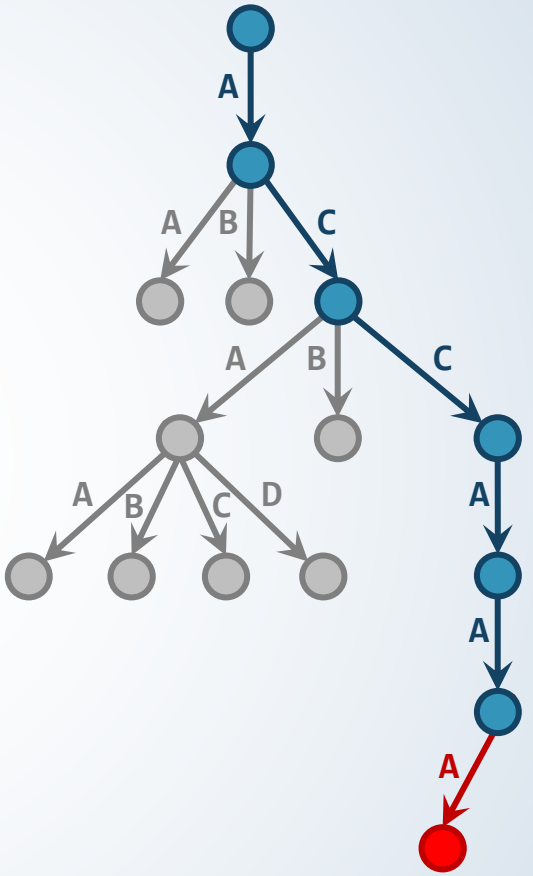
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
C	D	B	A
A	C	D	A
D			C

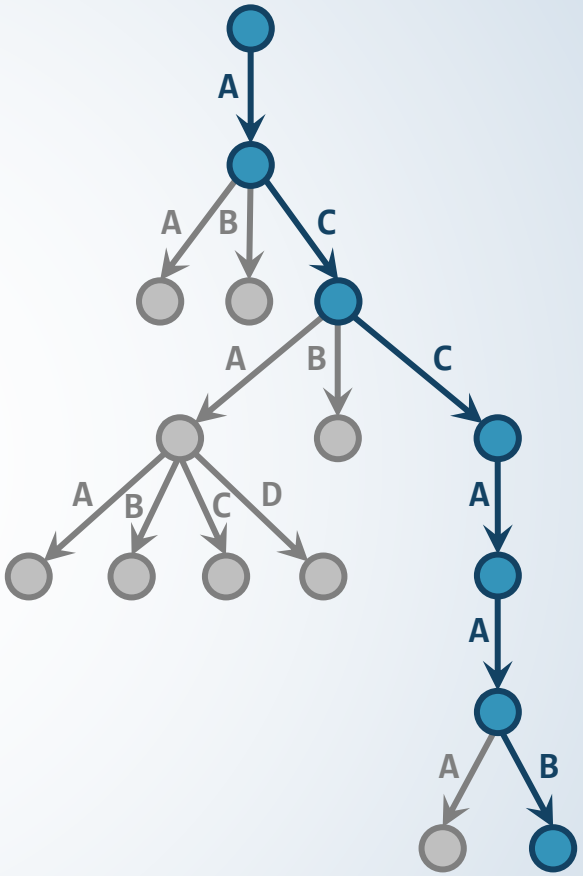
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
C	D	B	A
A	C	D	B
D			C

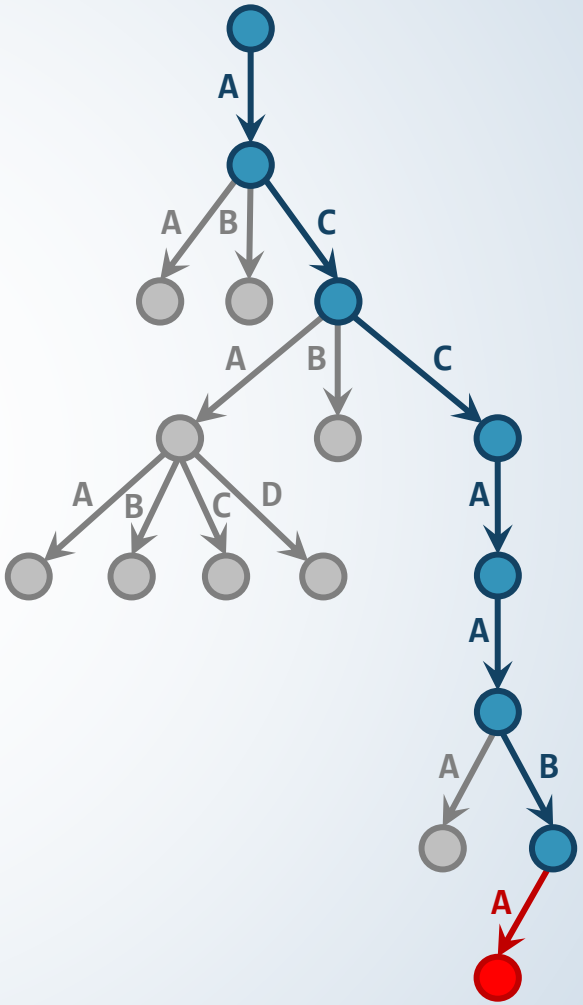
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
C	D	B	A
A	C	D	B
D	A		C

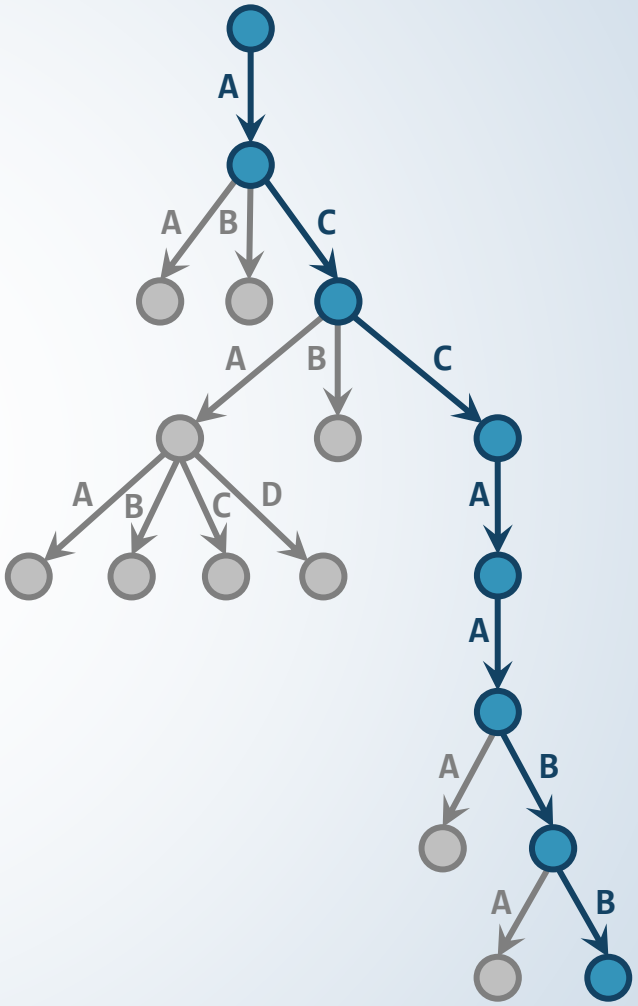
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
C	D	B	A
A	C	D	B
D	B		C

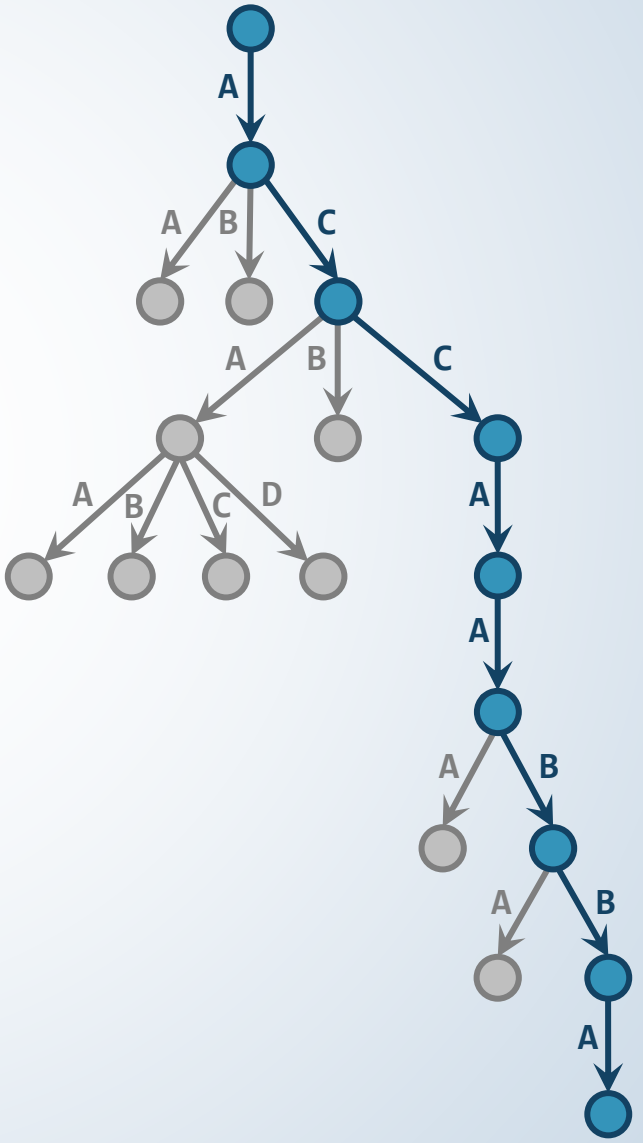
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
C	D	B	A
A	C	D	B
D	B	A	C

