Sistemas Inteligentes

2014/2

1) O que você espera desta disciplina?

2) Você imagina a aplicação do conteúdo da disciplina em sua carreira profissional? Exemplifique em caso afirmativo.

Resolução de problemas por meio de busca

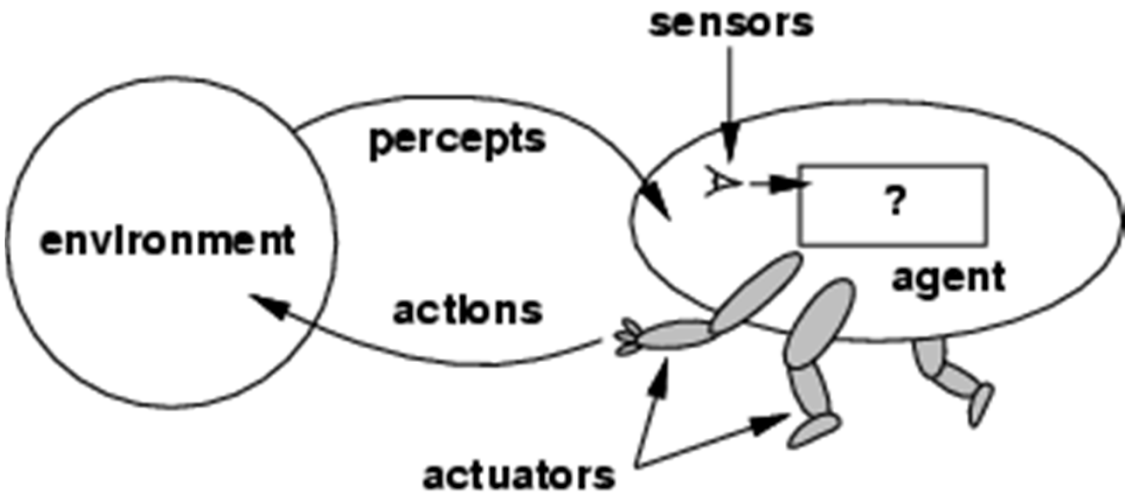
Russell & Norvig

Alguns conceitos do Cap. 2

Seções 3.1 a 3.5

Agentes

- Um **agente** é algo capaz de perceber seu **ambiente** por meio de **sensores** e de agir sobre esse ambiente por meio de **atuadores**.



PEAS

- Ao projetar um agente, a primeira etapa deve ser sempre especificar o ambiente de tarefa.
  - **P**erformance = Medida de desempenho
  - **E**nvironment = Ambiente
  - **A**ctuators = Atuadores
  - **S**ensors = Sensores

Propriedades de ambientes de tarefa

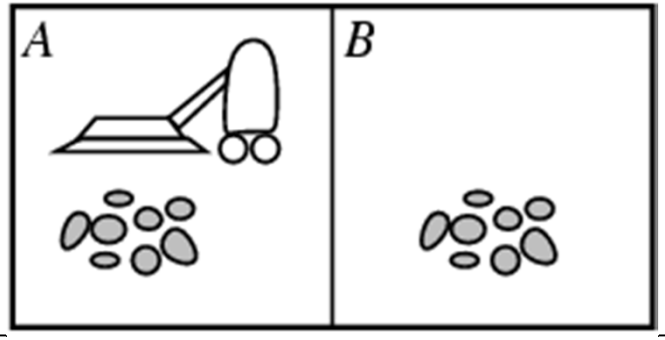
- Completamente observável (versus parcialmente observável)
- Determinístico (versus estocástico)
- Episódico (versus sequencial)
- Discreto (versus contínuo)
- Estático (versus dinâmico)

Exemplo

	Xadrez com relógio	Xadrez sem relógio	Direção de Táxi
Completamente observável	Sim	Sim	Não
Determinístico	Sim	Sim	Não
Episódico	Não	Não	Não
Estático	Semi	Sim	Não
Discreto	Sim	Sim	Não
Agente único	Não	Não	Não

- O mundo real é parcialmente observável, estocástico, seqüêncial, dinâmico, contínuo, multi-agente.

Agente reativo simples




```
Função AGENTE-ASPIRADOR-DE-PÓ-REATIVO([posição,estado])
   retorna uma ação
      se estado = Sujo então retorna Aspirar
      senão se posição = A então retorna Direita
      senão se posição = B então retorna Esquerda
```

- Regras condição-ação (regras se-então) fazem uma ligação direta entre a percepção atual e a ação.
- O agente funciona apenas se o ambiente for completamente observável e a decisão correta puder ser tomada com base apenas na percepção atual.

Agentes reativos baseados em modelo

Função AGENTE-REATIVO-COM-ESTADOS(*percepção*) **retorna** uma *ação*

Variáveis estáticas:

estado, uma descrição do estado atual do mundo

regras, um conjunto de regras condição-ação

ação, a ação mais recente, incialmente nenhuma

estado ← ATUALIZA-ESTADO(*estado*, *ação*, *percepção*)

regra ← REGRA-CORRESPONDENTE(*estado*, *regras*)

ação ← AÇÃO-DA-REGRA[regra]

retornar *ação*

Agentes de resolução de problemas

- Agentes reativos não funcionam em ambientes para quais o número de regras condição-ação é grande demais para armazenar.
- Nesse caso podemos construir um tipo de agente baseado em objetivo chamado de agente de resolução de problemas.

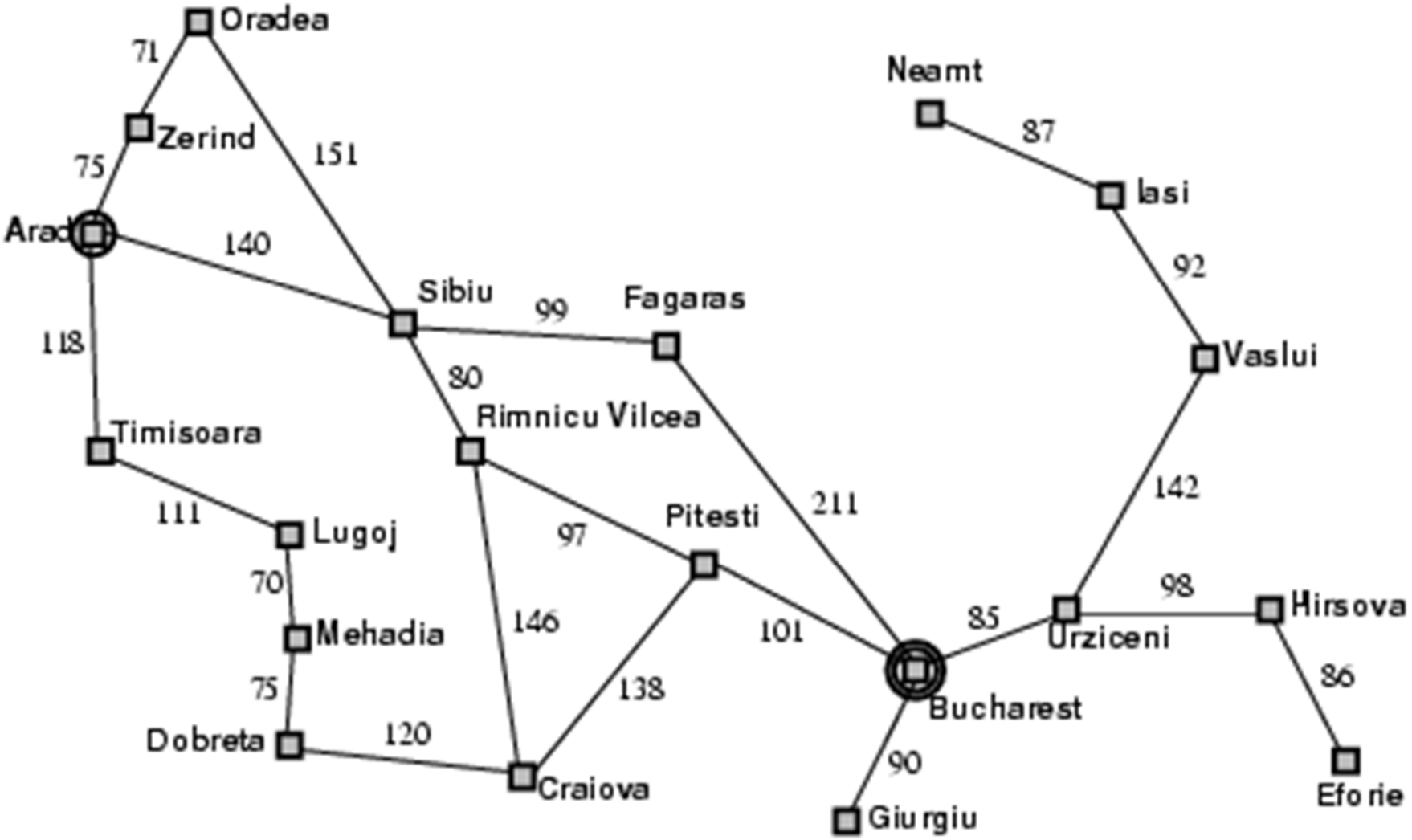
Busca

- Um agente com várias opções imediatas pode decidir o que fazer comparando diferentes sequências de ações possíveis.
- Esse processo de procurar pela melhor sequência é chamado de busca.
- Formular objetivo → buscar → executar

Exemplo: Romênia

- De férias na Romênia; atualmente em Arad.
- Vôo sai amanhã de Bucareste.
- Formular objetivo:
  - Estar em Bucareste
- Formular problema:
  - estados: cidades
  - ações: dirigir entre as cidades
- Encontrar solução:
  - sequência de cidades, ex., Arad, Sibiu, Fagaras, Bucareste.

Exemplo: Romênia



Formulação de problemas

Um problema pode ser definido por quatro itens:

1. Estado inicial ex., "em Arad"
2. Ações ou função sucessor *S(x)* = conjunto de pares estado-ação
   - ex., *S(Arad)* = {<*Arad* → *Zerind, Zerind*>, … }
3. Teste de objetivo, pode ser
   - explícito, ex., *x* = "em Bucharest"
   - implícito, ex., *Cheque-mate(x)*

4. Custo de caminho (aditivo)
   - ex., soma das distâncias, número de ações executadas, etc.
   - *c(x,a,y)* é o custo do passo, que deve ser sempre ≥ 0

- Uma solução é uma sequência de ações que levam do estado inicial para o estado objetivo.
- Uma solução ótima é uma solução com o menor custo de caminho.

Agente de resolução de problemas

```
função AGENTE-DE RESOLUÇÃO-DE-PROBLEMAS-SIMPLES(percepção) retorna uma ação
  persistente: seq, uma sequência de ações, inicialmente vazia
               estado, alguma descrição do estado atual do mundo
               objetivo, um objetivo, inicialmente nulo
               problema, uma formulação de problema
  estado ← ATUALIZAR-ESTADO(estado, percepção)
  se seq está vazia então faça
    objetivo ← FORMULAR-OBJETIVO(estado)
    problema ← FORMULAR-PROBLEMA(estado, objetivo)
    seq ← BUSCA(problema)
    se seq = falhar então retorne uma ação nula
  ação ← PRIMEIRO(seq)
  seq ← RESTO(seq)
  retornar ação
```

Supõe que ambiente é estático, observável, discreto e determinístico.

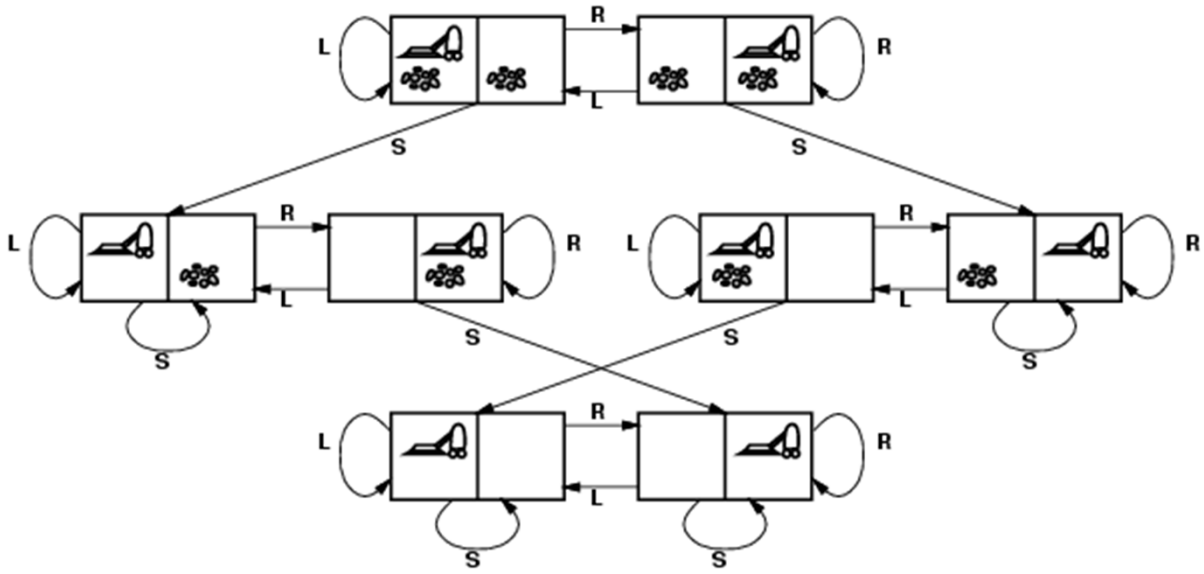
Espaço de estados

- O conjunto de todos os estados acessíveis a partir de um estado inicial é chamado de espaço de estados.
  - Os estados acessíveis são aqueles dados pela função sucessora.
- *O espaço de estados pode ser interpretado como um grafo em que os nós são estados e os arcos são ações*.

Selecionando um espaço de estados

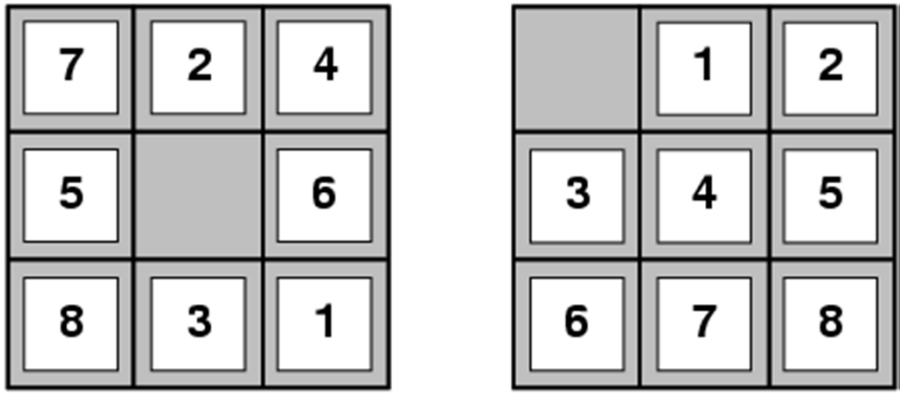
- O mundo real é absurdamente complexo
  → o espaço de estados é uma abstração
- Estado (abstrato) = conjunto de estados reais
- Ação (abstrata) = combinação complexa de ações reais
  - ex., "Arad → Zerind" representa um conjunto complexo de rotas, desvios, paradas, etc.
  - Qualquer estado real do conjunto “em Arad“ deve levar a algum estado real “em Zerind“.
- Solução (abstrata) = conjunto de caminhos reais que são soluções no mundo real
- A abstração é útil se cada ação abstrata é mais fácil de executar que o problema original

Exemplo 1: Espaço de estados do mundo do aspirador de pó



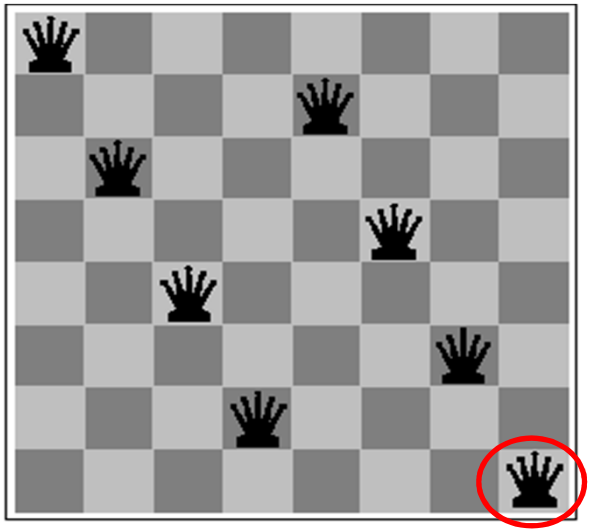
- **Estados:** Definidos pela posição do robô e sujeira (8 estados)
- **Estado inicial:** Qualquer um
- **Função sucessor:** pode-se executar qualquer uma das ações em cada estado (esquerda, direita, aspirar)
- **Teste de objetivo:** Verifica se todos os quadrados estão limpos
- **Custo do caminho:** Cada passo custa 1, e assim o custo do caminho é o número de passos do caminho

Exemplo 2:
O quebra-cabeça de 8 peças



- **Estados:** Especifica a posição de cada uma das peças e do espaço vazio
- **Estado inicial:** Qualquer um
- **Função sucessor:** gera os estados válidos que resultam da tentativa de executar as quatro ações (mover espaço vazio para esquerda, direita, acima ou abaixo)
- **Teste de objetivo:** Verifica se o estado corresponde à configuração objetivo.
- **Custo do caminho:** Cada passo custa 1, e assim o custo do caminho é o número de passos do caminho

Exemplo 3a: Oito rainhas
Formulação incremental



Quase solução

- **Estados:** qualquer disposição de 0 a 8 rainhas
- **Estado inicial**: nenhuma rainha
- **Função sucessor**: colocar 1 rainha em qualquer vazio
- **Teste**: 8 rainhas no tabuleiro, nenhuma atacada
- 64x63x...57 ≅ $1{,}8 \times 10^{14}$ sequências para investigar

Exemplo 3b: Oito rainhas
Backtracking

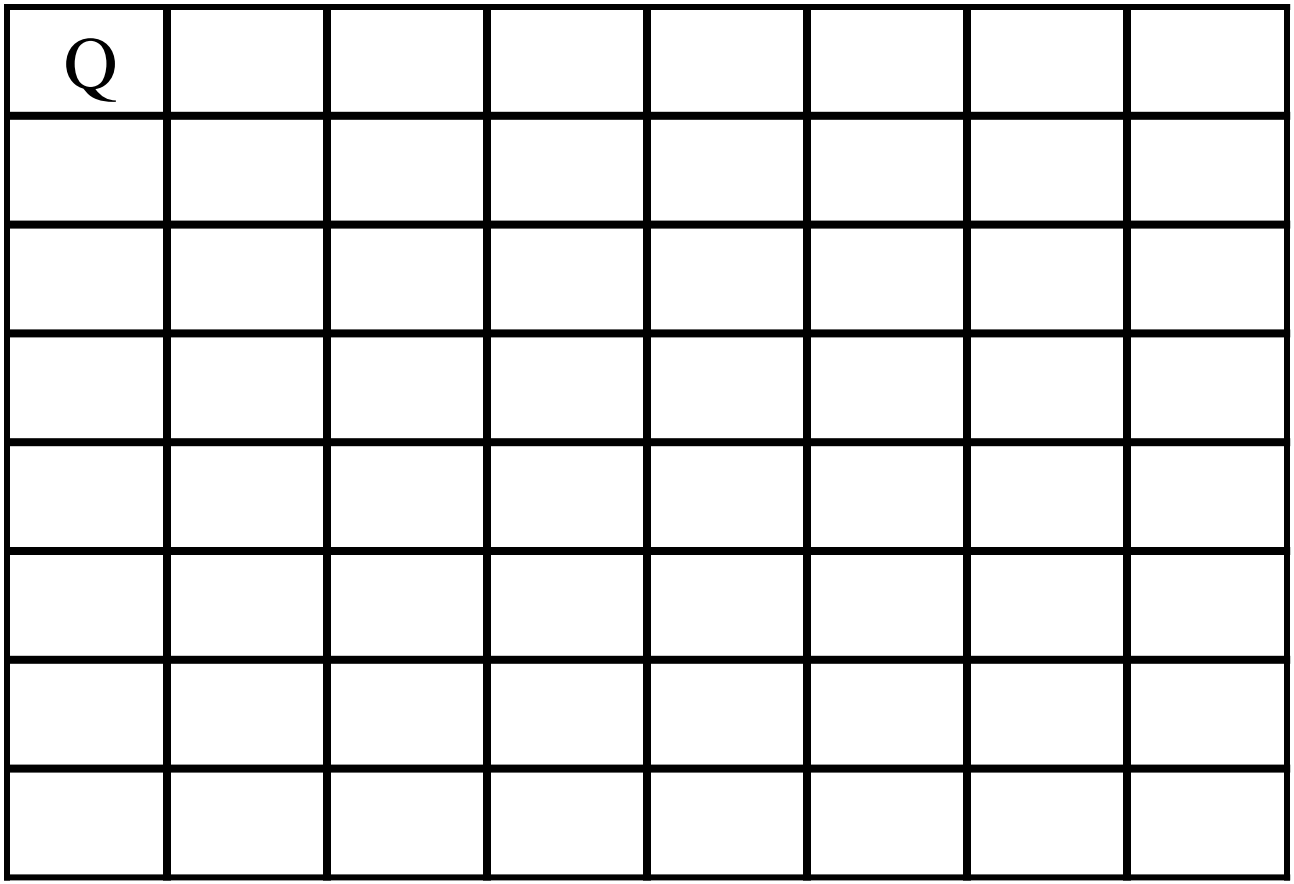
Técnica em procedimentos de busca que corresponde ao retorno de uma exploração.

