

Universidade de Aveiro

Inteligência Artificial (LEI, LECI)

Tópicos de IA:
Resolução Automática de Problemas

Ano lectivo 2025/2026

Regente: Luís Seabra Lopes

Tópicos de Inteligência Artificial

- Agentes
- Técnicas de resolução automática de problemas
 - Pesquisa em árvore no espaço de estados
 - Pesquisa com propagação de restrições
 - Pesquisa local
 - Planeamento
- Representação do conhecimento
- Tratamento probabilístico da incerteza

Tópicos de Inteligência Artificial

- Agentes
- Técnicas de resolução automática de problemas
 - Pesquisa em árvore no espaço de estados
 - Pesquisa com propagação de restrições
 - Pesquisa local
 - Planeamento
- Representação do conhecimento
- Tratamento probabilístico da incerteza

Resolução de problemas em IA

- Um *problema* é algo (um objectivo) cuja solução não é imediata
- Por isso, a resolução de um problema requer a *pesquisa de uma solução*

Resolução de problemas em IA

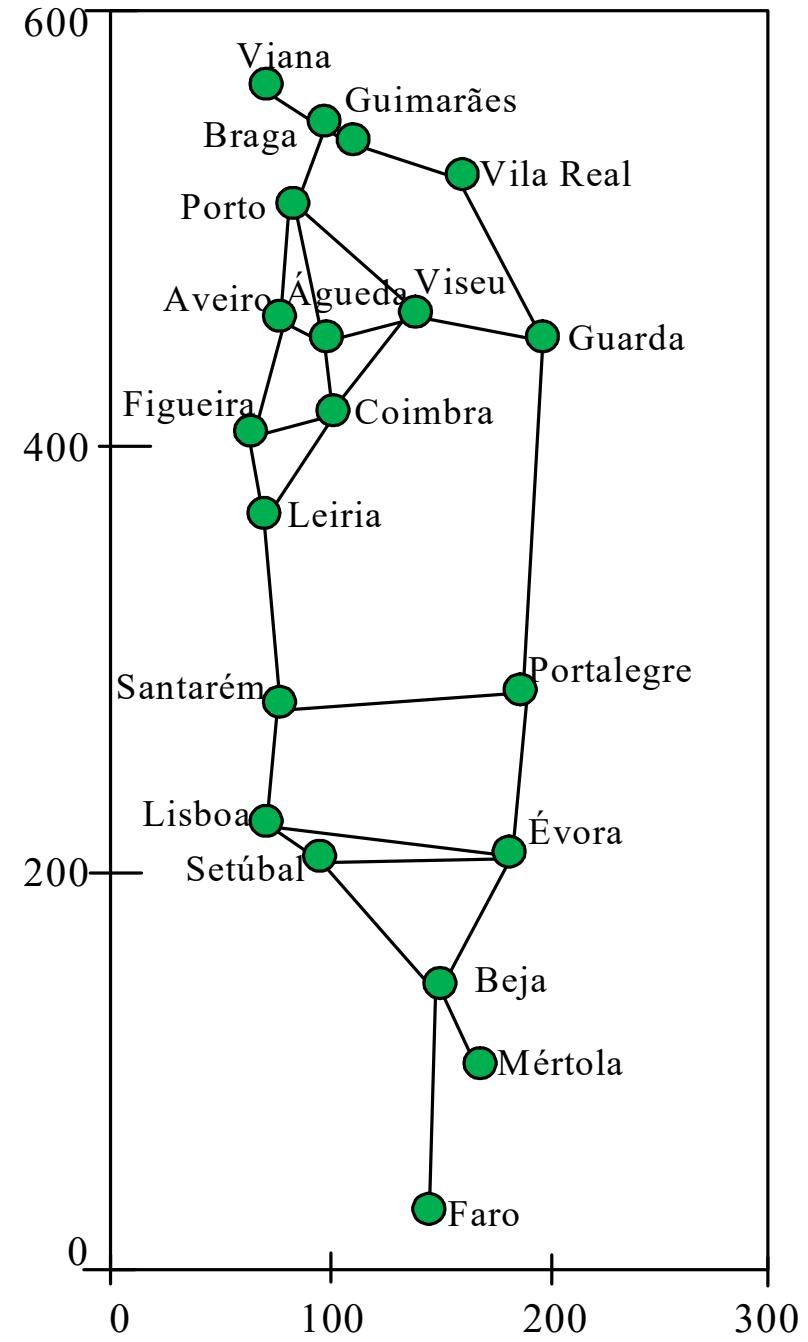
- Um *problema* é algo cuja solução não é imediata
- Exemplos de problemas:
 - Dado um conjunto de axiomas, demonstrar um novo teorema
 - Dado um mapa, determinar o melhor caminho entre dois pontos.
 - Dada uma situação num jogo de xadrez, determinar uma boa jogada.
 - Determinar a melhor distribuição das portas lógicas no circuito VLSI
 - Dada as peças de um produto a montar, determinar a melhor sequência de montagem.

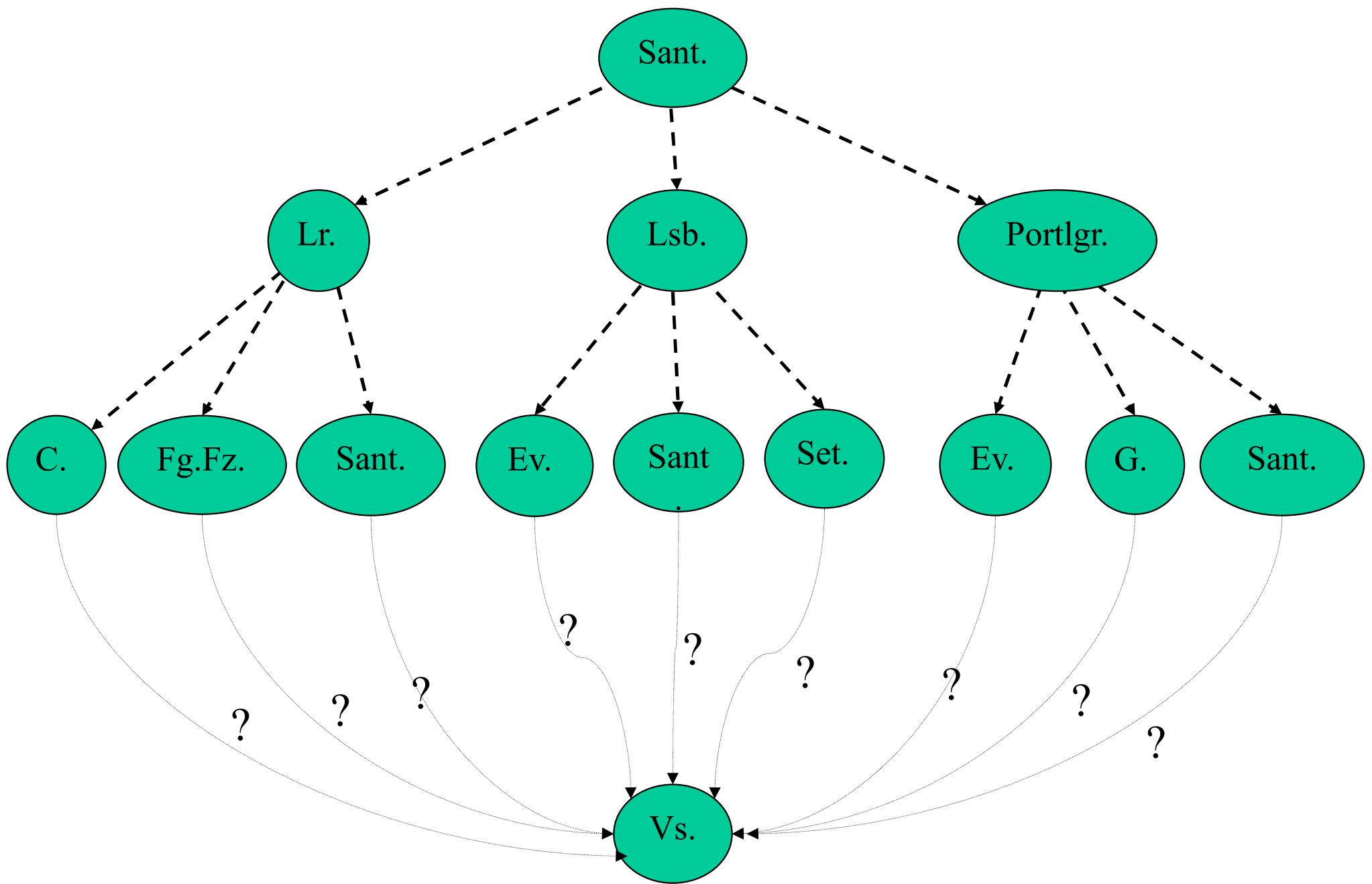
Formulação de problemas e pesquisa de soluções

- A formulação de um problema inclui:
 - Descrição do ponto de partida – o estado inicial
 - Exemplos
 - A situação no jogo de xadrez
 - A descrição de um mapa e a localização inicial do viajante
 - Um conjunto de transições de estados
 - Um função que diz se um dado estado satisfaz o objectivo
 - Por vezes também uma função que avalia o custo de uma solução
- A pesquisa de uma solução é um processo que, de forma recursiva ou iterativa, vai executando transições de estados até que um estado gerado satisfaça o objectivo.