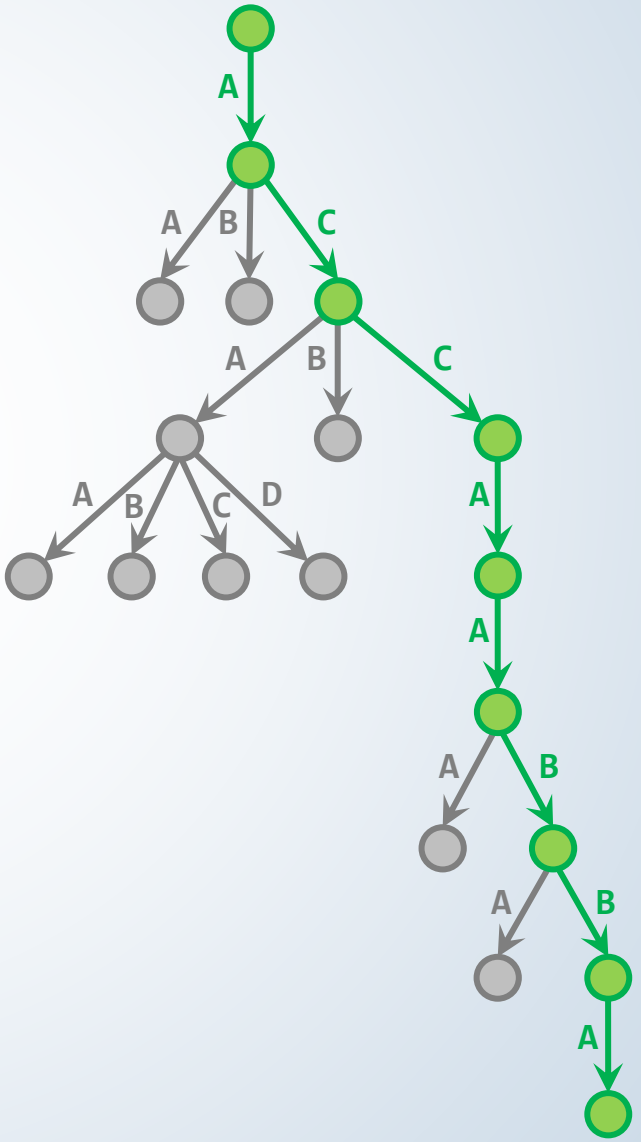
Árvore de Backtracking



Backtracking
Quadrado Latino

B	A	C	D
C	D	B	A
A	C	D	B
D	B	A	C

Árvore de Backtracking



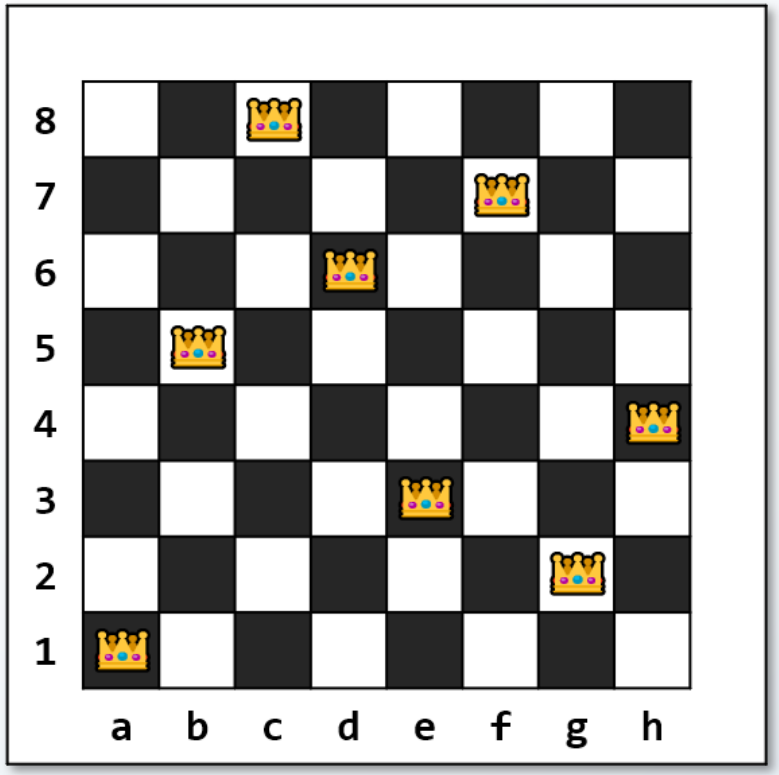
Backtracking
Quadrado Latino

- Implementação e Testes:
  - LatinSquareProblem.java
  - LatinSquareProblemTest.java

Backtracking

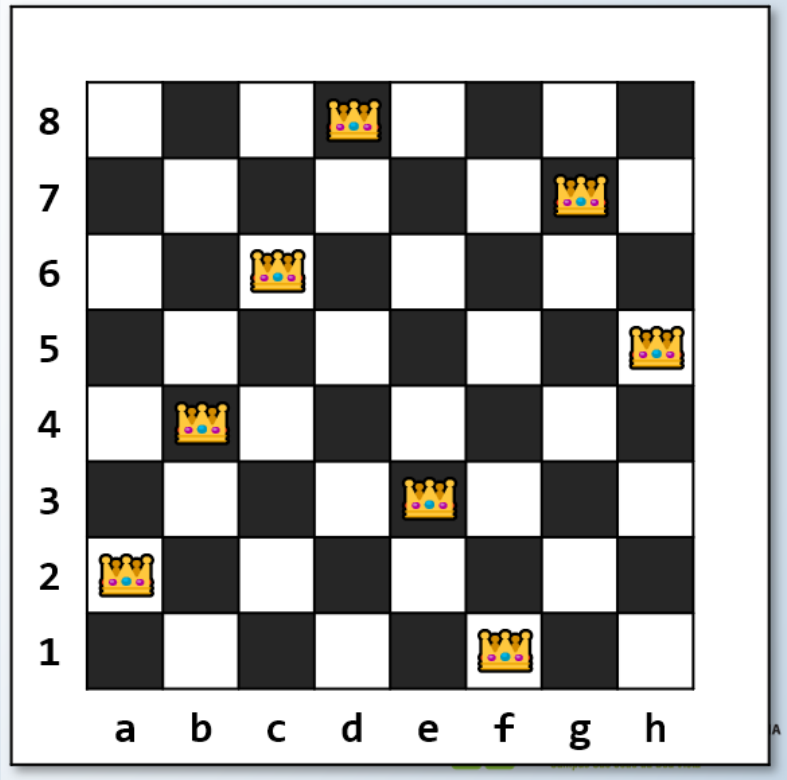
O Problema das Oito Rainhas

- **Problema:** Em um tabuleiro 8 × 8 é possível posicionar oito rainhas tal que elas não se ataquem umas às outras? Dada a posição inicial de uma rainha, devemos identificar todas as soluções possíveis.



Backtracking
O Problema das Oito Rainhas

- Uma solução ingênua para o problema tentaria todas as $8^8 \approx 17 \times 10^6$ de possibilidades, colocando cada rainha em uma casa e filtrando as soluções inviáveis;
- Sabe-se que duas rainhas não podem estar na mesma coluna, portanto pode-se reduzir o problema ao permutar a posição das rainhas nas linhas:
  - Por exemplo, `[2, 4, 6, 8, 3, 1, 7, 5]` representaria a solução da Figura, em que `linha[1] = 2` indica que a rainha da coluna 1 está posicionada na linha 2;
- Modelando o problema desta forma, diminuímos o espaço de busca de 8^8 para $8! = 40320$, tornando o problema 416 vezes menor!



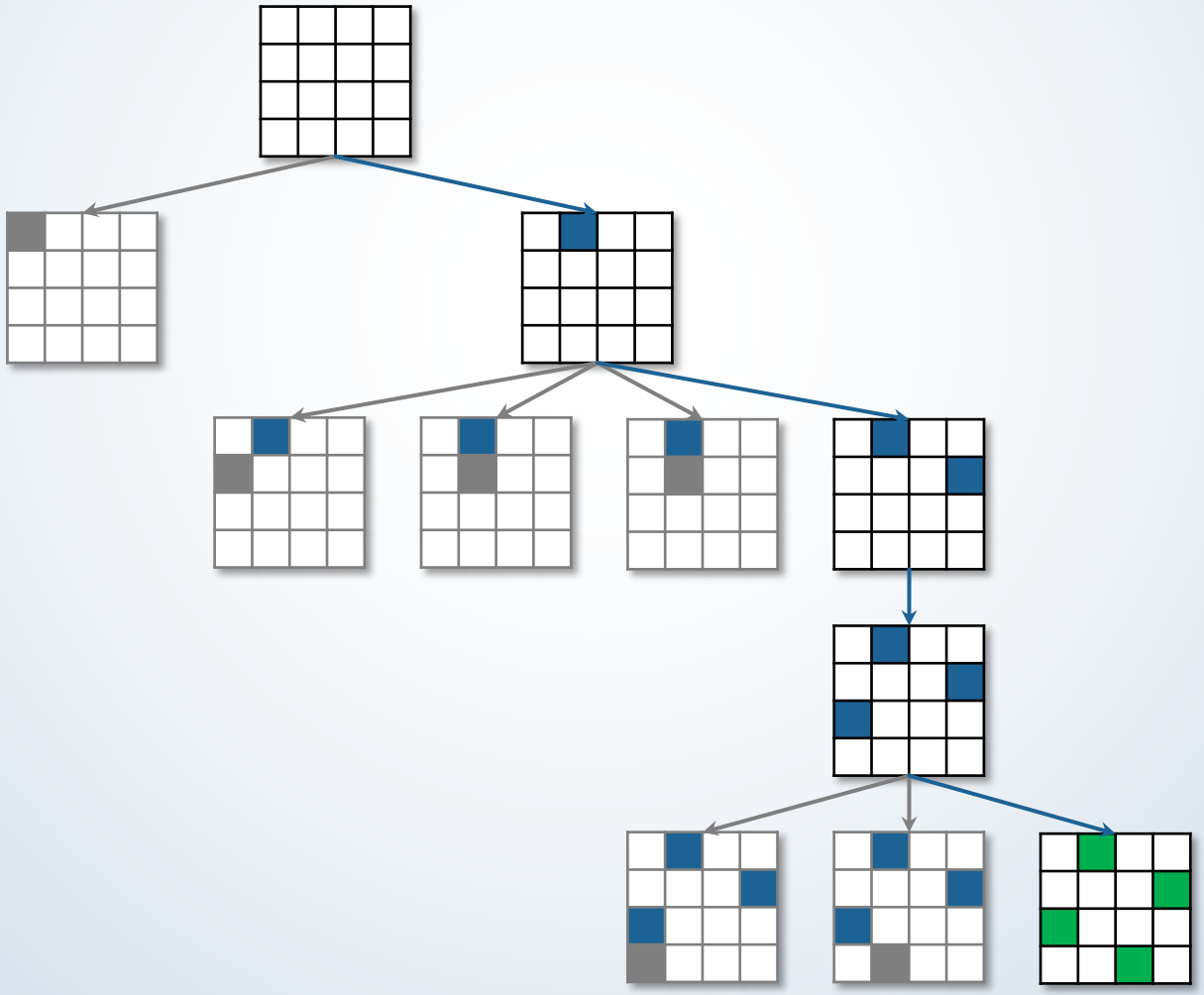
Backtracking
O Problema das Oito Rainhas

