CTC-17 Inteligência Artificial
Busca Competitiva e
Busca Iterativa

Prof. Paulo André Castro
pauloac@ita.br
www.comp.ita.br/~pauloac Sala 110,
IEC-ITA

Sumário

- **Busca Competitiva**
  - Para Ambientes multiagentes...

- Busca de Melhoria Iterativa
  - Quando a solução é um ponto não um caminho....

Busca Competitiva

- Jogos com adversários
  - Formulação simples (ações bem definidas)
  - Totalmente observável (geralmente)
  - “Sinônimo” de inteligência
    - Primeiro algoritmo para jogar Xadrez criado em 1950 (Claude Shannon)
  - Problemas bastante complexos:
    - Tamanho + limitação (aprox. 35^{100} nós em jogos de xadrez)
    - Incerteza devido as ações do oponente
    - Agente deve agir antes de completar a busca totalmente

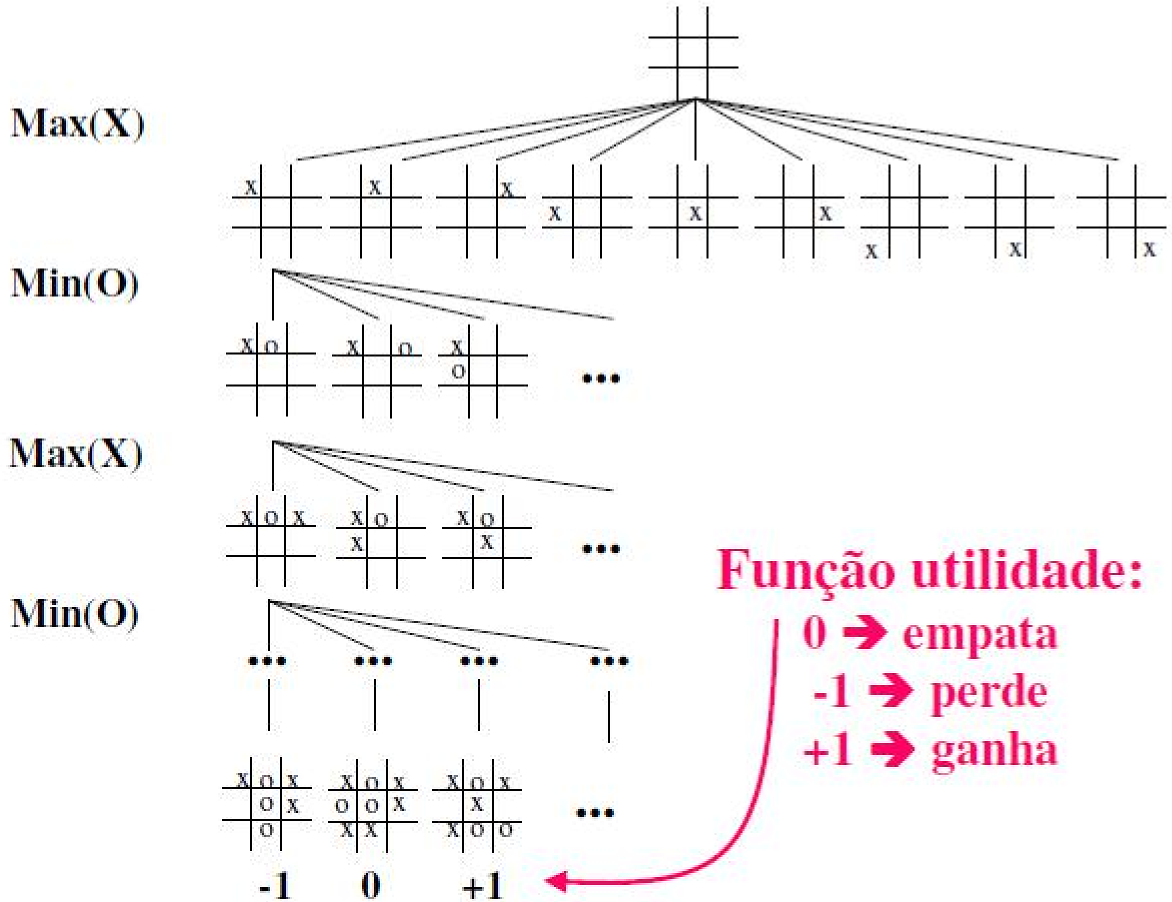
Busca Competitiva

- **2 jogadores, revezam o lance, são adversários**

- **Formulação**
  - **Estado inicial**: posições do tabuleiro + de quem é a vez
  - **Estado final**: posições em que o jogo acaba
  - **Operadores**: jogadas legais
  - **Função de utilidade**: valor numérico do resultado (pontuação)

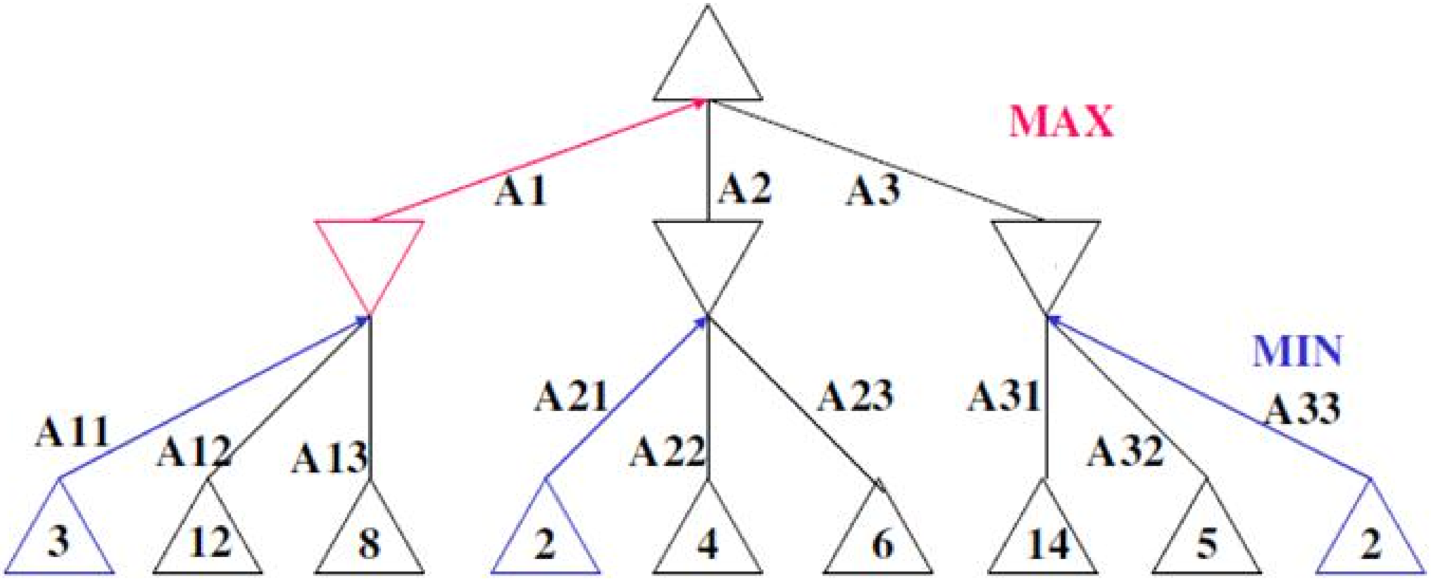
- **Busca: algoritmo minimax**
  - <u>**Idéia**</u>: **maximizar a utilidade** (ganho) supondo que o adversário vai tentar minimizá-la.
  - Minimax faz **busca cega em profundidade**.
  - O agente é MAX e o adversário é MIN.