Aplicação: determinar um percurso num mapa topológico

- Dados:
 - Distâncias por estrada entre cidades vizinhas
- Exemplo:
 - Determinar um caminho de Santarém para a Viseu





Pesquisa em árvore – algoritmo genérico

`pesquisa(Problema,Estratégia) retorna a Solução, ou ‘falhou’`

`Árvore ← árvore de pesquisa inicializada com o estado inicial do Problema`
`Ciclo:`

`se não há candidatos para expansão, retornar ‘falhou’`

`Folha ← uma folha escolhida de acordo com Estratégia`

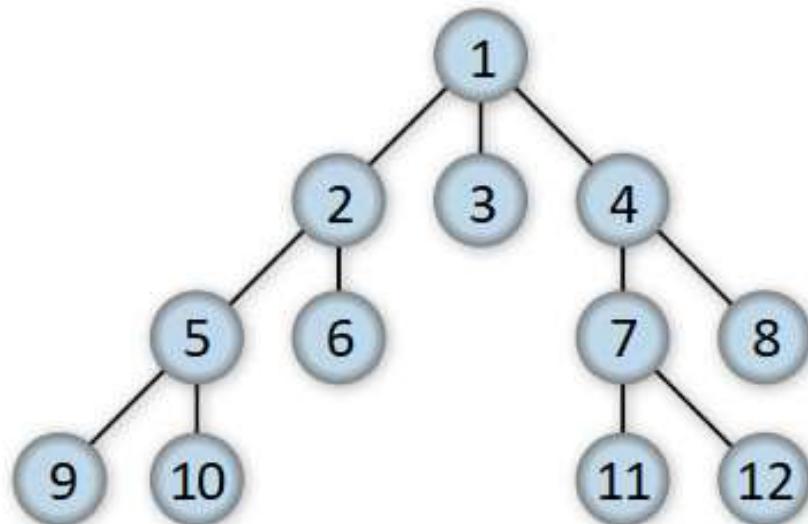
`se Folha contém um estado que satisfaz o objectivo`

`então retornar a Solução correspondente`

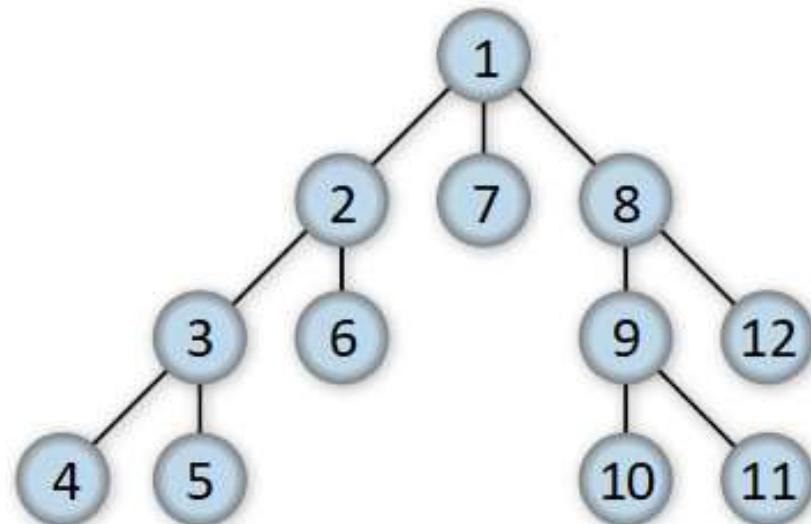
`senão expandir Folha e adicionar os nós resultantes à Árvore`

`Fim do ciclo;`

Percursos na árvore de pesquisa



Pesquisa em largura



Pesquisa em profundidade

(crédito das figuras: Alexander Drichel / Wikipedia)

Pesquisa em árvore – implementação baseada numa fila

`pesquisa_em_arvore(Problema, AdicionarFila)` **retorna** a Solução, ou ‘falhou’

`Fila` \leftarrow [`fazer_nó`(estado inicial do Problema)]

Ciclo

se Fila está vazia, **retornar** ‘falhou’

`Nó` \leftarrow `remover_cabeça`(Fila)

se estado(`Nó`) satisfaz o objectivo

então **retornar** a solução(`Nó`)

senão `Fila` \leftarrow `AdicionarFila`(Fila, `expansão`(`Nó`))

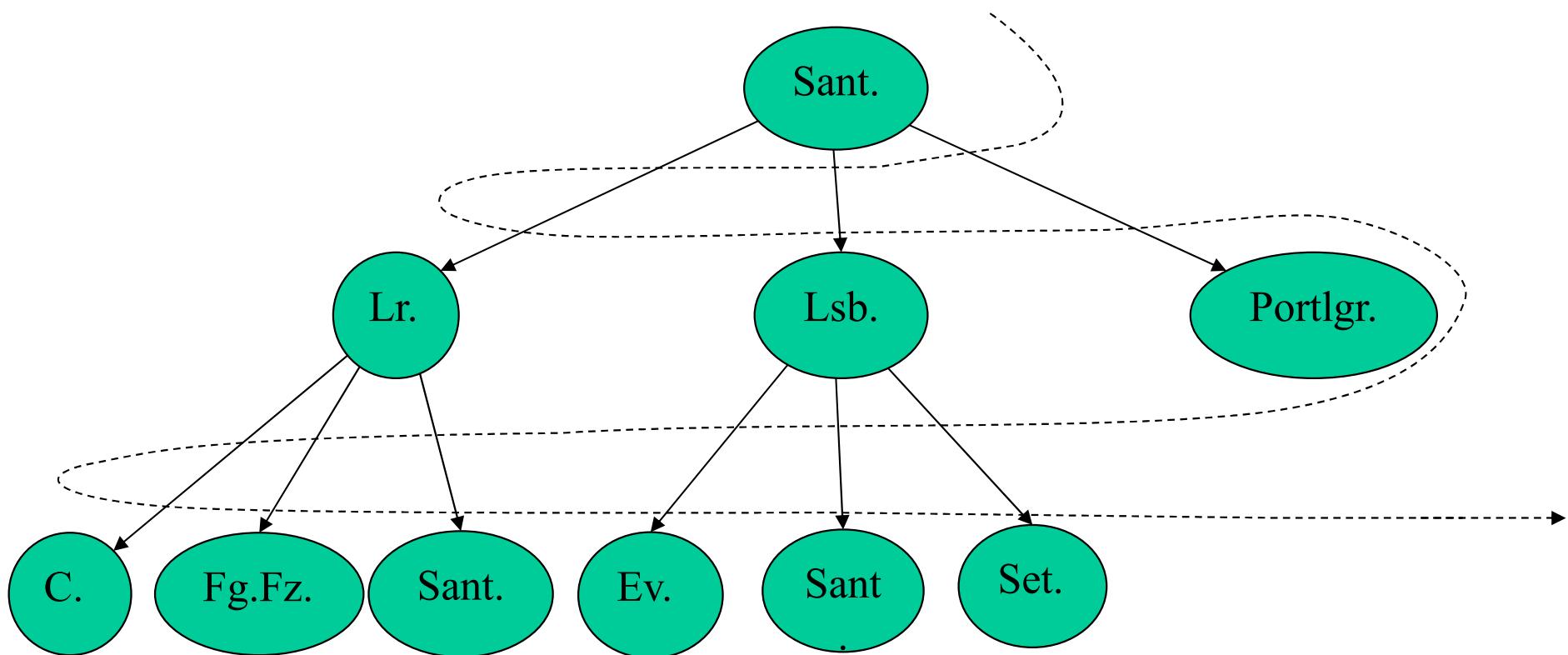
`pesquisa_em_largura(Problema)` **retorna** a Solução, ou ‘falhou’

retornar `pesquisa_em_arvore(Problema, juntar_no_fim)`

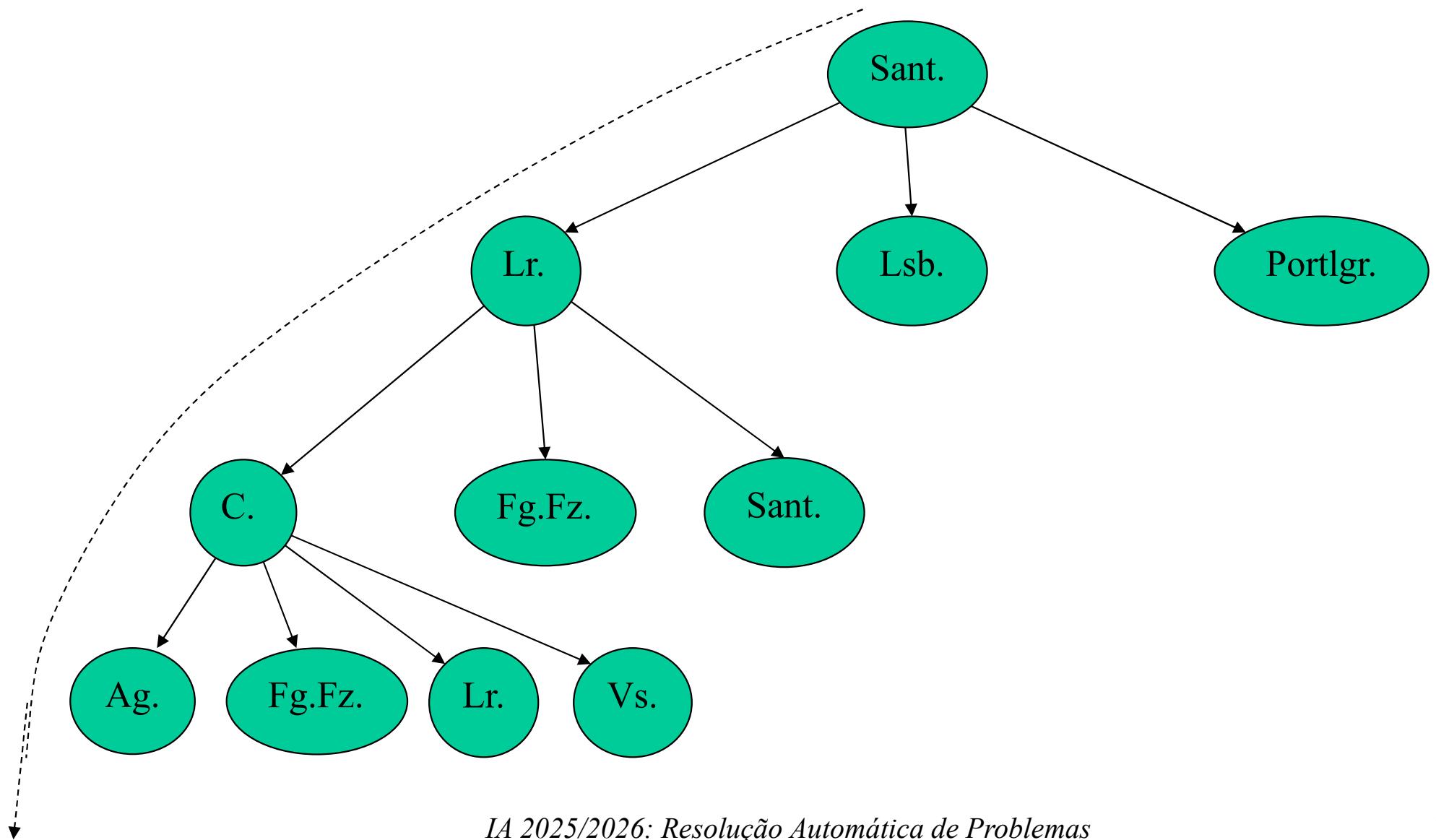
`pesquisa_em_profundidade(Problema)` **retorna** a Solução, ou ‘falhou’

retornar `pesquisa_em_arvore(Problema, juntar_à_cabeça)`

Pesquisa em largura



Pesquisa em profundidade



Pesquisa em Árvore em Python

- Vamos criar um conjunto de classes para suporte à resolução de problemas por pesquisa em árvore
 - Classe **SearchDomain()** – classe abstracta que formata a estrutura de um domínio de aplicação
 - Classe **SearchProblem(domain,initial,goal)** – classe para especificação de problemas concretos a resolver
 - Classe **SearchNode(state,parent)** – classe dos nós da árvore de pesquisa
 - Classe **SearchTree(problem)** – classe das árvores de pesquisa, contendo métodos para a geração de uma árvore para um dado problema