- Sabe-se também que duas rainhas não podem ocupar a mesma diagonal;
- Supondo que a rainha `A` está posicionada em `(i, j)` e que a rainha `B` está posicionada em `(k, l)`, elas se atacarão se $|i - k| = |j - l|$
- Uma solução por backtracking posicionaria as rainhas uma por uma, da coluna 1 até a coluna 8 respeitando as restrições do problema:
  - Caso o posicionamento de uma rainha viole alguma das restrições tem-se um erro e o último posicionamento/tentativa é desfeito, sendo realizado em outra casa, ou seja, uma nova tentativa;
  - No caso de não haver uma nova tentativa, o posicionamento da rainha anterior também é desfeito e assim sucessivamente, até haver a possibilidade de uma nova tentativa;
- Por fim é necessário verificar a condição de que uma das rainhas está na posição original descrita pelo problema, o que depende do enunciado.

Backtracking

O Problema das Oito Rainhas

- Ilustração parcial do bracktracking para a versão com 4 rainhas:



Backtracking
O Problema das Oito Rainhas

- Algumas melhorias podem ser aplicadas ao backtracking, de modo a não realizar uma busca completa:
  - Poda do espaço de soluções;
  - Utilização de simetrias;
  - Pré-computação.

Backtracking
O Problema das Oito Rainhas

- Ao gerar soluções, pode-se deparar com uma solução parcial que nunca se tornará uma solução completa viável;
- Pode-se podar esta parte do espaço de soluções e se concentrar em explorar outras partes;
- No problema das oito rainhas:
  - Se posicionarmos uma rainha em `linha[1] = 2` e depois posicionarmos outra rainha em `linha[2] = 1` ou `linha[2] = 3`, teremos um conflito em diagonal;
  - Continuar a buscar a partir de uma destas situações jamais nos levará a uma solução viável;
  - Desta forma, podamos estas ramificações e nos concentramos apenas nas posições válidas `[4, 5, 6, 7, 8]`, economizando tempo de execução.

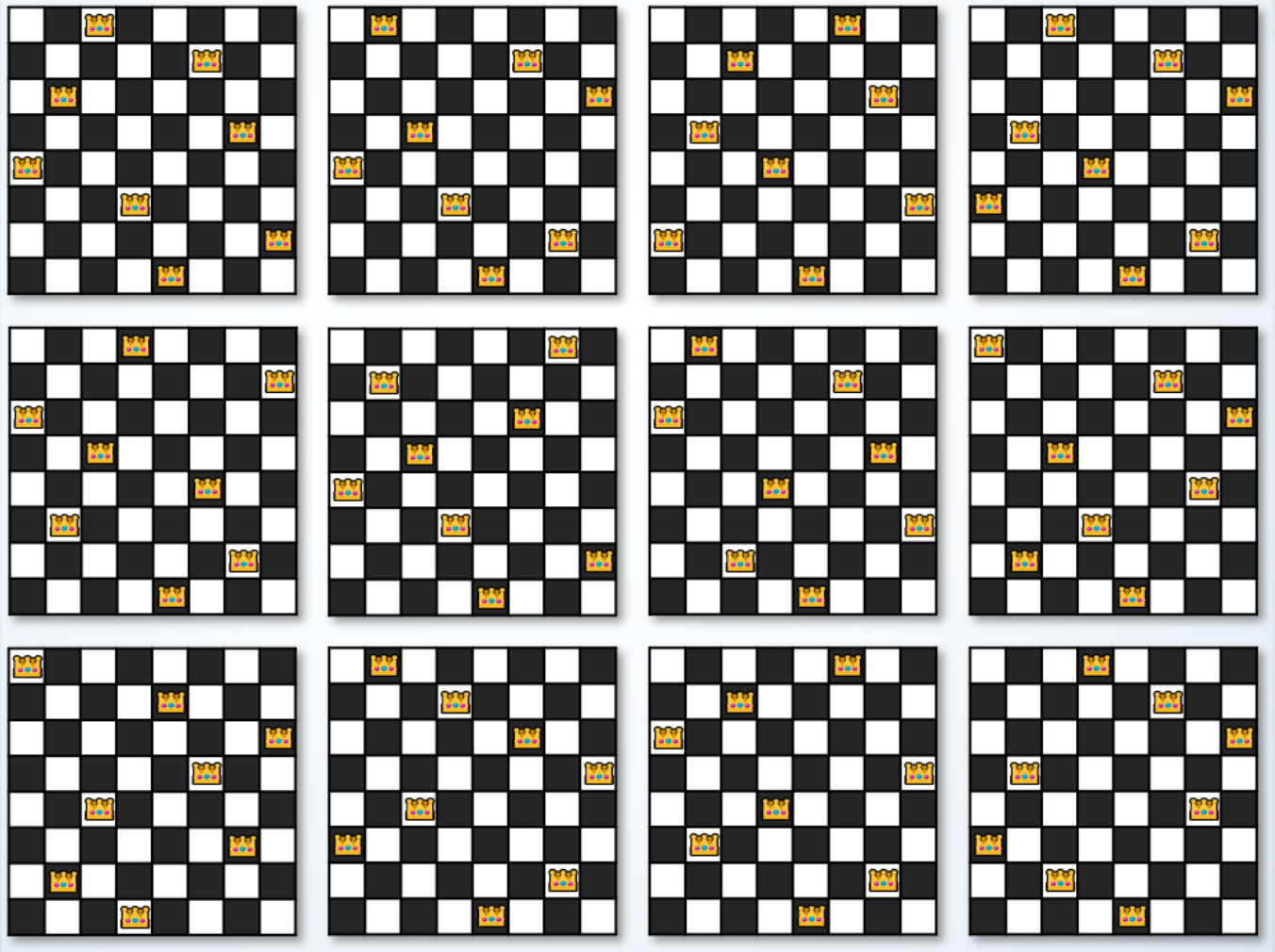
Backtracking
O Problema das Oito Rainhas

- Alguns problemas apresentam simetrias em sua estrutura e pode-se utilizá-las a favor;
- Desta forma, pode-se utilizar uma versão reduzida do espaço de soluções, reduzindo o tempo de execução do código;
- No problema das oito rainhas:
  - Sabe-se que existem somente 92 soluções, porém, apenas 12 delas são únicas, pois as outras são rotações e reflexões;
  - Pode-se gerar as 12 soluções únicas e, se necessário, todas as 92 por meio das rotações e reflexões das 12 originais.

Backtracking

O Problema das Oito Rainhas

- As 12 soluções originais do problema das oito rainhas:



Backtracking
O Problema das Oito Rainhas

- Pode ser útil gerar e preencher tabelas ou outras estruturas de dados que permitam uma busca rápida por valores, antes mesmo da execução do algoritmo em si;
- A pré-computação implica em uma relação de aumento de consumo de memória e diminuição do tempo de execução;
- No problema das oito rainhas:
  - Sabendo que existem 92 soluções, pode-se criar um array de dimensões 92×8 e então preenchê-lo com todas as 92 permutações válidas das 8 rainhas;
  - O algoritmo gerador levará algum tempo para preencher o array, mas será executado uma única vez. Outro algoritmo pode simplesmente buscar os valores na tabela posteriormente;
  - Desta forma o backtracking, ao invés de computar as soluções em tempo real, seria utilizado como um gerador;
  - A mesma lógica de pré-computação pode ser aplicada a problemas bem caracterizados e com a entrada limitada de alguma forma, como a geração de números da sequência Fibonnaci etc.

Backtracking
O Problema das Oito Rainhas

- Implementação e Testes:
  - EightQueensProblem.java
  - EightQueensProblemTest.java

Backtracking
Considerações Finais

- O backtracking é um paradigma utilizado quando é necessário obter soluções ótimas, enumerar soluções, ou quando não se sabe qual caminho seguir para buscar uma solução;
- É necessário que o problema possibilite a construção gradual de uma solução e que se organize como uma árvore;
- Deve ser levado em consideração o fator de ramificação do problema, o que pode levar a um crescimento rápido do espaço utilizado;
- Melhorias ao backtracking, modelagem correta e truques de programação auxiliam a reduzir o espaço de soluções, salvando tempo de execução.

Programação Dinâmica

Dynamic programming is a fancy name for [recursion] with a table. Instead of solving subproblems recursively, solve them sequentially and store their solutions in a table. The trick is to solve them in the right order so that whenever the solution to a subproblem is needed, it is already available in the table. Dynamic programming is particularly useful on problems for which divide-and-conquer appears to yield an exponential number of subproblems, but there are really only a small number of subproblems repeated exponentially often. In this case, it makes sense to compute each solution the first time and store it away in a table for later use, instead of recomputing it recursively every time it is needed" -

Ian Parberry, *Problems on Algorithms*

- A palavra programação na expressão "programação dinâmica" não tem relação direta com programação de computadores;
- Ela significa planejamento e refere-se à construção da tabela que armazena as soluções dos subproblemas.

Programação Dinâmica

- A programação dinâmica (ou PD/DP) é talvez o paradigma de solução de problemas mais desafiantes dentre os outros vistos;
- Este paradigma pode "parecer mágica", até que tenhamos visto exemplos o suficiente, tornando-o relativamente fácil de ser aplicado;
- É necessário nos assegurarmos de termos dominado todos os outros paradigmas anteriores antes de continuar.

Programação Dinâmica

- Se a soma do tamanho dos subproblemas é $O(n)$, temos uma boa probabilidade de que o algoritmo recursivo tenha complexidade polinomial;
- Entretanto, se temos n subproblemas de tamanho n−1 cada, a tendência é de que o algoritmo recursivo associado tenha complexidade exponencial;
- A Programação Dinâmica nos fornece uma maneira de projetar algoritmos que:
  - Sistematicamente exploram todas as possibilidades (corretude);
  - Evitam computação redundante (eficiência).