Ex: Busca-em-Profundidade (veremos a seguir)
Quando chegamos a um nó *v* pela primeira vez, cada aresta incidente a *v* é explorada e então o controle volta (***backtracks***) ao nó a partir do qual *v* foi alcançado.

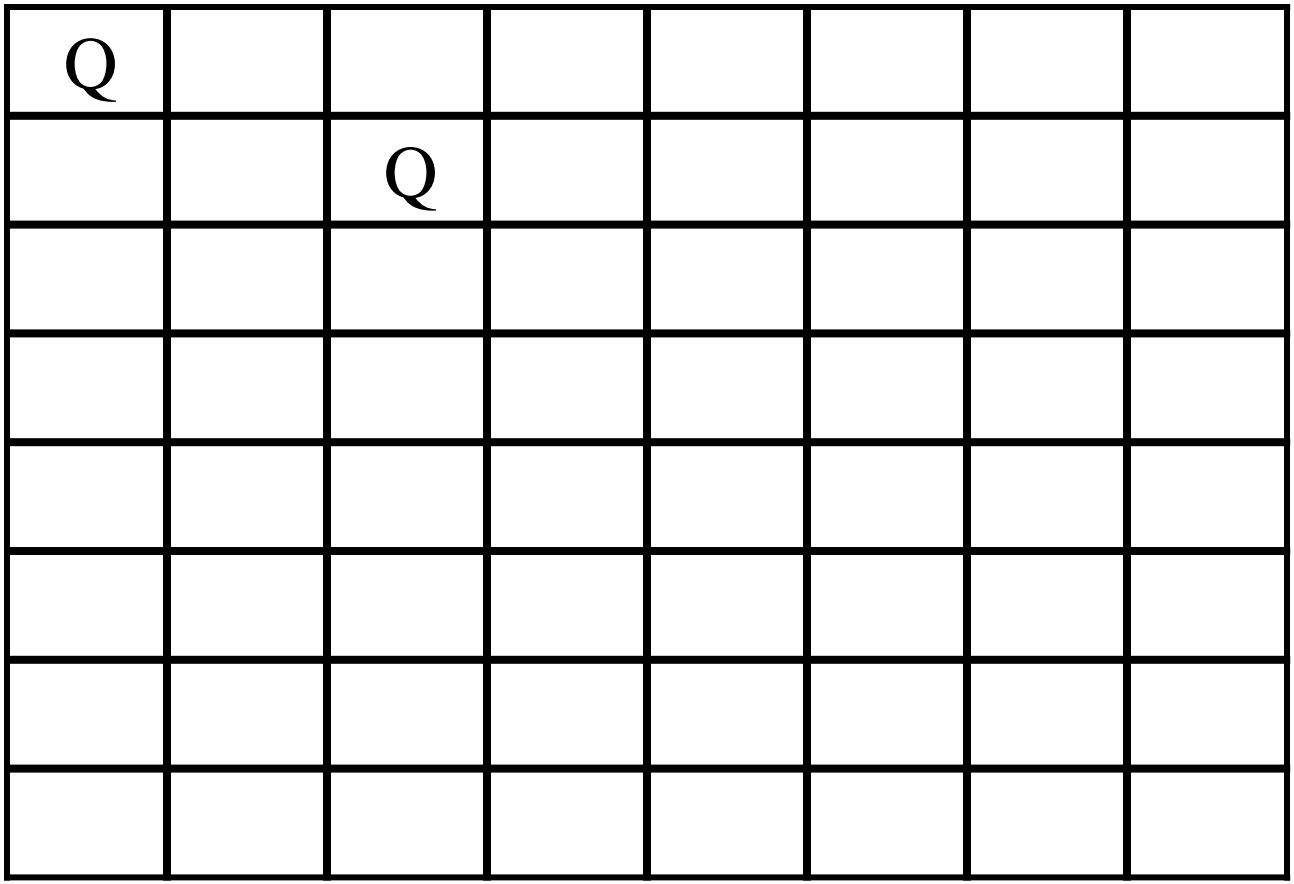
Algoritmo para
O Problema das Oito Rainhas

1. Coloque uma rainha na posição mais à esquerda da primeira linha.
2. Enquanto não houver oito rainhas no tabuleiro faça:
   Se na próxima linha existir uma coluna que não está sob ataque de uma rainha já no tabuleiro
   então coloque uma rainha nesta posição
   senão | volte à linha anterior **/* backtrack */**
   | mova a rainha o mínimo necessário para a direita de forma que ela não fique sob ataque

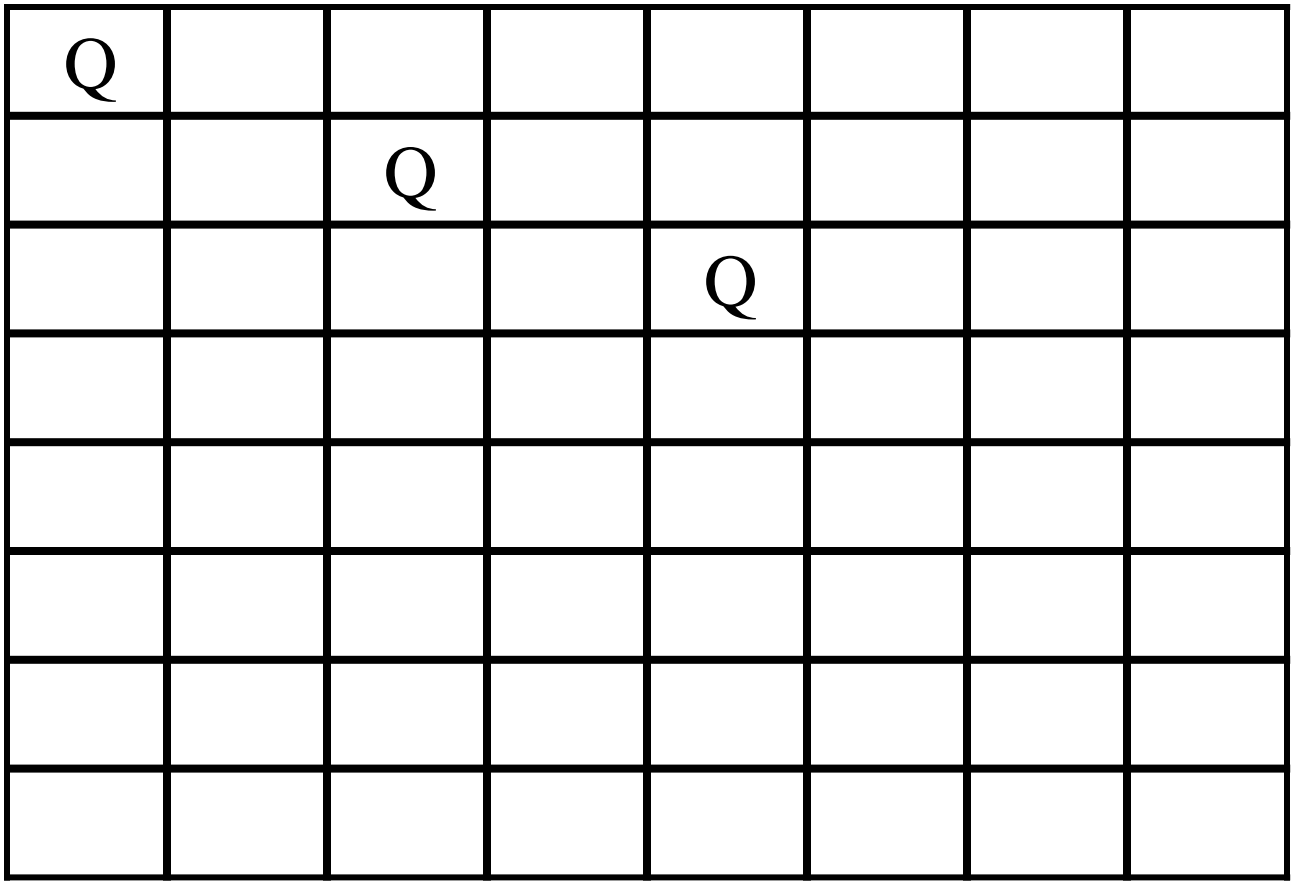
Algoritmo para
O Problema das Oito Rainhas



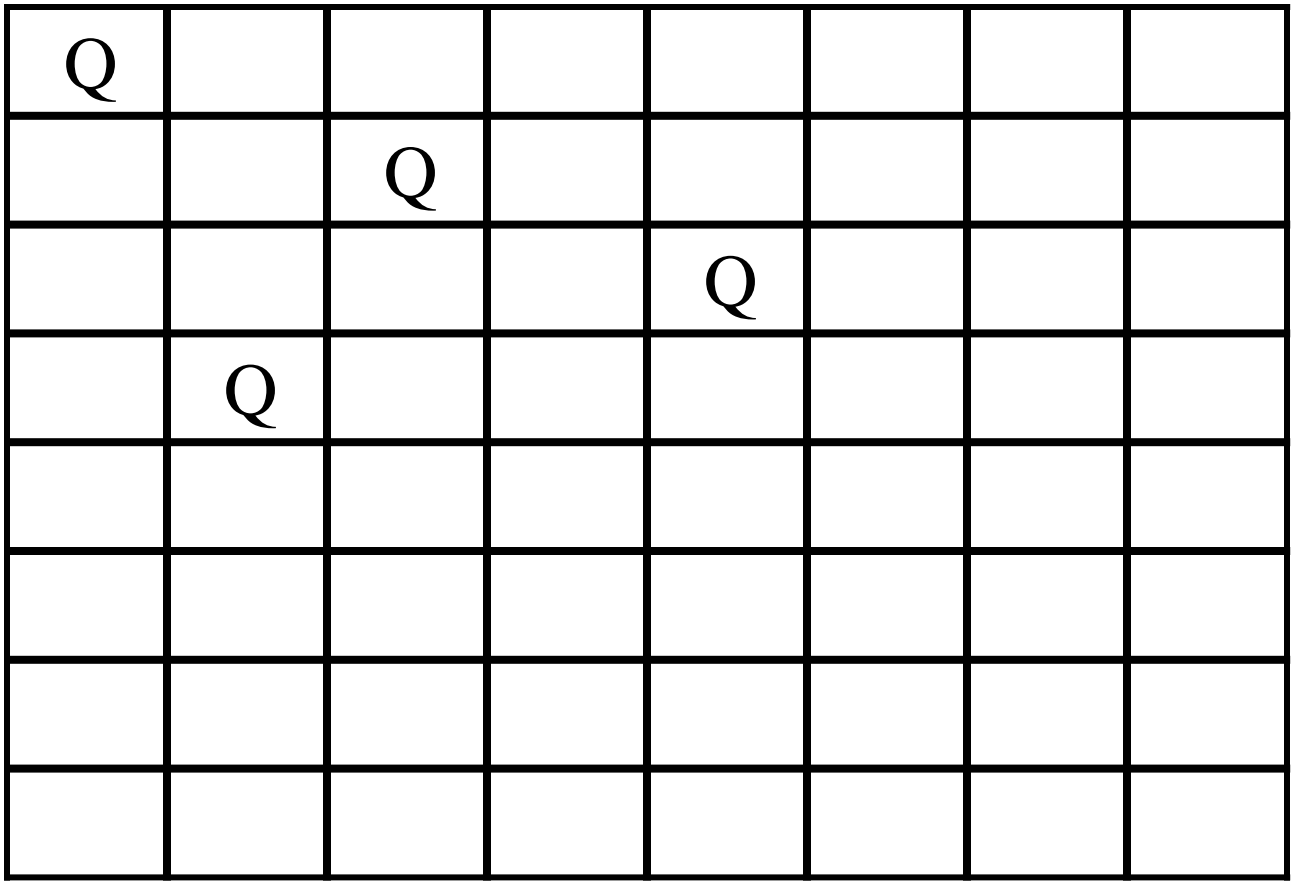
Algoritmo para
O Problema das Oito Rainhas



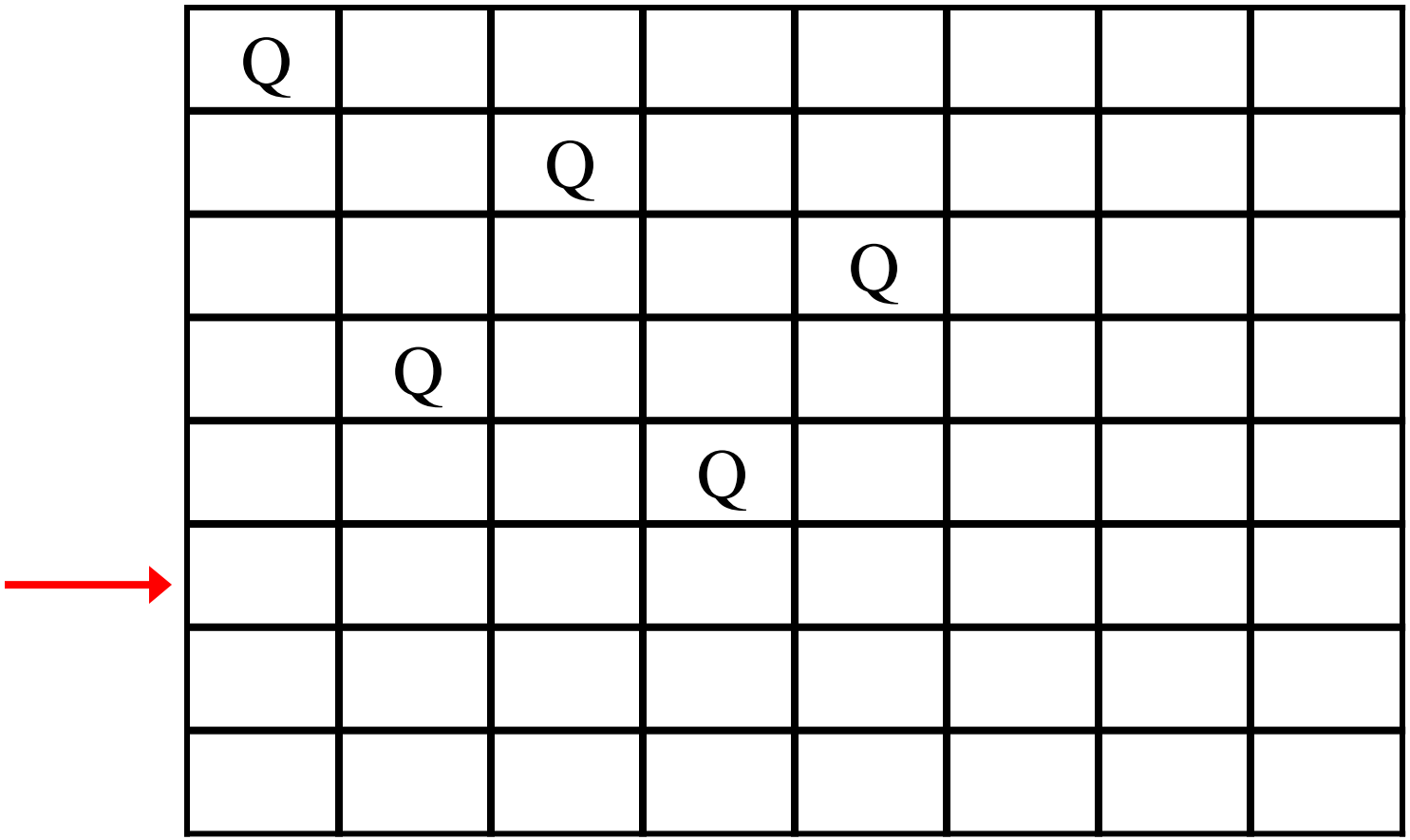
Algoritmo para
O Problema das Oito Rainhas



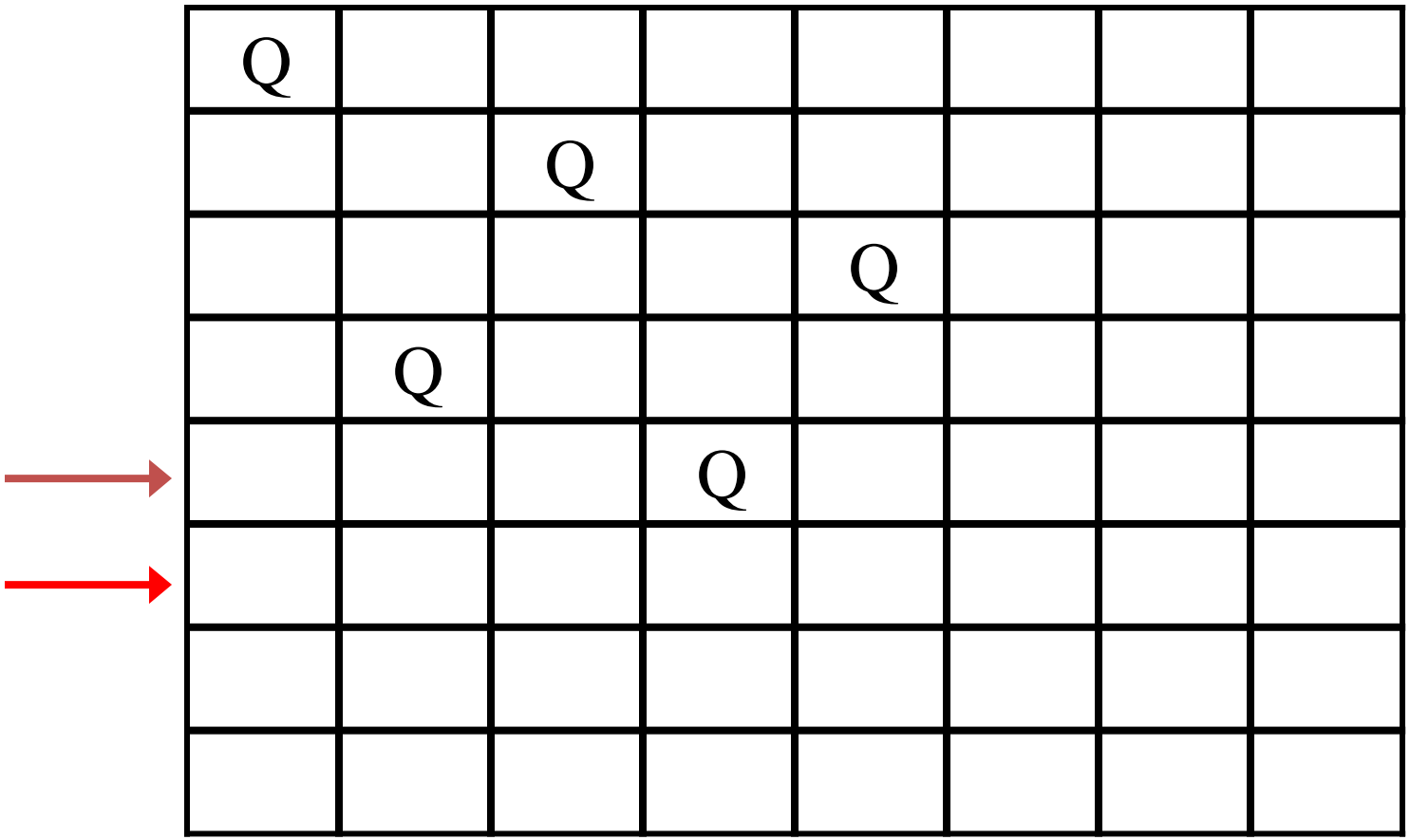
Algoritmo para
O Problema das Oito Rainhas