Jogo da velha - Minimax

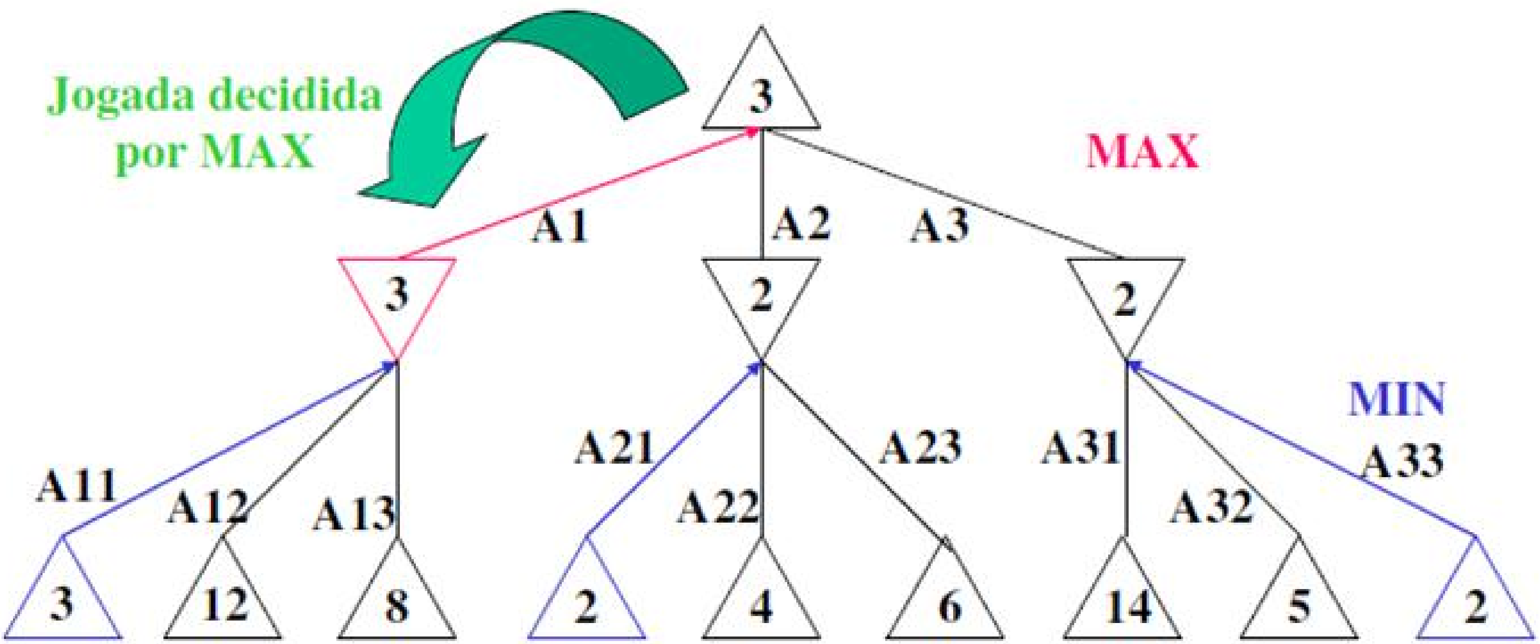


Minimax

- **Passos:**
  - Gera a árvore **inteira** até os estados terminais (ganha, perde ou empata).
  - Aplica a **função de utilidade** nas folhas.
  - Propaga os valores dessa função subindo a árvore através do minimax.
  - Determinar qual a ação que será escolhida por MAX.



