

Busca Heurística



49

Busca Heurística

- Muitos problemas possuem espaços de busca que são muito grandes para serem examinados completamente.
- É possível construir estratégias que não prometem a melhor solução, mas que encontram uma “boa” resposta rapidamente.

50

Heurística

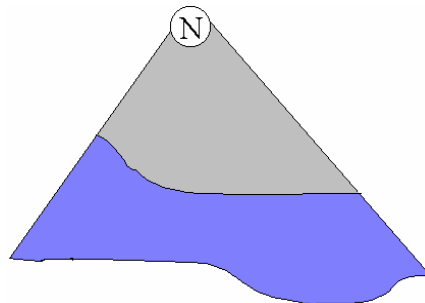
☛ Sobre busca Heurística:

- estimativa numérica da qualidade dos nós (em termos de possibilidade de dirigir a busca para uma solução)
- execução da busca sobre o nó mais promissor
- Avaliação dos nós pode ser baseada em uma informação incompleta.
- Uma *heurística* é usada para guiar o processo de busca.

51

Busca Heurística

- ## ☛ Estratégias de Busca Heurística usam
- informação do domínio para limitar a busca sobre áreas onde podem existir soluções.



52

Exercício

🦉 Crie heurísticas para:

- o que estudar para uma prova
- como resolver uma prova com questões objetivas
- prever qual será o clima de amanhã

53

Heurísticas para o problema do Caixeiro Viajante

🦉 Heurística do vizinho mais próximo:

1. Selecione qualquer cidade inicial
2. Enquanto houver cidades
escolha a cidade mais próxima à cidade corrente.

54

Utilização de Busca Heurística

- A solução não precisa ser ótima (qualquer solução é uma boa solução).
- O pior caso é raro.
- Entendimento de quando e como a busca heurística trabalha leva a um melhor entendimento do problema.

55

Usando conhecimento heurístico em um sistema baseado em regras

- Conhecimento Heurístico pode ser codificado diretamente em regras. Ex.: No problema dos jarros de água - não colocar água em um jarro cheio
- Pode ser uma função heurística que avalia estados do problema e impõe uma ordenação
- Xadrez - qual movimento deve ser explorado primeiro

56

Características de Problemas

• É útil identificar características para que um conjunto formado por estratégias aplicadas em problemas com características semelhantes possa ser obtido

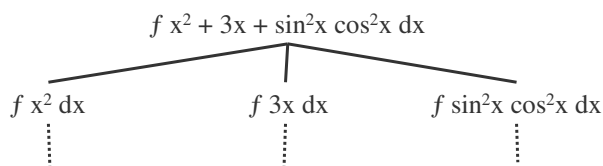
- Decomposição
- Passos ignorados ou desfeitos
- O universo é previsível?
- Precisa de uma solução ótima?
- A solução é um estado ou um caminho?
- Qual o papel do conhecimento?
- Humanos podem interagir com o sistema?

57

1. Decomposição

• O problema pode ser quebrado (decomposto) em vários problemas independentes de tamanho menor ?

• Sim - recursão pode ser utilizada, apenas os pequenos problemas são atacados diretamente



58

2. Passos podem ser ignorados ou desfeitos?

• Classes de problemas:

- Ignoráveis - passos da solução podem ser ignorados (a estratégia de busca não muda)
- Recuperáveis - passos podem ser desfeitos (retrocesso)
- Irrecuperáveis - passos não podem ser desfeitos

59

3. O Universo é previsível ?

- Algumas vezes o sistema controla completamente os estados e sabe exatamente a desvantagem de cada operação
- Outras vezes a desvantagem não é completamente previsível

60

4. Uma boa solução é relativa ou absoluta?

- Únicos vs. múltiplos caminhos para uma única solução
- Na busca pela solução absoluta não se pode utilizar heurística (todos os caminhos possuem possibilidades iguais e precisam ser percorridos)

61

5. A solução é um estado ou um caminho?

- Algumas vezes a solução é apenas um estado do espaço de estados
- Algumas vezes o caminho para o estado final é a resposta
- Todos os problemas podem ser reformulados para que a resposta seja apenas um estado

62

6. Qual é o papel do conhecimento?

- Alguns problemas usam o conhecimento para restringir a busca (a estratégia de busca faz parte do conhecimento)
- Outros problemas necessitam de conhecimento para avaliar estados (poder reconhecer uma solução)

63

7. Humanos podem interagir com o processo de busca?

- Algumas vezes humanos podem ajudar
- Humanos tendem a não confiar em sistemas se não entendem a lógica de como as respostas foram obtidas

64

Tipos de Problemas

- Várias características podem ser utilizadas para classificar problemas
- Existem técnicas adequadas para cada tipo de problema
- Exemplos de problemas genéricos:
 - classificação
 - Propor e refinar (problemas de projeto e planejamento)

65

Programas de Busca

- Busca pode ser vista como a travessia de uma árvore
- Árvore é formada pelas regras
- Representação Implícita vs. Explícita
- Algumas vezes é melhor pensar em busca como um grafo dirigido

66

Técnicas de Busca Heurística

Capítulo 3

- Gerar-e-testar
- Subida da Encosta
- Busca Best-First
- Redução de Problemas
- Satisfação de Restrições
- Análise Meios-Fins

67

Técnicas de Busca Heurística

- *Técnicas Diretas* nem sempre estão disponíveis (requerem muito tempo e memória)
- *Técnicas Fracas* podem ser suficientes se aplicadas corretamente (sobre o tipo correto de tarefas)
- Técnicas podem ser descritas genericamente

68

Gerar-e-testar

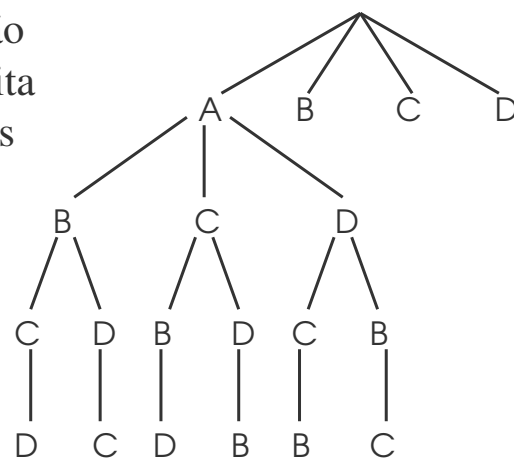
- Estratégia muito simples - baseada em previsão
 - enquanto a meta não for alcançada faça
 - gere uma solução possível
 - compare a solução com a meta
- Heurísticas podem ser usadas para determinar as regras específicas para geração de soluções

69

Exemplo de Gerar-e-Testar

- Caixeiro Viajante - geração de possíveis soluções é feita na ordem lexicográfica das cidades:

- 1. A - B - C - D
- 2. A - B - D - C
- 3. A - C - B - D
- 4. A - C - D - B
- ...