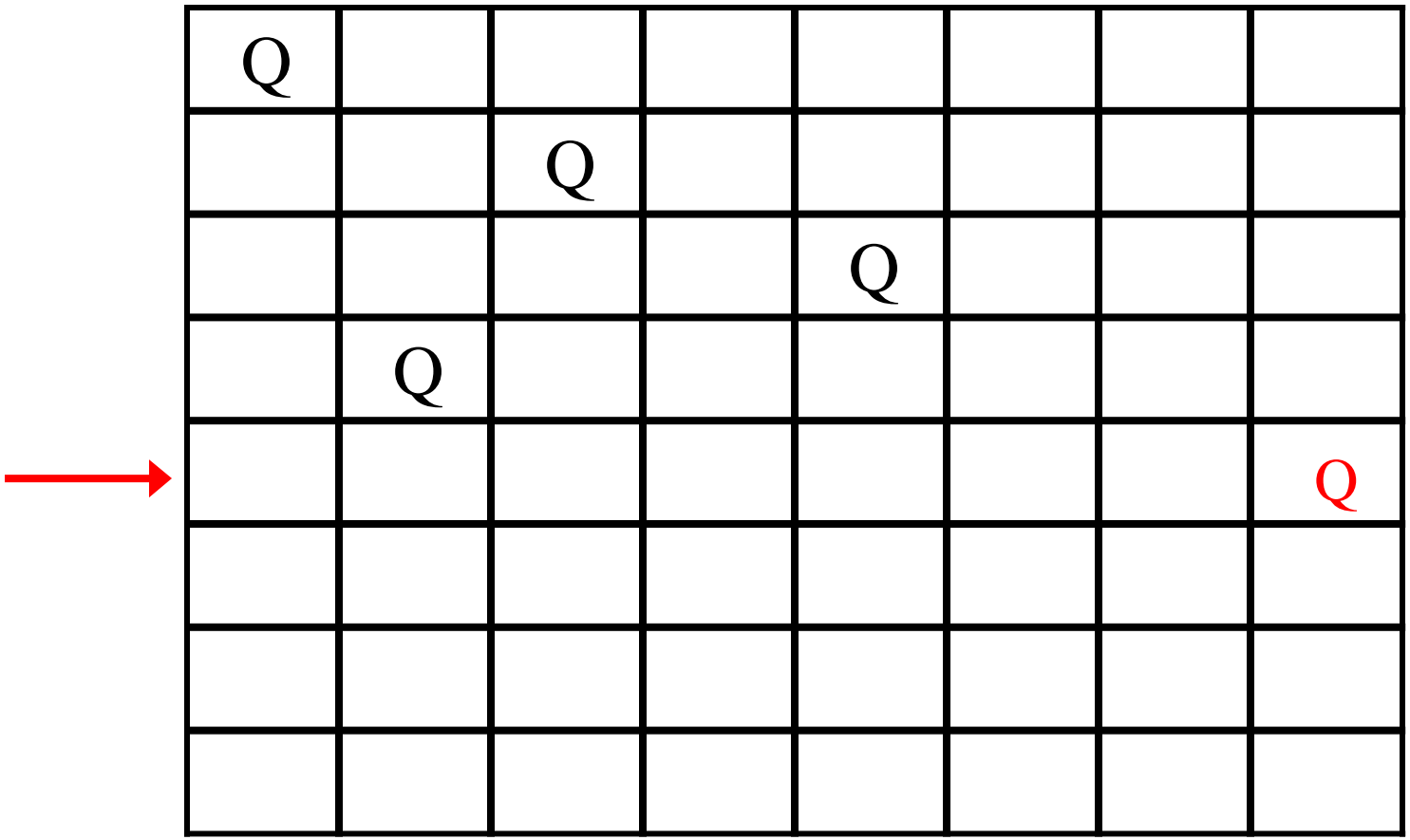
Algoritmo para
O Problema das Oito Rainhas



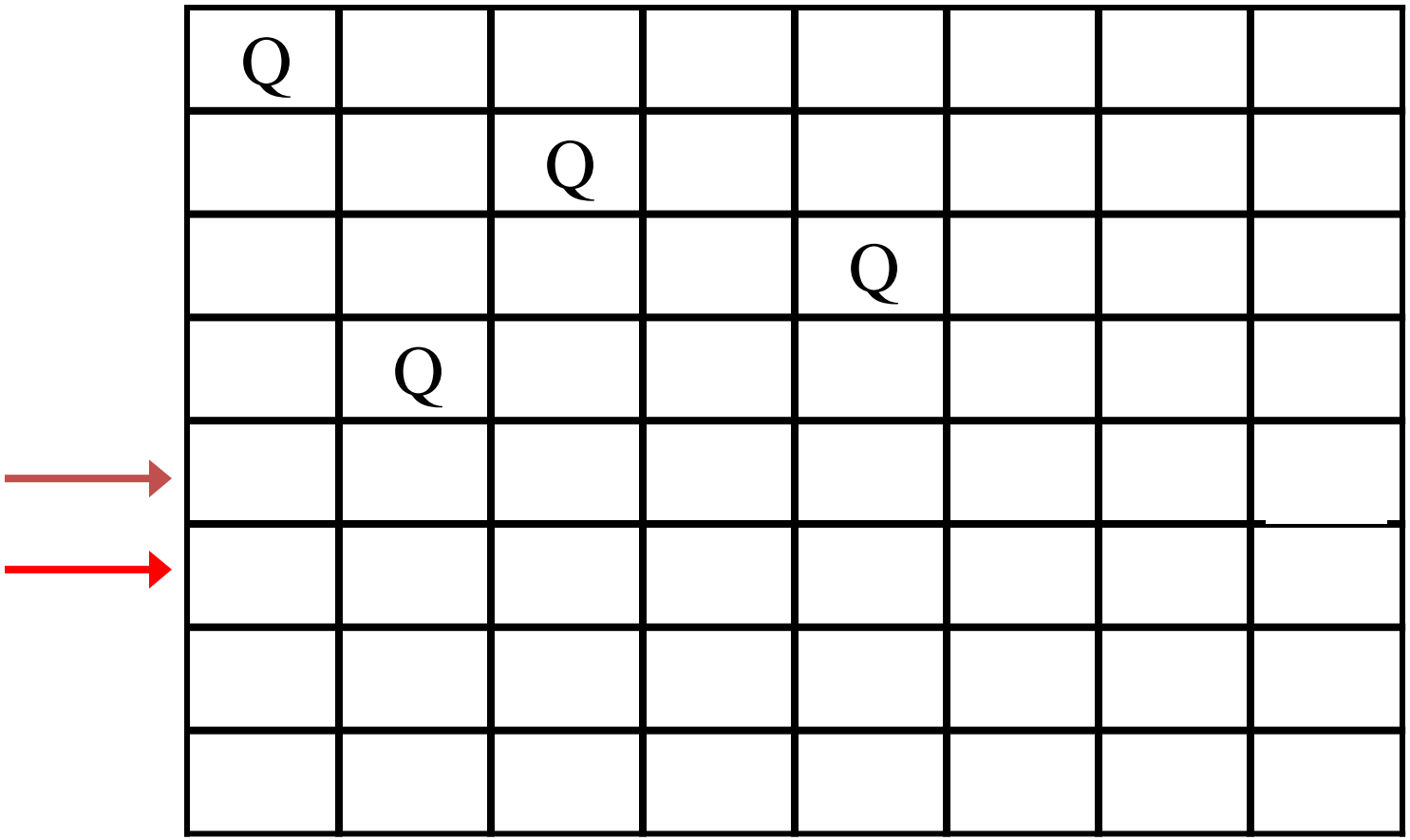
Algoritmo para
O Problema das Oito Rainhas



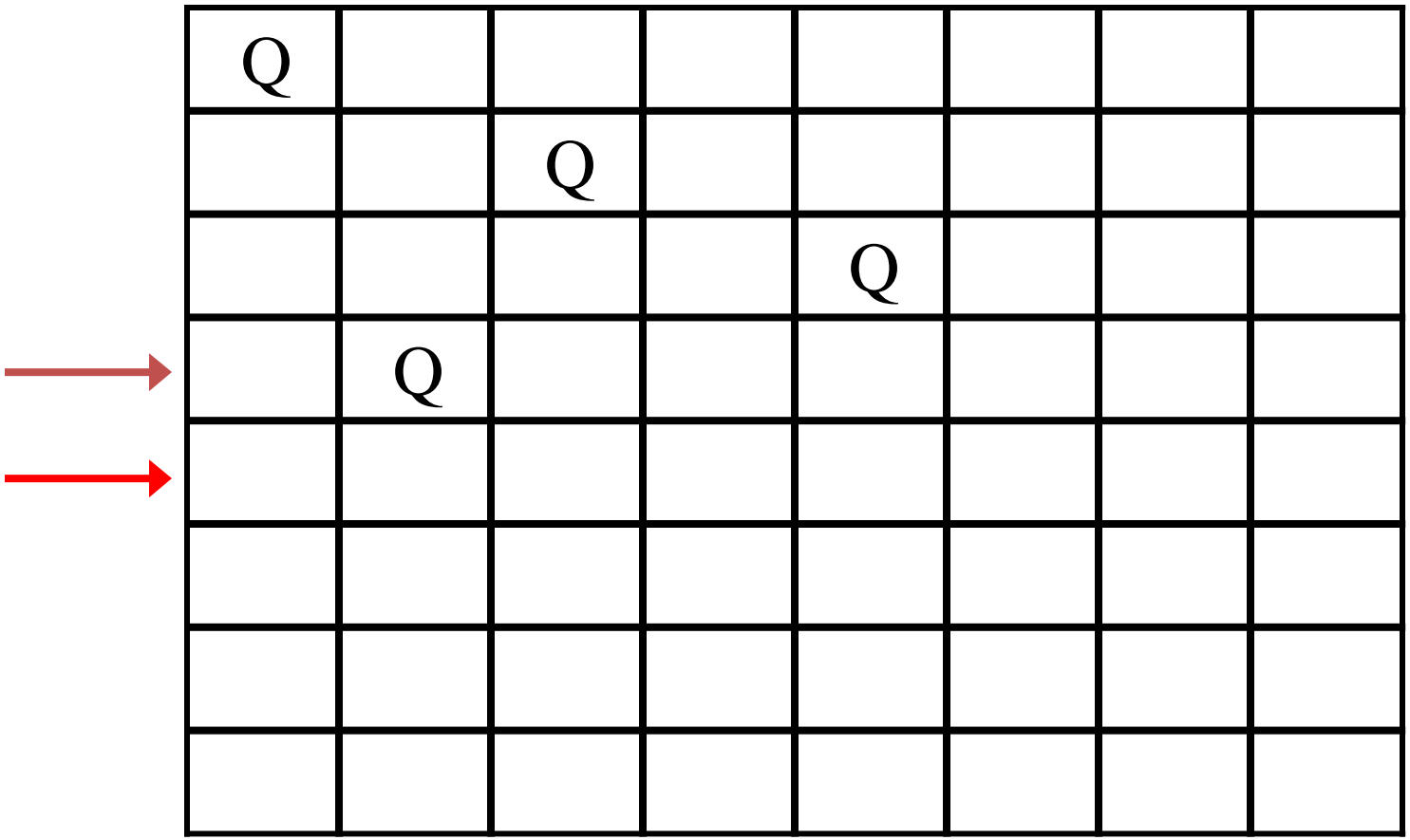
Algoritmo para
O Problema das Oito Rainhas



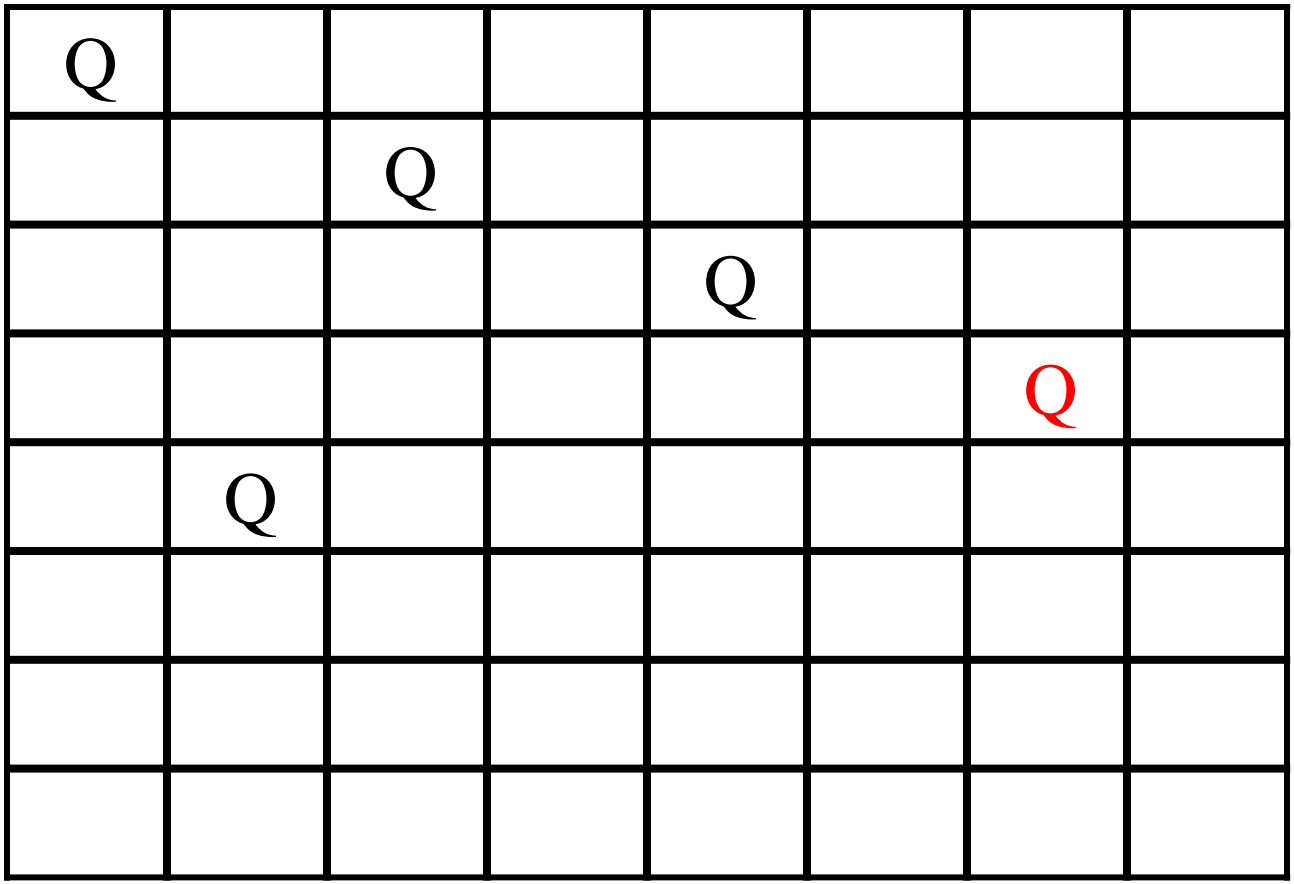
Algoritmo para
O Problema das Oito Rainhas



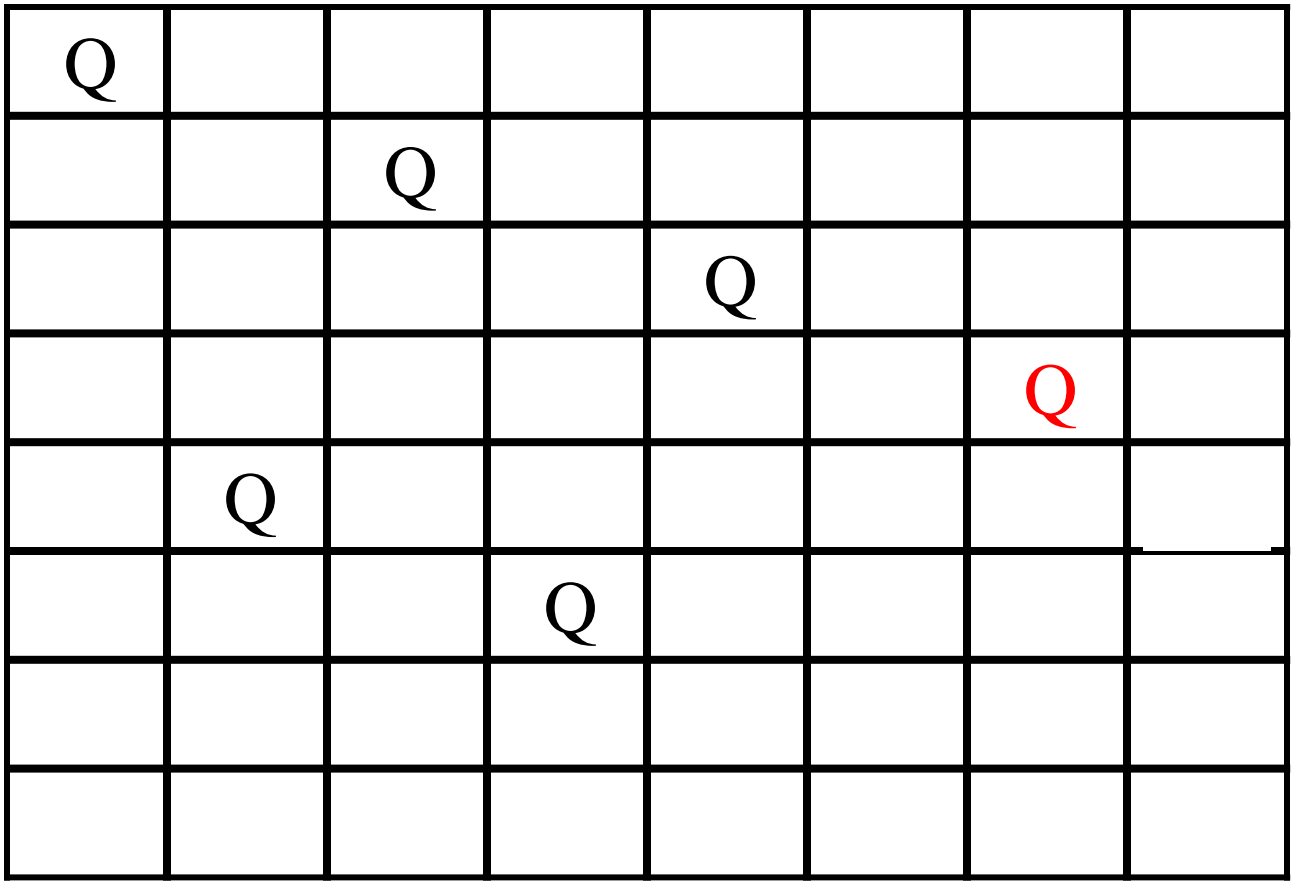
Algoritmo para
O Problema das Oito Rainhas



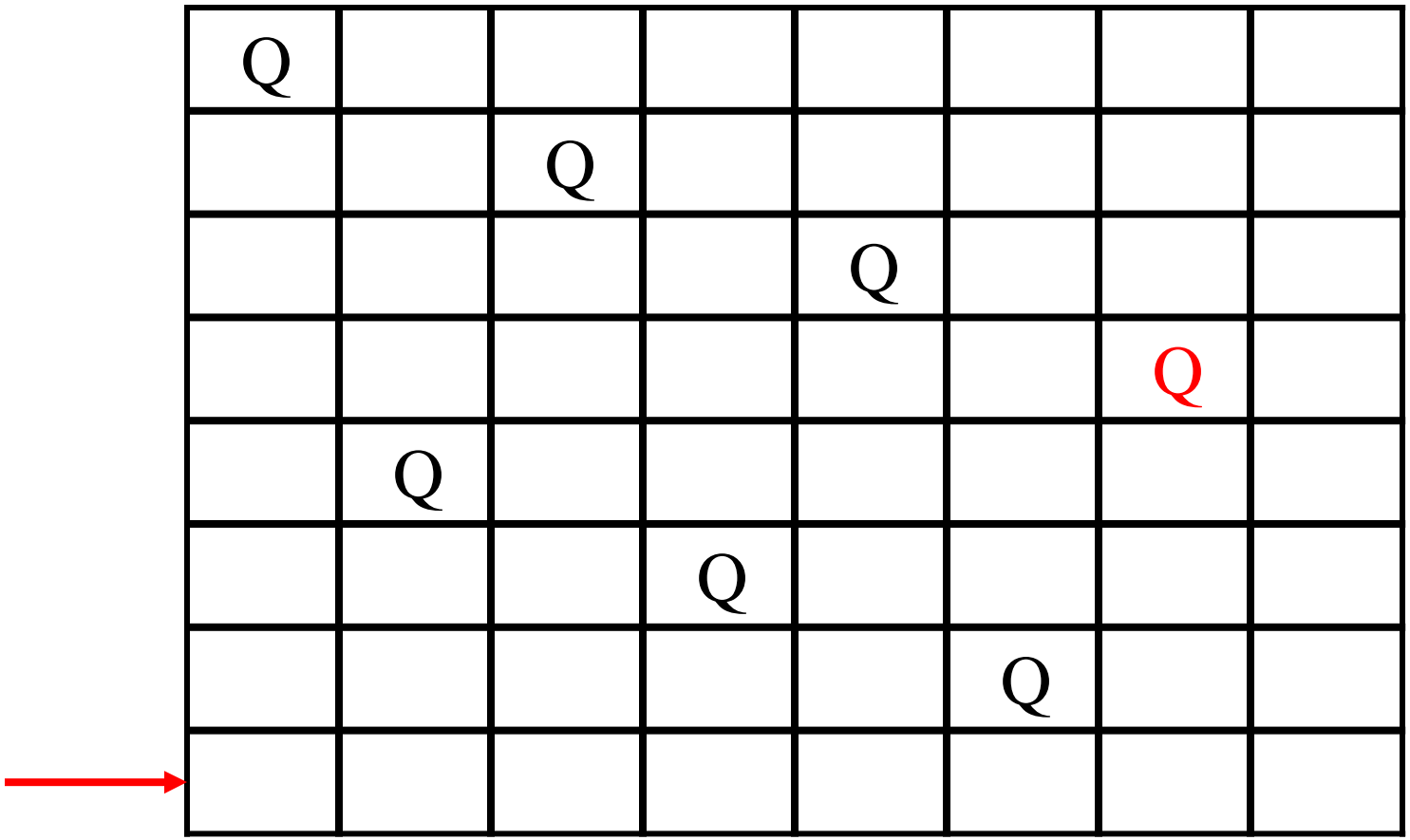
Algoritmo para
O Problema das Oito Rainhas