Resultado do algoritmo Minimax



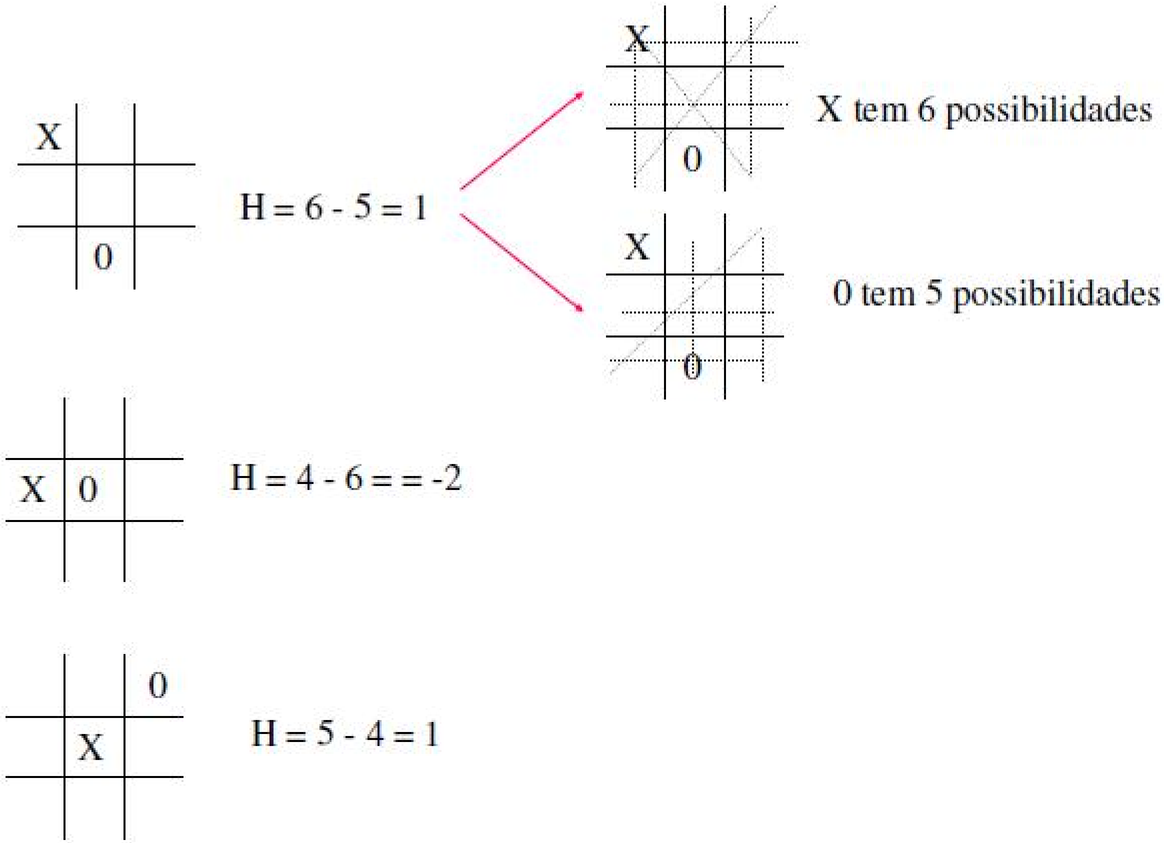
Avaliação do Minimax

- **Problemas**
  - Tempo gasto para determinar a solução ótima pode ser impraticável para muitos problemas reais (percorrer a árvore inteira – todas as folhas)
  - Complexidade: $O(b^m)$ – como em busca em profundidade

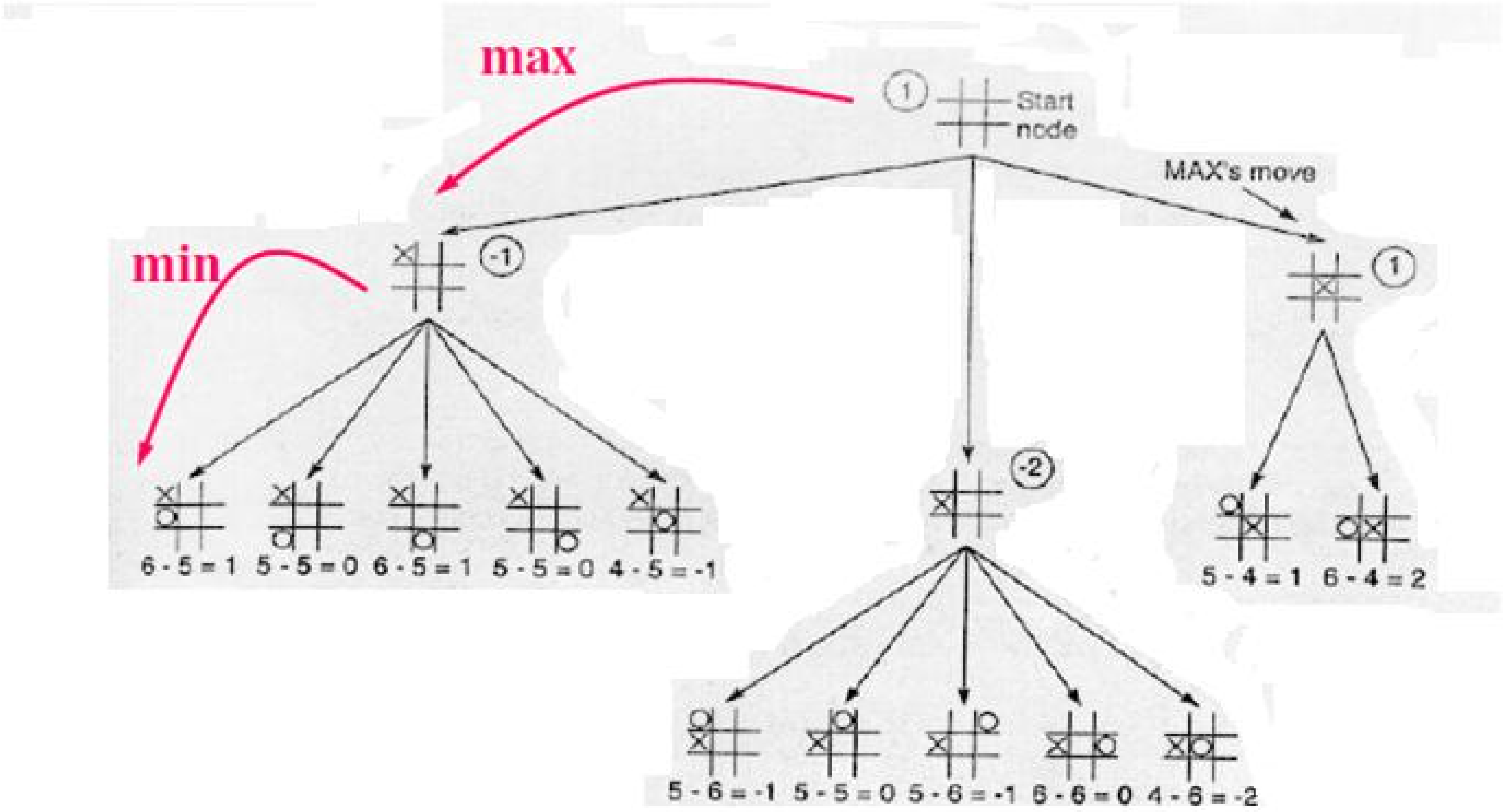
- Entretanto, Minimax traz solução ótima e pode ser modificado para gerar métodos mais eficientes

- Abordagens de Modificação
  - Substituir a função de utilidade por uma função de avaliação **heurística**, e assim limitar a profundidade
  - Podar a árvore e evitar subárvores irrelevantes: Poda **alpha-beta**