70

Problema dos Jarros de Água

- | | |
|--|--------------------|
| 1. $(x,y), x < 4 \rightarrow (4,y)$ | enche o jarro 4 |
| 2. $(x,y), y < 3 \rightarrow (x,3)$ | enche o jarro 3 |
| 3. $(x,y), x > 0 \rightarrow (0,y)$ | esvazia o jarro 4 |
| 4. $(x,y), y > 0 \rightarrow (x,0)$ | esvazia o jarro 3 |
| 5. $(x,y), x+y \geq 4, y > 0 \rightarrow (4, y-(4-x))$ | enche 4 com 3 |
| 6. $(x,y), x+y \geq 3, x > 0 \rightarrow (x-(3-y), 3)$ | enche 3 com 4 |
| 7. $(x,y), x+y \leq 4, y > 0 \rightarrow (x+y, 0)$ | conteúdo de 3 em 4 |
| 8. $(x,y), x+y \leq 3, x > 0 \rightarrow (0, x+y)$ | conteúdo de 4 em 3 |
| 9. $(0,2) \rightarrow (2,0)$ | 2 lt. de 3 para 4 |
| 10. $(x,2) \rightarrow (0,2)$ | esvazia 4 |

71

Subida da Encosta

• Variação sobre gerar e testar:

- *geração* do próximo estado depende do *procedimento de teste*
- *Teste* agora inclui uma função heurística que fornece uma previsão de quão bom é cada estado

• Há várias formas de utilizar a informação retornada pelo *procedimento de teste*

72

Subida da Encosta Simples

- Usa heurística para mudar para estados que são *melhores* que o estado corrente
- Sempre muda para o melhor estado quando possível
- O processo termina quando todos os operadores tiverem sido aplicados e nenhum dos estados resultantes são melhores que o estado corrente

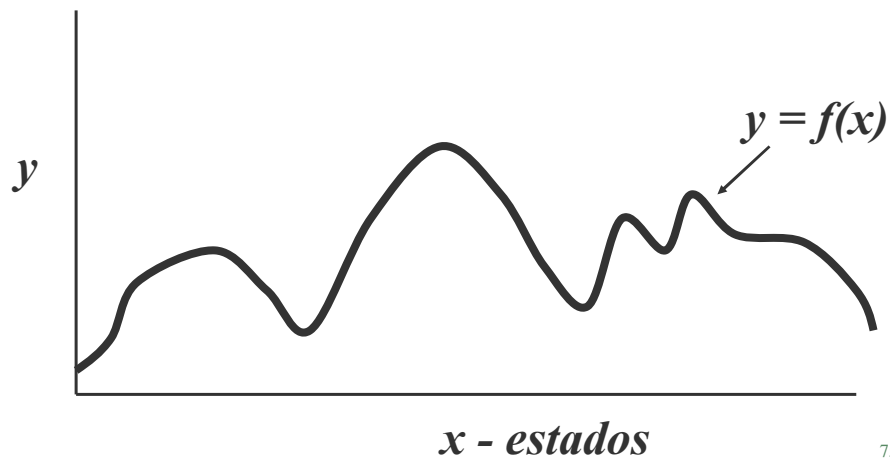
73

Subida da Encosta Simples - Algoritmo

- Se o estado inicial é a meta **então** termine
- **senão** *corrente* = estado inicial
- **enquanto** solução não encontrada **ou** não há novos operadores a serem aplicados
 - selecione um operador não aplicado sobre *corrente* para produzir um novo estado *novo*
 - **se** *novo* é meta **então** termine
 - **senão se** *novo* é melhor do que *corrente* **então**
corrente = *novo*
- **fim_enquanto**

74

Subida da Encosta Função de Otimização



75

Potenciais Problemas com Subida da Encosta

- Terminar o processo em um ponto ótimo local
- A ordem de aplicação dos operadores pode fazer uma grande diferença
- Não pode ver estados anteriores ao estado corrente

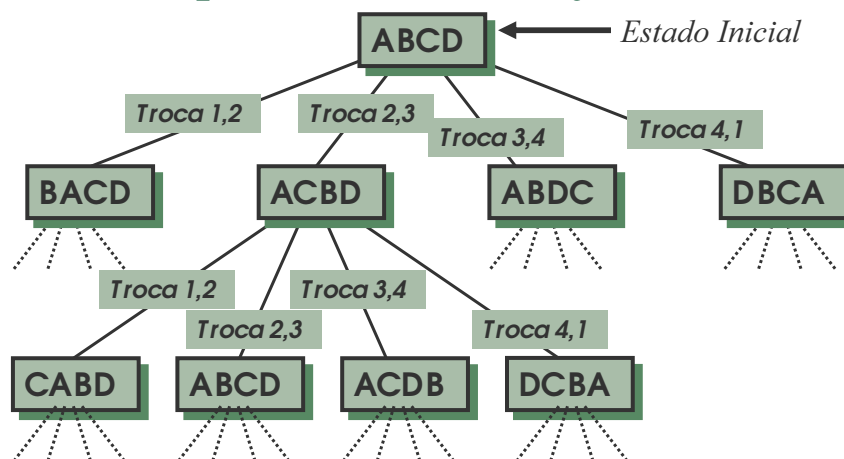
76

Exemplo de Subida da Encosta Simple

- Caixeiro Viajante - defina o espaço de estados como todos os trajetos possíveis
- Operadores trocam o estado de posição para cidades adjacentes em um trajeto
- Função Heurística é o tamanho do trajeto