Pesquisa em Árvore em Python

SearchTree:
problem:

SearchProblem
domain:

SearchDomain
actions()
result()
cost()
heuristic()

initial:

goal:

test_goal()

strategy:

search()

open_nodes:



- Nota: Ver módulo usado nas aulas práticas

SearchNode
state:
parent:

Pesquisa em profundidade - variantes

- Pesquisa em profundidade *sem repetição de estados* – para evitar ciclos infinitos, convém garantir que estados já visitados no caminho que liga o nó actual à raiz da árvore de pesquisa não são novamente gerados
- Pesquisa em profundidade *com limite* – não são considerados para expansão os nós da árvore de pesquisa cuja profundidade é igual a um dado limite
- Pesquisa em profundidade *com limite crescente* – consiste no seguinte procedimento:
 - 1) Tenta-se resolver o problema por pesquisa em profundidade com um dado limite N
 - 2) Se foi encontrada uma solução, retornar.
 - 3) Incrementar N .
 - 4) Voltar ao passo 1.

Pesquisa informada (“melhor primeiro”)

`pesquisa_informada(Problema,FuncAval)` **retorna** a Solução, ou ‘falhou’

Estratégia \leftarrow estratégia de gestão de fila de acordo com FuncAval

`pesquisa_em_arvore(Problema,Estratégia)`

Avaliação das estratégias de pesquisa

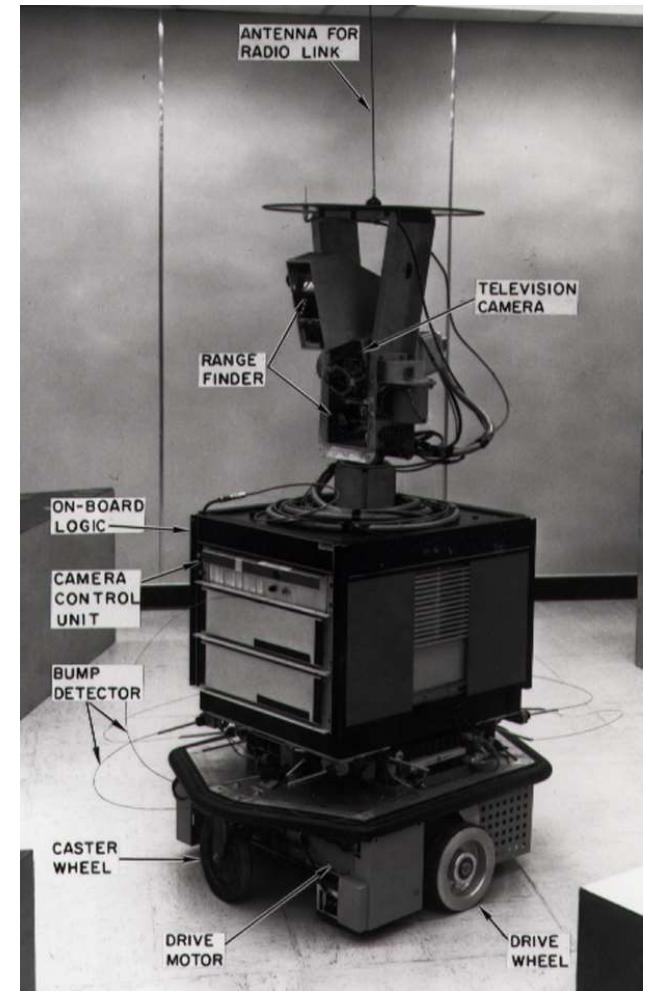
- Completude – uma estratégia é completa se é capaz de encontrar uma solução quando existe uma solução
- Complexidade temporal – quanto tempo demora a encontrar a solução
- Complexidade espacial – quanto espaço de memória é necessário para encontrar uma solução
- Optimalidade – a primeira solução que a estratégia de pesquisa consegue encontrar é a melhor solução.

Pesquisa A*

- Escolhe-se o nó em que a função de custo total $f(n)=g(n)+h(n)$ tem o menor valor
 - $g(n)$ = custo desde o nó inicial até ao nó n
 - $h(n)$ = custo estimado desde o nó n até à solução [heurística]
- A função heurística $h(n)$ diz-se *admissível* se nunca sobrestima o custo real de chegar a uma solução a partir de n .
- Se for possível garantir que $h(n)$ é admissível, então a pesquisa A* encontra sempre (um)a solução óptima.
- A pesquisa A* é também completa.

Shakey the Robot

- A pesquisa A* foi inventada em 1968 para optimizar o planeamento de caminhos deste robô



Pesquisa A* - variantes

- *Pesquisa de custo uniforme*
 - $h(n) = 0$
 - $f(n) = g(n)$
 - É um caso particular da pesquisa A*
 - Também conhecido como algoritmo de Dijkstra
 - Tem um comportamento parecido com o da pesquisa em largura
 - Caso exista solução, a primeira solução encontrada é óptima
- *Pesquisa gulosa*
 - Ignora custo acumulado $g(n)$
 - $f(n) = h(n)$
 - Dado que o custo acumulado é ignorado, não é verdadeiramente um caso particular da pesquisa A*
 - Tem um comportamento que se aproxima da pesquisa em profundidade
 - Ao ignorar o custo acumulado, facilmente deixa escapar a solução óptima

Pesquisa num grafo de estados - motivação

- Em inglês: “*graph search*”
- Frequentemente, o espaço de estados é um grafo.
- Ou seja, transições a partir de diferentes estados podem levar ao mesmo estado.
- Isto leva a que a pesquisa fique menos eficiente.
- Portanto, o que se deve fazer é memorizar os estados já visitados por forma a evitar tratá-los novamente.
- Memoriza-se apenas o melhor caminho até cada estado

Pesquisa num grafo de estados

- Tal como no algoritmo anterior, trabalha-se com uma fila de nós
 - Chama-se fila de nós ABERTOS (nós ainda não expandidos, ou folhas)
 - Em cada iteração, o primeiro nó em ABERTOS é seleccionado para expansão
- Adicionalmente, usa-se também uma lista de nós FECHADOS (os já expandidos)
 - Necessário para evitar repetições de estados

Pesquisa num grafo de estados - algoritmo

- 1. Inicialização
 - $N_0 \leftarrow$ nó do estado inicial; $ABERTOS \leftarrow \{ N_0 \}$
 - $FECHADOS \leftarrow \{ \}$
- 2. Se $ABERTOS = \{ \}$, então acaba sem sucesso.
- 3. Seja N o primeiro nó de $ABERTOS$.
 - Retirar N de $ABERTOS$.
 - Colocar N em $FECHADOS$.
- 4. Se N satisfaz o objectivo, então retornar a solução encontrada.
- 5. Expandir N :
 - $CV \leftarrow$ conjunto dos vizinhos sucessores de N
 - Para cada $X \in CV - (ABERTOS \cup FECHADOS)$, ligá-lo ao antecessor directo, N
 - Para cada $X \in CV \cap (ABERTOS \cup FECHADOS)$, ligá-lo a N caso o melhor caminho passe por N
 - Adicionar os novos nós a $ABERTOS$
 - Reordenar $ABERTOS$
- 6. Voltar ao passo 2.

Pesquisa num grafo de estados

- Tal como a pesquisa em árvore, a “pesquisa em grafo” ou “graph search” utiliza uma árvore de pesquisa
- No entanto, a pesquisa em árvore normal ignora a possibilidade de o espaço de estados ser um grafo
 - Mesmo que o espaço de estados seja um grafo, a pesquisa em árvore trata-o como se fosse uma árvore
- Pelo contrário, a pesquisa em grafo leva em conta que o espaço de estados é normalmente um grafo e garante que a árvore de pesquisa não tem mais do que um caminho para cada estado

Avaliação da pesquisa em árvore

- factores de ramificação

- Seja:
 - N – número de nós da árvore de pesquisa no momento em que se encontra a solução
 - X – Número de nós expandidos (não terminais)
 - d – comprimento do caminho na árvore correspondente à solução
- *Ramificação média* – número médio de filhos por nó expandido:

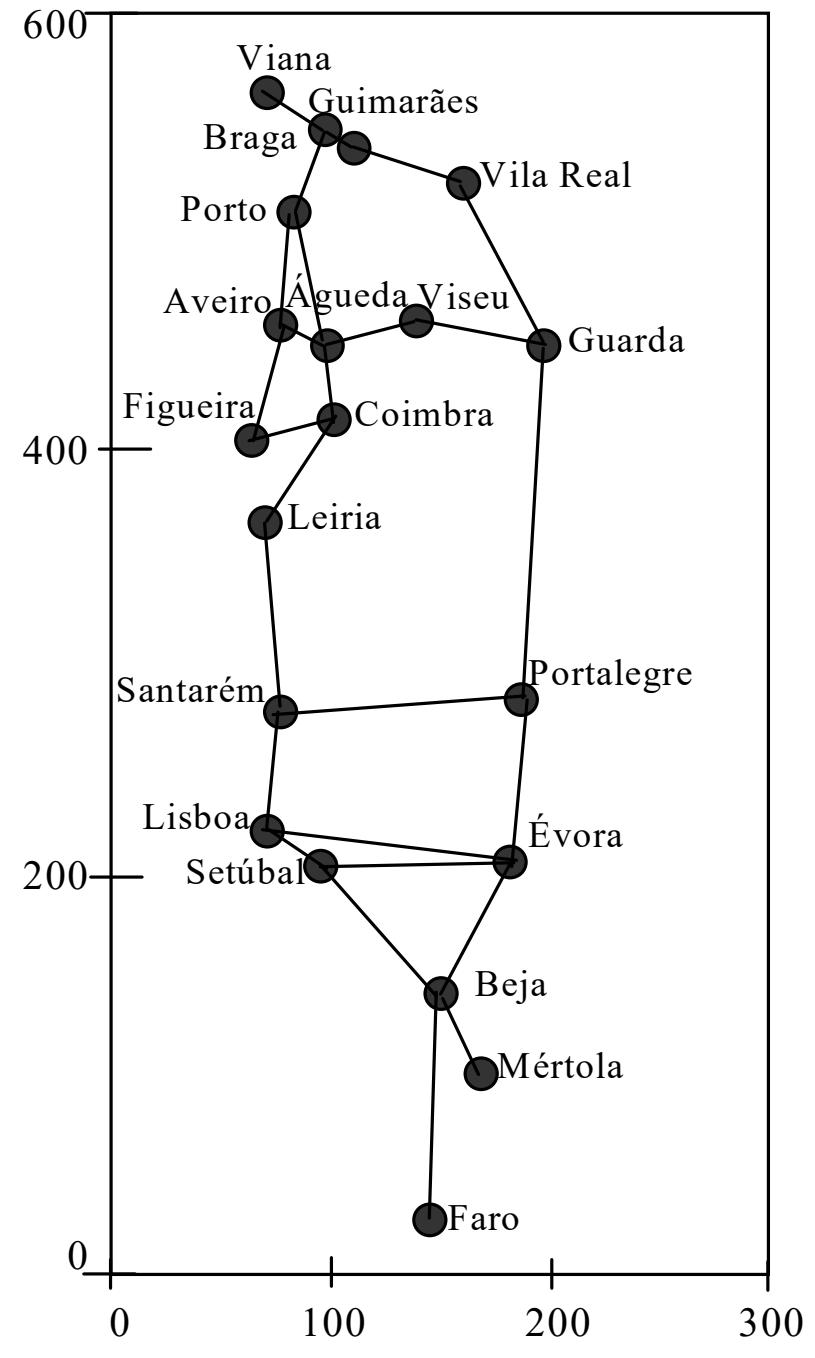
$$RM = \frac{N - 1}{X}$$

Nota: a ramificação média é um indicador da dificuldade do problema.

- *Factor de ramificação efectivo* – número de filhos por nó, B, numa árvore com ramificação constante e com profundidade constante d . Portanto:
$$1 + B + B^2 + \dots + B^d = N$$
 ou seja:
$$\frac{B^{d+1} - 1}{B - 1} = N$$
 (resolve-se por métodos numéricos).
 - O factor de ramificação efectiva é um indicador da eficiência da técnica de pesquisa utilizada.

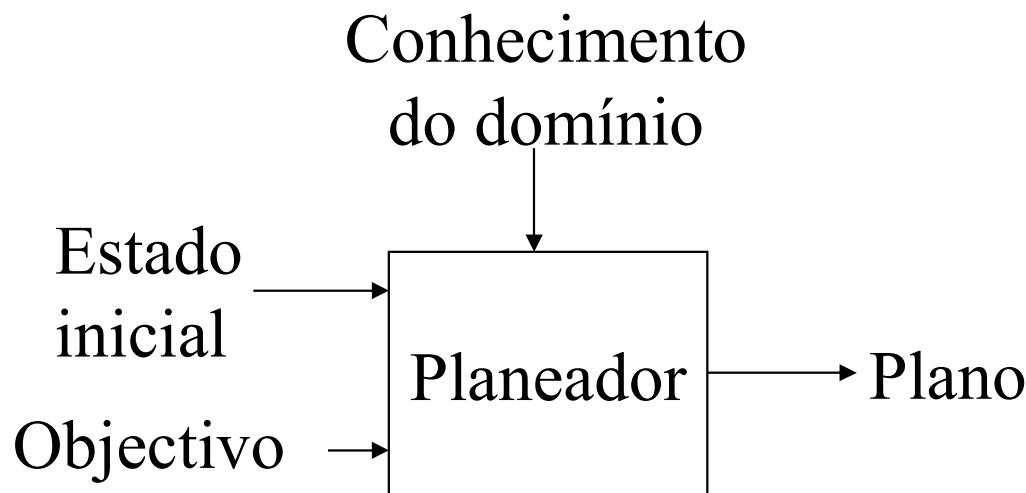
Aplicação: planear um passeio turístico

- Dados:
 - Coordenadas entre cidades
 - Distâncias por estrada entre cidades vizinhas
- Calcular:
 - O melhor caminho entre duas cidades.
- Usando:
 - Pesquisa em largura
 - Pesquisa A*



Aplicação: planeamento de sequências de acções

- O problema consiste em determinar uma sequência de acções a desempenhar por um agente por forma a que, partindo de um dado *estado inicial*, se atinja um dado *objectivo*.
- O *conhecimento do domínio* inclui uma descrição das *condições de aplicabilidade* e *efeitos* das acções possíveis.

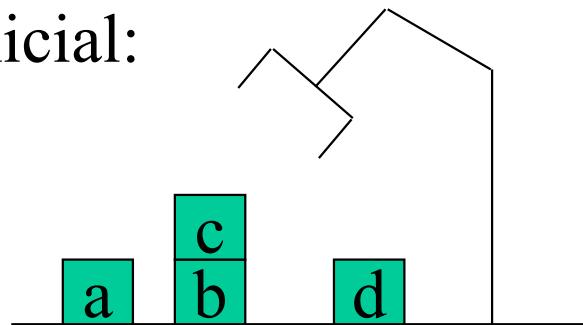


Representação de acções em problemas de planeamento

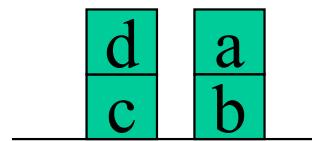
- STRIPS – planeador desenvolvido por volta de 1970, por Fikes, Hart e Nilsson
- A funcionalidade de um dado tipo de operação é definida, no formalismo STRIPS, através de uma estrutura chamada *operador*, que inclui a seguinte informação:
 - *Pré-condições* - um conjunto de fórmulas atómicas que representam as condições de aplicabilidade deste tipo de operação.
 - *Efeitos negativos (delete list)* – um conjunto de fórmulas atómicas que representam propriedades do mundo que deixam de ser verdade ao executar-se a operação.
 - *Efeitos positivos (add list)* – um conjunto de fórmulas atómicas que representam propriedades do mundo que passam a ser verdade ao executar-se a operação.

Exemplo: planeamento no mundo dos blocos

Estado inicial:



Objectivo:



Plano:
[desempilhar(c,b),
poisar(c),
levantar(d),
empilhar(d,c),
levantar(a),
empilhar(a,b)]

Especificação de operadores. Exemplo: empilar(X,Y)

- Pré-condições: [no_robot(X), livre(Y)]
- Efeitos negativos: [no_robot(X), livre(Y)]
- Efeitos positivos: [em_cima(X,Y), robot_livre]

Pesquisa A* - heurísticas

- Uma heurística é tanto melhor quanto mais se aproximar do custo real
 - A qualidade de uma heurística pode ser medida através do factor de ramificação efectiva
 - Quanto melhor a heurística, mais baixo será esse factor
- Em alguns domínios, há funções de estimação de custos que naturalmente constituem heurísticas admissíveis
 - Exemplo: Distância em linha recta no domínio dos caminhos entre cidades
- Em muitos outros domínios práticos, não há uma heurística admissível que seja óbvia
 - Exemplo: Planeamento no mundo dos blocos

Pesquisa A* - cálculo de heurísticas admissíveis em problemas simplificados

- Um problema simplificado (*relaxed problem*) é um problema com menos restrições do que o problema original
 - É possível gerar automaticamente formulações simplificadas de problemas a partir da formulação original
 - A resolução do problema simplificado será feita usando pouca ou nenhuma pesquisa
 - Pode-se assim “inventar” heurísticas, escolhendo a melhor, ou combinando-as numa nova heurística
- IMPORTANTE: O custo de uma solução óptima para um problema simplificado constitui uma heurística admissível para o problema original

Pesquisa A* - combinação de heurísticas

- Se tivermos várias heurísticas admissíveis (h_1, \dots, h_n), podemos combiná-las numa nova heurística:
 - $H(n) = \max(\{h_1(n), \dots, h_n(n)\})$
- Esta nova heurística tem as seguintes propriedades:
 - Admissível
 - Dado que é uma melhor aproximação ao custo real, vai ser uma heurística melhor do que qualquer das outras

Pesquisa A* em aplicações práticas

- Principais vantagens
 - Completa
 - Óptima
- Principais desvantagens
 - Na maior parte das aplicações, o consumo de memória e tempo de computação têm um comportamento exponencial em função do tamanho da solução
 - Em problemas mais complexos, poderá ser preciso utilizar algoritmos mais eficientes, ainda que sacrificando a optimalidade
 - Ou então, usar heurísticas com uma melhor aproximação média ao custo real, ainda que não sendo estritamente admissíveis, e não garantindo portanto a optimalidade da pesquisa