Abordagem 1: Função Heurística



Aplicação do Minimax



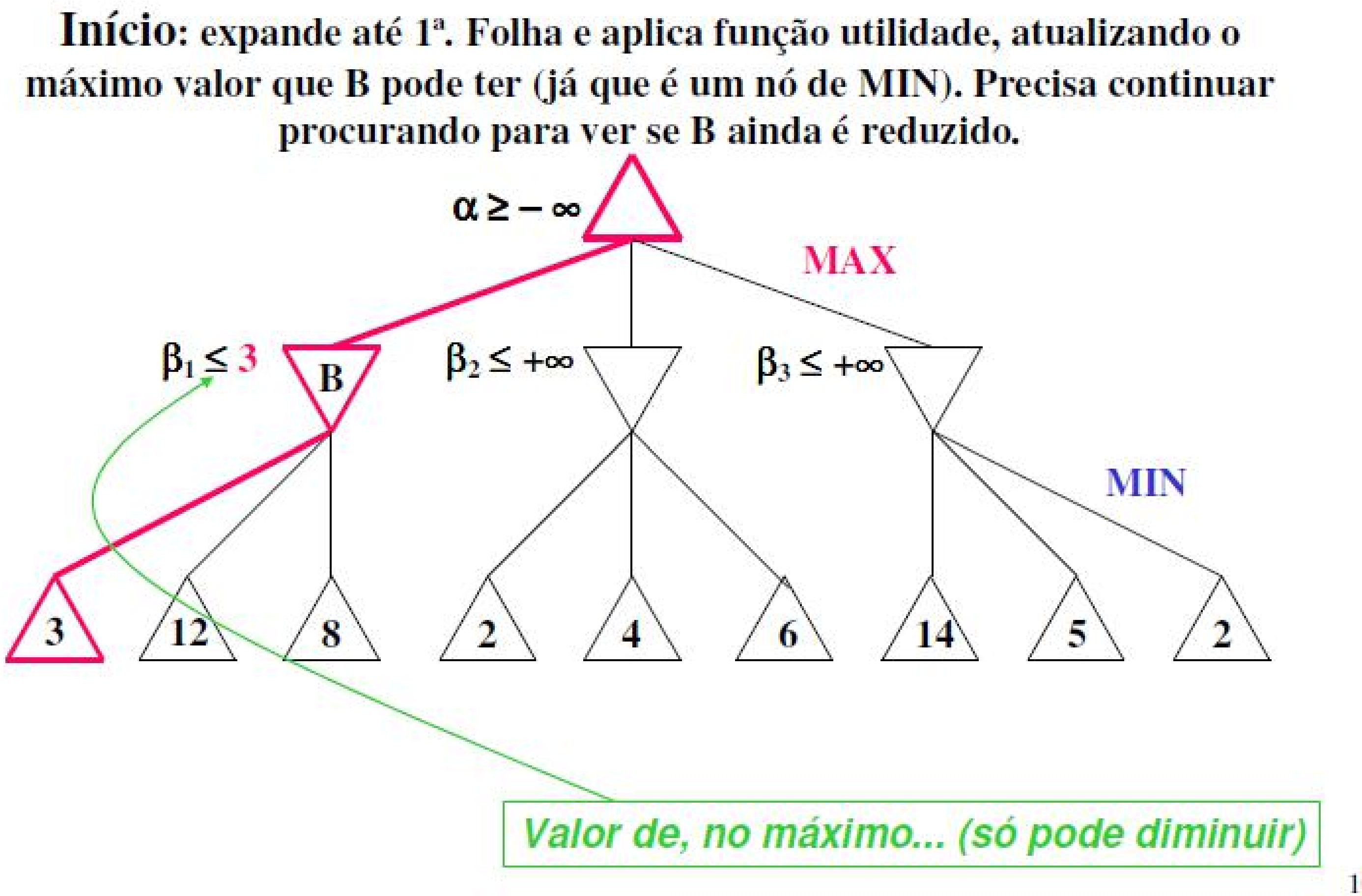
Poda Alpha-Beta

- **Objetivo**: Não expandir subárvores desnecessariamente durante a busca
- Idéia: *Ninguém* escolhe uma opção pior do que uma opção já disponível
- Manter dois parâmetros
  - α - melhor valor para MAX
  - β - melhor valor para MIN

Poda Alpha-Beta (α - β)

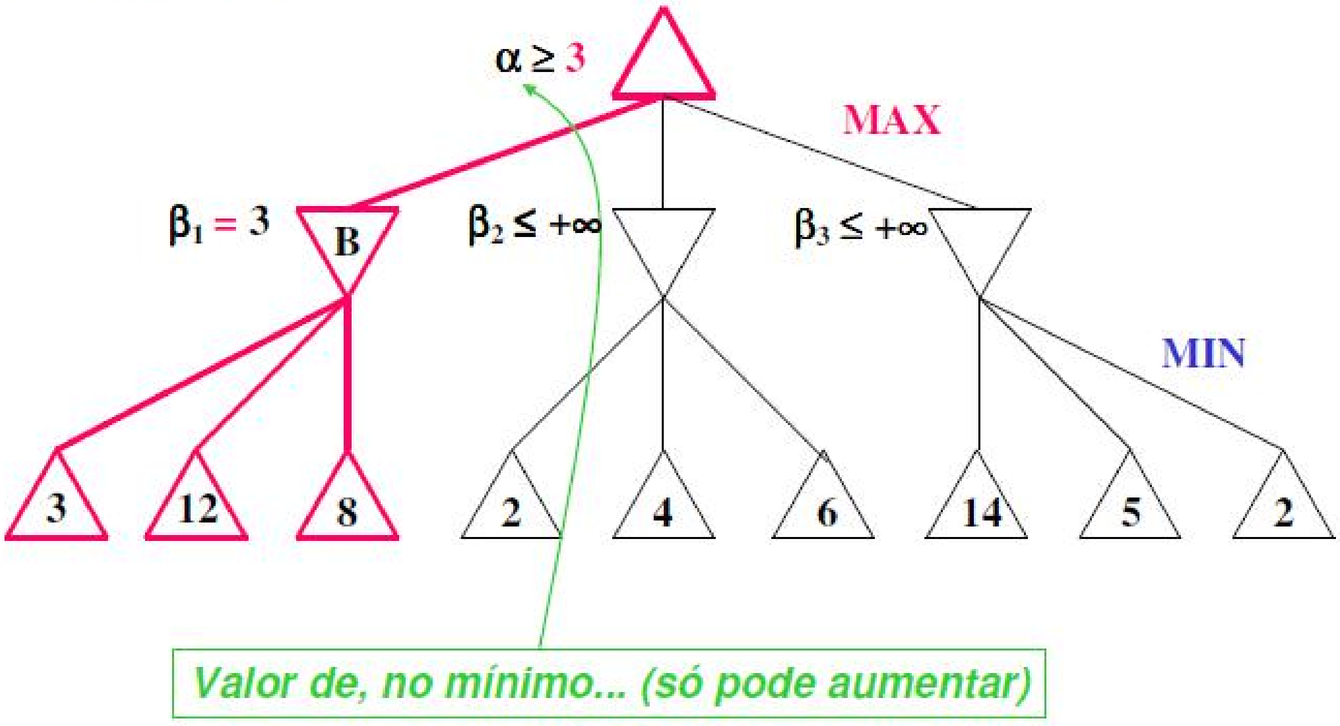
- Teste de expansão - MAX não aceita α pior e MIN não aceita β pior logo:
  - α – não pode diminuir (não pode ser menor que um ancestral)
    - α já encontrado funciona como um limitante inferior
  - β – não pode aumentar (não pode ser maior que um ancestral)
    - β já encontrado funciona como um limitante superior