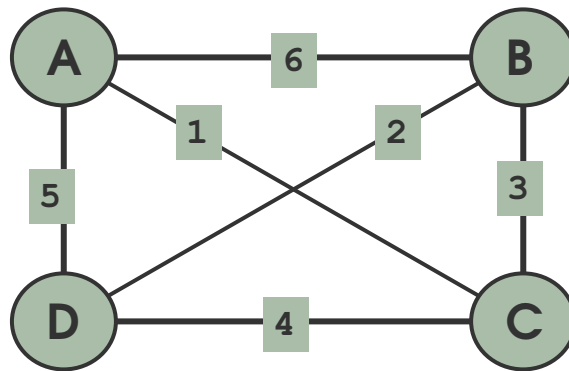
77

Espaço de Estados na Subida da Encosta para Caixeiro Viajante



78

Exemplo Caixeiro Viajante



79

Subida da Encosta pela Trilha mais Íngreme

- Uma variação da Subida da Encosta simples
- Ao invés de mover para o *primeiro* estado melhor, move-se para o melhor estado possível
- A ordem dos operadores não importa
- Como na subida simples, também não garante a subida para o melhor estado

80

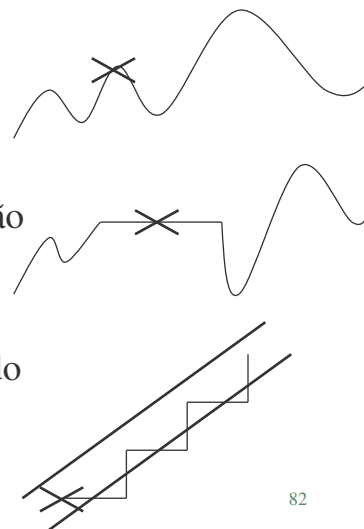
Subida da Encosta pela Trilha mais Íngreme - Algoritmo

- Se o estado inicial é a meta **então** termine
- **senão** *corrente* = estado inicial
- **enquanto** solução não encontrada **ou** uma iteração completa não modifique o estado *corrente*
 - *SUCC* = um estado tal que qualquer sucessor de *corrente* seja melhor que *SUCC*
 - **para** cada operador aplicável a *corrente* **faça**
 - aplique o operador e gere um novo estado *novo*
 - **se** *novo* é um estado meta **então** termine
 - **senão** se *novo* é melhor que *SUCC* então *SUCC* = *novo*
 - **fim_para**
 - Se *SUCC* é melhor que *corrente* então *corrente* = *SUCC*
- **fim_enquanto**

81

Perigos da Subida da Encosta

- Máximo Local: todos os estados vizinhos são piores que o atual
- Platô - todos os estados vizinhos são iguais que o atual
- Cume - máximo local que é causado pela incapacidade de aplicar 2 operadores de uma vez



82

Dependência Heurística

- Subida da encosta está baseada no valor associado aos estados por uma função heurística
- A heurística usada pelo algoritmo subida da encosta não precisa ser uma função estática de um simples estado
- A heurística pode indicar muitos estados ou pode usar outros meios para obter um valor para um estado

83

Cozimento Simulado

- Baseado no processo físico de cozimento de metal para obter o melhor estado (mínima energia)
- Subida da Encosta com modificações:
 - permite movimentos para baixo (para estados piores)
 - inicia permitindo grandes decidas (para estados piores) e gradualmente permite apenas pequenos movimentos de descida

84

Cozimento Simulado (Cont.)

- A busca inicialmente faz grandes saltos, explorando muitas regiões do espaço
- Os saltos são gradualmente reduzidos e a busca torna-se uma subida de encosta simples (busca por um ótimo local)

85

Cozimento Simulado (Cont.)

- O movimento dos estados é feito a partir dos estados com maior energia para os estados de menor energia
- Existe uma probabilidade de ocorrer um movimento para um estado de maior energia, calculada por:

$$p = e^{-\Delta E / kT}$$

- ΔE é a mudança positiva do nível de energia
- T é a temperatura
- k é a constante de Boltzmann

86

Cozimento Simulado – Algoritmo Simplificado

```
➤ candidato <- s0;  
  T <- T0;  
  repita  
➤ próximo <- vizinho de 'candidato' tomado aleatoriamente;  
  deltaE <- energia(próximo) - energia(candidato);  
  se deltaE <= 0  
    então candidato <- próximo  
    senão faça candidato <- próximo com probabilidade  $\exp(-\text{deltaE}/T)$ ;  
  T <- próximaTemperatura(T);  
➤ até T < Tfinal  
  retorna candidato;
```

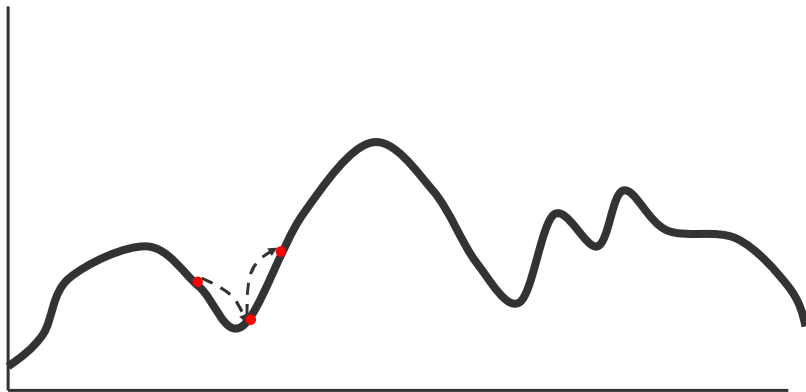
➤ onde:

- s0: estado (candidato a mínimo) inicial;
- T0/Tfinal: temperatura inicial/final;
- próximaTemperatura: função que calcula a temperatura vigente na próxima iteração;

Principais diferenças entre Cozimento Simulado e Subida da Encosta

- O cronograma da têmpora precisa ser mantido
- Movimentos para estados piores podem ser mantidos
- É interessante armazenar o melhor estado já encontrado até o momento (caso o estado final seja pior devido à escolha feita pelo algoritmo)

Cozimento Simulado (Cont.)