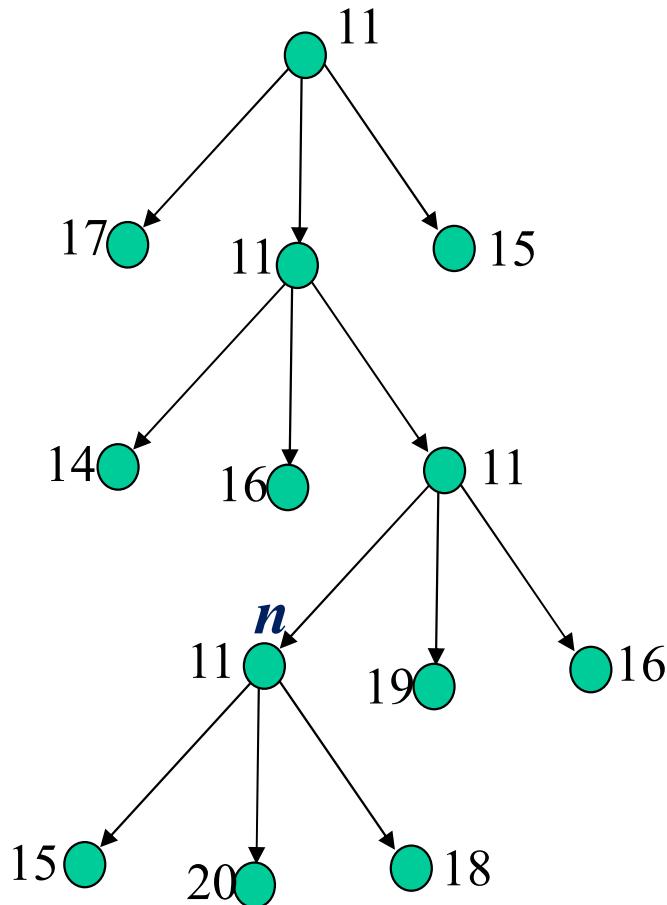
IDA* (Iterative Deepening A*)

- Semelhante à pesquisa em profundidade com aprofundamento iterativo
- A limitação à profundidade é estabelecida indirectamente através de um limite na função de avaliação $f(n) = g(n) + h(n) \leq f_{max}$
 - Ou seja: Qualquer nó n com $f(n) > f_{max}$ não será expandido
- Passos do algoritmo:
 1. $f_{max} = f(\text{raiz})$
 2. Executar pesquisa em profundidade com limite f_{max}
 3. Se encontrou solução, retornar solução encontrada
 4. $f_{max} \leftarrow$ menor $f(n)$ que tenha sido superior a f_{max} na última execução do A*
 5. Voltar ao passo 2

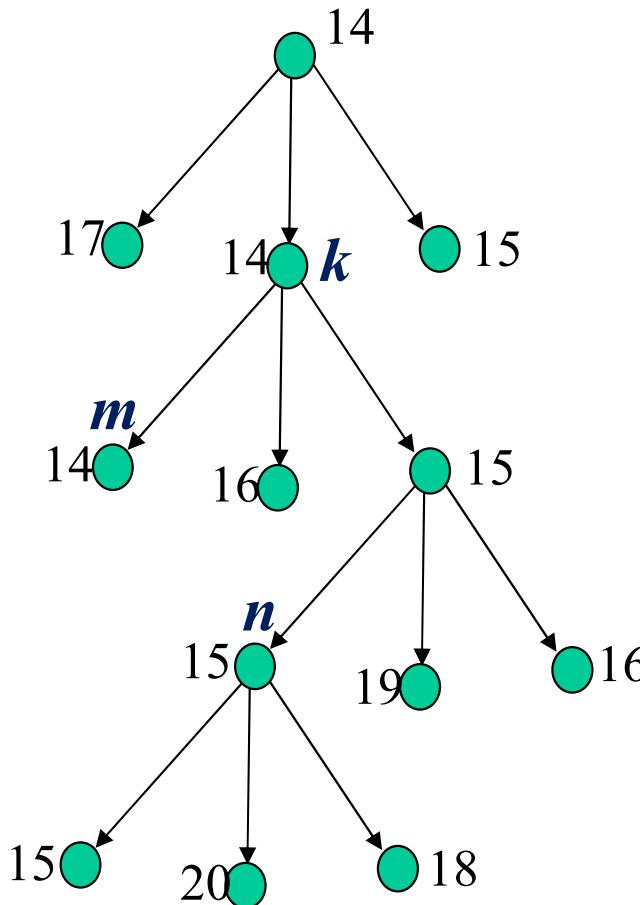
RBFS

- Pesquisa recursiva melhor-primeiro (*Recursive Best-First Search*)
- Para cada nó n , o algoritmo não guarda o valor da função de avaliação $f(n)$, mas sim o menor valor $f(x)$, sendo x uma folha descendente do nó n
 - Sempre que um nó é expandido, os custos armazenados nos ascendentes são actualizados
- Funciona como pesquisa em profundidade com retrocesso
 - Quando a folha m com menor custo $f(m)$ não é filha do último nó expandido n , então o algoritmo retrocede até ao ascendente comum de m e n