Poda Alpha-Beta - Exemplo



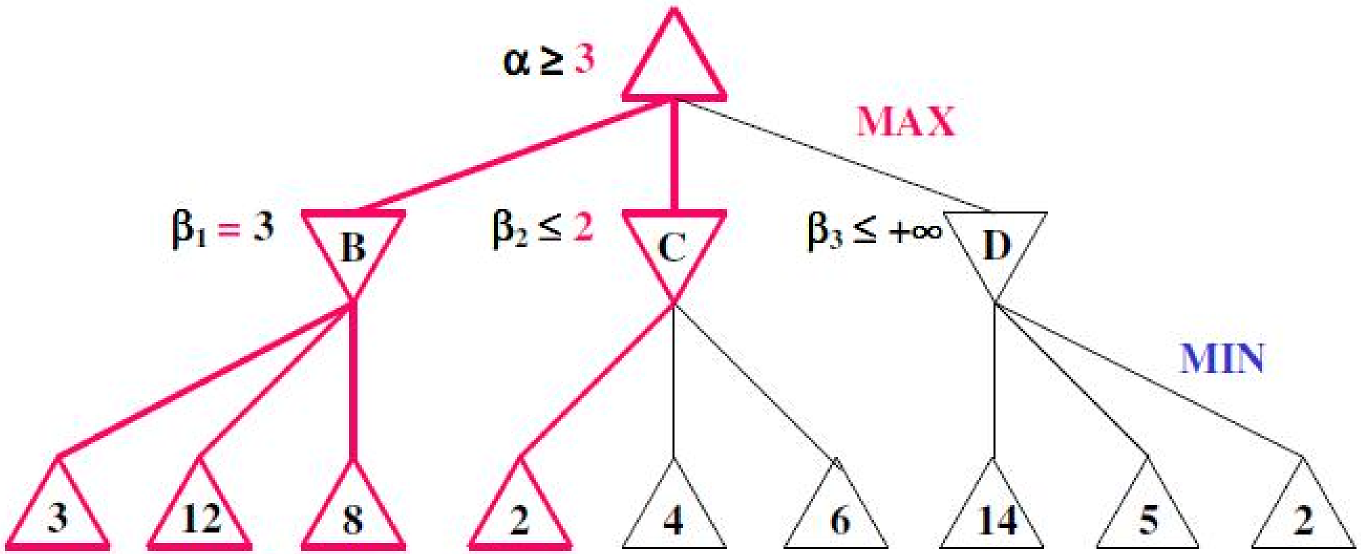
Poda Alpha-Beta – Continuação do Exemplo

Continua expansão de B: folha com 12 > 3 (B não muda seu máximo), depois folha com 8>3 (B não muda seu máximo). B não tem mais filhos, portanto $\beta_1 = 3$ e α do pai pode ser iniciado. Continua expansão, com α limitando a busca.



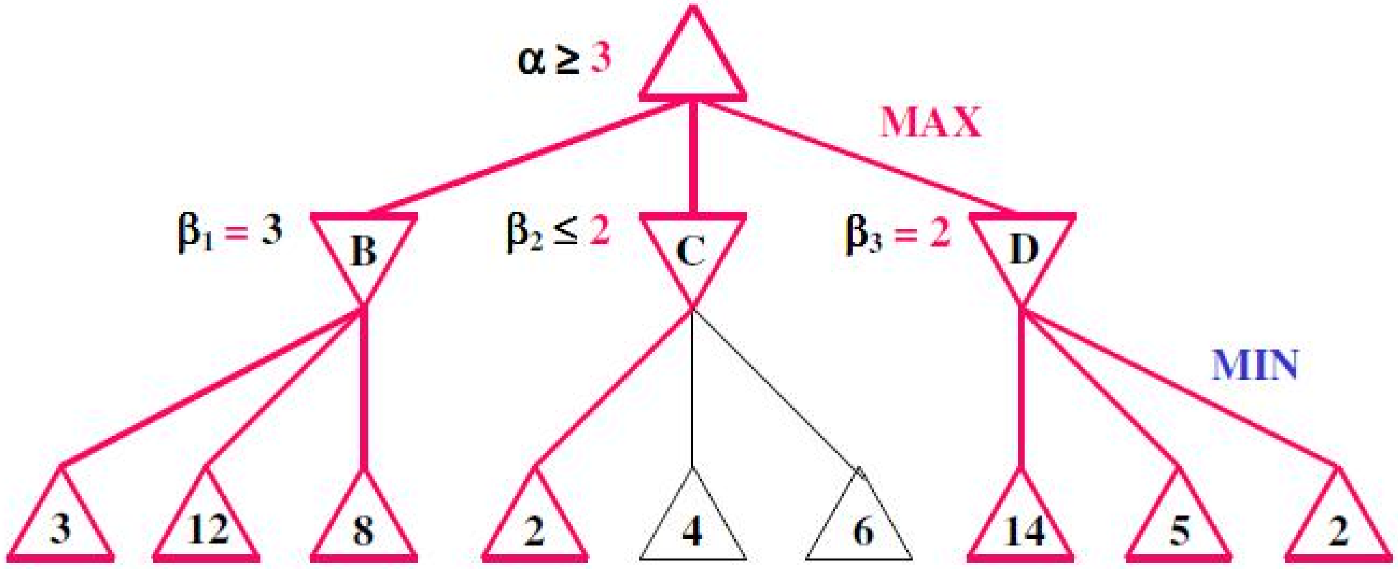
Poda Alpha-Beta – Continuação do Exemplo

Expande C: folha com 2, atualiza $\beta_2 \leq 2$. Como $2 < \alpha$ do nó pai, C não precisa mais ser expandido. Deve-se verificar se $\beta_3 > 3$ para mudar α.
Justificativa: se novo filho tiver valor MAIOR que 2, como β_2 não pode aumentar, nada será mudado; se novo filho tiver valor MENOR que 2, β_2 reduzirá mas não afetará α, que só pode aumentar e já está com valor 3.