89

Cozimento Simulado - Algoritmo

- Se o estado inicial é a meta **então** termine
- **senão** *corrente* = estado inicial e *MELHOR_MOMENTO* = *corrente*
- Inicialize *T*
- **enquanto** solução não encontrada **ou** não houver mais operadores aplicáveis a *corrente*
 - selecione um operador não aplicado a *corrente* e gere o estado *novo*
 - *Delta* = (valor do *corrente*) - (valor do *novo*)
 - **se** o *novo* é meta **então** termine
 - **senão se** *novo* é melhor que *corrente*
 - **então** *corrente* = *BEST_MOMENTO* = *novo*
 - **senão** *corrente* = NOVO com probabilidade p'
 - Recalcule *T*
- **fim_enquanto**
- **Retorne** *MELHOR_MOMENTO* como resposta

90

Busca Best-First

- Combina as vantagens das buscas em Largura e Profundidade
 - Profundidade: segue um caminho único, não precisa gerar todos os possíveis caminhos
 - Largura: não tem problemas com loops ou caminhos sem solução
- Busca Best First : explora o caminho mais promissor visto

91

Busca Best-First (Cont.)

- **enquanto** meta não alcançada:
 - Gere todos os potenciais estados sucessores e os adicione a uma lista de estados
 - Escolha o melhor estado na lista e realize busca sobre ele
- **fim_enquanto**
- Similar a encosta mais íngreme, mas não realiza busca em caminhos não escolhidos

92

Best First - Avaliação dos Estados

- Valor de cada estado é uma combinação de:
 - o custo do caminho para o estado
 - custo estimado de alcançar uma meta a partir um estado.
- A idéia é usar uma função para determinar (parcialmente) o ranking dos estados quando comparados a outros estados.
- Isto não ocorre em BL ou BF, mas é útil para busca Best-First.

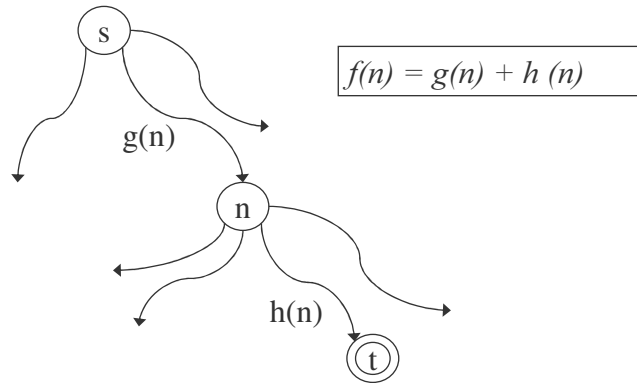
93

Busca Best First - Por que é necessária a avaliação

- Considere uma busca best-first que gera o mesmo estado muitas vezes.
- Quais dos caminhos que levam ao estado é o melhor?
- Lembre-se de que o caminho para uma meta pode ser a resposta (por exemplo, o problema dos jarros de água)

94

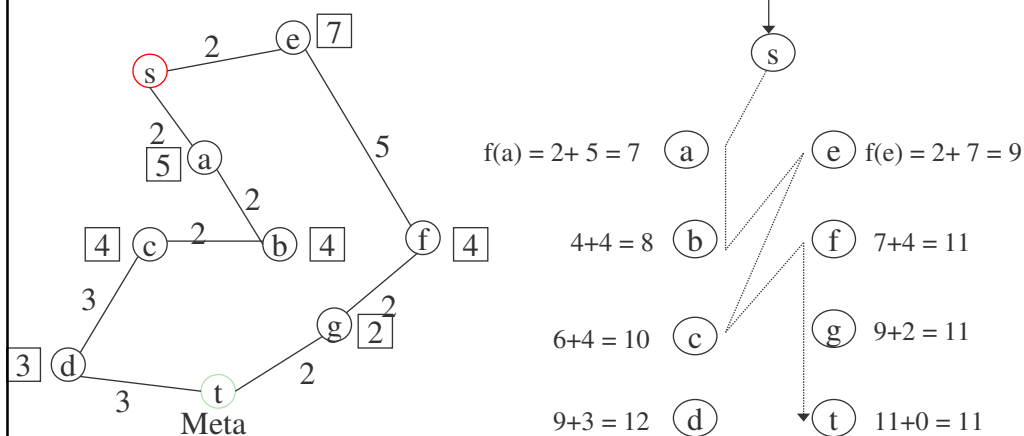
Busca Best-First



- $f(n)$ é uma função que estima o valor heurístico do nó n .
- s é o nó inicial, t é uma solução

95

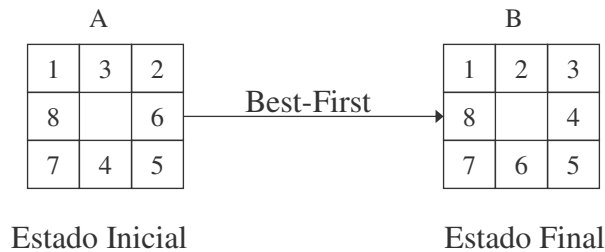
Busca Best-First - Caixeiro Viajante



- \square Representa a distância do nó até a solução (em linha reta)
- Os arcos representam as distâncias entre os nós

96

Best-First Aplicada ao Puzzle



Exercício:

Pense numa heurística que pode ser utilizada na busca Best-First para se encontrar uma solução rapidamente

97

Best-First Aplicada ao Puzzle - Solução do Exercício Anterior

🐼 Distancia_Total = somatório da diferença das distâncias do n. de cada quadrado e sua posição correta. Ex.: Na figura A do slide anterior

$$DT = |1 - 1| + |3 - 2| + |2 - 3| + |6 - 4| + |5 - 5| + |4 - 6| + |7 - 7| + |8 - 8| = 6$$

Sequência = somatório dos pesos associados a cada quadrado da seguinte forma:

- se há valor no centro então some 1
- se o quadrado não é central e é seguido pelo seu sucessor e ambos NÃO estão nas posições corretas, some 0
- se o quadrado não é central e não é seguido pelo seu sucessor e ambos ESTÃO na posição correta, some 2

$$Seq = (0) + (0) + (2 + 2 + 2 + 2 + 2 + 2) = 12$$

$H = DT + 3 * seq$

98

Algoritmo A*

- O Algoritmo A* usa uma função de avaliação e a busca Best-First.
- A* minimiza o custo total do caminho.
- Sob as condições corretas A* fornece uma solução com custo menor em tempo ótimo!