Programação Dinâmica

- Tais algoritmos são quase sempre definidos por recursividade:
  - Definem a solução para um problema em termos da solução de problemas menores;
  - De certa forma, lembram divisão e conquista e algoritmos gulosos;
- Um possível defeito na busca recursiva é a computação redundante de subproblemas, ou a exploração redundante do espaço de busca:
  - Para contornar esta situação, pode-se armazenar os resultados dos subproblemas já resolvidos;
  - Em parte, por isto o backtracking é ineficiente.

Programação Dinâmica

- À medida em que resolvemos subproblemas, armazenamos os resultados parciais, acelerando o algoritmo:
  - Para cada subproblema inédito, o resolvemos e armazenamos o resultado;
  - Para os subproblemas repetidos, apenas consultamos o resultado;
- É importante primeiro nos certificarmos de que o algoritmo recursivo é correto, depois o aceleramos;
- As soluções dos subproblemas são mantidas em uma tabela, a qual é consultada a cada subproblema encontrado;
- Cada entrada na tabela é chamada de estado ou estágio.

Programação Dinâmica

- É aplicável a problemas que possuam as seguintes propriedades:
  - **Formulação Recursiva Bem Definida:** no caso de formulação recursiva, esta não deve possuir ciclos;
  - **Subestrutura Ótima:** O problema pode ser dividido sucessivamente e a combinação das soluções ótimas dos subproblemas corresponde à solução ótima do problema original;
  - **Superposição de Subproblemas:** O espaço de subproblemas é pequeno e eles se repetem com frequência durante a solução do problema original.

Programação Dinâmica

- Pode ser empregada de duas formas:
  - **Top-Down:**
    - O problema original é decomposto em subproblemas que são resolvidos recursivamente e combinados para obtenção da solução;
    - As entradas da tabela são preenchidas de acordo com a necessidade, ao longo da recursão. Pode não preencher toda a tabela;
  - **Bottom-Up:**
    - Não se utiliza recursão. Os subproblemas são resolvidos e combinados sucessivamente de maneira a construir a solução do problema original;
    - As entradas da tabela são preenchidas "em ordem" e a tabela é totalmente preenchida.

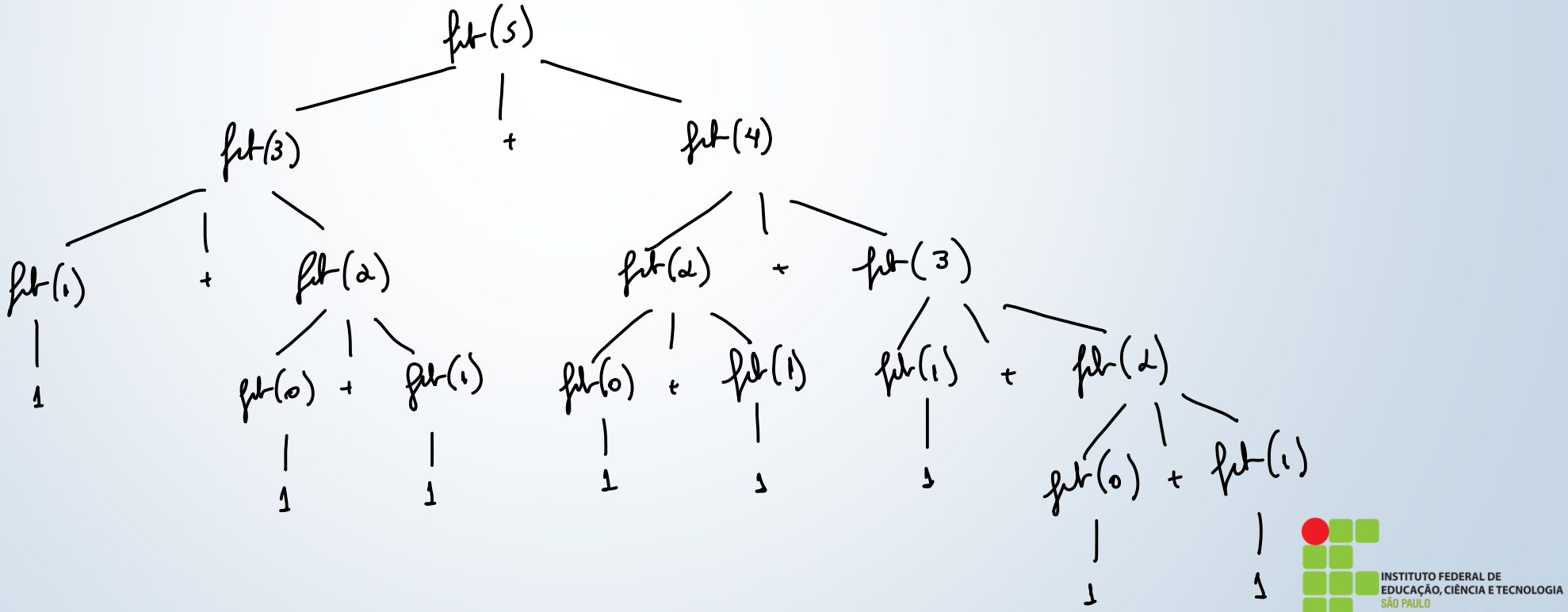
Programação Dinâmica
O Problema de Fibonacci

- **Problema:** Gerar um termo arbitrário da série de Fibonacci. Sabe-se que a série de Fibonacci é dada pela seguinte equação, onde n representa o termo. Note que o primeiro termo é o 0, o segundo é o 1 e assim sucessivamente.

$$fib(n) = \begin{cases} 1, & n \leq 1 \\ fib(n-2) + fib(n-1), & n > 1 \end{cases} \mid n \in \mathbb{N}$$

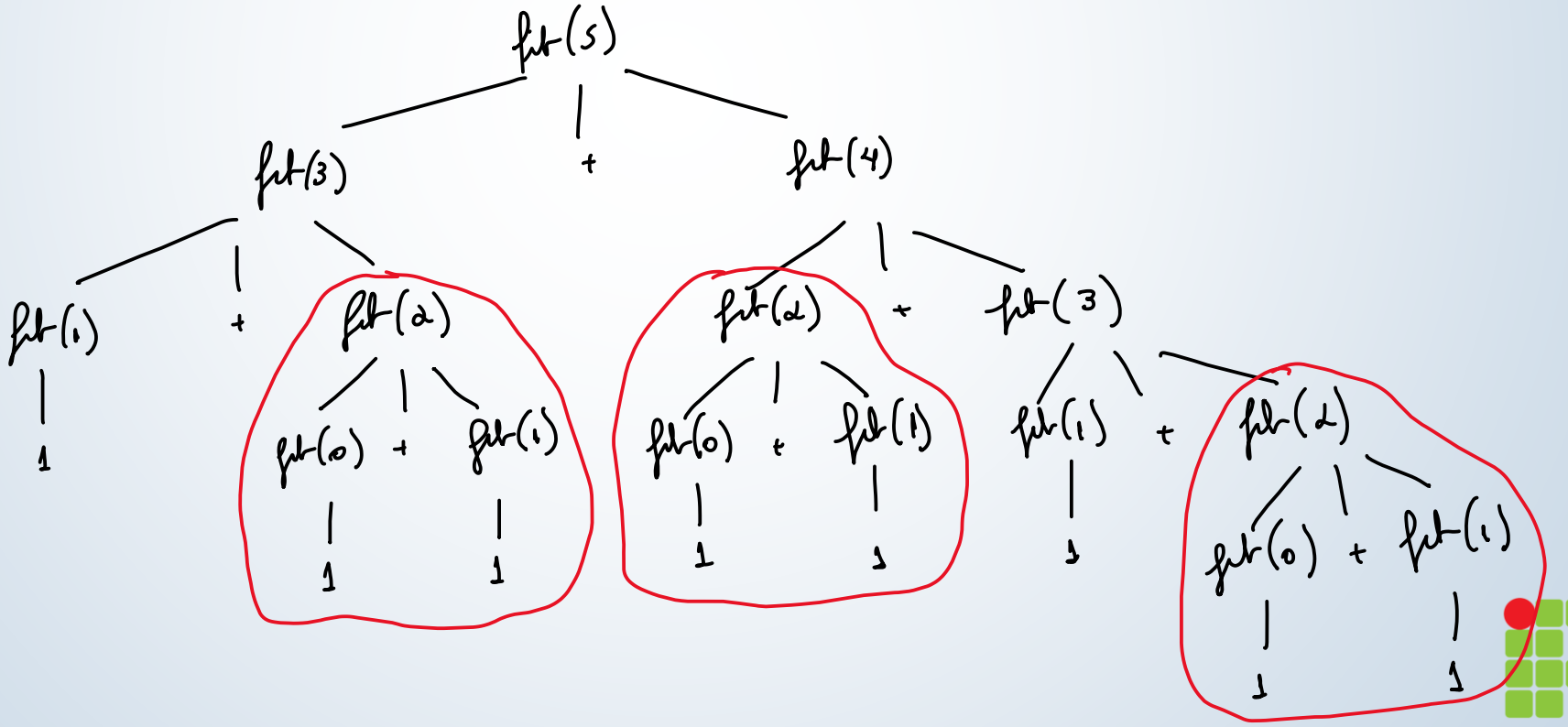
Programação Dinâmica
O Problema de Fibonacci

- Por exemplo, se aplicarmos o algoritmo recursivo para o cálculo de termos da série de Fibonacci, notaremos o recálculo de vários termos. Abaixo é mostrado o cálculo do sexto termo. Note a repetição de vários cálculos.