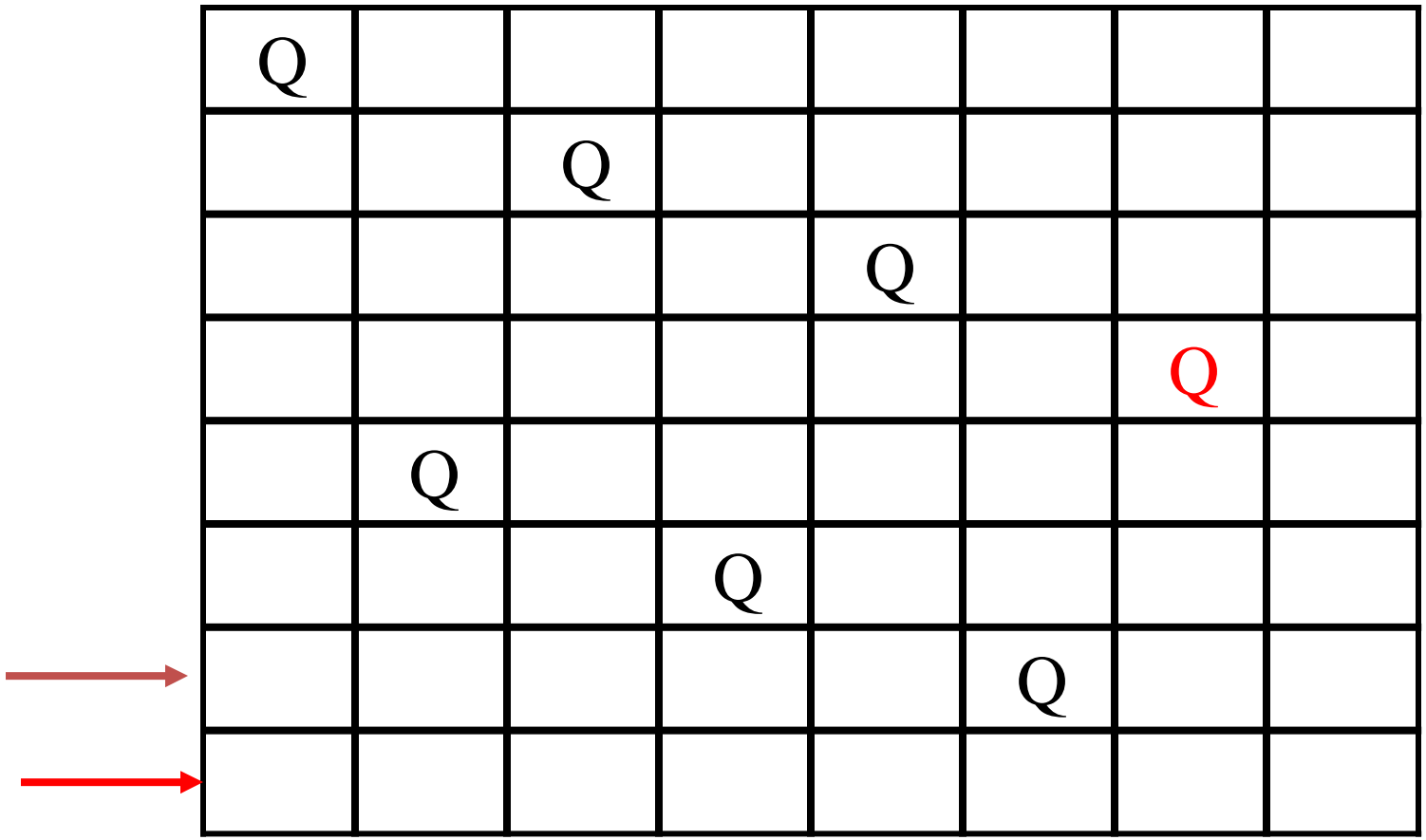
Algoritmo para
O Problema das Oito Rainhas



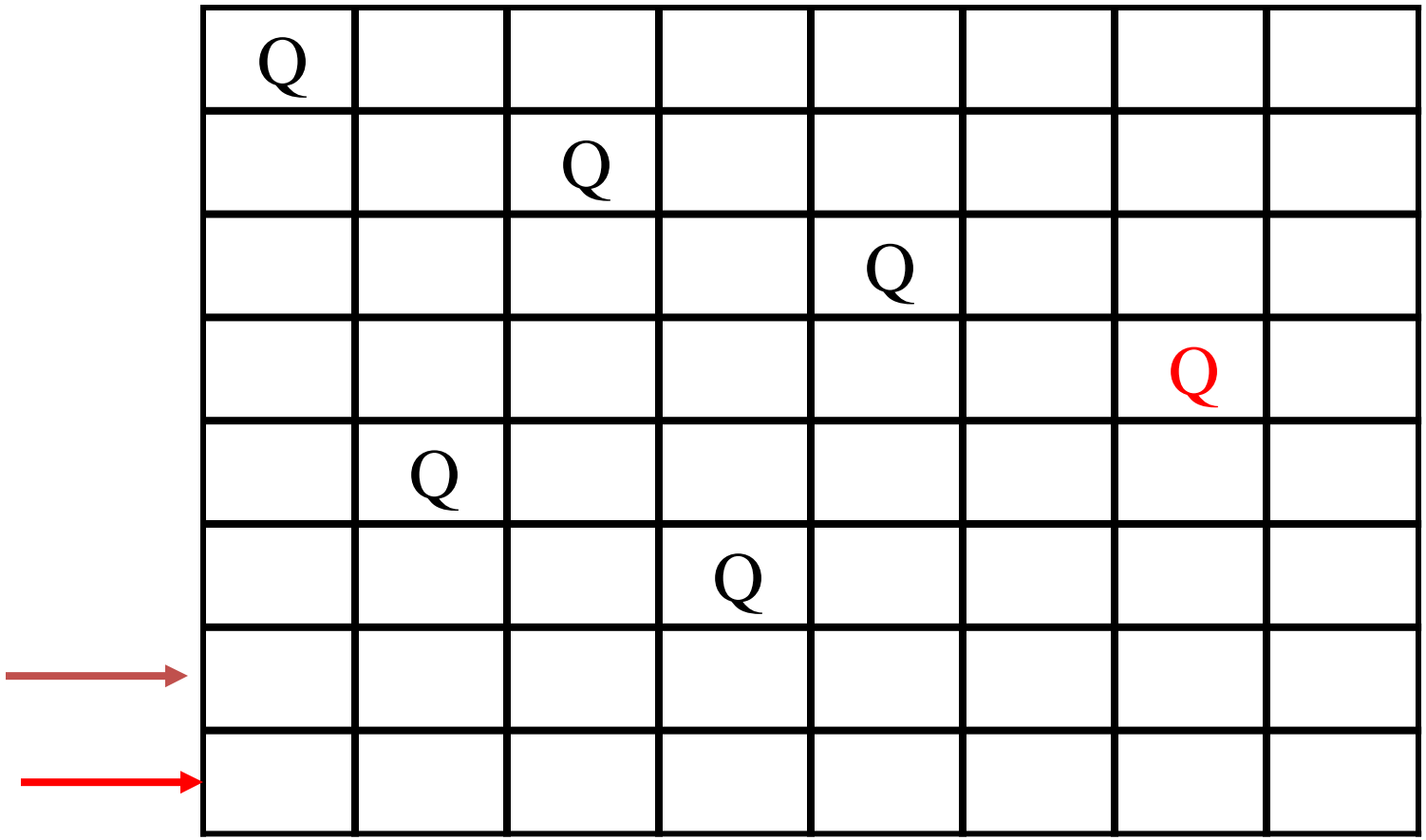
Algoritmo para
O Problema das Oito Rainhas



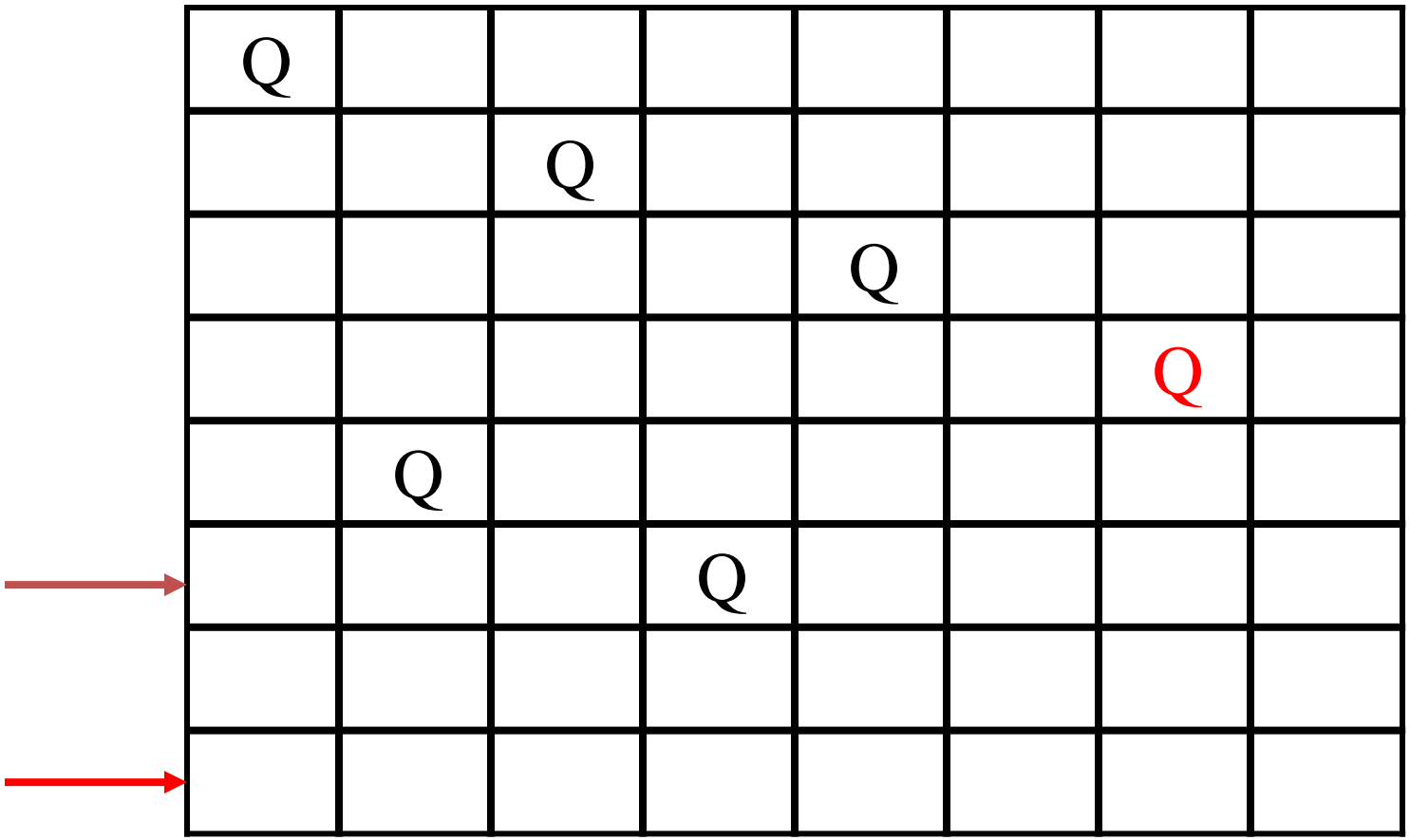
Algoritmo para
O Problema das Oito Rainhas



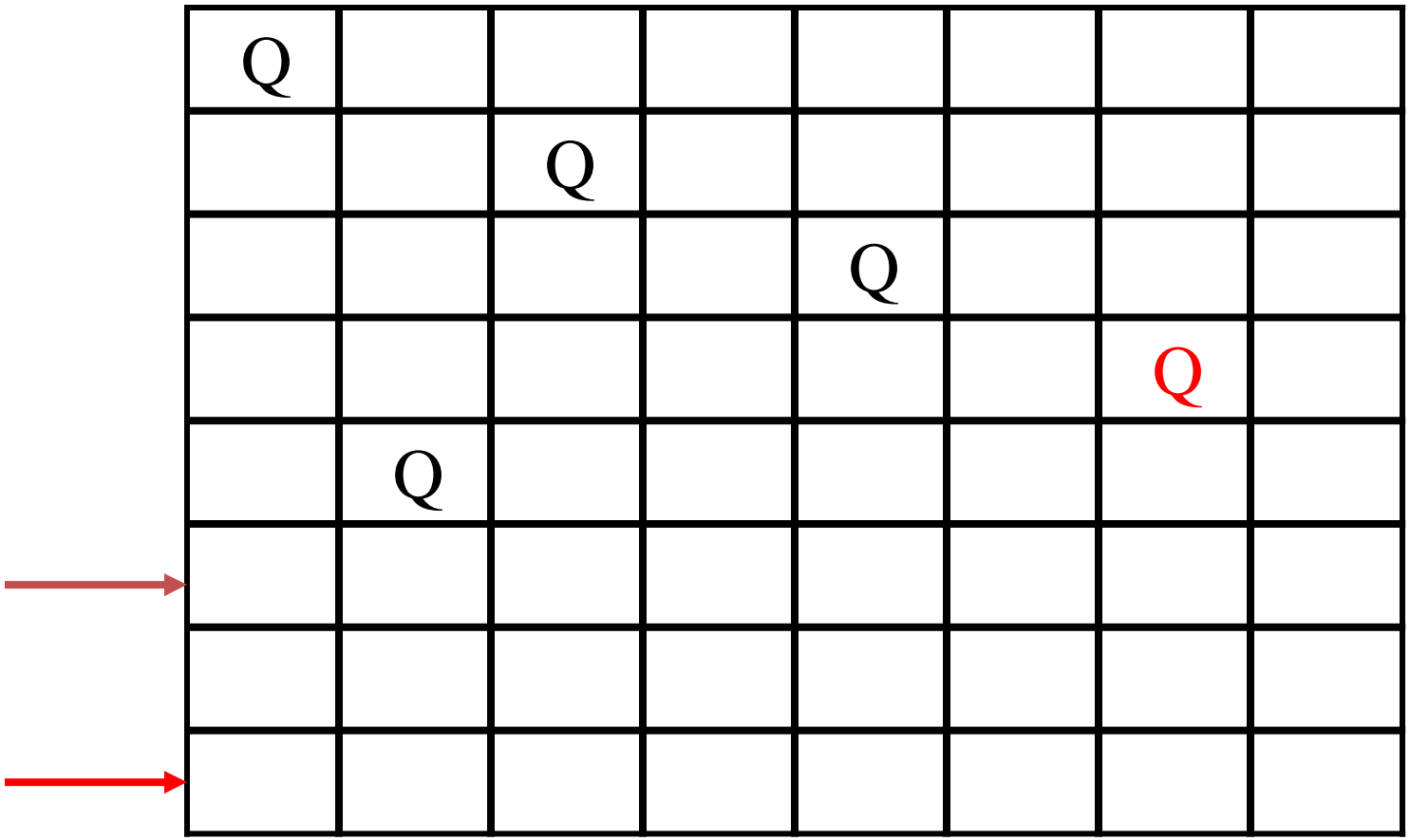
Algoritmo para
O Problema das Oito Rainhas



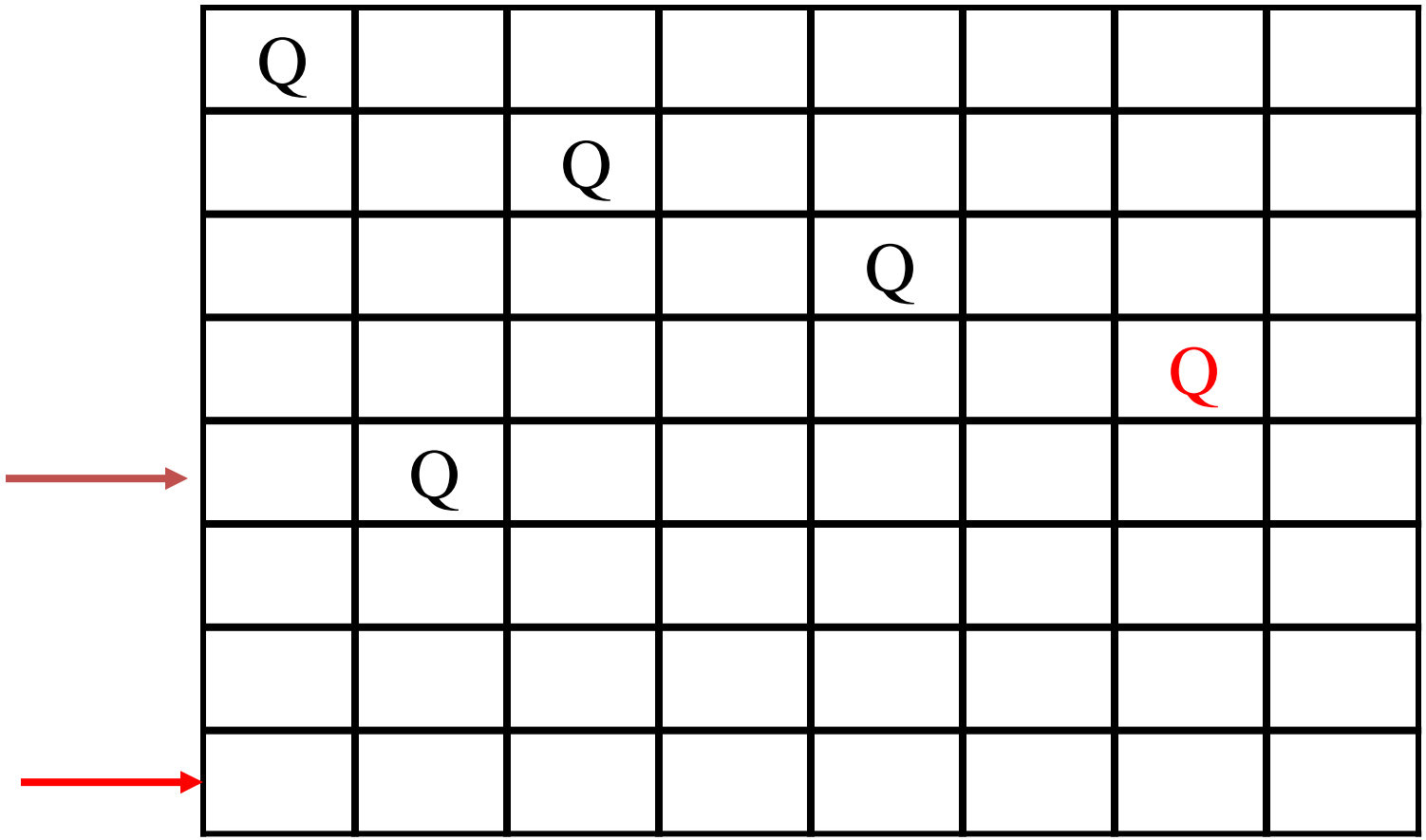
Algoritmo para
O Problema das Oito Rainhas