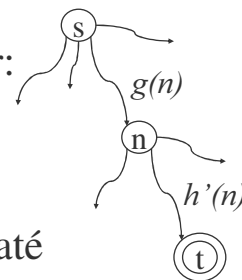
99

Função de Avaliação A*

- A função de avaliação f' é uma estimativa do valor de um nó dada por:

$$f'(x) = g(x) + h'(x)$$

- $g(x)$ é o custo do estado inicial até x .
- $h'(x)$ é o custo estimado a partir de x até o estado meta (heurística).



100

Algoritmo A*

♣ Idéia Geral:

- loops são desconsiderados - nenhum estado é expandido mais de uma vez.
- Informação sobre o caminho até o estado meta é retida.
- A lista de hipóteses é formada por caminhos de estados e não apenas por um estado

101

Algoritmo A*

1. Crie uma fila de caminhos parciais (inicialmente a raiz);
2. **enquanto** a fila não está vazia e meta não encontrada:
 - obtenha o melhor caminho x da fila;
 - se** o estado corrente de x não é o estado meta **então**
 - Forme novos caminhos estendendo o estado corrente de x a todos os caminhos possíveis;
 - Adicione os novos caminhos a fila;
 - Ordene a fila usando f';
 - Remova duplicações* da fila (usando f');
 - fim_se**
- fim_enquanto**

* duplicações são caminhos cujo final estão no mesmo estado.

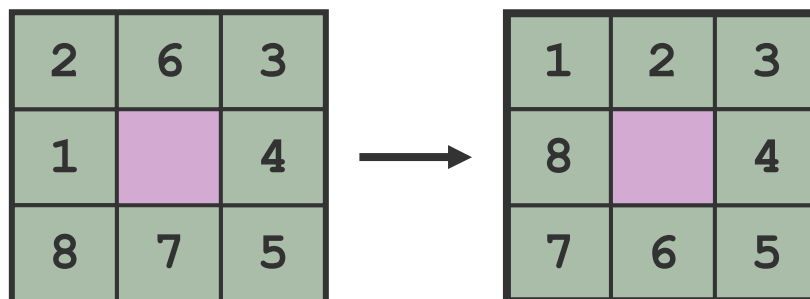
102

A* Otimização and Completude

- Se a função heurística h' é *admissível* o algoritmo encontrará o caminho mais barato para a solução no número mínimo de passos
- Uma heurística é *admissível* se jamais superestima o custo de um estado até o estado meta.

103

Exemplo A* - 8 Puzzle



104

Exemplo A*

🌳 A partir do mapa a seguir, crie a árvore de busca do A* gerada para a rota tendo como ponto inicial Arad e destino Bucharest (exemplo extraído de Norvig)