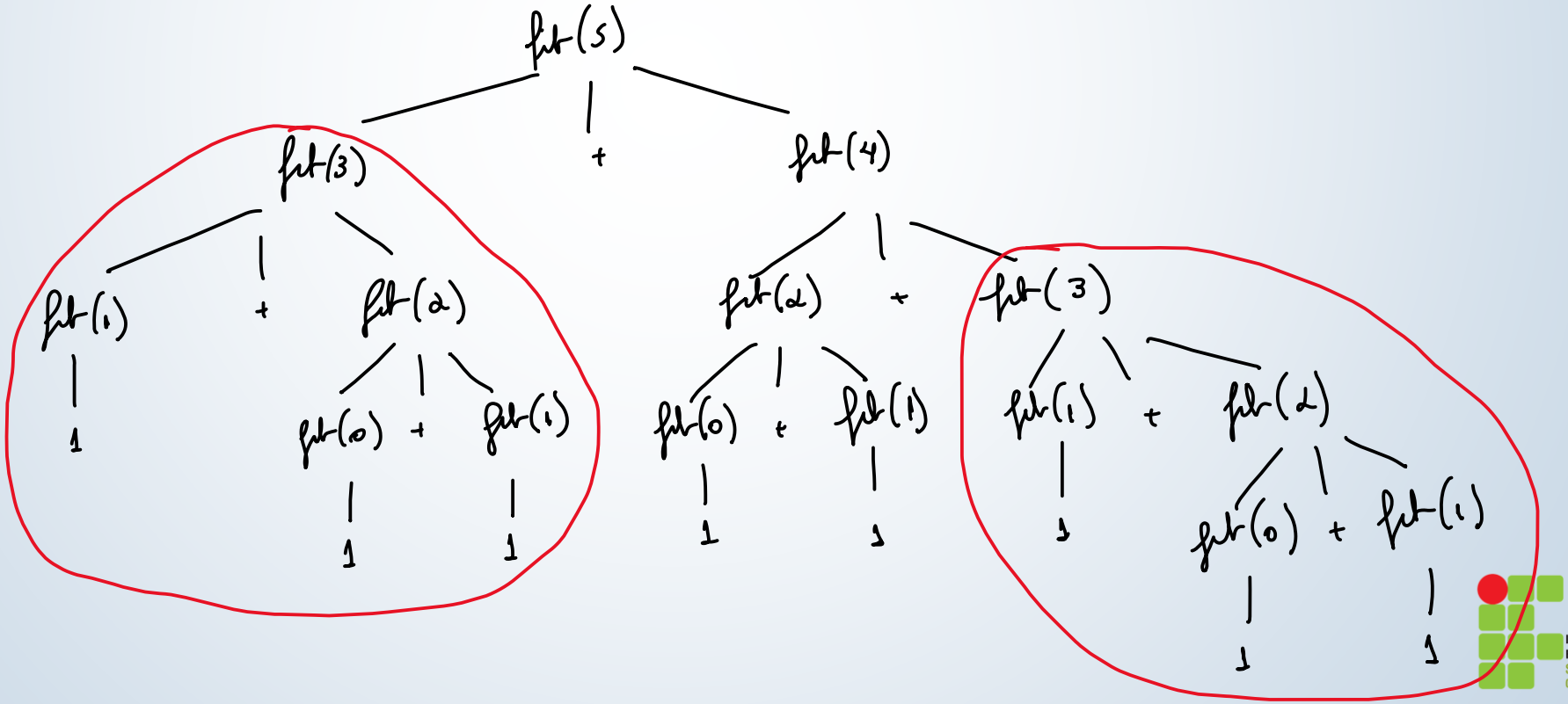
Programação Dinâmica
O Problema de Fibonacci

- Por exemplo, se aplicarmos o algoritmo recursivo para o cálculo de termos da série de Fibonacci, notaremos o recálculo de vários termos. Abaixo é mostrado o cálculo do sexto termo. Note a repetição de vários cálculos.



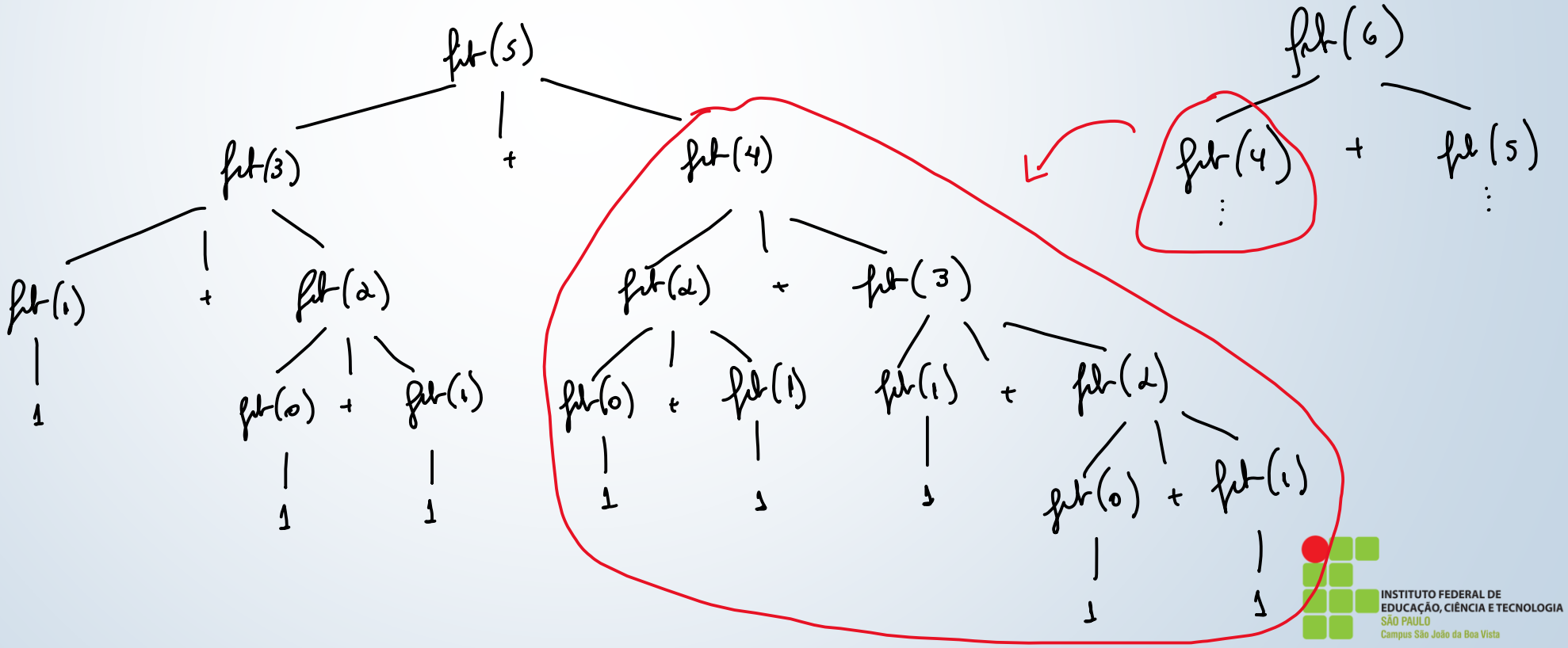
Programação Dinâmica
O Problema de Fibonacci

- Por exemplo, se aplicarmos o algoritmo recursivo para o cálculo de termos da série de Fibonacci, notaremos o recálculo de vários termos. Abaixo é mostrado o cálculo do sexto termo. Note a repetição de vários cálculos.



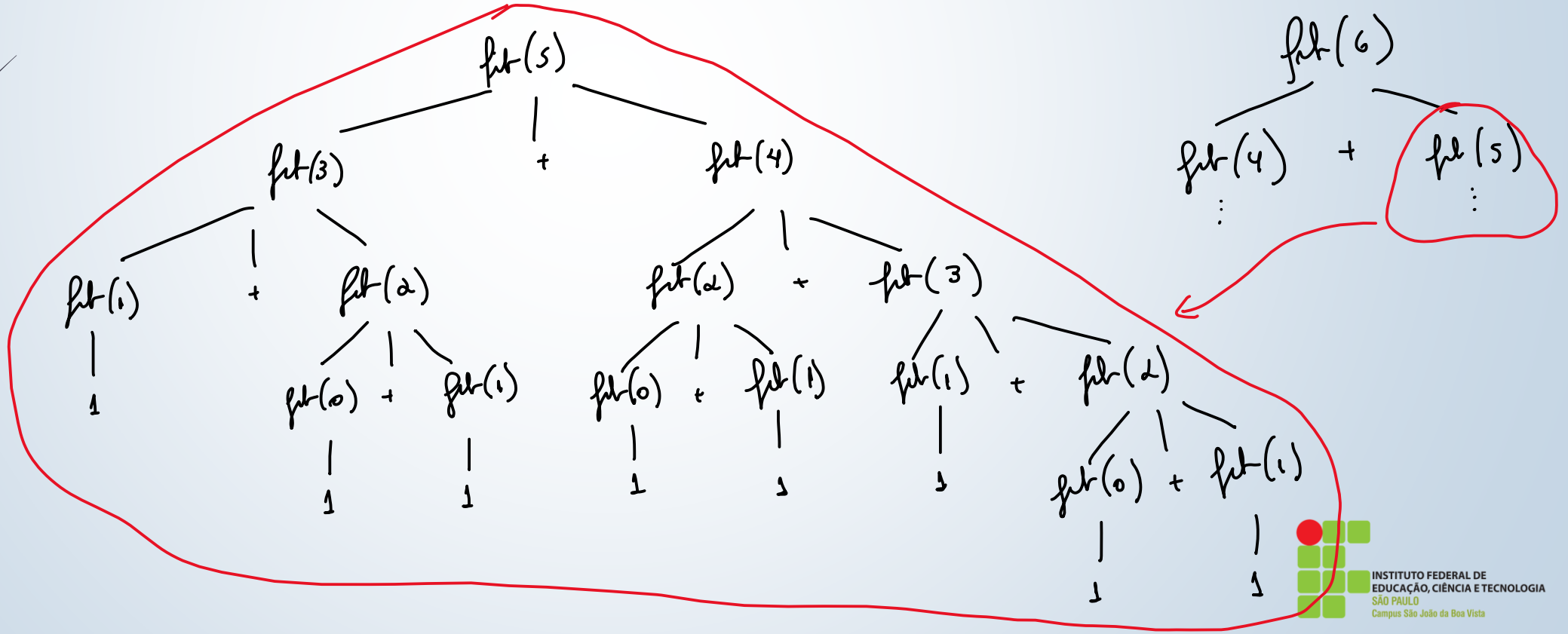
Programação Dinâmica
O Problema de Fibonacci

- Por exemplo, se aplicarmos o algoritmo recursivo para o cálculo de termos da série de Fibonacci, notaremos o recálculo de vários termos. Abaixo é mostrado o cálculo do sexto termo. Note a repetição de vários cálculos.