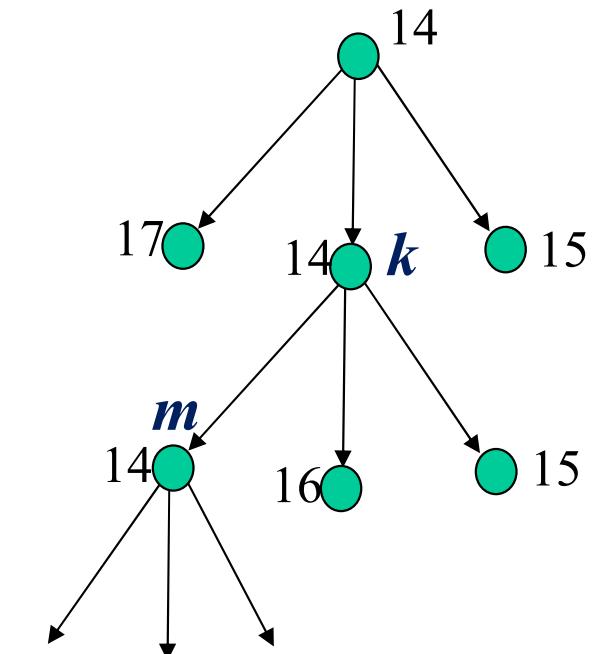
RBFS – exemplo



Nó n acaba de ser expandido



Custos foram actualizados

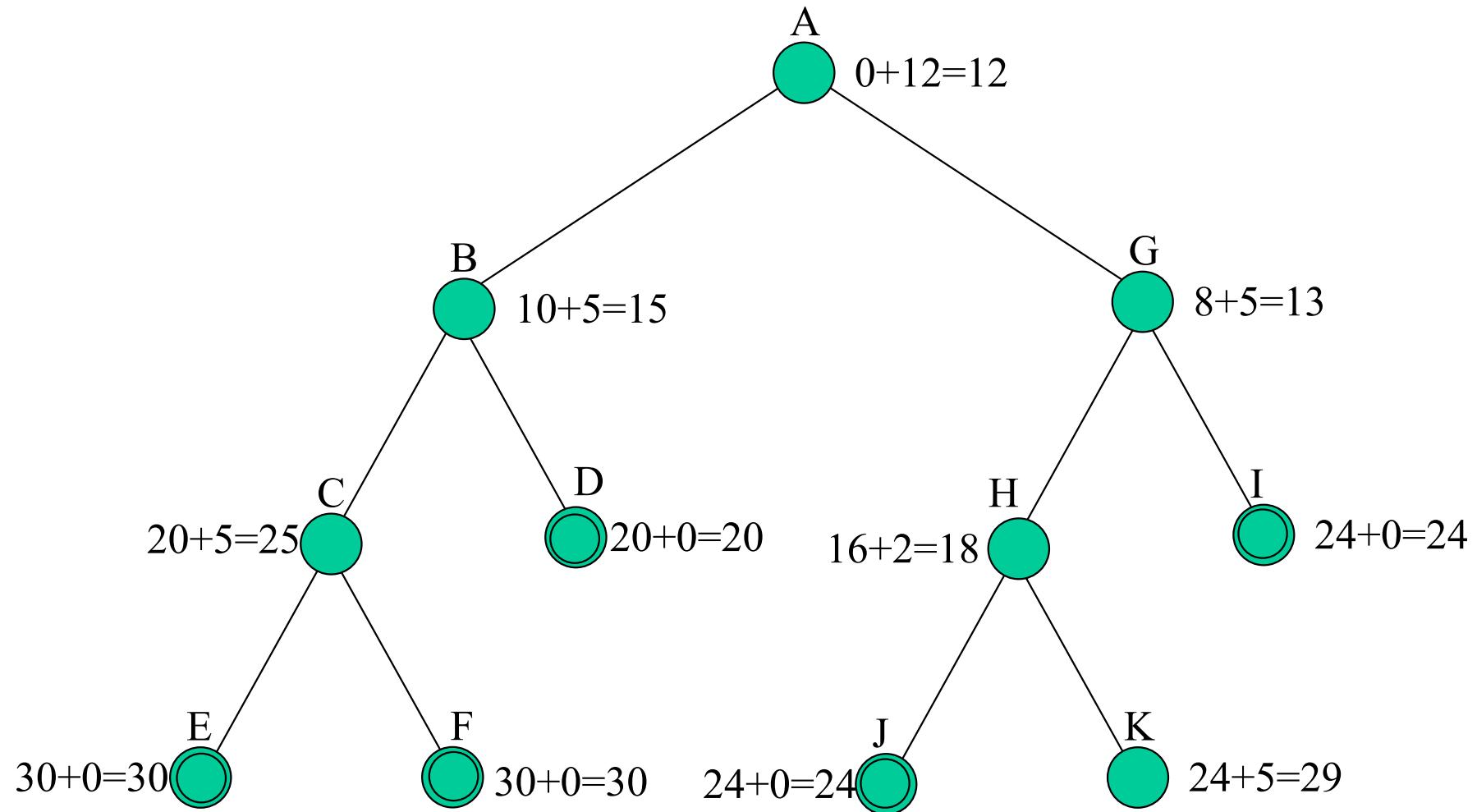


Algoritmo retrocedeu até ao nó k ; Expansão segue pelo nó m

SMA*

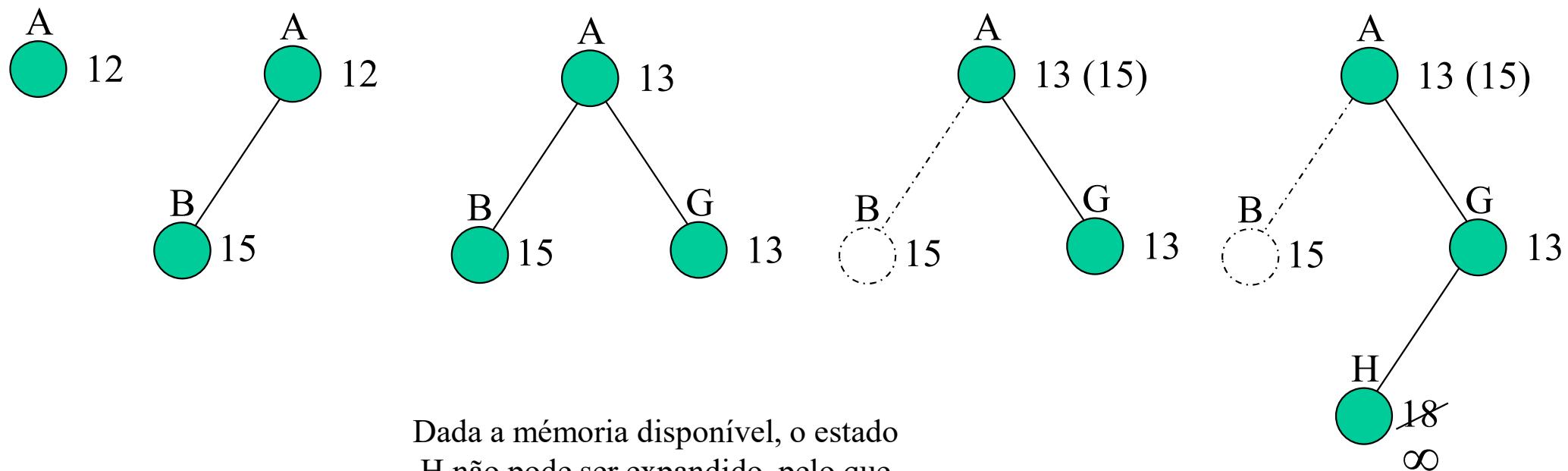
- A* com memória limitada simplificado (*simplified memory-bounded A**)
- Usa a memória disponível
 - Contraste com IDA* e RBFS: estes foram desenhados para poupar memória, independentemente de ela existir de sobra ou não
- Quando a memória chega ao limite, esquece (remove) o nó n com maior custo $f(n)=g(n)+h(n)$, actualizando em cada um dos nós ascendentes o “custo do melhor nó esquecido”
- Só volta a gerar o nó n quando o custo do melhor nó esquecido registado no antecessor de n for inferior aos custos dos restantes nós
- Em cada iteração, é gerado apenas um nó sucessor
 - Existindo já um ou mais filhos de um nó, apenas se gera ainda outro se o custo do nó pai for menor do que qualquer dos custos dos filhos
 - Quando se gerou todos os filhos de um nó, o custo do nó pai é actualizado como no RBFS

SMA* - exemplo – espaço de estados

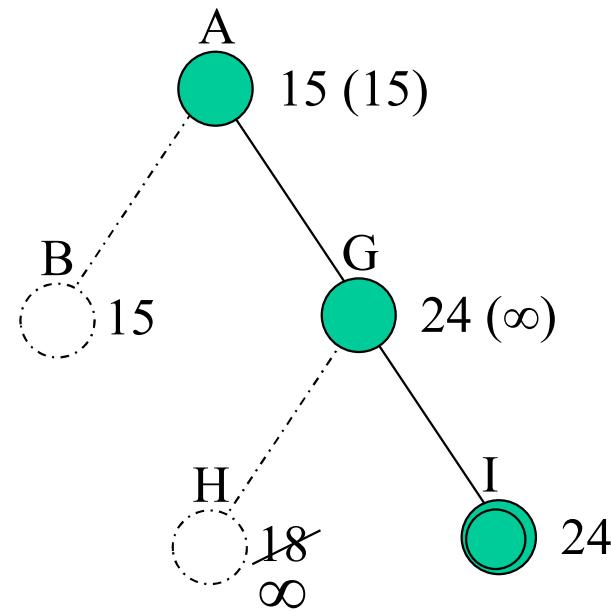
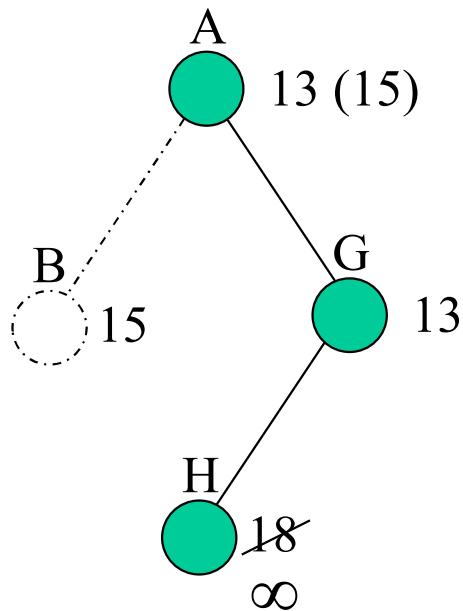


SMA* - exemplo

- Neste exemplo: memória = 3 nós
 - Melhores custos de nós esquecidos anotados entre parêntesis



SMA* - exemplo (cont.)



- Chegámos a uma solução (estado I)
- Se quisermos continuar: Das restantes folhas já exploradas, a que tinha o estado B era a melhor, por isso a pesquisa retrocede e continua expandindo esse folha

Tópicos de Inteligência Artificial

- Agentes
- Técnicas de resolução automática de problemas
 - Pesquisa em árvore no espaço de estados
 - Pesquisa com propagação de restrições
 - Pesquisa local
 - Planeamento
- Representação do conhecimento
- Tratamento probabilístico da incerteza

Pesquisa para problemas de atribuição

- Nos problemas de atribuição pretende-se atribuir valores a um conjunto de variáveis, respeitando um conjunto de restrições.
- Exemplos:
 - Problema das 8 rainhas - distribuir 8 rainhas num tabuleiro de xadrez de forma a que haja uma e uma só rainha em cada linha e em cada coluna e não haja mais do que uma rainha em cada diagonal.
 - Invenção de palavras cruzadas – dada uma matriz de palavras cruzadas vazia, preencher os espaços brancos com letras, de forma a que a matriz possa ser usada como passatempo de palavras cruzadas.
- Técnicas de resolução de problemas de atribuição:
 - Método construtivo – usando técnicas de pesquisa em árvore
 - Em cada passo da pesquisa atribui-se um valor a uma variável
 - Método construtivo combinado com propagação de restrições
 - Resolução por melhorias sucessivas

Pesquisa com propagação de restrições em problemas de atribuição

- Construir um grafo de restrições:
 - Em cada nó do grafo está uma variável
 - Um arco dirigido liga um nó i a um nó j se o valor da variável de j impõe restrições ao valor da variável de i .
 - Um arco (i,j) é *consistente* se, para cada valor da variável i , existe um valor da variável j que não viola as restrições.
- Tipicamente, usa-se uma estratégia de pesquisa em profundidade; em cada iteração da pesquisa, faz-se o seguinte:
 - 1) Seleciona-se arbitrariamente um dos valores possíveis para uma das variáveis (descartam-se os restantes)
 - 2) Restringem-se os conjuntos de valores possíveis das restantes variáveis por forma a que os arcos do grafo de restrições continuem consistentes.
- Nota: Neste caso, cada estado da pesquisa não representa uma situação ou configuração possível do mundo, como acontece no problema dos blocos; o estado é constituído pelos conjuntos de valores possíveis para as variáveis.

Pesquisa com propagação de restrições em problemas de atribuição - algoritmo

1. Inicialização: o nó inicial da árvore de pesquisa é composto por todas as variáveis e todos os valores possíveis para cada uma delas
2. Se pelo menos uma variável tem um conjunto de valores vazio, falha e retrocede; se não puder retroceder, a pesquisa falha
3. Se todas as variáveis têm exactamente um valor possível, tem-se uma solução; retornar com sucesso
4. Expansão: Escolher arbitrariamente uma variável V_k e, de entre os valores possíveis, um dado valor X_{kl} – descartar os restantes valores possíveis dessa variável
5. Propagação de restrições: para cada arco (i,j) no grafo de restrições, remover os valores na variável V_i por forma a que o arco fique consistente
6. Caso tenha sido preciso remover valores na origem de algum arco, voltar a repetir o passo 5.
7. Voltar ao passo 2.

Propagação de restrições - algoritmo

- Os passos 5 e 6 do algoritmo anterior executam a propagação de restrições
- Esta parte do processo é suportada por uma fila de arestas do grafo de restrições
 - Inicialmente, a fila contém as arestas que apontam para a variável cujo valor foi fixado

```
propagarRestricoes(grafoRestricoes,FilaArestas) retorna o grafo de restrições com
domínios possivelmente mais limitados
enquanto FilaArestas não vazia fazer {
    ( $X_j, X_i$ )  $\leftarrow$  remover cabeça de FilaArestas
    remover valores inconsistentes em  $X_j$ 
    se removeu valores, então
        para cada vizinho  $X_k$ , acrescentar ( $X_k, X_j$ ) a FilaArestas
}
```

Tipos de restrições

- Restrições unárias – envolvem apenas uma variável
 - Podem ser satisfeitas através de pré-processamento do domínio de valores da variável – aproveitam-se apenas os valores que satisfazem a restrição
- Restrições binárias – envolvem duas variáveis
 - Uma restrição binária é directamente representada por uma aresta no grafo de restrições
- Restrições de ordem superior – envolvem três ou mais variáveis
 - Através da introdução de variáveis auxiliares, uma restrição de ordem superior pode ser transformada num conjunto de restrições binárias e/ou unárias

Exemplo: quebra-cabeças critpoaritmético

$$\begin{array}{r} \text{T} \ \text{W} \ \text{O} \\ + \ \text{T} \ \text{W} \ \text{O} \\ \hline \text{F} \ \text{O} \ \text{U} \ \text{R} \end{array}$$

- Variáveis principais: F, O, R, T, U, W ($\in \{ 0 .. 9 \}$)
 - Todas as variáveis são diferentes
- Variáveis internas: X_1 (transporte das unidades para as dezenas) e X_2 (transporte das dezenas para as centenas) ($\in \{ 0, 1 \}$)
- Restrições:
 - $2 \cdot O = R + 10 \cdot X_1$ [restrição sobre 3 variáveis]
 - $2 \cdot W + X_1 = U + 10 \cdot X_2$ [restrição sobre 4 variáveis]
 - $2 \cdot T + X_2 = O + 10 \cdot F$ [restrição sobre 4 variáveis]