105

Exemplo A*

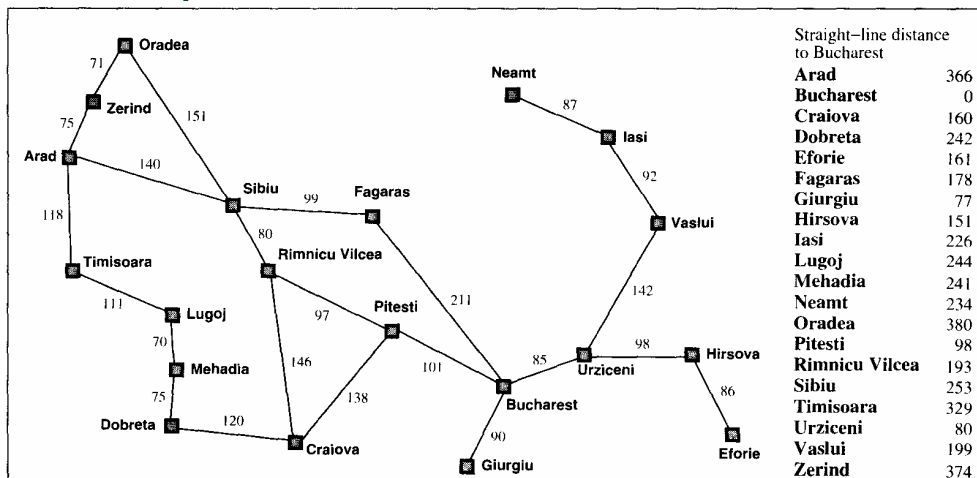
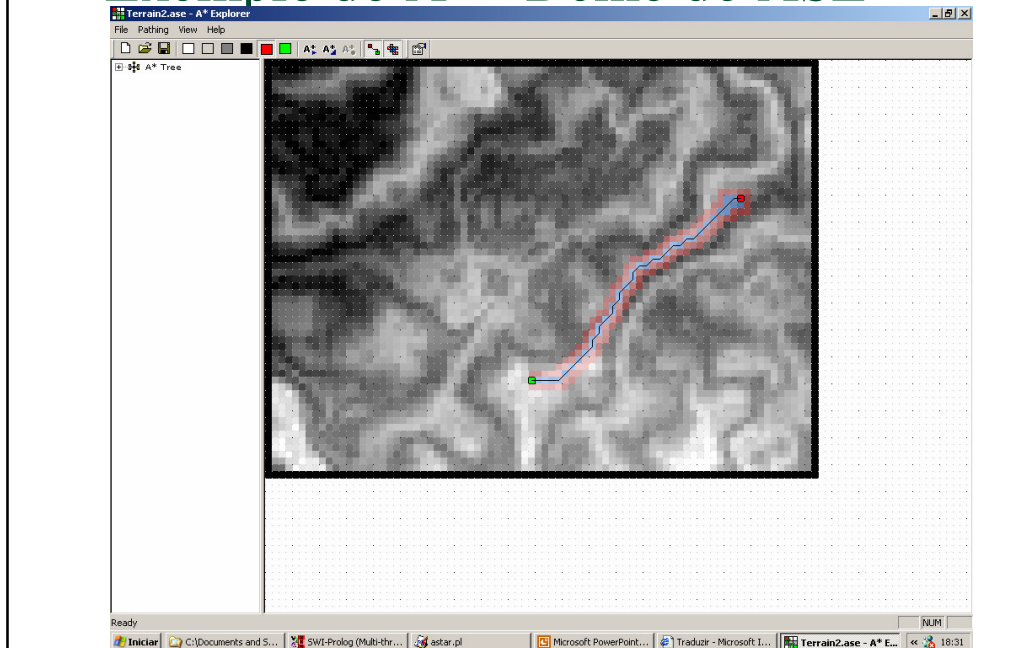
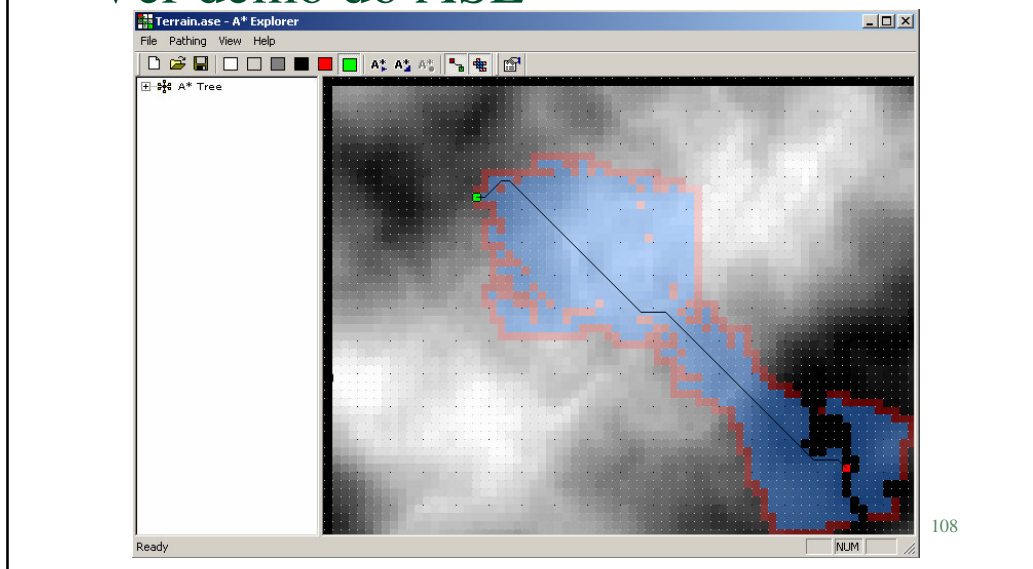


Figure 4.2 Map of Romania with road distances in km, and straight-line distances to Bucharest.

Exemplo do A* - Demo do ASE



Ver demo do ASE



Redução de Problemas

- Problemas que podem ser decompostos requerem algoritmos especiais para tirar proveito de soluções parciais.
- Busca em árvores/grafos até aqui não empregavam redução de problema.
- Grafos AND/OR fornecem a base para muitos algoritmos de redução.
- Um grafo AND/OR é diferente de um espaço de estados

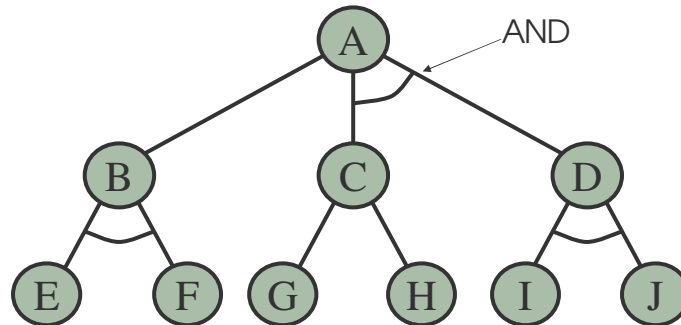
109

Árvores/Grafos AND/OR

- Cada nó (estado) na árvore de busca tem filhos que representam possíveis estados sucessores.
- Cada nó é AND ou OR.
- Um nó é resolvido se:
 - AND: nó e todos os seus filhos são resolvidos.
 - OR: nó e pelo menos 1 filho é resolvido

110

Árvore AND/OR



Para resolver A, é necessário resolver B *ou* C e D.

Para resolver B, é necessário resolver E e F.

...

111

Busca em Grafos AND/OR

🐼 Algoritmo básico de busca AND/OR

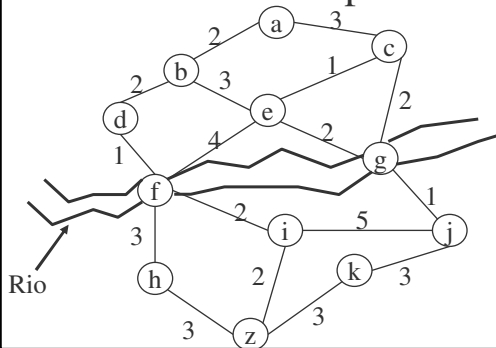
- **se** Nó é uma solução **então** pare
- **se** Nó tem sucessores OR *sucessoresOR*
 - **então** resolva *sucessoresOR* (tente encontrar pelo menos uma solução a partir de *sucessores*)
- **se** Nó tem sucessores AND *sucessoresAND*
 - **então** resolva *sucessoresAND* (tente encontrar pelo menos uma solução para cada nó de *sucessoresAND*)

* Se o algoritmo anterior não encontrou nenhuma resposta, então o problema não possui solução