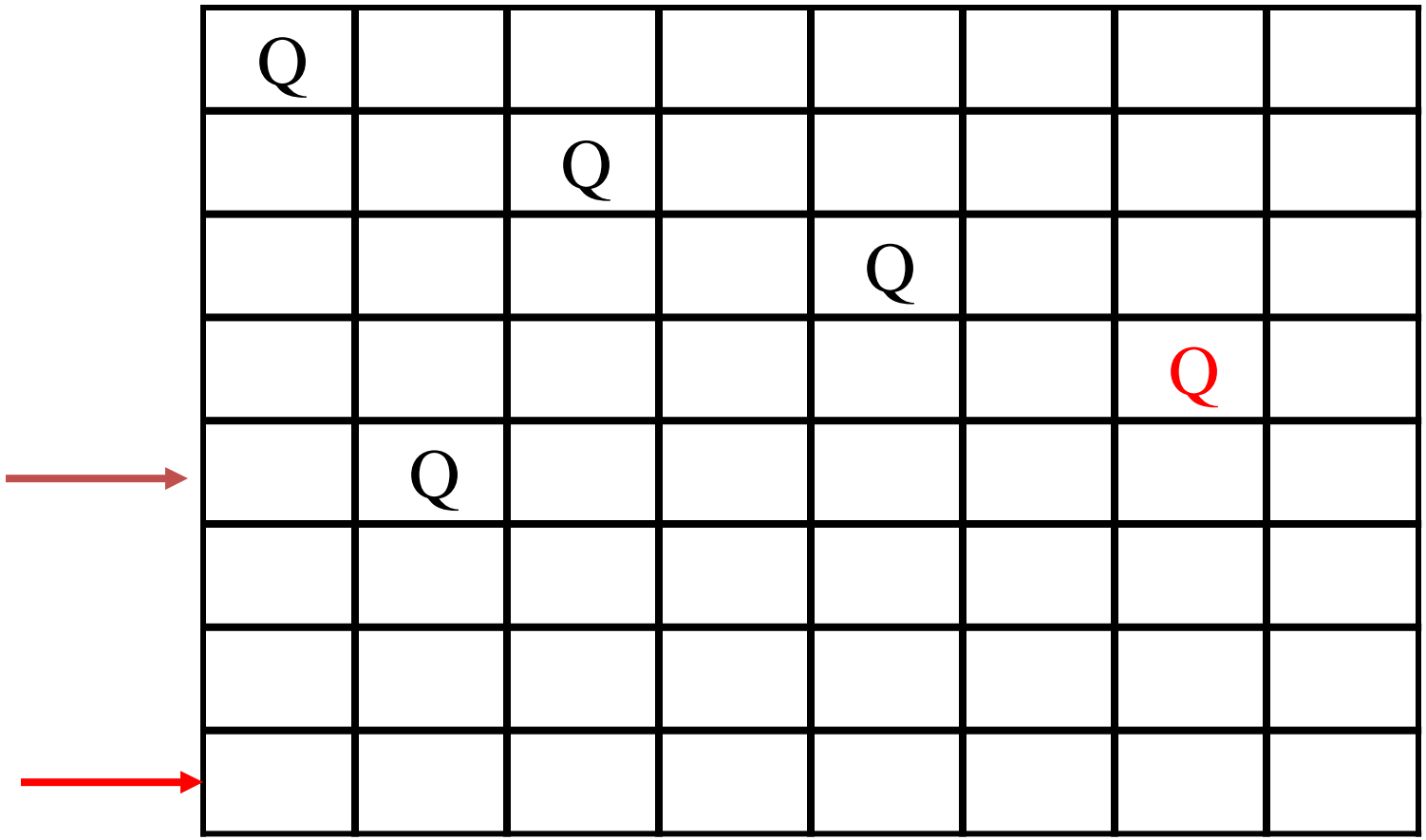
Algoritmo para
O Problema das Oito Rainhas



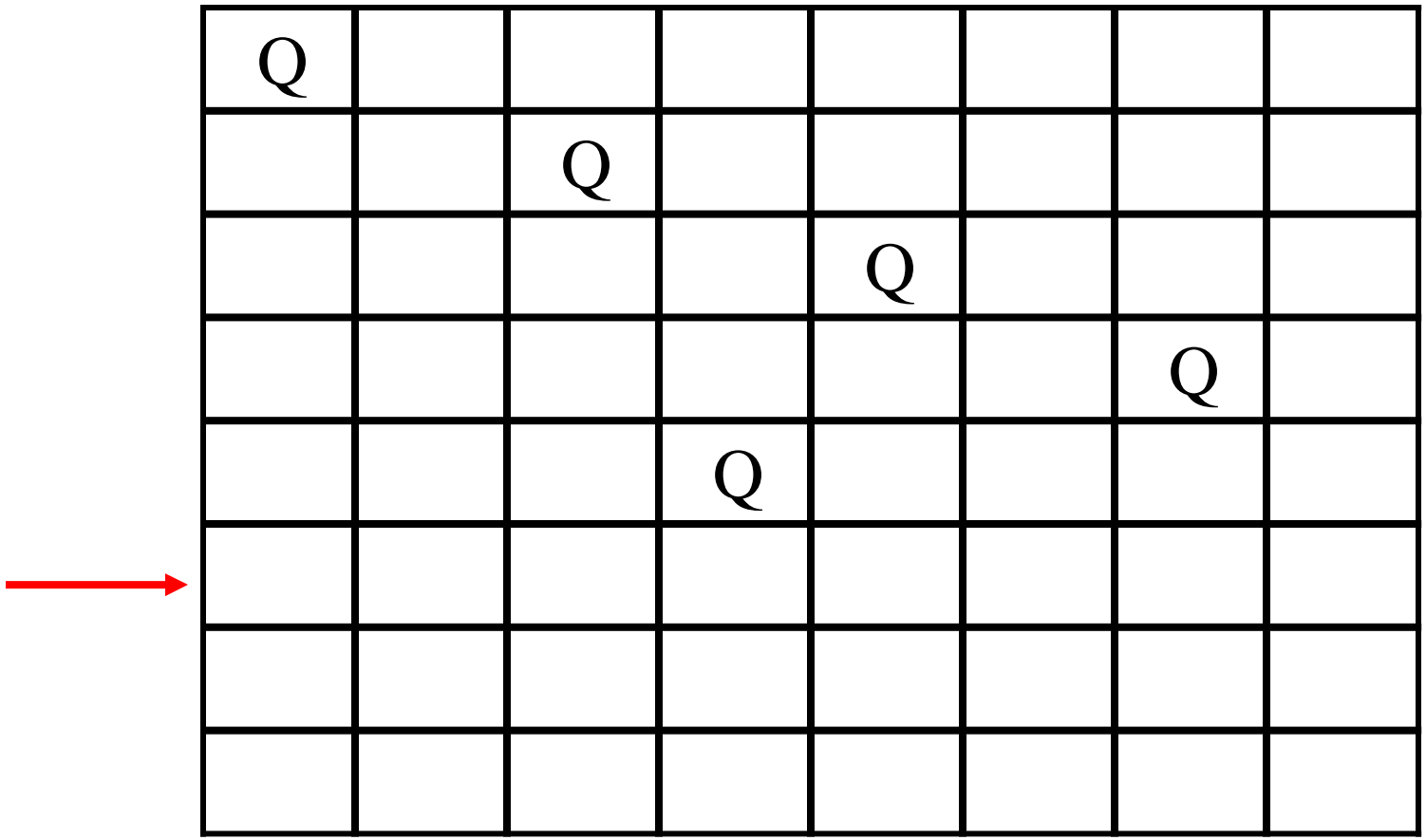
Algoritmo para
O Problema das Oito Rainhas



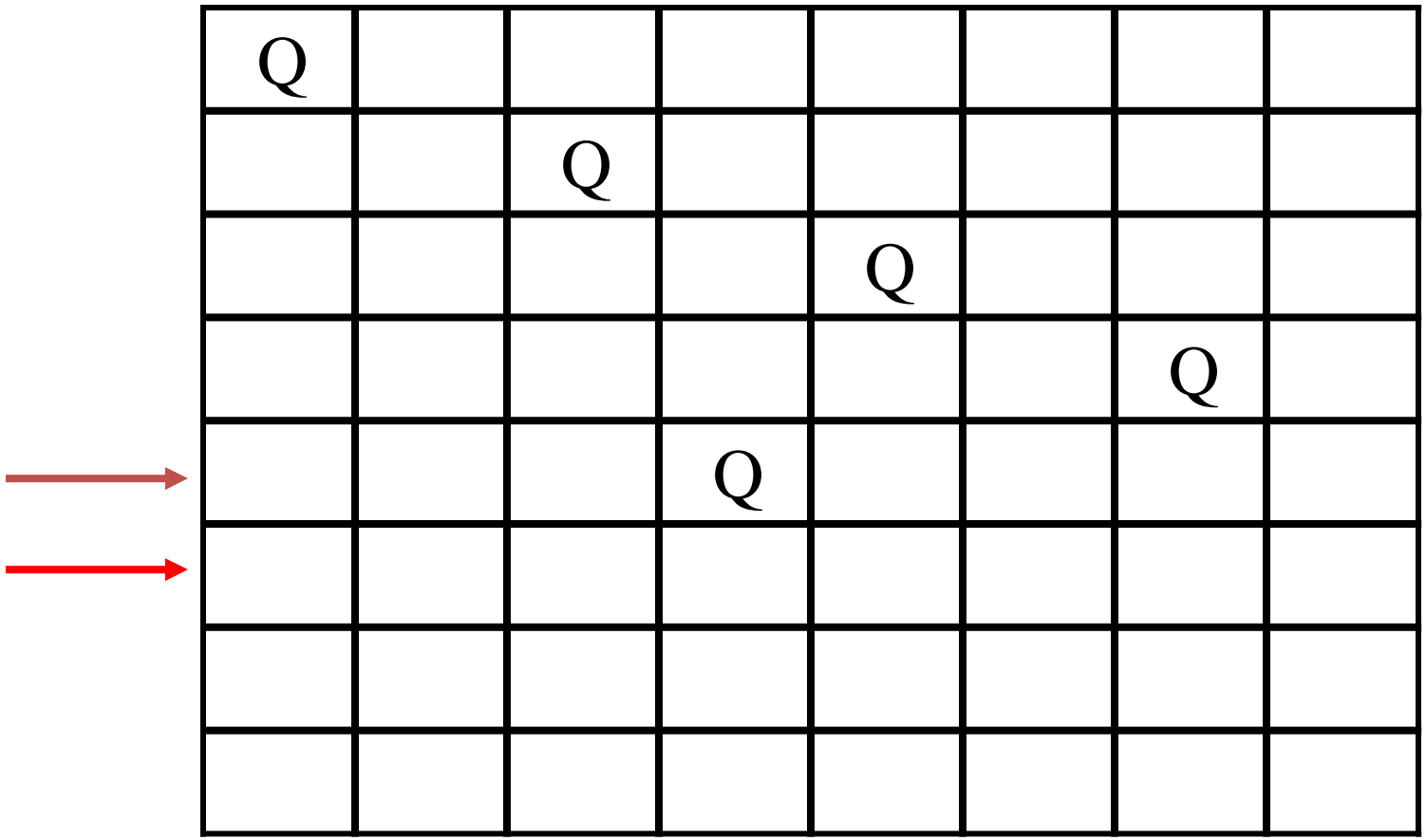
Algoritmo para
O Problema das Oito Rainhas



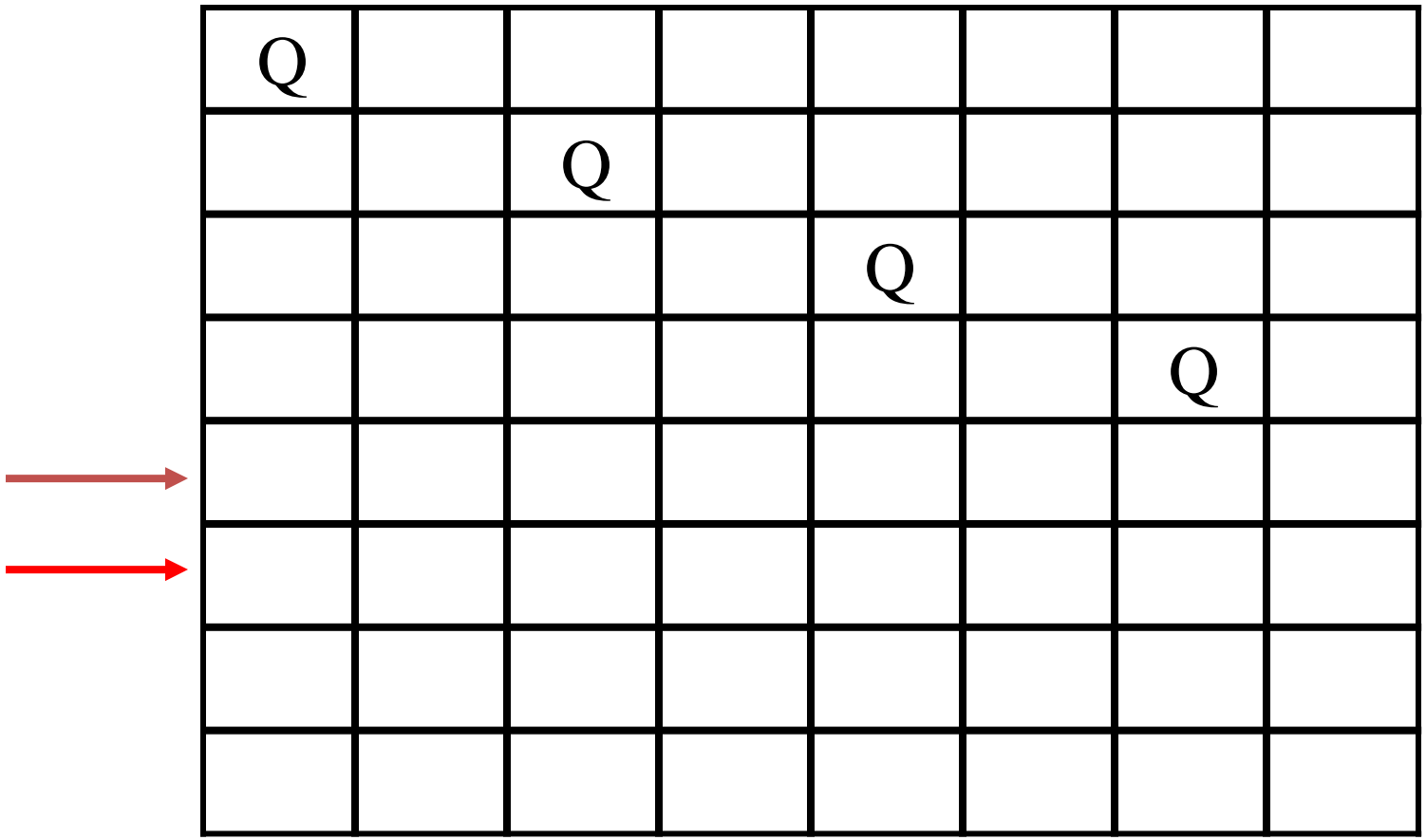
Algoritmo para
O Problema das Oito Rainhas



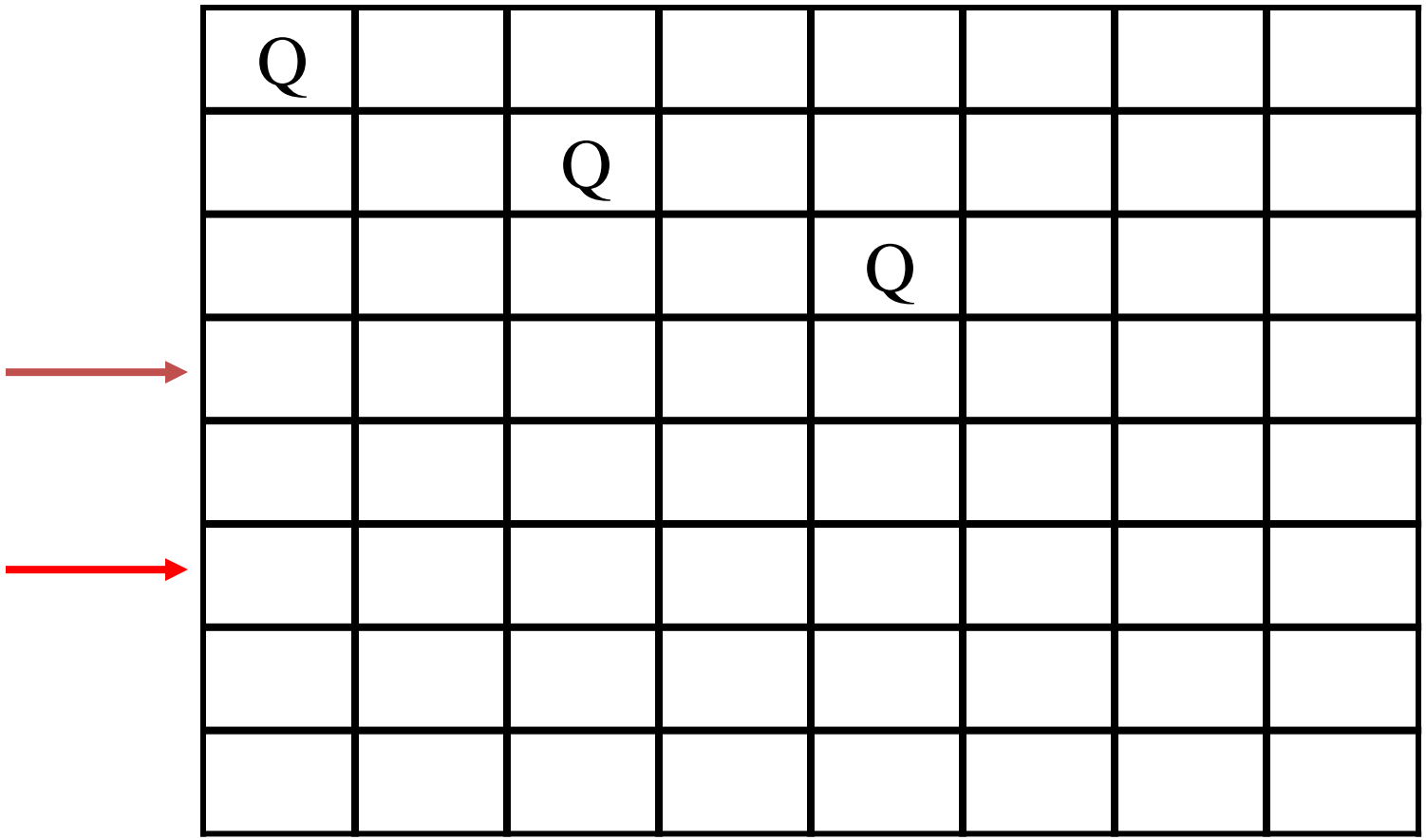
Algoritmo para
O Problema das Oito Rainhas



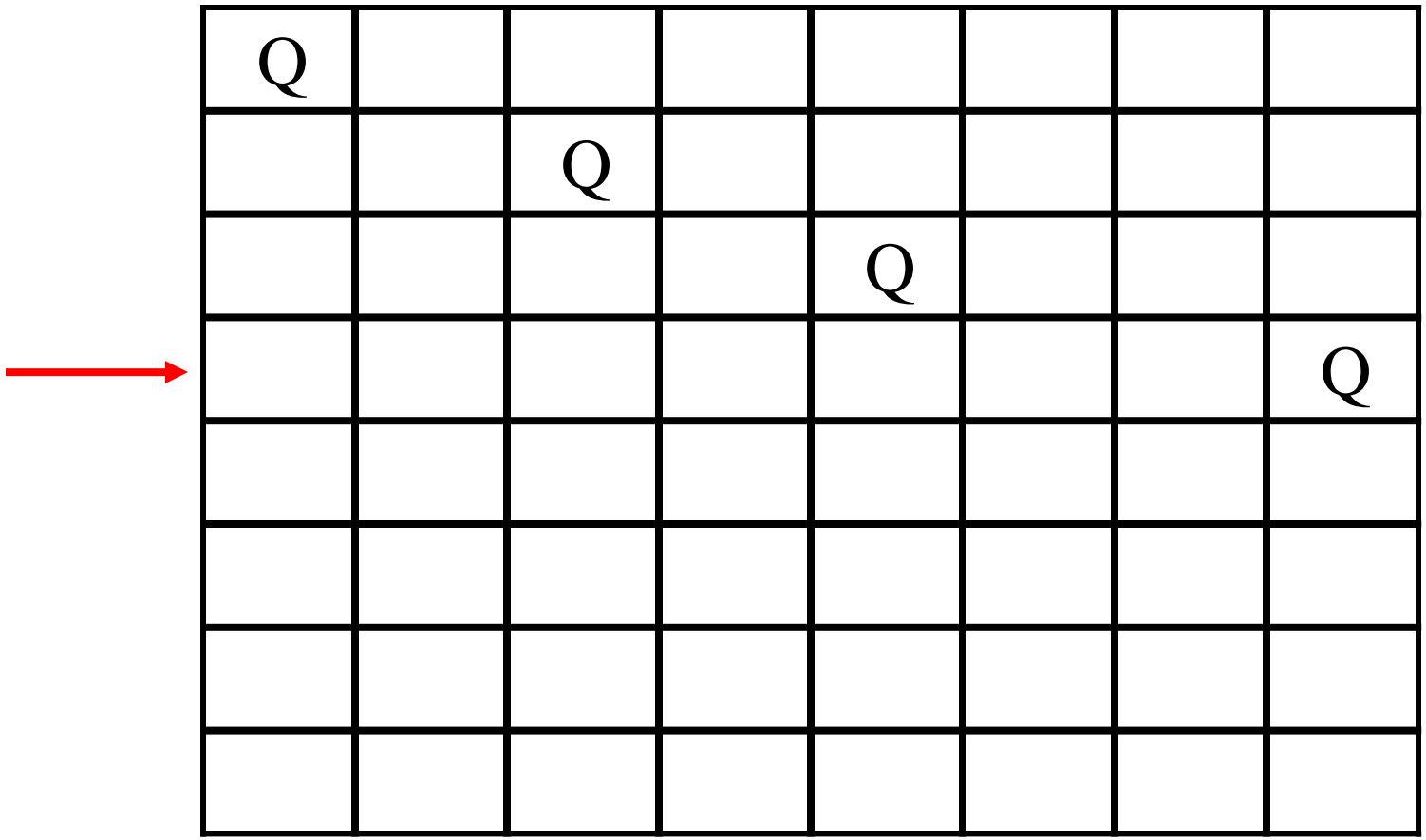
Algoritmo para
O Problema das Oito Rainhas