Restrições de ordem superior – conversão para restrições binárias

- No exemplo anterior, a restrição ternária $2 \cdot O = R + 10 \cdot X_1$ pode ser transformada no seguinte conjunto de restrições:
 - Restrições binárias:
 - $O = \text{primeiro}(Aux)$
 - $R = \text{segundo}(Aux)$
 - $X_1 = \text{terceiro}(Aux)$
 - Restrição unária:
 - $2 \cdot \text{primeiro}(Aux) = \text{segundo}(Aux) + 10 \cdot \text{terceiro}(Aux)$
- Aux é uma variável auxiliar cujo domínio é o produto cartesiano dos domínios de O , R e X_1 .
 - Ou seja: $Aux \in \{0..9\} \times \{0..9\} \times \{0,1\}$
 - A restrição unária sobre Aux pode ser satisfeita através de pré-processamento

Tópicos de Inteligência Artificial

- Agentes
- Técnicas de resolução automática de problemas
 - Pesquisa em árvore no espaço de estados
 - Pesquisa com propagação de restrições
 - Pesquisa local
 - Planeamento
- Representação do conhecimento
- Tratamento probabilístico da incerteza

Diferentes técnicas

- Atingir um objectivo a partir de um estado inicial
 - Pesquisa em árvore
 - No espaço de estados
- Atribuir valores a variáveis satisfazendo restrições
 - Pesquisa em árvore, com propagação de restrições
 - No espaço de soluções, começando com solução vazia
- Vamos ver agora: optimizar uma dada solução
 - Pesquisa local (não em árvore)
 - No espaço de soluções

Pesquisa local

- Também conhecida como **pesquisa por melhorias sucessivas**
 - A partir de uma dada configuração inicial, aplicam-se melhorias ou refinamentos sucessivos até obter uma configuração satisfatória
 - A solução inicial pode ser aleatória
- Técnicas mais comuns:
 - Reparação heurística
 - É a versão mais básica deste tipo de pesquisa: reparações à solução inicial vão sendo aplicadas de acordo com uma heurística local.
 - No caso de problemas de satisfação de restrições, a heurística pode ser:
 - Fazer a reparação que, naquele momento, mais contribui para reduzir os conflitos entre variáveis, dadas as restrições.
 - Montanhismo, reconhecimento simulado, algoritmos genéticos, etc.

Pesquisa local: montanhismo

- A pesquisa é vista como um problema de optimizar uma função
- O espaço de soluções é visto como uma paisagem de vales (zonas de soluções menos satisfatórias) e colinas (zonas de soluções melhores).
- Tem semelhanças com a pesquisa em profundidade e com a pesquisa gulosa, diferenciando-se pelo seguinte:
 - Escolhe-se sempre o sucessor com melhor valor da função de avaliação
 - Não há retrocesso (*backtracking*)
 - Quando o valor da função no nó actual é superior ao valor da função em qualquer dos seus sucessores, a pesquisa pára. (atingiu-se um máximo local)
- Problemas:
 - Máximos locais, planaltos, ravinas

Pesquisa local: variantes do montanhismo

- **Montanhismo estocástico** – escolhe aleatoriamente entre os sucessores que melhoraram a função de avaliação
- **Montanhismo de primeira escolha** – escolhe sucessores aleatoriamente até encontrar um com melhor função de avaliação que o estado actual
- **Montanhismo com reinício aleatório** – executar o montanhismo várias vezes, partindo de estados iniciais aleatórios, e escolhe a melhor solução
- **Recozimento simulado** (página seguinte)

Pesquisa local: **recozimento simulado**

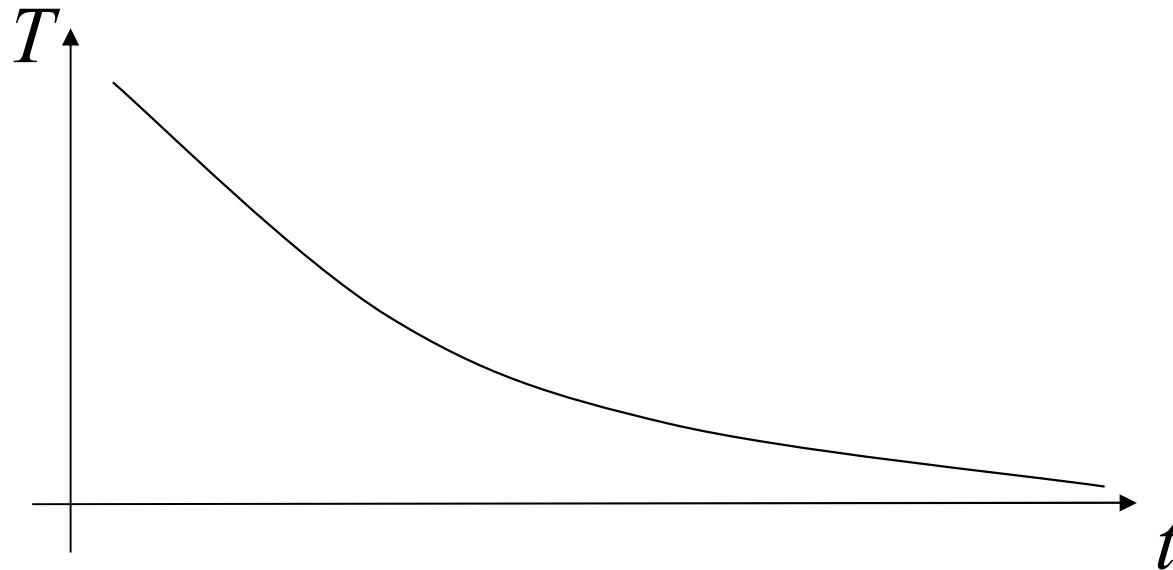
- *Recozimento simulado (Simulated Annealing)* é uma variante da pesquisa por montanhismo na qual podem ser aceites refinamentos que, localmente, piorem a solução.
 - O nome inspira-se no processo industrial chamado *recozimento*.
 - Recozer = “deixar esfriar lentamente (um produto de cerâmica ou de vidro) num forno especial, logo após o seu fabrico”.
- Começou a ser usado circa 1980 para resolver problemas de configuração de circuitos VLSI
- Particularidades:
 - O sucessor é seleccionado aleatoriamente
 - Quando o valor da função no nó actual é superior ao valor da função no sucessor, o sucessor é aceite com uma probabilidade que diminui exponencialmente em função da perda na função de avaliação.
 - Pesquisa termina quando um indicador designado “temperatura” chega a zero.

Recozimento simulado: algoritmo

```
recozimento_simulado(Problema,Regime_termico,Aval)
(* A função Regime_termico dá a temperatura em função do tempo. *)
Nó ← fazer_nó(estado inicial do Problema)
repetir para  $t=0 \dots \infty$ : {
     $T \leftarrow \text{Regime\_termico}(t)$ 
    se  $T=0$ , retornar a solução de Nó
     $\text{Prox} \leftarrow \text{um sucessor de } \text{Nó \ gerado aleatoriamente}$ 
     $\text{Ganho} \leftarrow \text{Aval}(\text{Prox}) - \text{Aval}(\text{Nó})$ 
    se  $\text{Ganho} > 0$ ,  $\text{Nó} \leftarrow \text{Prox}$ 
    senão, com probabilidade  $\exp(\text{Ganho}/T)$ , fazer:  $\text{Nó} \leftarrow \text{Prox}$ 
}
```

- Nota: Se a temperatura T diminuir de forma suficientemente lenta, o recozimento simulado encontra um máximo global (solução óptima) com uma probabilidade que tende para 1.

Recozimento simulado: regime térmico

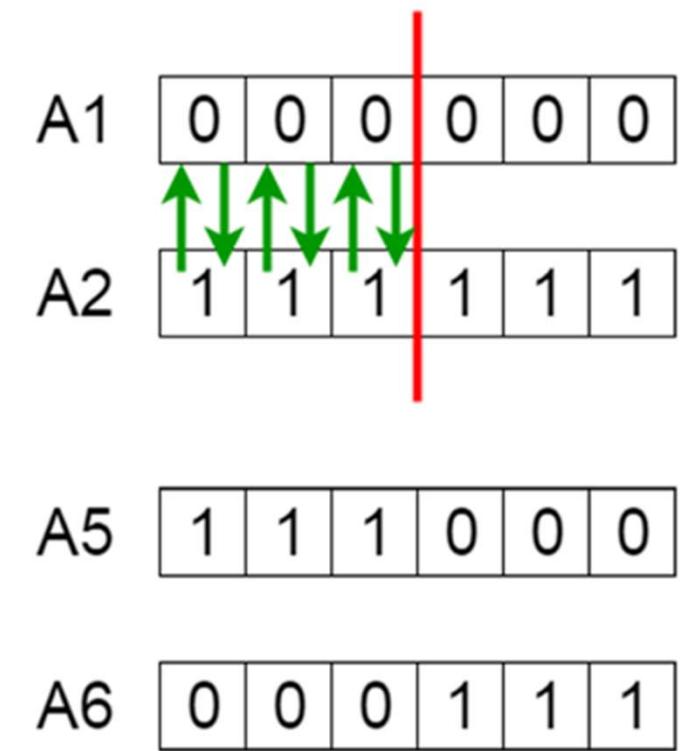
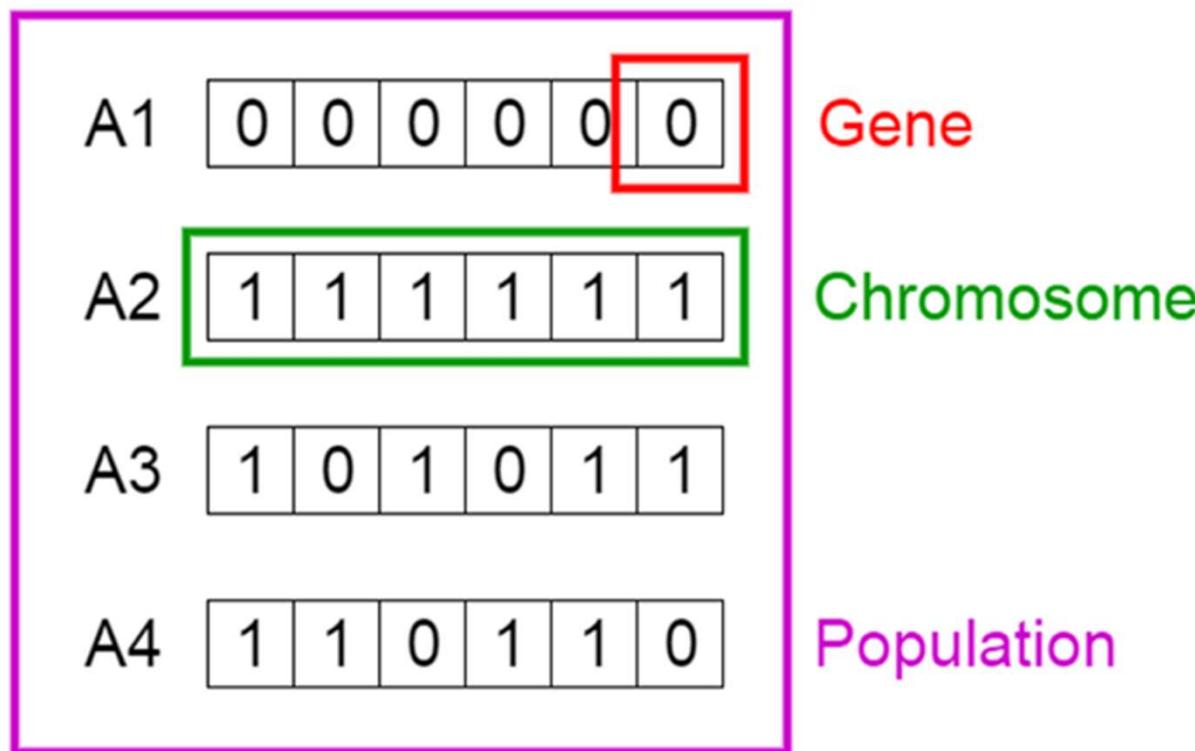