112

Representação de Grafos AND/OR

- Decompõe problemas em subproblemas
- Subproblemas são independentes e podem ser resolvidos separadamente

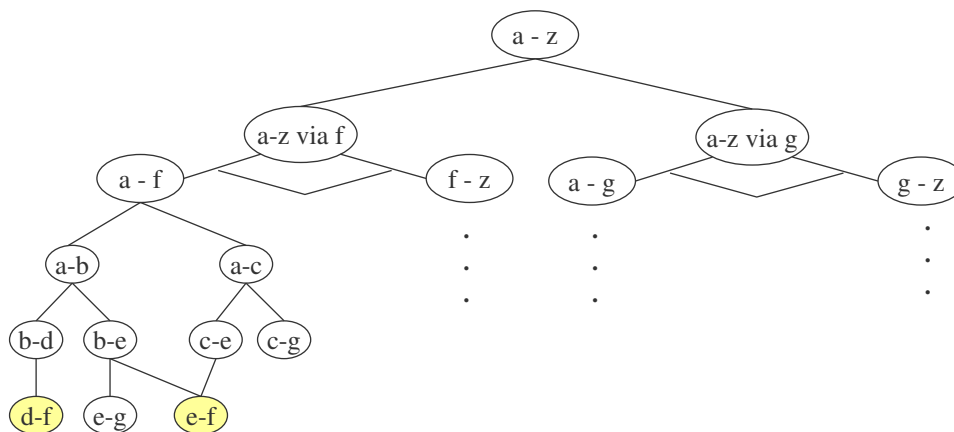


Exercício:

Encontre o grafo and/or com as rotas possíveis entre as cidades a e z que obrigatoriamente passem por f e g.

113

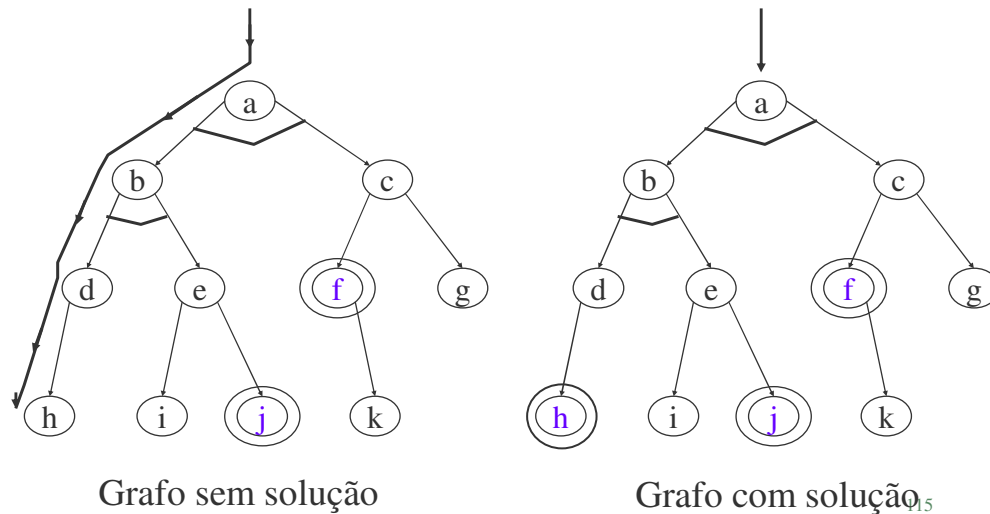
Representação de Grafos AND/OR - Solução do Exercício Anterior



114

Representação de Grafos AND/OR

(Cont.)



Busca AND/OR

Exercício: Escreva um programa Prolog que realize busca AND/OR simples em um grafo AND/OR qualquer

Solução:

```
and_or(No,N) :-
    pertence(No,N).
and_or(No,N) :-
    filhos_or(No,Filhos_or),
    solucao_or(Filhos_or,N).
and_or(No,N) :-
    filhos_and(No,Filhos_and),
    solucao_and(Filhos_and,N).
```

```
solucao_or(Nos,N) :-
    pertence(X,Nos),
    and_or(X,N),!.

solucao_and([],_).
solucao_and([No|Nos],N) :-
    and_or(No,N),
    solucao_and(Nos,N).
```

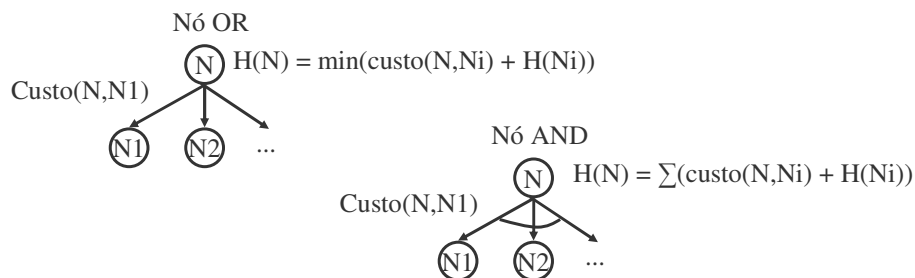
Algoritmo AO* (Encontra o caminho com menor custo)

- Aplicado sobre grafos AND/OR
- Cada nó do Grafo possui uma função h' (estimativa do custo do caminho entre o nó e um conjunto de soluções)
- Não é necessário armazenar g' (custo do nó inicial até o nó corrente)
- Apenas os nós do melhor caminho serão expandidos
- Cada nó do grafo aponta para seus sucessores e predecessores diretos

117

Algoritmo AO*

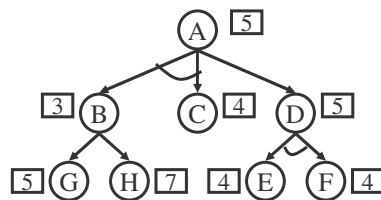
- A partir de um nó expande o filho mais promissor
- A qualidade de um nó é dada por uma função $h(n)$
- Exemplo considera apenas arcos com peso 1



118

Exemplo do AO*

➤ Dado o problema representado no grafo and/or a seguir, ilustre o processo de pesquisa pelo melhor caminho no grafo a partir do nó A

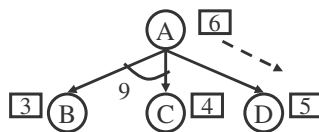


119

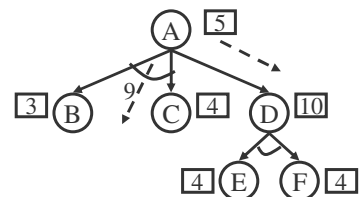
Exemplo do AO*



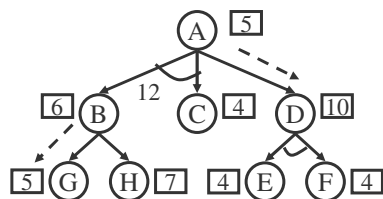
(a)



(b) A é expandido



(b) D é expandido



(b) B é expandido

120

Algoritmo AO* (cont.)