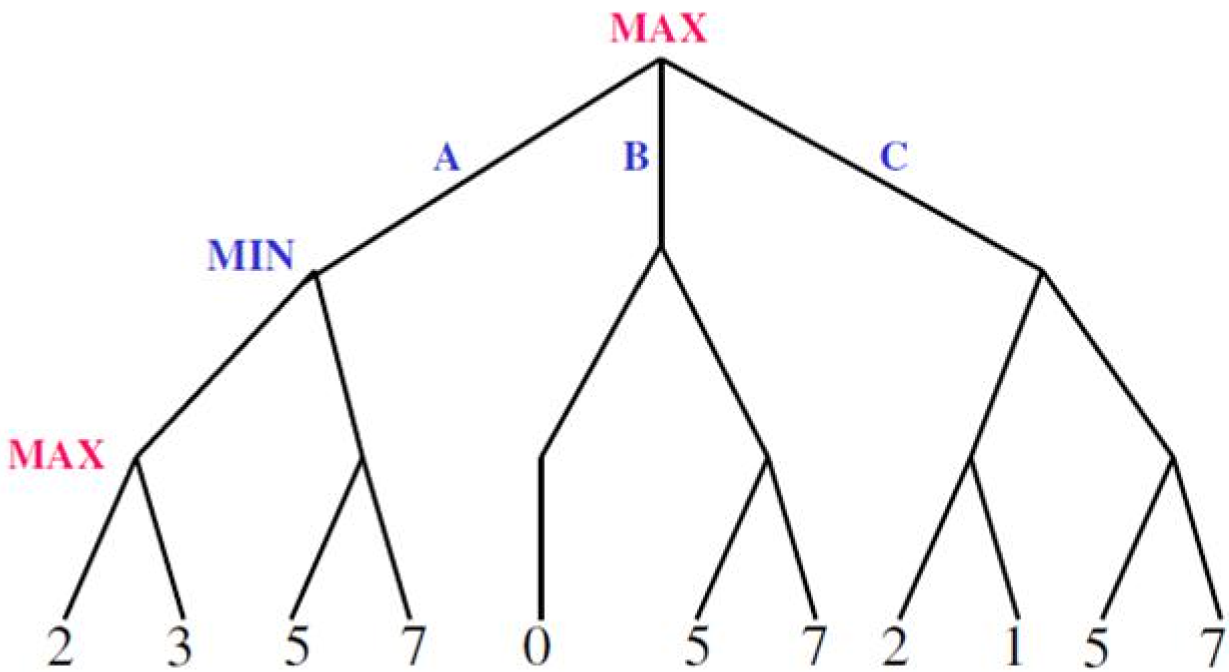
Poda Alpha-Beta – Continuação do Exemplo

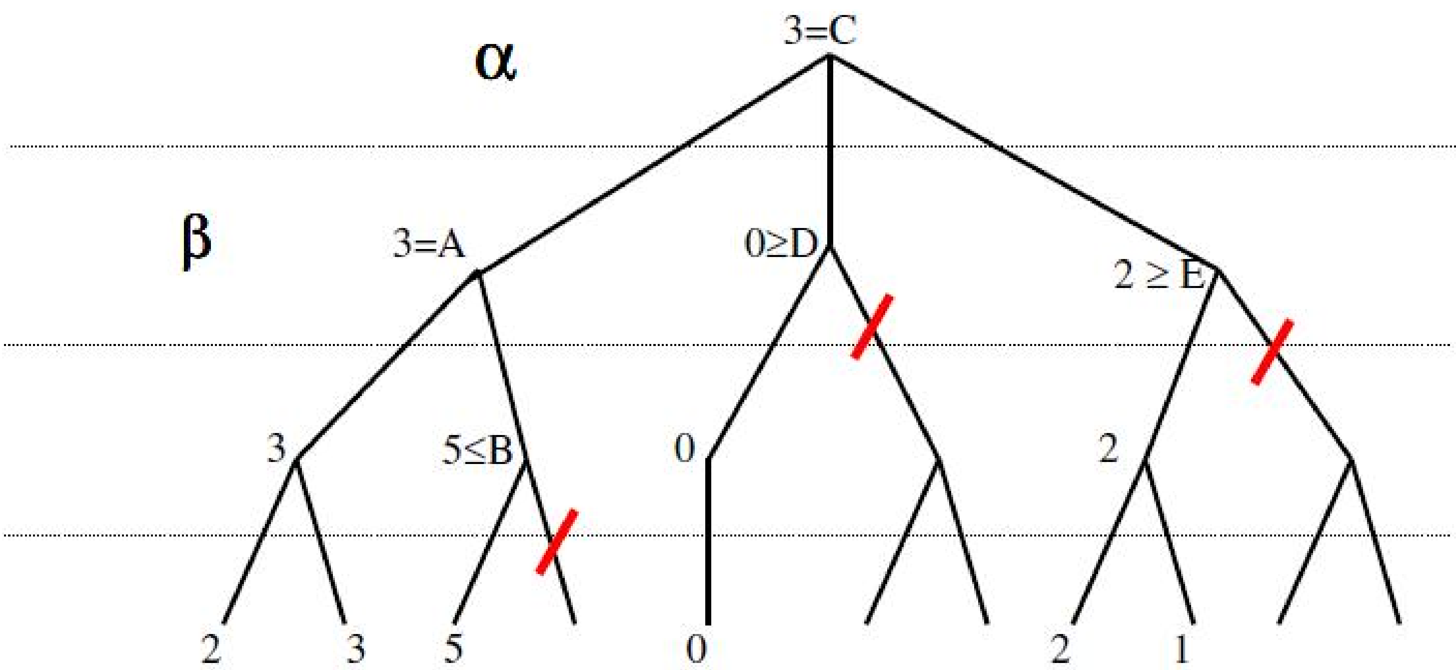
Expande D: folha com 14, atualiza $\beta_3 \leq 14$ e como $14 > \alpha$ e β_3 ainda pode diminuir, continua a expansão de D. Folha com 5, reduz β_3 e como $5 > \alpha$ e β_3 ainda pode diminuir, continua a expansão de D. Última folha tem 2: define valor de $\beta_3 = 2$ e verifica se atualiza α; como $\alpha > 2$, ele não muda e a busca termina, com escolha da jogada B.



Exercício

Decidir a jogada de MAX (A, B ou C) considerando as utilidades fornecidas nas folhas. Adotando a poda alfa-beta, indicar quais arestas/subárvores serão podadas.



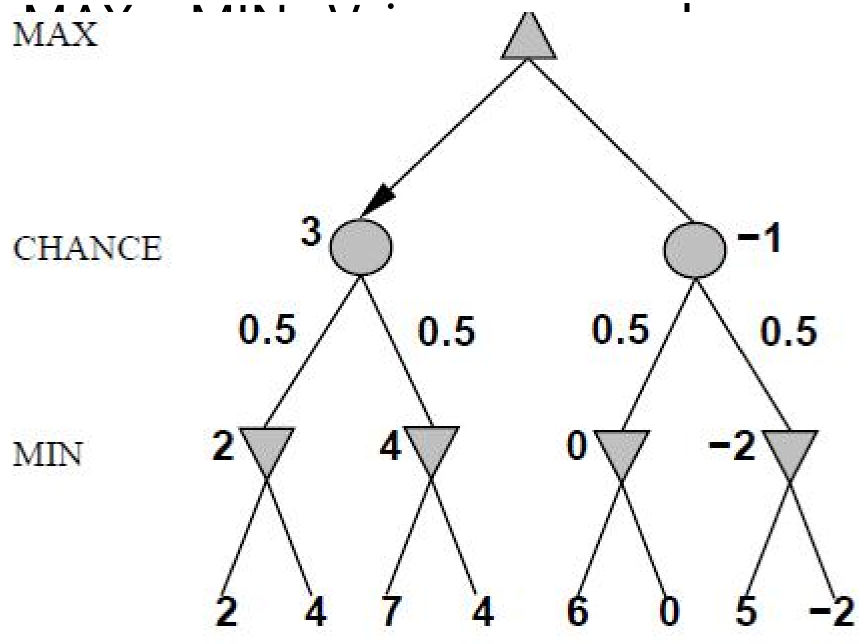


A tem β=3; B será podado por β, já que 5>3
D é podado por α, já que 0<3
E é podado por α, já que 2<3
C é 3.