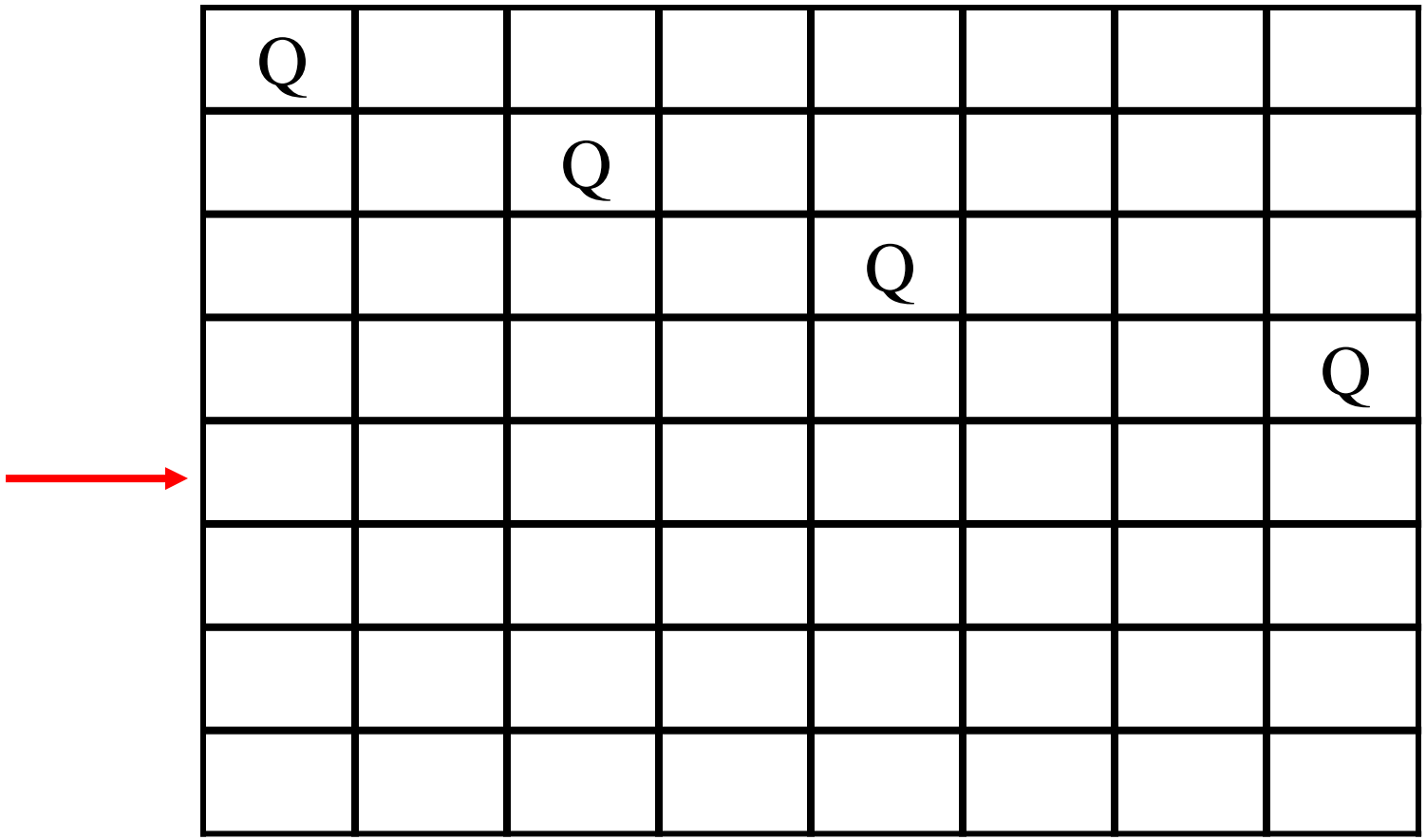
Algoritmo para
O Problema das Oito Rainhas



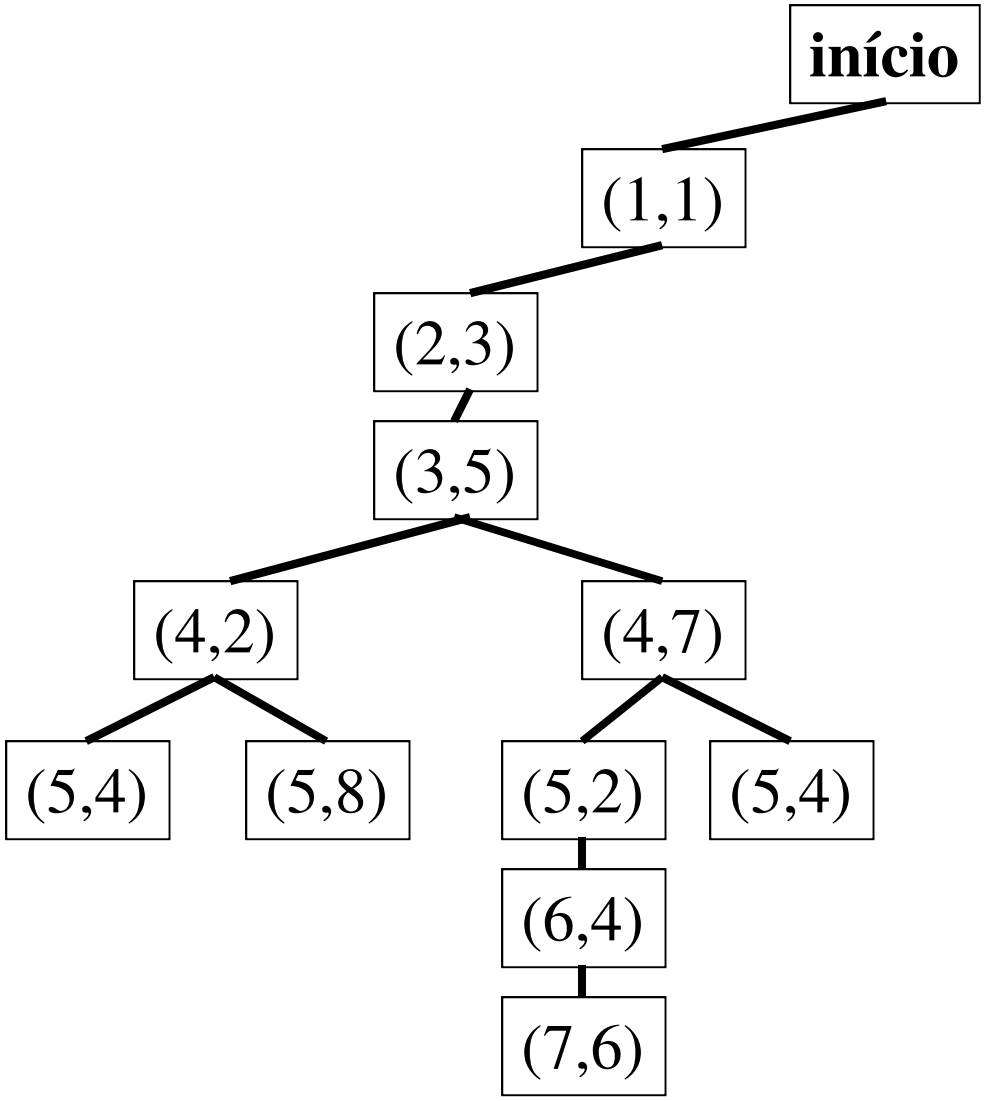
Algoritmo para
O Problema das Oito Rainhas



Algoritmo para
O Problema das Oito Rainhas



Algoritmo para
O Problema das Oito Rainhas



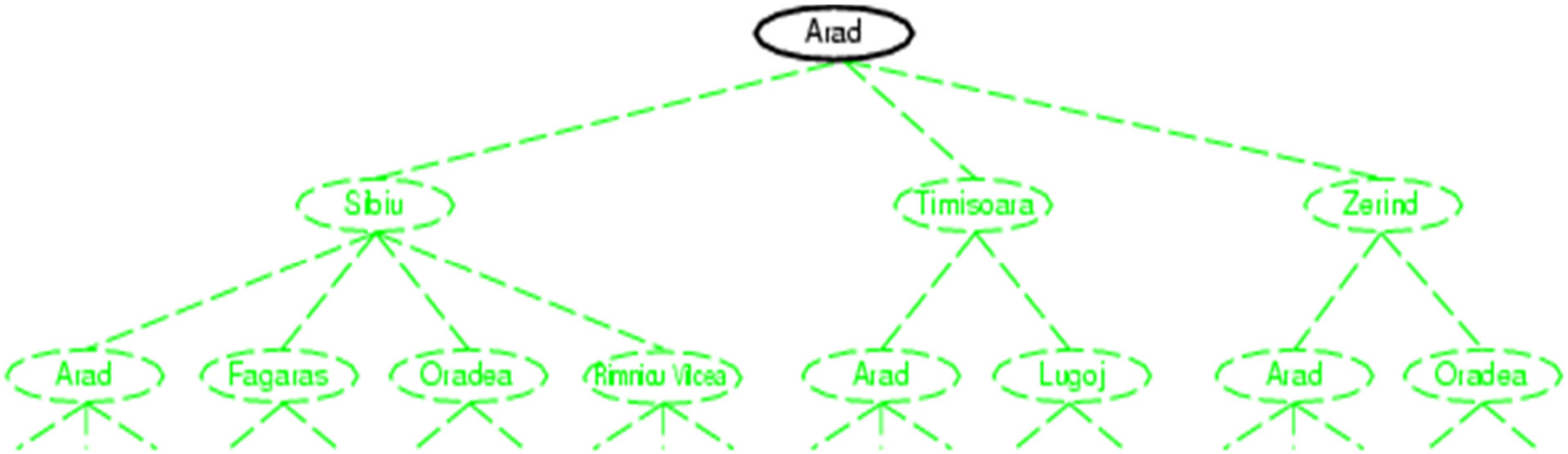
Problemas do mundo real

- Problema de roteamento
  - encontrar a melhor rota de um ponto a outro (aplicações: redes de computadores, planejamento militar, planejamento de viagens aéreas)
- Problemas de tour
  - visitar cada ponto pelo menos uma vez
- Caixeiro viajante
  - visitar cada cidade exatamente uma vez
  - encontrar o caminho mais curto
- Layout de VLSI
  - posicionamento de componentes e conexões em um chip
- Projeto de proteínas
  - encontrar uma sequência de aminoácidos que serão incorporados em uma proteína tridimensional para curar alguma doença.
- Pesquisas na Web
  - é fácil pensar na Web como um grafo de nós conectados por links

Busca de soluções

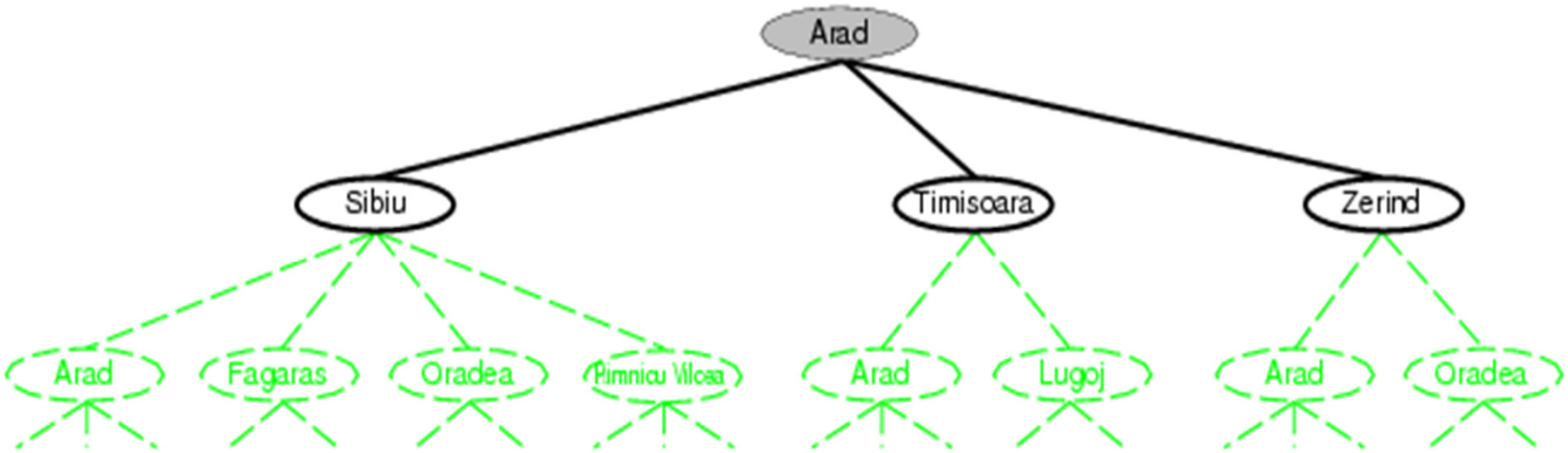
- Idéia: Percorrer o espaço de estados a partir de uma ***árvore de busca***.
- *Expandir* o estado atual aplicando a função sucessor, *gerando* novos estados.
- Busca: seguir um caminho, deixando os outros para depois.
- A *estratégia de busca* determina qual caminho seguir.

Exemplo de árvore de busca



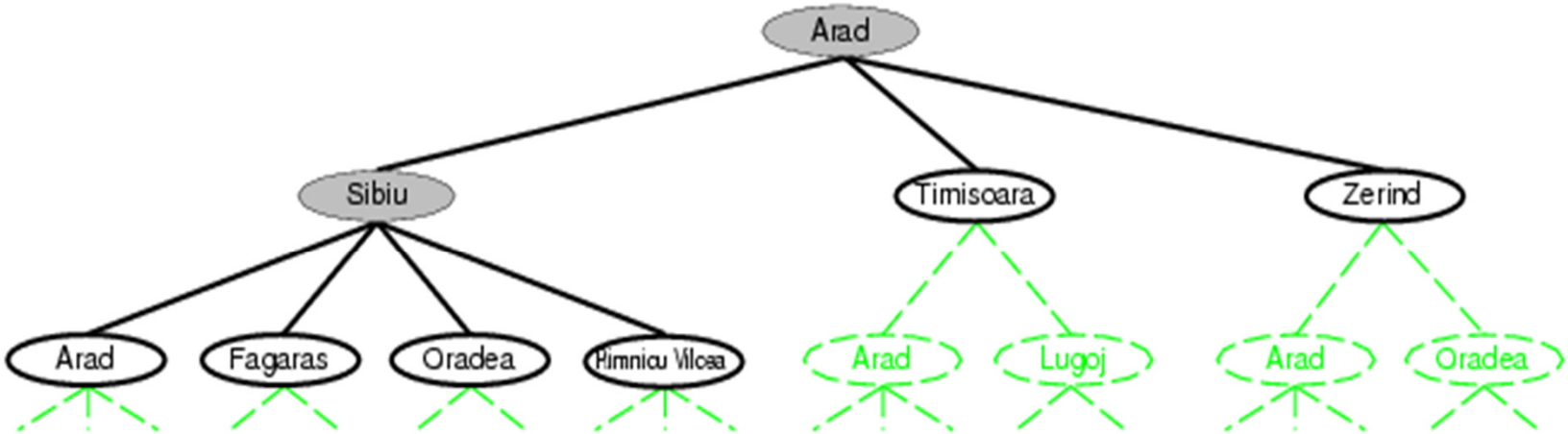
Estado inicial

Exemplo de árvore de busca



Depois de expandir Arad

Exemplo de árvore de busca



Depois de expandir Sibiu

Descrição informal do algoritmo geral de busca em árvore

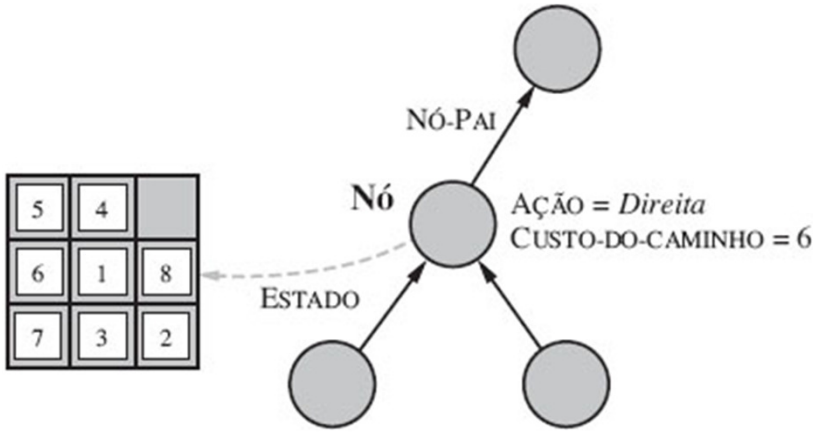
```
função BUSCA-EM-ÁRVORE(problema) retorna uma solução ou falha
    inicializar a borda utilizando o estado inicial do problema
    repita
        se borda vazia então retornar falha
        escolher um nó folha e o remover da borda
        se o nó contém um estado objetivo então retornar a solução correspondente
        expandir o nó escolhido, adicionando os nós resultantes à borda
```

Estados vs. nós

- Um estado é (representação de) uma configuração física
- Um nó é uma estrutura de dados que é parte da árvore de busca
- A função **`Expand`** cria novos nós, preenchendo os vários campos e usando a função sucessor do problema para gerar os estados correspondentes.
- A coleção de nós que foram gerados, mas ainda não foram expandidos é chamada de borda (ou *fringe* ou lista lista aberta)
- Para evitar caminhos redundantes, deve-se estabelecer um conjunto explorado (ou lista fechada)

Infraestrutura para os algoritmos de busca

- Representação de um nó *n* da árvore
  - *n*.ESTADO: estado no espaço de estado
  - *n*.PAI: o nó na árvore de busca que gerou este
  - *n*.AÇÃO: a ação que foi aplicada ao pai para gerar o nó
  - *n*.CUSTO-DO-CAMINHO: o custo, denotado por *g*(*n*), do caminho do estado inicial até o nó, indicado pelos ponteiros para os pais



Controle de borda

- **Lista** como estrutura de dados adequada
  - VAZIA?(*lista*) – devolve *true* se não existir elementos
  - POP(*lista*) – remove e devolve primeiro
  - INSERIR(*elemento*, *lista*) – insere um elemento
- Variantes quanto à ordem
  - FIFO (*first in, first out*) ou fila
  - LIFO (*last in, first out*) ou pilha
  - Fila de prioridade
- Tabela hash pode ser usada para controle de estados repetidos

Estratégias de busca

- Uma estratégia de busca é definida pela escolha da ordem da expansão de nós

- Estratégias são avaliadas de acordo com os seguintes critérios:
  - completeza: o algoritmo sempre encontra a solução se ela existe?
  - complexidade de tempo: número de nós gerados
  - complexidade de espaço: número máximo de nós na memória
  - otimização: a estratégia encontra a solução ótima?

- Complexidade de tempo e espaço são medidas em termos de:
  - ***b:*** máximo fator de ramificação da árvore (número máximo de sucessores de qualquer nó)
  - ***d:*** profundidade do nó objetivo menos profundo
  - ***m*****:** o comprimento máximo de qualquer caminho no espaço de estados (pode ser ∞)

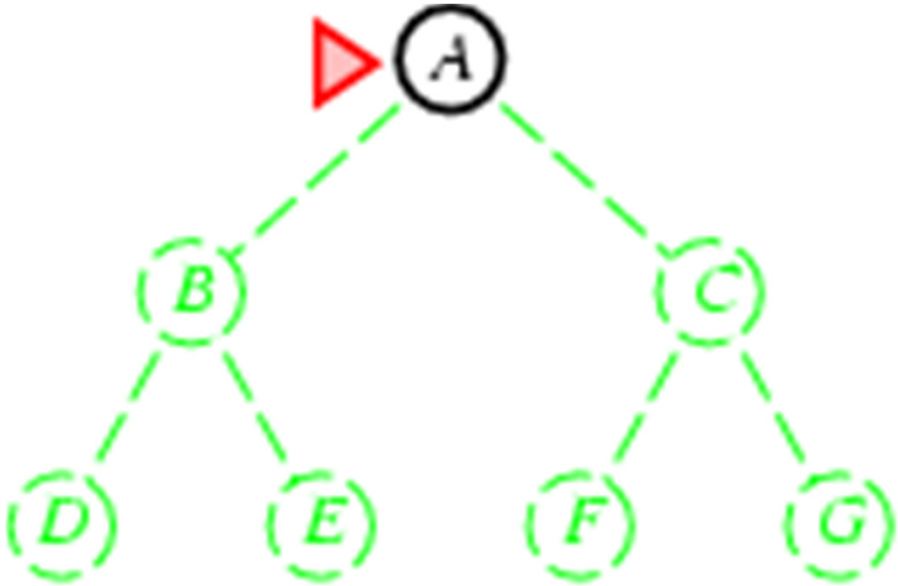
Estratégias de Busca Sem Informação (ou Busca Cega)

- Estratégias de busca sem informação usam apenas a informação disponível na definição do problema.
  - Apenas geram sucessores e verificam se o estado objetivo foi atingido.