- $t \rightarrow \infty$
- $T \rightarrow 0$
- $Ganho/T \rightarrow -\infty$ (dado que o Ganho é negativo)
- Probabilidade: $\exp(Ganho/T) \rightarrow 0$
- Ou seja: À medida que o tempo passa, a pesquisa arrisca cada vez menos quanto a aceitar alterações com ganho negativo

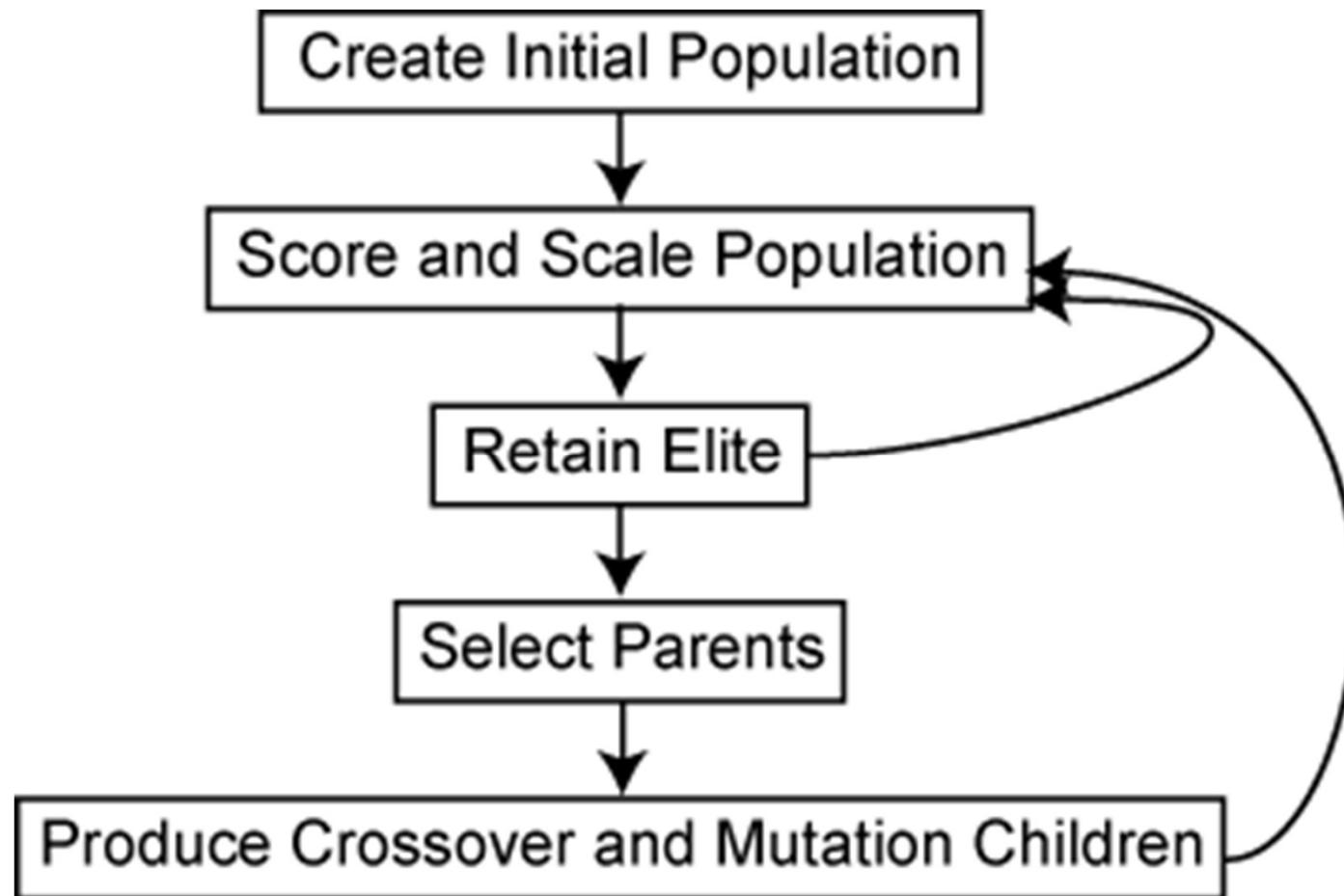
Pesquisa local alargada (*local beam search*)

- **Pesquisa local alargada** – semelhante ao montanhismo mas, em cada iteração, são mantidos k estados, e os melhores k sucessores são passados para a iteração seguinte [NOTA: podem ser seleccionados vários sucessores de alguns dos k estados e nenhum sucessor de alguns dos outros]
- **Pesquisa alargada estocástica** – semelhante à pesquisa local alargada, mas os k sucessores são seleccionados aleatoriamente
- **Algoritmos genéticos** – variante da pesquisa alargada estocástica em que os sucessores são gerados por combinação de dois estados, e não apenas por modificação de um único estado

Algoritmos Genéticos (1)



Algoritmos Genéticos (2)



Tópicos de Inteligência Artificial

- Agentes
- Técnicas de resolução automática de problemas
 - Pesquisa em árvore no espaço de estados
 - Pesquisa com propagação de restrições
 - Pesquisa por melhorias sucessivas
 - Planeamento
- Representação do conhecimento
- Tratamento probabilístico da incerteza

Planeamento no espaço de soluções

- Partindo de um dado estado inicial do mundo, podemos executar pesquisa em árvore no espaço de estados de planeamento, correspondendo cada nó da pesquisa a um estado → *planeamento no espaço de estados*.
- Uma técnica alternativa consiste em partir de um plano vazio e adicionar sucessivamente operações e restrições de sequenciamento → *planeamento no espaço de soluções*.
- Neste caso, cada nó da pesquisa corresponde a uma solução parcial para o problema.
- Operações de transformação da solução:
 - Adicionar um operador
 - Re-ordenar operadores
 - Instanciar um operador

Planeamento Hierárquico – a técnica ABSTRIPS

- O planeamento é realizado numa hierarquia de níveis de abstracção
- Um valor de “criticalidade” é atribuído a cada uma das condições que podem aparecer na descrição do estado do mundo
- Algoritmo:
 1. $CM \leftarrow$ valor inicial para o nível de criticalidade mínima.
 2. Gerar um um plano que satisfaça todas as condições com nível de criticalidade $\geq CM$.
 3. $CM \leftarrow CM-1$
 4. Usando o plano anterior como guia, gerar um plano que satisfaça todas as condições com criticalidade $\geq CM$.
 5. **se** todas as condições estão satisfeitas, **retornar** a solução.
 6. **voltar** ao passo 3.

Operadores com fórmulas não atómicas e condicionais

- Literal – uma formula atómica (literal positivo) ou negação de uma fórmula atómica (literal negativo)
- Fórmula de aplicabilidade do operador pode ser:
 - Fórmula atómica
 - Negação de uma fórmula
 - Conjunção de fórmulas
 - Disjunção de fórmulas
 - Fórmula quantificada existencialmente
 - Fórmula quantificada universalmente
- Fórmula de efeitos do operador pode ser:
 - Literal
 - Conjunção de literais
 - Efeitos condicionais: when <fórmula de aplicabilidade> <fórmula de efeitos>
 - Fórmula de efeitos quantificada universalmente
 - Conjunção de fórmulas de efeitos
- Ver “PDDL - Planning Domain Definition Language”.

PDDL - exemplo

```
(:action stop
:parameters (?f - floor)
:precondition (lift-at ?f)
:effect (and
          (forall (?p - passenger)
                 (when (and (boarded ?p) (destin ?p ?f) )
                           (and (not (boarded ?p)) (served ?p))))
          (forall (?p - passenger)
                 (when (and (origin ?p ?f) (not (served ?p)))
                           (boarded ?p))))))
```

PDDL - exemplo

```
(:action drive-truck
:parameters (?truck - truck
            ?loc-from ?loc-to - location
            ?city - city)
:precondition (and (at ?truck ?loc-from) (in-city ?loc-from ?city)
                    (in-city ?loc-to ?city))
:effect (and (at ?truck ?loc-to)
              (not (at ?truck ?loc-from))
              (forall (?x - obj)
                     (when (and (in ?x ?truck))
                           (and (not (at ?x ?loc-from))
                                (at ?x ?loc-to)))))))
```