Programação Dinâmica
O Problema de Fibonacci

- Por exemplo, se aplicarmos o algoritmo recursivo para o cálculo de termos da série de Fibonacci, notaremos o recálculo de vários termos. Abaixo é mostrado o cálculo do sexto termo. Note a repetição de vários cálculos.



Programação Dinâmica
O Problema de Fibonacci

- Note que o consumo de tempo é exponencial, pois o algoritmo resolve subproblemas muitas vezes;
- Como já vimos, podemos resolver usando programação dinâmica de duas formas:
  - **Top-Down:** utiliza memorização, ou seja, registra valores para buscá-los posteriormente se necessário;
  - **Bottom-Up:** normalmente depende de uma noção de "tamanho" do problema e de uma noção de "ordem" dos subproblemas, tal que resolver um subproblema em particular depende de termos resolvidos os subproblemas menores anteriores;
- As duas abordagens normalmente geram algoritmos com tempo de execução assintoticamente iguais, exceto em circunstâncias em que a Top-Down não examina todos os subproblemas recursivamente;
- A PD Bottom-Up possui, normalmente, constantes melhores, devido ao menor overhead por chamadas recursivas de funções.

Programação Dinâmica
O Problema de Fibonacci

- Implementação e Testes:
  - FibonacciProblem.java
  - FibonacciTestProblem.java

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)

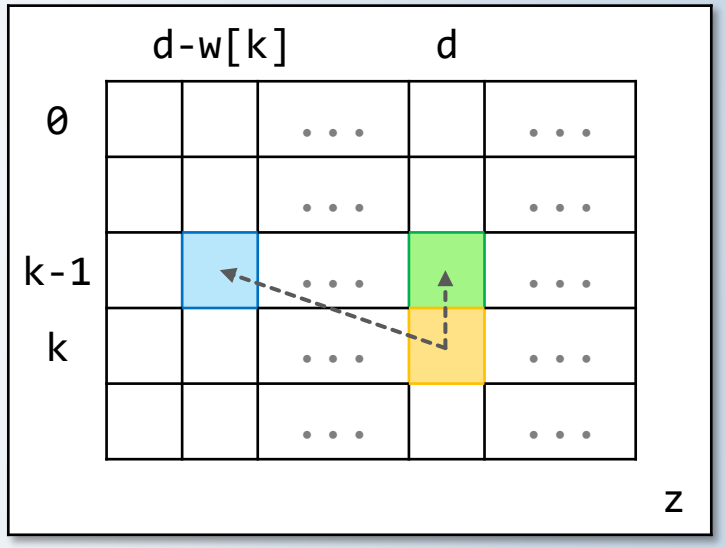
- **Problema:** Dada uma mochila de capacidade c, um inteiro em unidades de peso, e um conjunto de n itens distintos e únicos, enumerados de 0 a $n-1$, cada um com um peso w_i (inteiro) e um valor v_i associado a cada item i, deseja-se determinar quais itens devem ser colocados na mochila de modo a maximizar o valor total transportado, respeitando sua capacidade. Para cada item, deve-se escolher se ele estará incluso na solução ou não.
- **Exemplo:**
  - $c = 10$, $n = 4$, $w = [8,\ 1,\ 5,\ 4]$ e $v = [500,\ 1000,\ 300,\ 210]$
  - Resposta: 1510, selecionando os itens 1, 2 e 3 $(1 + 5 + 4)$ com capacidade total de 10 unidades de peso.

Programação Dinâmica
O Problema da Mochila (*Knapsack Problem*)

- Como pode-se projetar um algoritmo para resolver o problema?
- Existem 2^n possíveis subconjuntos de itens, sendo assim um algoritmo de força bruta é impraticável;
- É um problema de otimização. Será que tem subestrutura ótima?
  - Se o item n estiver na solução ótima, o valor desta solução será v_n mais o valor da melhor solução do problema da mochila com capacidade $c - w_n$ considerando-se só os $n - 1$ primeiros itens;
  - Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade c considerando-se só os $n - 1$ primeiros itens.

Programação Dinâmica
O Problema da Mochila (*Knapsack Problem*)

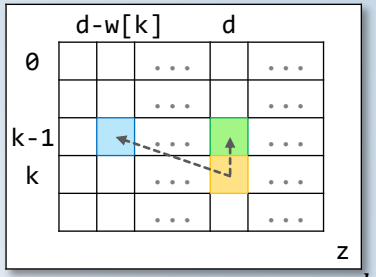
- $z[\,][\,]$é a tabela para armazenar os valores calculados;
- $z[k][d]$:
  - k: quantidade de itens momentânea
  - d: capacidade momentânea
- Solução: $z[k][d]$
- Seja $z[k][d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original;
- A fórmula de recorrência para computar $z[k][d]$ para todo valor de d e k é:



$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d-w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

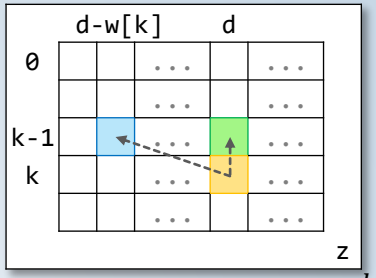
k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							

z

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

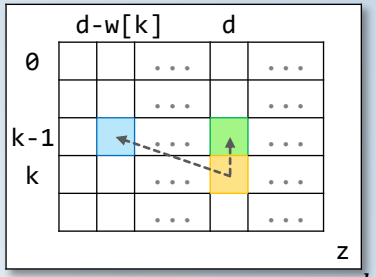
k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0							
3	0							
4	0							

z

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

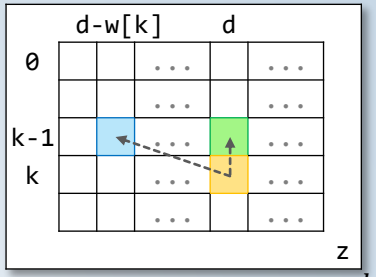
k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0							
3	0							
4	0							

z

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

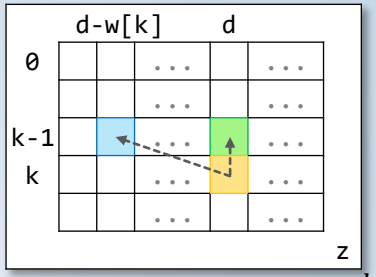
k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0							
4	0							

z

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

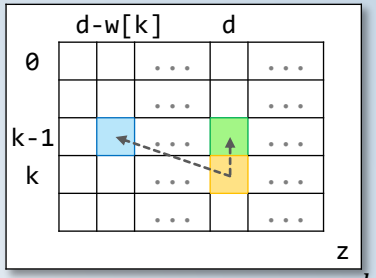
k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0							
4	0							

z

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

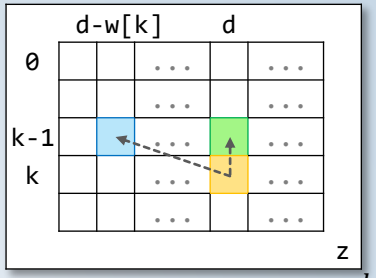
k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0							

z

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

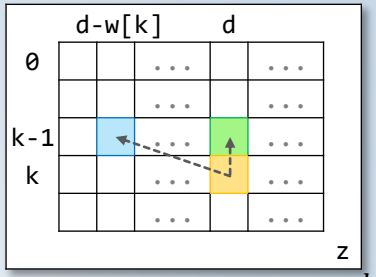
k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0							

z

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

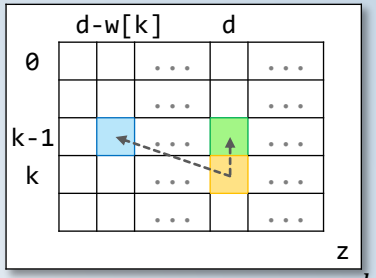
k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

z

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

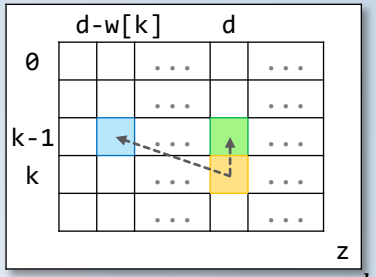
k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

z

Programação Dinâmica

O Problema da Mochila (*Knapsack Problem*)



- **Exemplo:** $c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

$$z[k][d] = \begin{cases} 0, & k = 0 \text{ ou } d = 0 \\ z[k-1][d], & w_{k-1} > d \\ \max(z[k-1][d], z[k-1][d - w_{k-1}] + v_{k-1}), & w_{k-1} \leq d \end{cases}$$

k: quantidade de itens momentânea
d: capacidade momentânea

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

z

Programação Dinâmica
O Problema da Mochila (*Knapsack Problem*)

- **Observações:**
  - A implementação de exemplo não retorna o subconjunto de valor total máximo, apenas o valor máximo;
  - No exemplo, o subconjunto de valor total máximo é apresentado à direita:
    - para chegar em 34, somou-se 24 a 10;
    - para chegar em 10, somou-se 10 a 0
    - 10 corresponde ao item de peso 2;
    - 25 corresponde ao item de peso 5;
    - a soma desses dois itens atinge 7, o máximo permitido na mochila.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

z

$c = 7$, $w = \{2, 1, 6, 5\}$ e $v = \{10, 7, 25, 24\}$

k: quantidade de itens momentânea
d: capacidade momentânea

Programação Dinâmica
O Problema da Mochila (*Knapsack Problem*)

- Implementação e Testes:
  - KnapsackProblem.java
  - KnapsackProblemTest.java

Programação Dinâmica
O Problema da Subcadeia Comum Máxima

- **Problema:** Uma subcadeia de uma sequência é uma seleção de alguns elementos da sequência. A sequência "aba", por exemplo, é uma subcadeia de "aberração". Para comparar duas sequências, frequentemente é interessante determinar a subcadeia comum mais longa entre os duas;
- **Definição:** Dada duas sequências $X = x_1 x_2 \dots x_m$ e $Y = y_1 y_2 \dots y_n$ de um alfabeto, determinar a maior subcadeia comum de X e Y.