Nem todos os jogos são determinísticos...

- Como tratar problemas onde a situação do ambiente não depende exclusivamente das decisões dos agentes? Ex. Gamão, War, poquêr, black jack, etc. ...

- Pode-se modelar o fator aleatório como um terceiro jogador..Por exemplo, em jogos com dados, os dados seriam um terceiro jogador que age entr



Algoritmo Expectminimax

EXPECTIMINIMAX proporciona jogo perfeito

Igual à MINIMAX, mas considerando-se nós fortuitos:

...

se *estado* é um nó fortuito **então** uso

EXPECTIMINIMAX-VALUE de SUCCESSORS(*estado*)

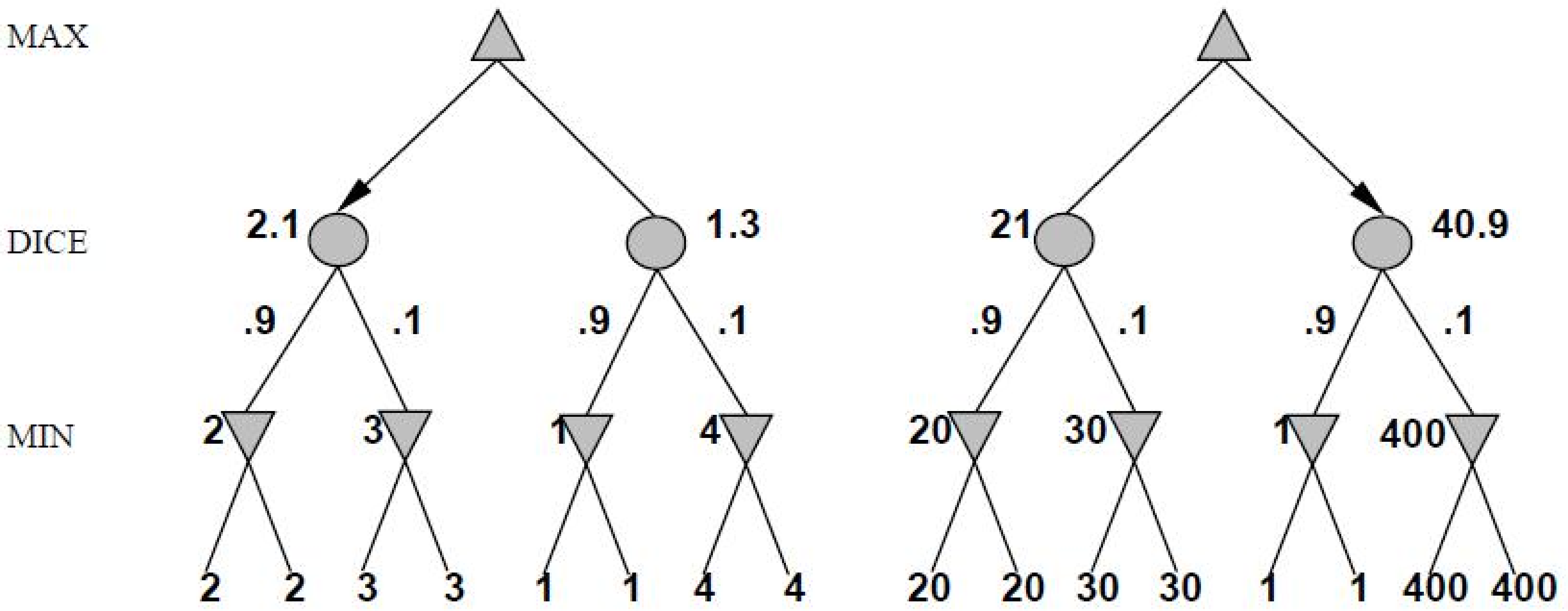
...

$$\text{expectmax}\ (C) = \sum_i P(d_i) \max_{s \in S(C,d_i)} (\text{utilidade}(s)) \tag{1}$$

$$\text{expectmin}\ (C) = \sum_i P(d_i) \min_{s \in S(C,d_i)} (\text{utilidade}(s)) \tag{2}$$

Uma versão de poda alpha-beta é possível

Valores exatos das utilidades importam



- O comportamento não é mais preservado se utilizarmos um escalonamento que apenas preserve a ordem das utilidades dos nós folhas
- Para preservar o comportamento, deve-se utilizar uma transformação linear positiva (Teoria da decisão)

Algoritmos Expectminimax

- A introdução do elemento ao acaso, faz aumentar enormemente a árvore de busca e o tempo para analisá-la
- A poda alfa-beta pode ser usada, porém é muito menos efetiva

- Jogos com incerteza são muitas vezes tratados com outras técnicas tais como Teoria da decisão, Modelo decisório de Markov, e redes bayesianas que serão estudadas no segundo bimestre

Sumário

- Busca Competitiva
- **Busca de Melhoria Iterativa**

Algoritmos de Melhoria Iterativa

Em muitos problemas de otimização, a trajetória é *irrelevante*;
o próprio estado-objetivo é a solução

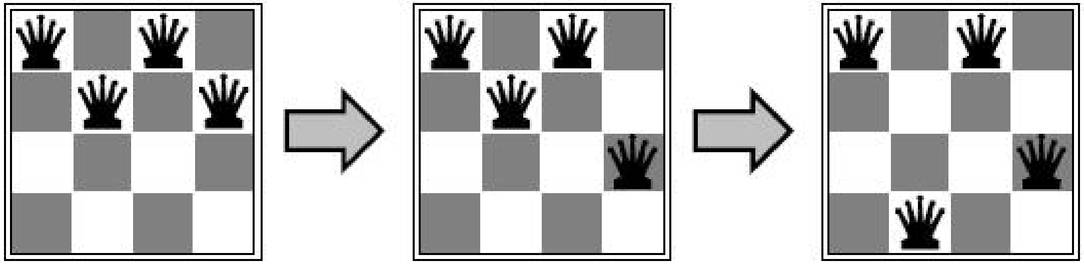
Espaço de estados = conjunto de configurações "completas"
ache configuração *ótima* (e.g. PCV),
ou ache configuração que satisfaz restrições (e.g. prob. das n-rainhas)

Em tais casos, pode-se usar um algoritmo de **melhoria iterativa**;
mantém-se um único estado "atual", tentando-se melhorá-lo