- As estratégias de busca sem informação se distinguem pela ordem em que os nós são expandidos.
  - Busca em extensão ou em largura (*Breadth-first*)
  - Busca de custo uniforme
  - Busca em profundidade (*Depth-first*)
  - Busca em profundidade limitada
  - Busca de aprofundamento iterativo

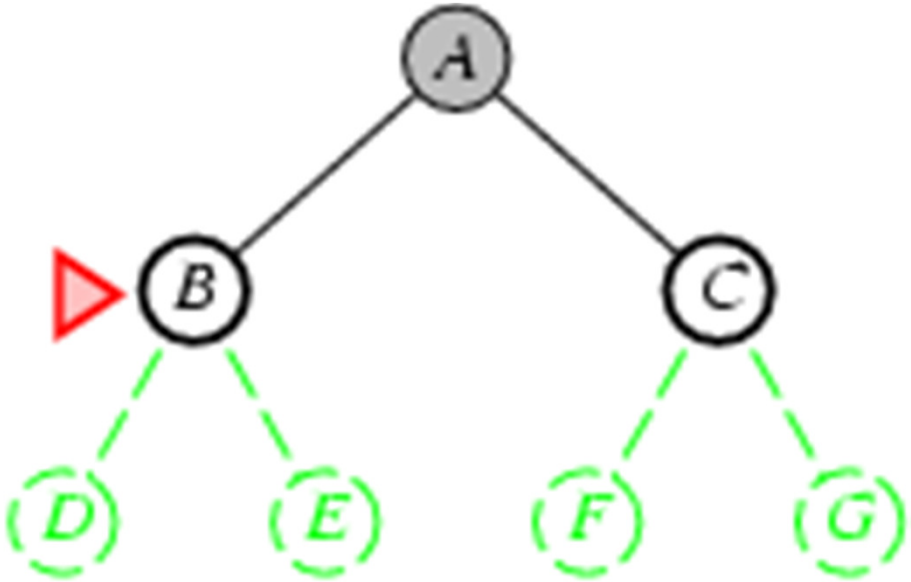
Busca em extensão (largura)

- Expandir o nó não-expandido mais perto da raiz.
- Implementação:
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.



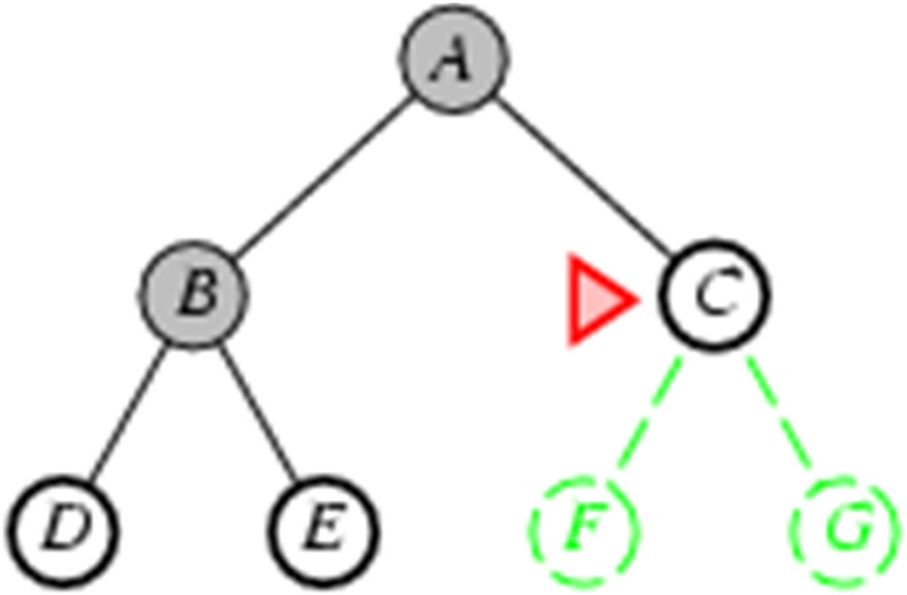
Busca em extensão (largura)

- Expandir o nó não-expandido mais perto da raiz.
- Implementação:
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.



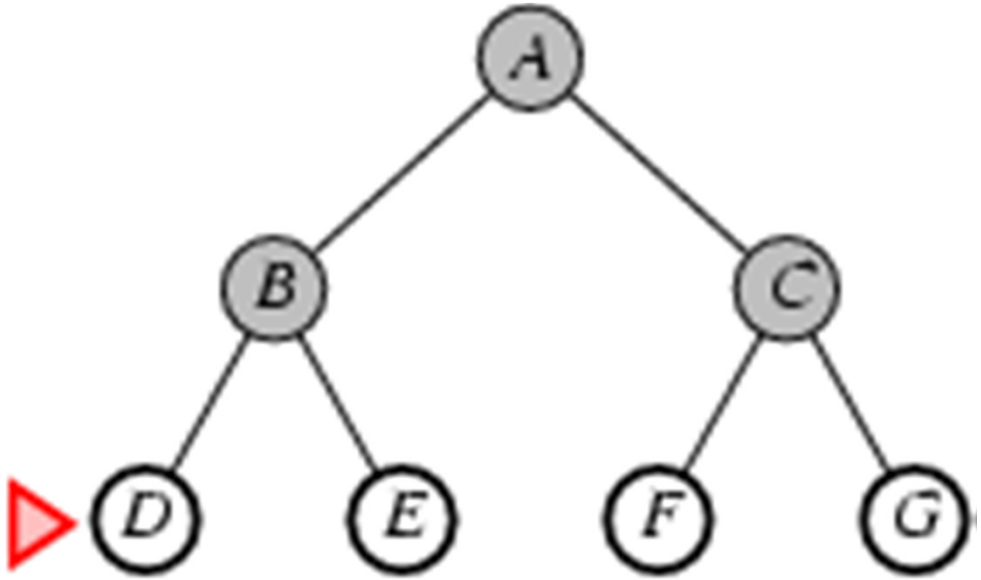
Busca em extensão (largura)

- Expandir o nó não-expandido mais perto da raiz.
- Implementação:
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.



Busca em extensão (largura)

- Expandir o nó não-expandido mais perto da raiz.
- Implementação:
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.



Propriedades da busca em extensão (largura)

- Completa? Sim (se b é finito)
- Tempo? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^d)$
- Espaço? $O(b^d)$ (mantém todos os nós na memória)
- Ótima? Sim (se todas as ações tiverem o mesmo custo)

Requisitos de Tempo e Memória para a Busca em Extensão (Largura)

- Busca com fator de ramificação b=10.
- Supondo que 10.000 nós possam ser gerados por segundo e que um nó exige 1KB de espaço.

Profundidade	Nós	Tempo	Memória
2	1100	0.11 segundo	1 megabyte
4	111.100	11 segundos	106 megabytes
6	10^7	19 minutos	10 gigabytes
8	10^9	31 horas	1 terabyte
10	10^{11}	129 dias	101 terabytes
12	10^{13}	35 anos	10 petabytes
14	10^{15}	3.523 anos	1 exabyte

Requisitos de Tempo e Memória para a Busca em Extensão (Largura)

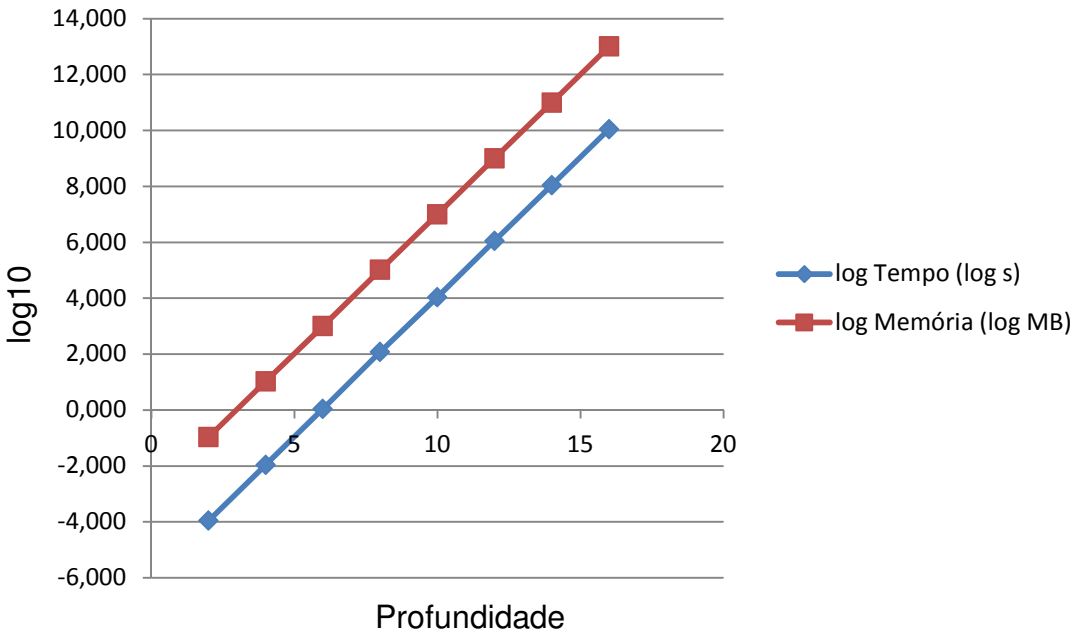
- Busca com fator de ramificação b=10.
- Supondo que 10.000 nós possam ser gerados por segundo e que um nó exige 1KB de espaço.

Profundidade	Nós	Tempo	Memória
2	110	0,11 milisegundo	107 kilobytes
4	11.110	11 milisegundos	10,6 megabytes
6	10^6	1,1 segundo	1 gigabyte
8	10^8	2 minutos	103 gigabytes
10	10^{10}	3 horas	10 terabytes
12	10^{12}	13 dias	1 petabytes
14	10^{14}	3,5 anos	99 petabytes
16	10^{15}	350 anos	10 exabytes

Requisitos de Tempo e Memória para a Busca em Extensão (Largura)

- Busca com fator de ramificação b=10.
- Supondo que 10.000 nós possam ser gerados por segundo e que um nó exige 1KB de espaço.

Profundidade	log Tempo (log s)	log Memória (log MB)
2	-3,959	-0,971
4	-1,959	1,025
6	0,041	3,000
8	2,079	5,013
10	4,033	7,000
12	6,050	9,000
14	8,043	10,996
16	10,043	13,000



Busca de custo uniforme

- Expande o nó não-expandido que tenha o caminho de custo mais baixo.

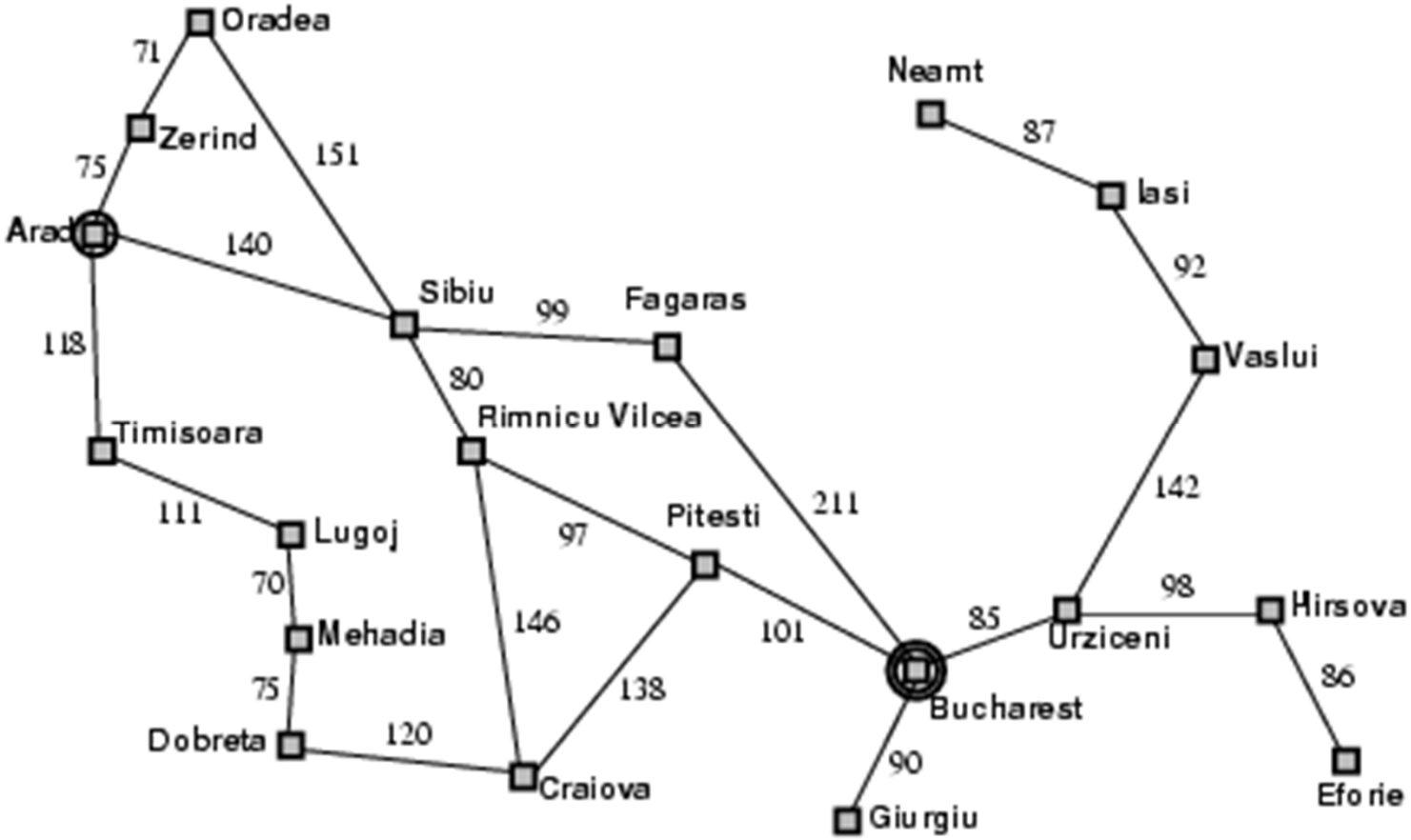
- Implementação:
  - *borda* = fila ordenada pelo custo do caminho

- Equivalente a busca em extensão (em largura) se os custos são todos iguais

- Completa? Sim, se o custo de cada passo ≥ ε
- Tempo? # de nós com $g \leq$ custo da solução ótima, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$ onde C^* é o custo da solução ótima
- Espaço? de nós com $g \leq$ custo da solução ótima, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
- Ótima? Sim pois os nós são expandidos em ordem crescente de custo total.

Exercício

- Aplicar busca de custo uniforme para achar o caminho mais curto entre Arad e Bucareste.

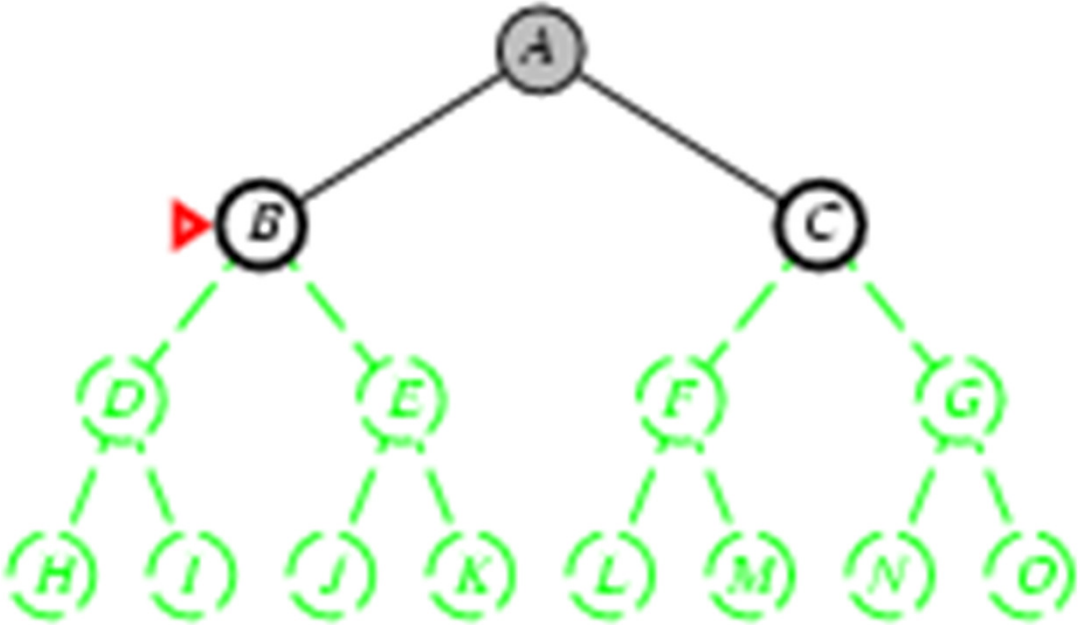


Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha

Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha

Busca em Profundidade

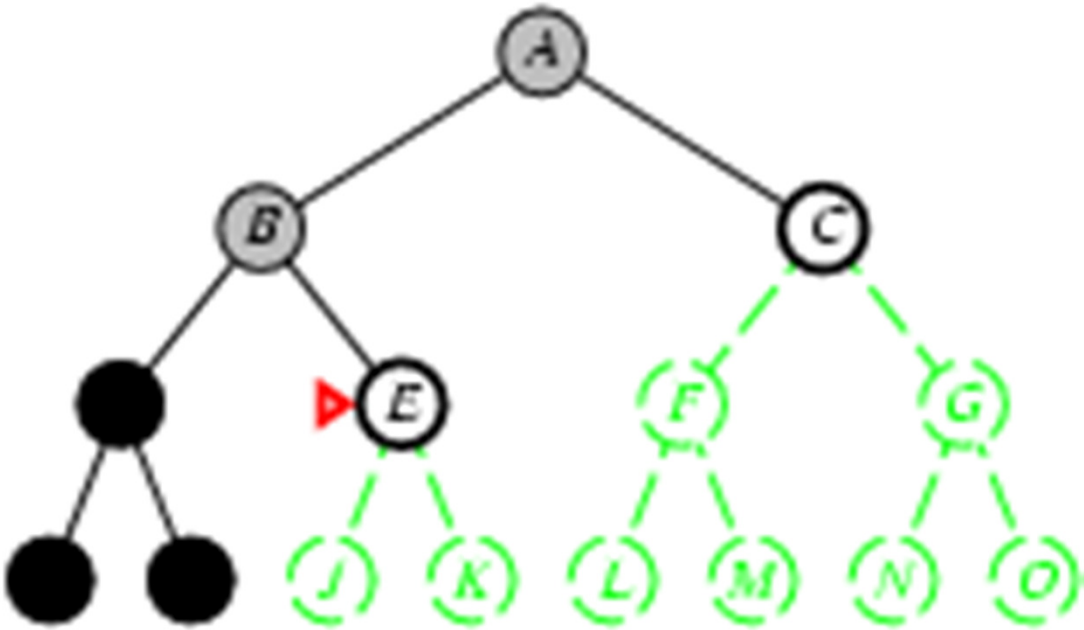
- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha

Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha

Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha

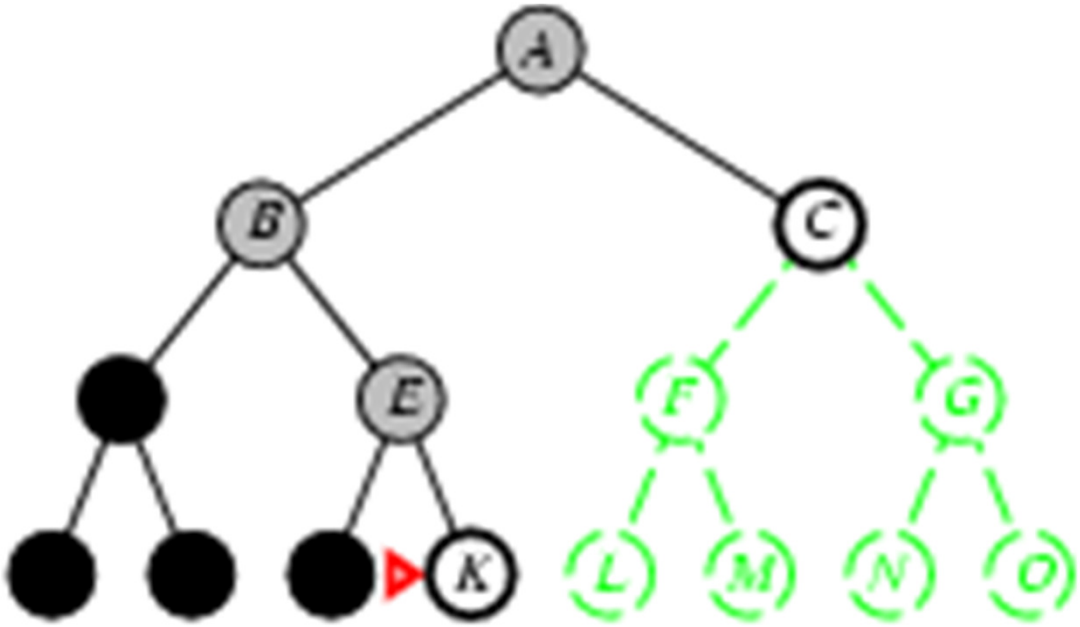


Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha

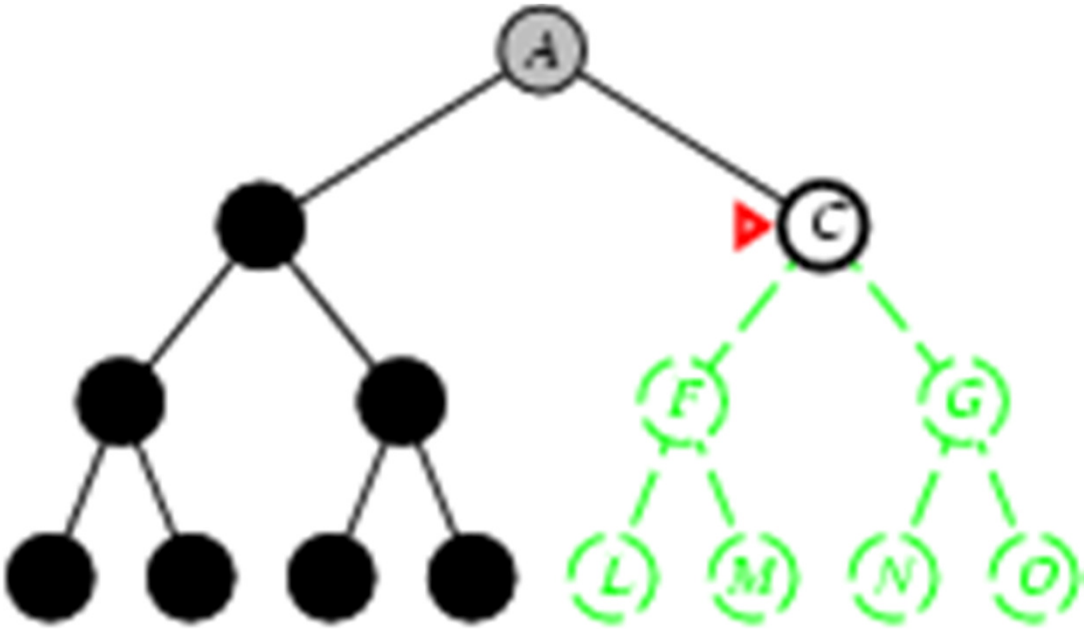
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



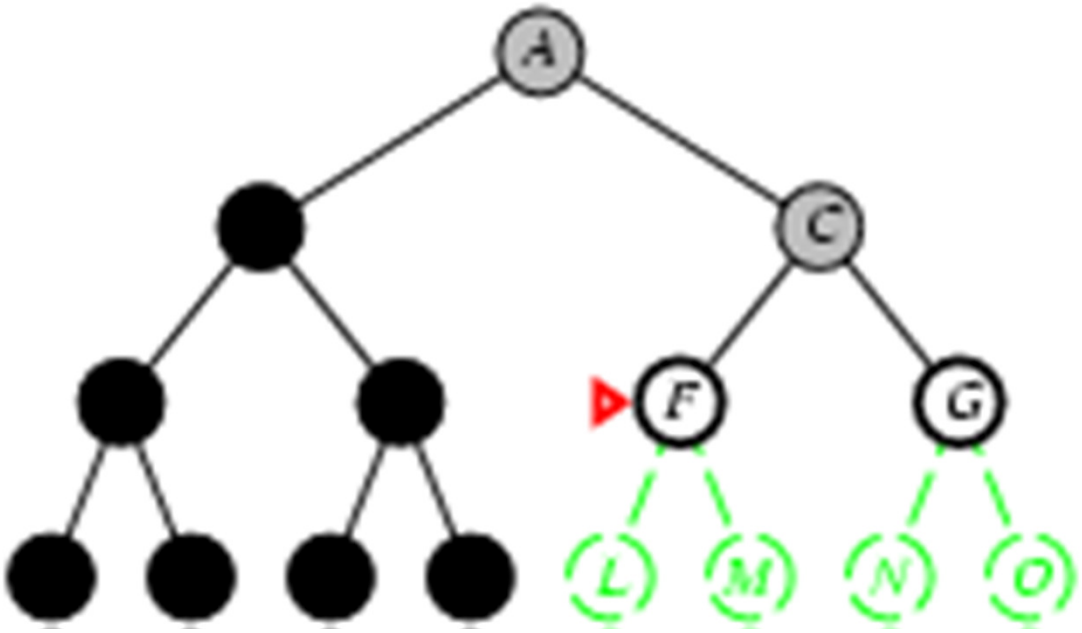
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



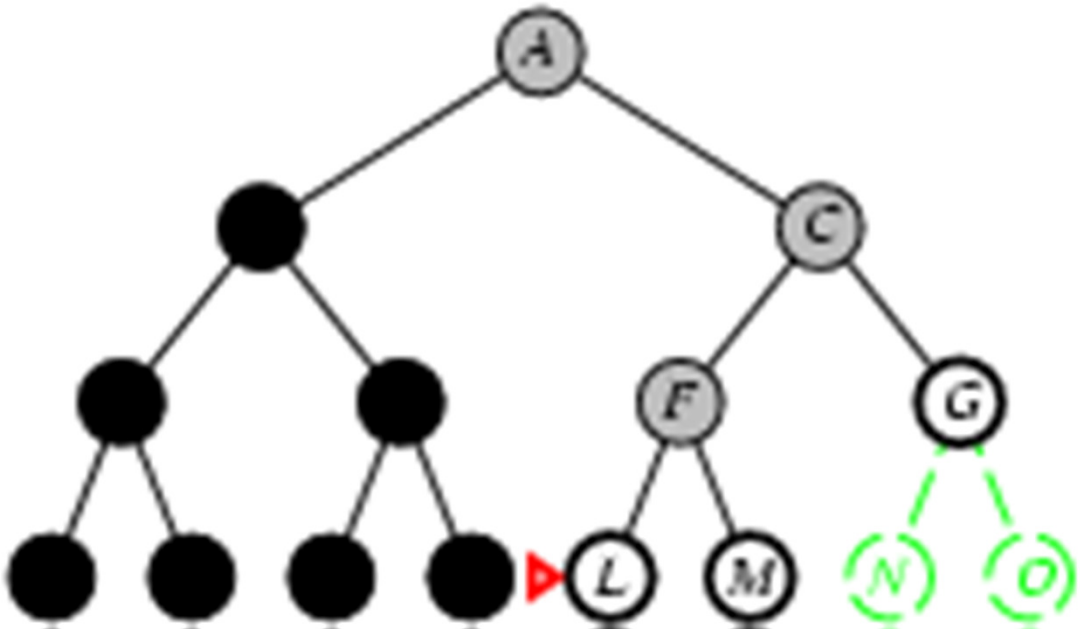
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



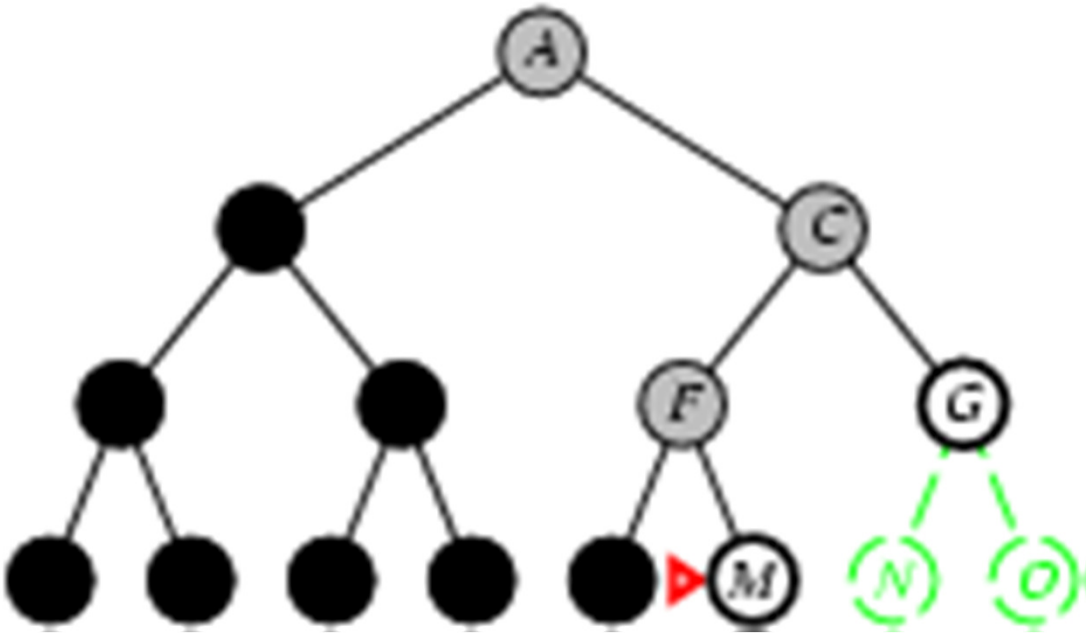
Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



Busca em Profundidade

- Expande o nó não-expandido mais profundo.
- Implementação:
  - *borda* = fila LIFO (*last-in, first-out*) = pilha



Propriedades da Busca em Profundidade

- Completa? Não: falha em espaços com profundidade infinita, espaços com loops
  - Se modificada para evitar estados repetidos é completa para espaços finitos
- Tempo? $O(b^m)$: péssimo quando m é muito maior que d.
  - mas se há muitas soluções pode ser mais eficiente que a busca em extensão (em largura)
- Espaço? $O(b^m)$
- Ótima? Não

Busca em Profundidade Limitada

= busca em profundidade com limite de profundidade *l*,

isto é, nós com profundidade *l* não tem sucessores

- Implementação Recursiva:

```
função BUSCA-EM-PROFUNDIDADE-LIMITADA(problema, limite) retorna uma solução ou falha/corte
retornar BPL-RECURSIVA (CRIAR-NÓ(problema, ESTADO-INICIAL), problema, limite)
função BPL-RECURSIVA(nó, problema, limite) retorna uma solução ou falha/corte
  se problema. TESTAR-OBJETIVO (nó.ESTADO) então, retorna SOLUÇÃO (nó)
  se não se limite = 0 então retorna corte
  senão
    corte_ocorreu? ← falso para cada ação no problema.AÇÕES(nó.ESTADO) faça
      filho ← NÓ-FILHO (problema, nó, ação)
      resultado ← BPL-RECURSIVA (criança, problema limite − 1)
      se resultado = corte  então corte_ocorreu? ← verdadeiro
      senão se resultado ≠ falha então retorna resultado
    se corte_ocorreu? então retorna corte senão retorna falha
```

Propriedades da Busca em Profundidade Limitada

- Completa? Não; a solução pode estar além do limite.
- Tempo? $O(b^l)$
- Espaço? $O(bl)$
- Ótima? Não

Busca de Aprofundamento Iterativo em Profundidade

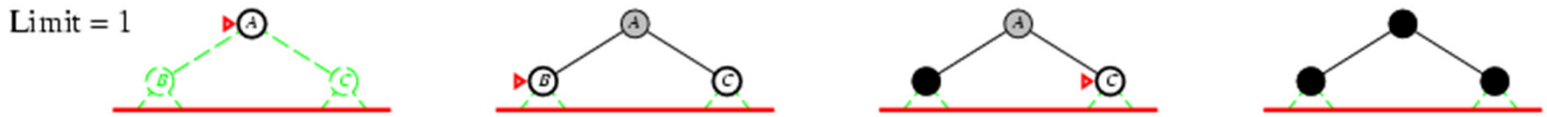
função BUSCA-DE-APROFUNDAMENTO-ITERATIVO(*problema*) **retorna** uma solução ou falha

    para profundidade ← 0 **até** ∞ **faça**
        resultado ← BUSCA-EM-PROFUNDIDADE-LIMITADA(*problema*, *profundidade*)
        se *resultado* ≠ *corte* **então retornar** *resultado*

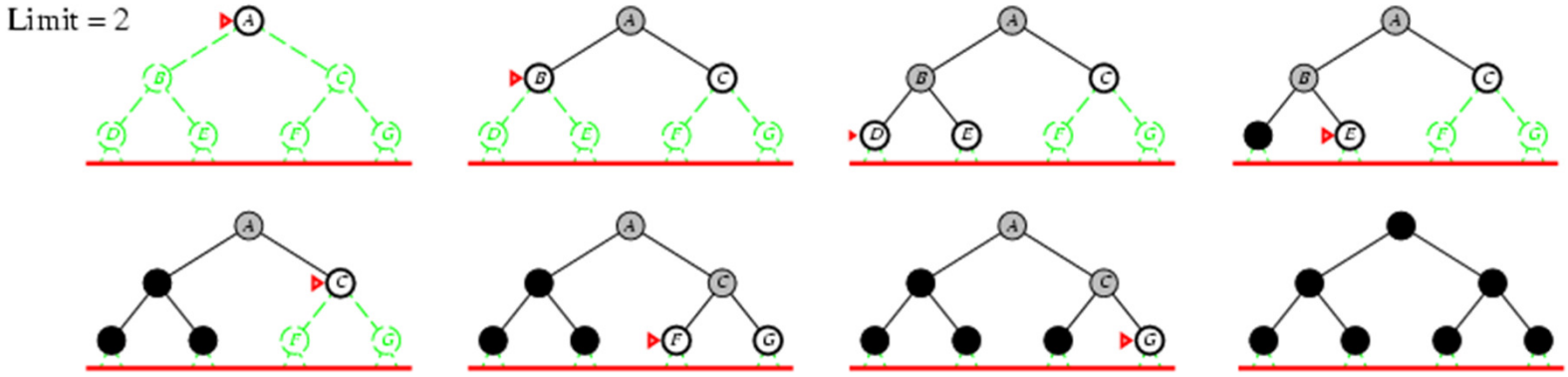
Busca de Aprofundamento Iterativo em Profundidade / =0

Limit = 0

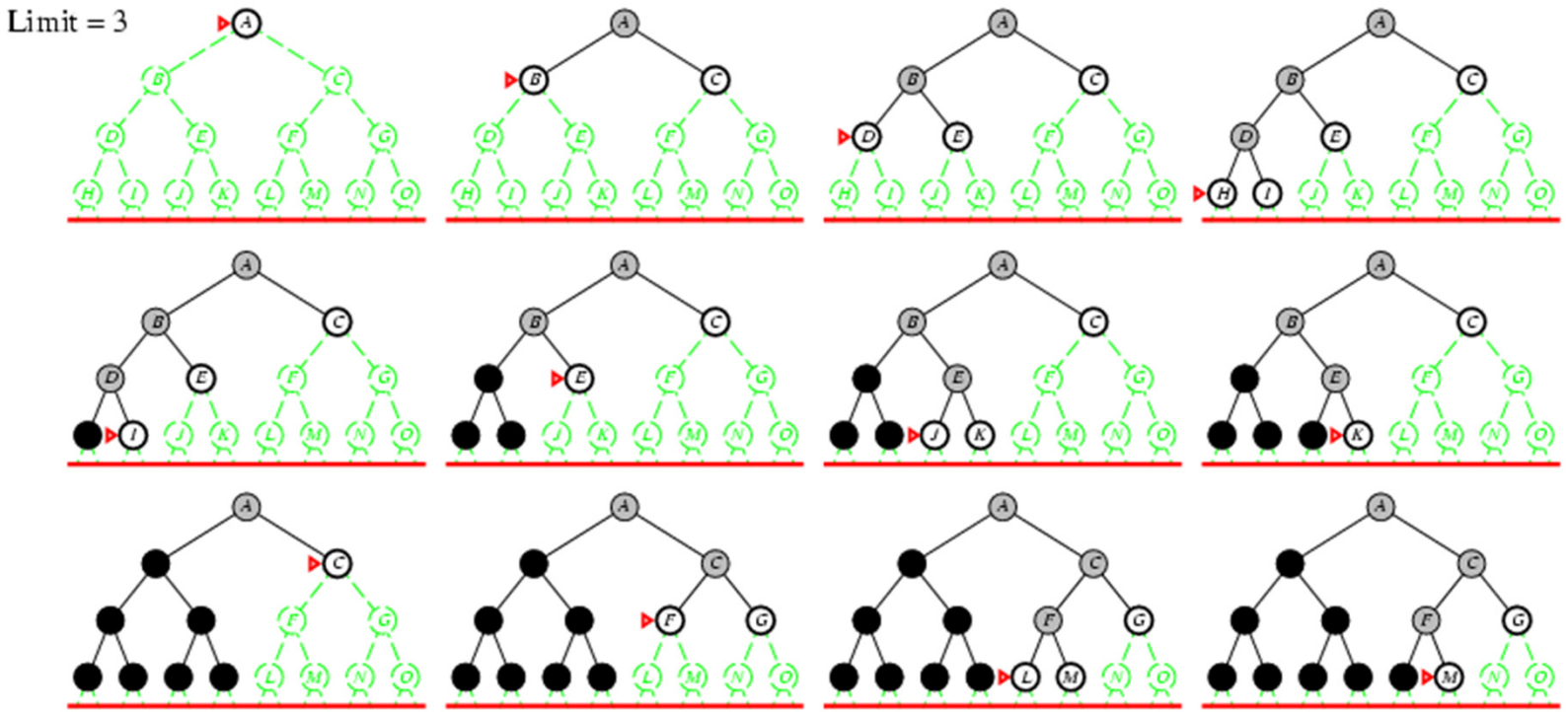
Busca de Aprofundamento Iterativo em Profundidade / =1



Busca de Aprofundamento Iterativo em Profundidade *l* =2



Busca de Aprofundamento Iterativo em Profundidade *l* =3



Propriedades da busca de aprofundamento iterativo

- Completa? Sim
- Tempo? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Espaço? $O(b^d)$
- Ótima? Sim, se custo dos passos são todos idêntidos

Resumo dos algoritmos

Critério	Em largura	Custo uniforme	Em profundidade	Em profundidade limitada	Aprofundamento Iterativo
Completa?	Sim[a]	Sim[a,b]	Não	Não	Sim[a]
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Espaço	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Ótima?	Sim[c]	Sim	Não	Não	Sim[c]

Notações da avaliação de estratégias de busca em árvore:

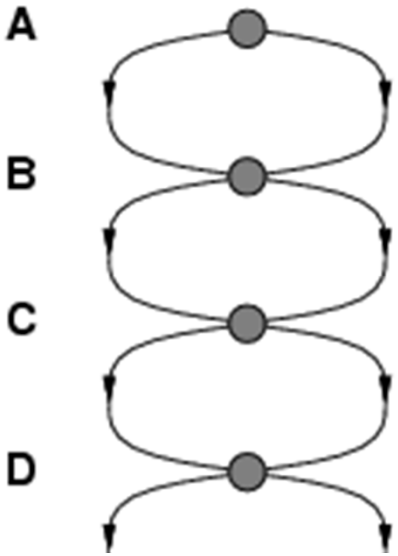
a: completa se *b* é finito
b: completa se custo do passo é ≥ ϵ para ϵ positivo
c: ótima se os custos dos passos são todos idênticos

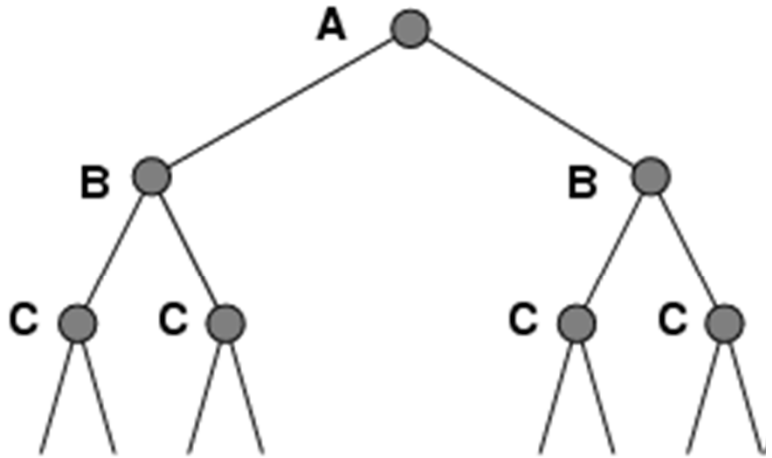
Estados repetidos

- O processo de busca pode perder tempo expandindo nós já explorados antes
  - Estados repetidos podem levar a loops infinitos
  - Estados repetidos podem transformar um problema linear em um problema exponencial

Estados Repetidos

- Não detectar estados repetidos pode transformar um problema linear em um problema exponencial.





Detecção de estados repetidos

- Comparar os nós prestes a serem expandidos com nós já visitados.
  - Se o nó já tiver sido visitado, será descartado.
  - Lista “closed” (fechado) armazena nós já visitados.
  - A busca percorre um grafo e não uma árvore.

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Resumo

- A formulação de problemas usualmente requer a abstração de detalhes do mundo real para que seja definido um espaço de estados que possa ser explorado através de algoritmos de busca.

- Há uma variedade de estratégias de busca sem informação (ou busca cega).

- A busca de aprofundamento iterativo usa somente espaço linear e não muito mais tempo que outros algoritmos sem informação.