Programação Dinâmica
O Problema da Subcadeia Comum Máxima

- **Teorema (subestrutura ótima):** Seja $Z = z_1 z_2 \dots z_k$ a subcadeia comum máxima de X e Y, denotado por $Z = \text{SCM}(X, Y)$
  - Se $x_m = y_n$ então $z_k = x_m = y_n$ e $Z_{k-1} = \text{SCM}(X_{m-1}, Y_{n-1})$
  - Se $x_m \neq y_n$ então $z_k \neq x_m$ implica que $Z = \text{SCM}(X_{m-1}, Y)$
  - Se $x_m \neq y_n$ então $z_k \neq y_n$ implica que $Z = \text{SCM}(X, Y_{n-1})$
- Fórmula de Recorrência:

$$c[i][j] = \begin{cases} 0, & i = 0 \text{ ou } j = 0 \\ c[i-1][j-1] + 1, & i, j > 0 \; e \; x_i = y_j \\ \max(c[i-1][j], c[i][j-1]), & i, j > 0 \; e \; x_i \neq y_j \end{cases}$$

Programação Dinâmica

O Problema da Subcadeia Comum Máxima

- **Exemplo:** X=abcb e Y=bdcab, $m = 4$ e $n = 5$

		Y	b	d	c	a	b
		0	1	2	3	4	5
	0						
a	1						
b	2						
c	3						
b	4						

X

c

$$c[i][j] = \begin{cases} 0, & i = 0 \text{ ou } j = 0 \\ c[i-1][j-1] + 1, & i,j > 0 \; e \; x_i = y_j \\ \max(c[i-1][j], c[i][j-1]), & i,j > 0 \; e \; x_i \neq y_j \end{cases}$$

Programação Dinâmica

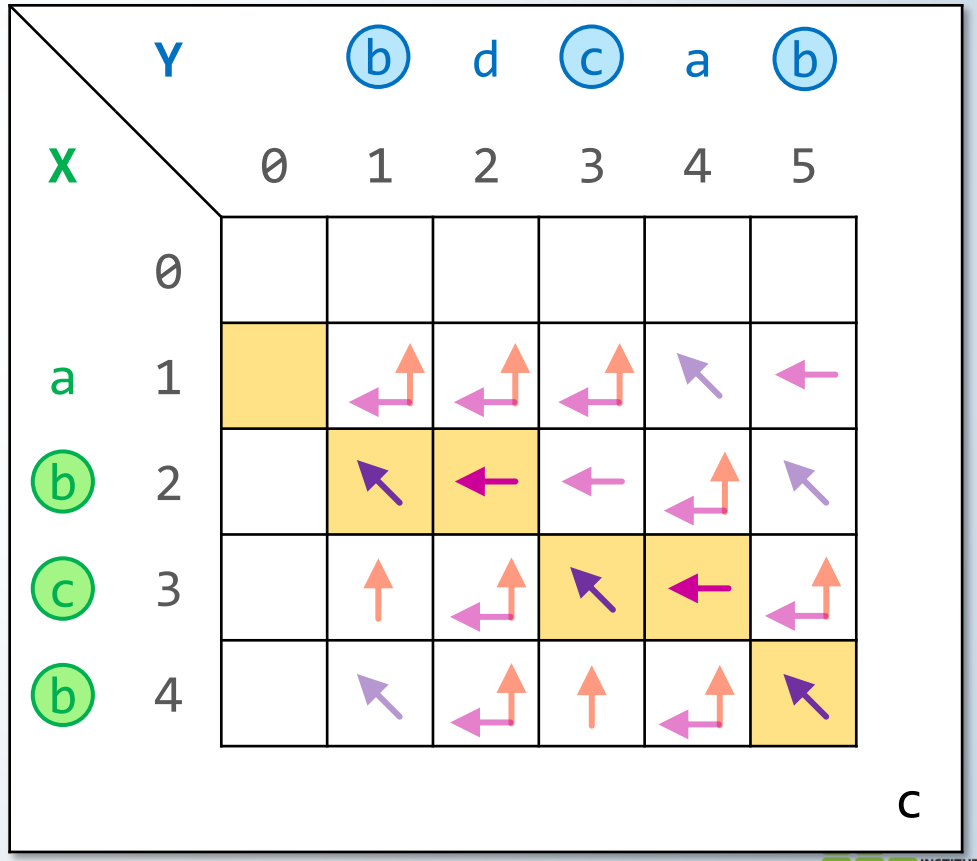
O Problema da Subcadeia Comum Máxima

- **Exemplo:** X=abcb e Y=bdcab, $m = 4$ e $n = 5$

$$c[i,j] = \begin{cases} 0, & i = 0 \text{ ou } j = 0 \\ c[i-1][j-1] + 1, & i,j > 0 \text{ e } x_i = y_j \\ \max(c[i-1][j], c[i][j-1]), & i,j > 0 \text{ e } x_i \neq y_j \end{cases}$$

X		Y	b	d	c	a	b
		0	1	2	3	4	5
	0	0	0	0	0	0	0
a	1	0	0	0	0	1	1
b	2	0	1	1	1	1	2
c	3	0	1	1	2	2	2
b	4	0	1	1	2	2	3

c



Programação Dinâmica
O Problema da Subcadeia Comum Máxima

- Implementação e Testes:
  - LongestCommonSubstringProblem.java
  - LongestCommonSubstringProblemTest.java

Programação Dinâmica

O Problema do Troco de Frobenius (*Coin Change Problem*)

- **Problema:** Dada uma quantia v e uma lista d de denominações de n moedas, qual é o número mínimo de moedas que devemos utilizar para obter a quantia v?
- **Nota:** Nesta versão do problema consideramos ter um suprimento infinito de moedas de qualquer denominação;
- **Lembrete:** Para algumas denominações de moedas pode-se utilizar algoritmos gulosos com sucesso.

Programação Dinâmica

O Problema do Troco de Frobenius (*Coin Change Problem*)

- **Exemplo 01:**
  - $v = 10, n = 2$ e $d = \{1, 5\}$
  - **Solução:**
    - Dez moedas de 1 centavo = 10 moedas;
    - Uma moeda de 5 centavos + cinco moedas de 1 centavo = 6 moedas;
    - Duas moedas de 5 centavos = 2 moedas;
- **Exemplo 02:**
  - $v = 7, n = 4$ e $d = \{1, 3, 4, 5\}$
  - **Solução:**
    - A solução gulosa seria 3, ou seja, uma moeda de 5 centavos e duas moedas de 1 centavo;
    - Entretanto a solução ótima é 2, ou seja, uma moeda de 3 centavos e uma moeda de 4 centavos.

Programação Dinâmica

O Problema do Troco de Frobenius (*Coin Change Problem*)

- **Solução Ótima:**
  - Usamos a recorrência de busca completa:

$$\text{CoinChange}(v) = \begin{cases} 0, & v = 0 \\ -\infty, & v < 0 \\ 1 + \min(\text{CoinChange}(v - d_i)), & i \in \{0,1, \dots, n-1\} \end{cases}$$

  - A resposta é determinada por $\text{CoinChange}(v)$.

Programação Dinâmica

O Problema do Troco de Frobenius (*Coin Change Problem*)

$$\text{CoinChange}(v) = \begin{cases} 0, & v = 0 \\ -\infty, & v < 0 \\ 1 + \min(\text{CoinChange}(v - d_i)), & i \in \{0,1,\dots,n-1\} \end{cases}$$

- **Exemplo:**
  - $v = 10$, $n = 2$ e $d = \{1, 5\}$
  - **Solução:**

< 0	0	1	2	3	4	5	6	7	8	9	10
$-\infty$	0	1	2	3	4	1	2	3	4	5	2

    - $\text{CoinChange}(< 0) = -\infty$
    - $\text{CoinChange}(0) = 0$, o caso base
    - $\text{CoinChange}(1) = 1$, obtido de $1 + \text{CoinChange}(1 - 1)$. $\text{CoinChange}(1 - 5)$ é inviável $(-\infty)$
    - $\text{CoinChange}(2) = 2$, obtido de $1 + \text{CoinChange}(2 - 1)$. $\text{CoinChange}(2 - 5)$ é inviável $(-\infty)$
    - $\text{CoinChange}(3) = 3$, obtido de $1 + \text{CoinChange}(3 - 1)$. $\text{CoinChange}(3 - 5)$ é inviável $(-\infty)$
    - $\text{CoinChange}(4) = 4$, obtido de $1 + \text{CoinChange}(4 - 1)$. $\text{CoinChange}(4 - 5)$ é inviável $(-\infty)$
    - $\text{CoinChange}(5) = 1$, obtido de $1 + \text{CoinChange}(5 - 5)$, menor do que $1 + \text{CoinChange}(5 - 1) = 5$
    - $\text{CoinChange}(6) = 2$, obtido de $1 + \text{CoinChange}(6 - 5)$ ou de $1 + \text{CoinChange}(6 - 1)$
    - $\text{CoinChange}(7) = 3$, obtido de $1 + \text{CoinChange}(7 - 5)$. $\text{CoinChange}(7 - 1)$ é inviável (futuro)
    - $\text{CoinChange}(8) = 4$, obtido de $1 + \text{CoinChange}(8 - 5)$ ou de $1 + \text{CoinChange}(8 - 1)$
    - $\text{CoinChange}(9) = 5$, obtido de $1 + \text{CoinChange}(9 - 5)$, menor do que $1 + \text{CoinChange}(9 - 1) = 5$
    - $\text{CoinChange}(10) = 2$, obtido de $1 + \text{CoinChange}(10 - 5)$, menor do que $1 + \text{CoinChange}(10 - 1) = 6$

Programação Dinâmica

O Problema do Troco de Frobenius (*Coin Change Problem*)

- Implementação e Testes:
  - CoinChangeProblem.java
  - CoinChangeProblemTest.java

Programação Dinâmica

O Problema do Subarranjo Máximo de Maior Soma

- **Problema:** Dado um array de n inteiros diferentes de zero, determine o intervalo de maior soma.
- **Exemplo:** Consideremos a sequência -2, 1, -3, 4, -1, 2, 1, -5, 4. A subsequência contígua de números de soma máxima é 4, -1, 2, 1, cuja soma é 6.
- **Observação:** Existem $O(n^2)$ possibilidades de intervalos, porém, há um algoritmo de divisão e conquista com tempo $O(n \lg n)$, entretanto o algoritmo de Kadane resolve este problema em tempo $O(n)$.