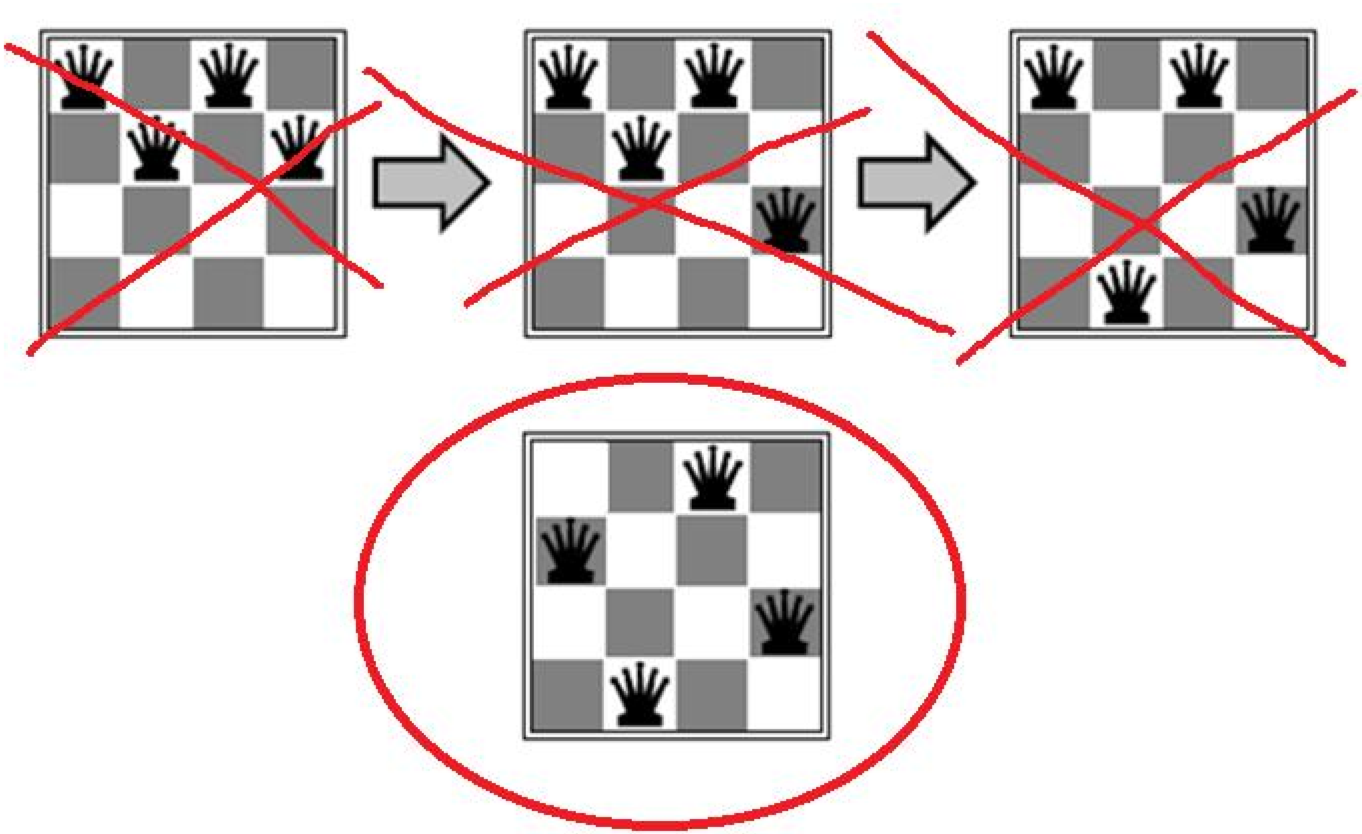
Espaço constante, adequado para busca *online* ou *offline*

Exemplo: Estado Objetivo

Problema das n-rainhas: Ponha n rainhas em um tabuleiro $n \times n$, sem que duas rainhas fiquem na mesma linha, coluna, ou diagonal



Solução?

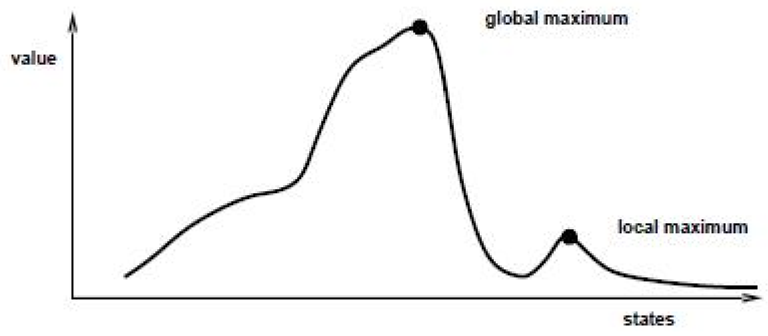


Subida de Encosta (Hill-Climbing) ou subida de gradiente

```
function HILL-CLIMBING( problem) returns a solution state
    inputs: problem, a problem
    local variables: current, a node
                     next, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        next ← a highest-valued successor of current
        if VALUE[next] < VALUE[current] then return current
        current ← next
    end
```

Dependendo do estado inicial, pode ficar preso em máximos locais



É como escalar uma montanha com amnésia e em névoa espessa....

Hill Climbing

- Determinar mínimos: basta encontrar os máximos da função objetiva negativada: -f(x)
- Subida de Encosta com reinicio aleatório: Ao encontrar um plateau ou um máximo local não satisfatório. Reinicie o algoritmo a partir de outro ponto inicial.
  - Abordagem: "Se não tiver sucesso na primeira vez, continue tentando."
- Subida de encosta estocástica: seleciona um movimento aleatório com certa probabilidade ao invés de sempre seguir a direção de subida. A probabilidade da seleção pode variar de com o grau de declividade.
  - Por exemplo, quanto menor a declividade maior a probabilidade de selecionar aleatoriamente

Busca de Têmpera Simulada (Simmulated Annealing)

Idéia: fuga de máximos locais permitindo-se alguns movimentos "ruins" mas **gradualmente diminuindo-se o tamanho e frequência destes**

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if $T=0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

Busca de Têmpera Simulada (Simmulated Annealing) - 2

Observe que $\Delta E < 0$ para os estados "ruins".

T diminuiu lentamente o bastante $\Longrightarrow$ sempre atinge-se o melhor estado

esta é necessariamente uma garantia interessante??

Inventado por Metropolis em 1953, para modelamento de processos físicos

Extensivamente usado em projetos de VLSI, programação de rotas aéreas, etc.

Resumo

- Buscas de Melhoria Iterativa permitem resolver problemas onde o caminho não é relevante e apenas a configuração final importa

- Outras possíveis idéias para melhorar busca local:
  - Paralelizar : buscas a partir de diversos pontos simultâneamente
  - Mesclar pontos intermediários que aparentam ser "boas" configurações intermediárias, para criar novas configurações
  - Fazer pequenos desvios aleatórios na configuração...
  - Isso lembra algo?
  - Seleção Natural e algoritmos genéticos