Programação Dinâmica
O Problema do Subarranjo Máximo de Maior Soma

- O princípio do algoritmo de Kadane é manter a soma dos elementos já analisados e reiniciar o valor em 0 caso a soma se torne negativa;
- Isto porque é melhor começar de zero do que continuar de uma soma "ruim";
- Embora possa também ser interpretado como um algoritmo guloso, este algoritmo possui características de programação dinâmica na medida em que a cada passo possui duas opções: incrementa a soma do intervalo anterior ou inicia um novo intervalo.

Programação Dinâmica

O Problema do Subarranjo Máximo de Maior Soma

- **Exemplo:** Para a sequência -2, 1, -3, 4, -1, 2, 1, -5, 4, encontrar o intervalo de maior soma.

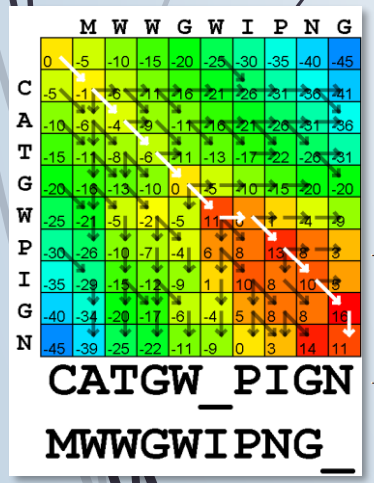
	0	1	2	3	4	5	6	7	8
sequence	-2	1	-3	**4**	**-1**	**2**	**1**	-5	4
sums	-2/0	1	-2/0	4	3	5	6	1	5
results	0	1	1	4	4	5	6	6	**6**

Programação Dinâmica
O Problema do Subarranjo Máximo de Maior Soma

- Implementação e Testes:
  - MaximumSumSubarrayProblem.java
  - MaximumSumSubarrayProblemTest.java

Programação Dinâmica
O Problema do Alinhamento de Proteínas

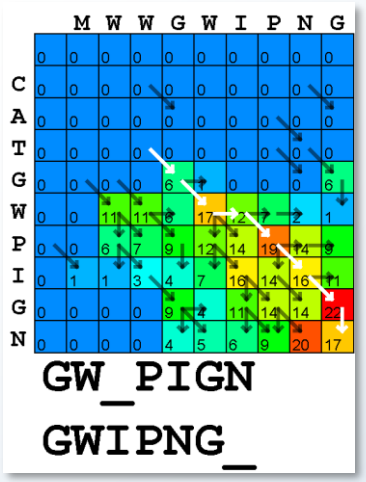
- Outro exemplo de aplicação de programação dinâmica são os algoritmos de alinhamento de sequências biológicas como DNAs, RNAs e proteínas;
- A ideia básica desses algoritmos é similar à ideia do algoritmo de solução do problema da subcadeia comum máxima, com a diferença de utilizarem matrizes de pontuação para calcular o *score* (pontuação) na comparação de duas bases nitrogenadas ou dois aminoácidos.



Alinhamento Global
Algoritmo de Needleman-Wunsch

$$F(0,0) = 0$$

$$F(i,j) = max \begin{cases} F(i-1,j-1) + s(s_j^1, s_i^2) \\ F(i-1,j) + d \\ F(i,j-1) + d \end{cases}$$



Alinhamento Local
Algoritmo de Smith-Waterman

$$F(0,0) = 0$$

$$F(i,j) = max \begin{cases} F(i-1,j-1) + s(s_j^1, s_i^2) \\ F(i-1,j) + d \\ F(i,j-1) + d \\ 0 \end{cases}$$

Programação Dinâmica
O Problema do Alinhamento de Proteínas

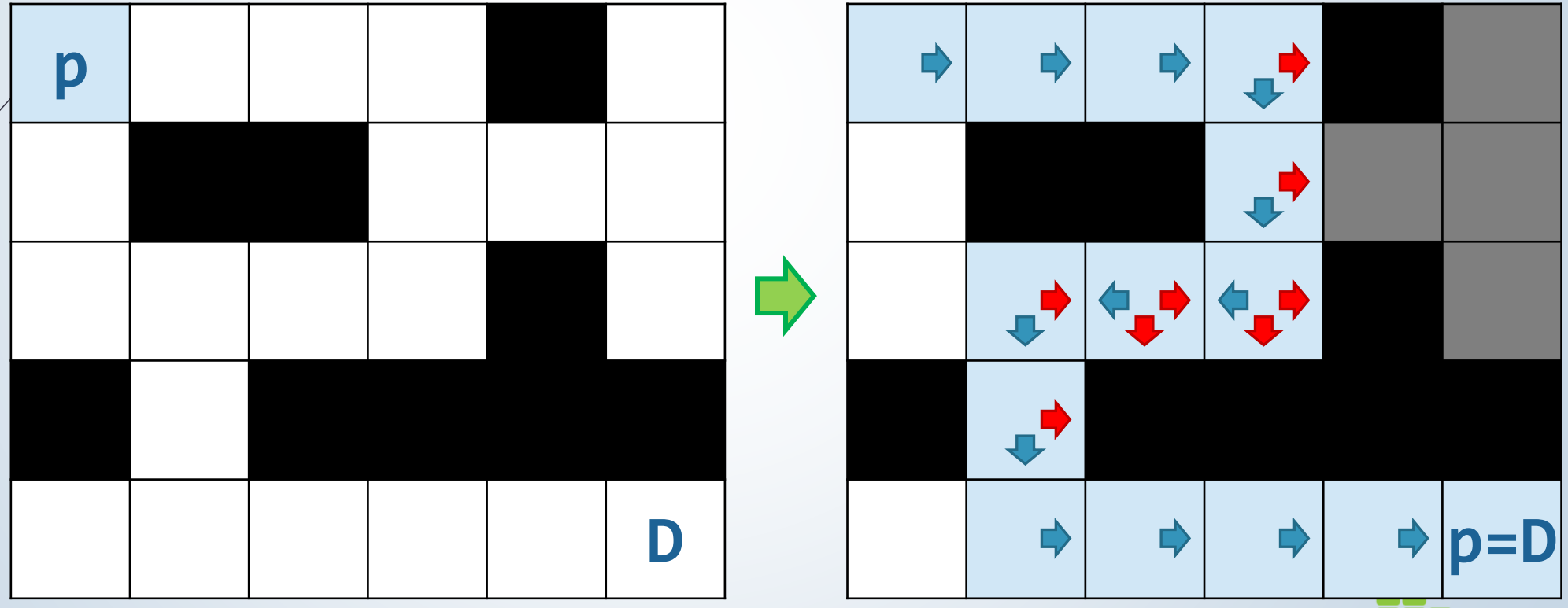
- Implementação e Testes:
  - AlinhamentosDidaticos (Repositório, projeto NetBeans)

Programação Dinâmica
Considerações Finais

- **Vantagens:**
  - Economiza computação de soluções em problemas que possuem superposição de problemas e subestrutura ótima, gerando melhoria no desempenho;
- **Desvantagens:**
  - O número de soluções armazenadas na tabela pode crescer rapidamente caso o espaço de soluções não seja pequeno, exigindo assim muita memória;
- **Observação Final:**
  - A Programação Dinâmica usa memória adicional para economizar em tempo de computação, um exemplo claro de trade-off entre tempo e memória:
    - Implica um conflito de escolha e uma consequente relação de compromisso;
    - Visa à resolução de problema mas acarreta outro.

Backtracking
Exercícios Escritos

- **Exercício e2.1:** Usando como base a apresentação da árvore de exploração do backtracking com poda do quadrado latino, apresente a árvore de exploração do exemplo de solução do labirinto.



Programação Dinâmica
Exercícios Escritos

- **Exercício e2.2:** Aplicar o algoritmo de resolução do Problema da Mochila para a resolução do problema com as condições abaixo:
  - $c = 12,\ v = \{400, 900, 200, 110\},\ w = \{7, 2, 4, 3\}$
- **Exercício e2.3:** Aplicar o algoritmo para determinar a Subcadeia Comum Máxima entre as strings `abcbdab` e `bdcaba`.
- **Exercício e2.4:** Aplicar o algoritmo de resolução do Problema do Troco para as condições abaixo:
  - $v = 7,\ n = 4$ e $d = \{1, 3, 4, 5\}$
- **Exercício e2.5:** Aplicar o algoritmo de Kadane para solução do Problema do Subarranjo Máximo de Maior Soma para as condições abaixo:
  - $\text{sequence} = \{4, -5,\ 4, -3,\ 4,\ 4, -4,\ 4, -5\}$ e $n = 9$

Backtracking
Exercícios de Implementação

- **Exercício i2.1:** Usando a implementação de exemplo do resolvedor do labirintos, implemente um resolvedor de labirintos para três dimensões.
- **Exercício i2.2:** Usando a implementação de exemplo do resolvedor do quadrado latino, implemente um resolvedor de Sudokus.

Programação Dinâmica
Exercícios de Implementação

- **Exercício i2.3:** Modifique a implementação de exemplo do resolvedor do Problema da Mochila para que além de mostrar o valor máximo dos itens que poderão ser carregados, mostre também quais os itens que entrarão na mochila.
- **Exercício i2.4:** Modifique a implementação de exemplo do resolvedor do Problema do Troco para que além de mostrar qual a quantidade de moedas que serão usadas, mostre também a quantidade e o valor de cada moeda que fará a composição do troco.
- **Exercício i2.5:** Modifique a implementação de exemplo do resolvedor do Problema do Subarranjo Máximo de Maior Soma para que além de mostrar o valor da soma, mostre também o intervalo em que o subarranjo em questão está situado.

Bibliografia

SEDGEWICK, R.; WAYNE, K. **Algorithms**. 4. ed. Boston: Pearson Education, 2011. 955 p.

GOODRICHM M. T.; TAMASSIA, R. **Estruturas de Dados & Algoritmos em Java**. Porto Alegre: Bookman, 2013. 700 p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos – Teoria e Prática**. 3. ed. São Paulo: GEN LTC, 2012. 1292 p.

Obs: esse material foi baseado no material da disciplina de Estruturas de Dados II do Prof. Dr. Breno Lisi Romano