1. *Grafo* consiste apenas do nó inicial *Inicial*
2. **enquanto** *Inicial* não é resolvido **ou** $h' > limite$
 - a) selecione um dos sucessores de *Inicial*, *Nó*, que ainda não foi expandido
 - b) gere os sucessores de *Nó*
 - **se** *Nó* não tiver sucessor, atribua a *limite* o valor de $h'(Nó)$
 - **se** *Nó* tiver sucessores para cada nó *sucessor* que não seja antecessor
 - acrescente *sucessor* a *Grafo*
 - **se** *sucessor* é solução **então** marque-o como resolvido e $h'(sucessor) = 0$
 - **se** *sucessor* não é solução **então** calcule $h'(sucessor)$

121

Algoritmo AO* (cont.)

- c) S = conjunto de Nós marcados como Resolvidos ou cujo valor de h' tenha sido alterado
 - **enquanto** S não está vazio
 - escolha, se possível, um nó que não tenha nenhum descendente em *GRAFO* ocorrendo em S , caso não exista, escolha qualquer nó (*corrente*) em S e o remova
 - $h'(corrente)$ = mínimo do somatório dos custos dos arcos que dele emergem
 - marque o melhor caminho que parte de *corrente* como aquele com o menor custo calculado anteriormente
 - marque *corrente* como resolvido se todos os nós ligados a ele pelo arco obtido tiverem sido marcados como resolvidos
 - **se** *corrente* está resolvido **ou** o custo de *corrente* foi alterado **então** acrescente os *ancestrais* de *corrente* a S

122

Satisfação de Restrição

- Alguns problemas têm metas que podem ser modeladas como um conjunto de restrições satisfeitas.
- Estados são definidos por valores de um conjunto de variáveis.
- Utilização de restrições para limitar movimentos no espaço de busca pode melhorar a eficiência.

123

Satisfação de Restrição - Algoritmo

- 1. Formule todas as restrições R a partir do estado inicial
- 2. Encontre um novo estado S
- 3. Aplique as restrições R sobre S gerando um novo conjunto de restrições R'
- 3. Se S satisfaz todas as restrições R' , retorne S
- 4. Se S produz uma contradição então retorne fracasso
- 5. Se não existir outro estado retorne fracasso senão retorne ao passo 2

124

Exemplo de Problemas de Satisfação de Restrições

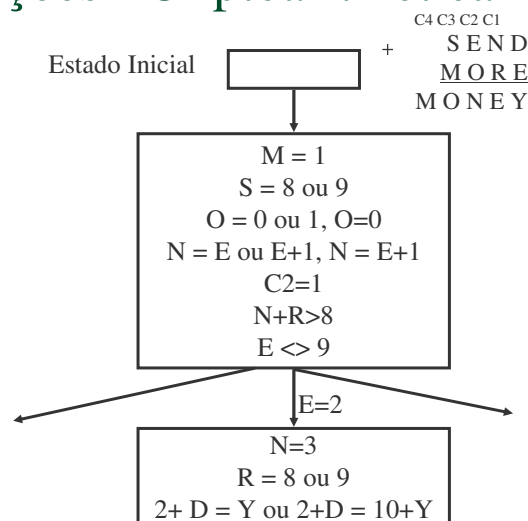
♣️ Criptoaritmética:

- problemas aritméticos representados em letras. A meta é associar um dígito diferente a cada letra de modo a poder utilizá-las matematicamente.

$$\begin{array}{r}
 + \quad \text{S E N D} \\
 \quad \text{M O R E} \\
 \hline
 \text{M O N E Y}
 \end{array}$$

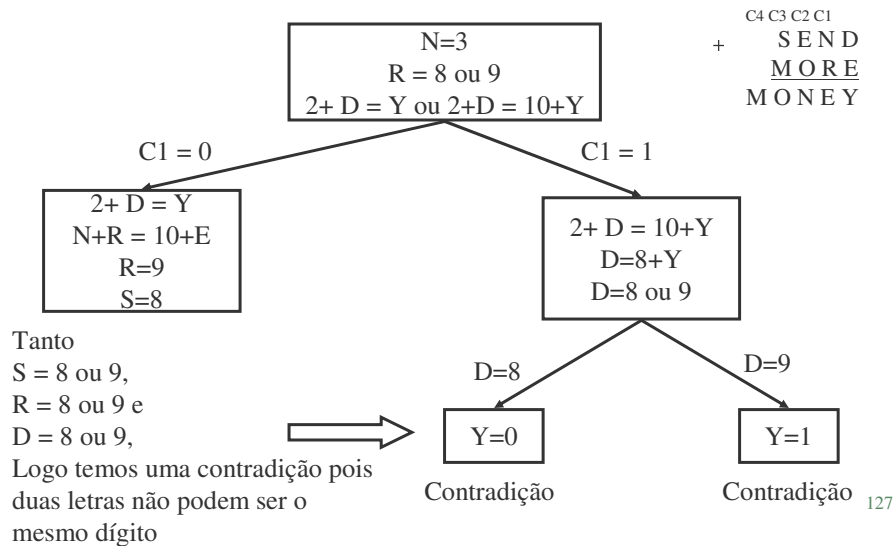
125

Exemplo de Problemas de Satisfação de Restrições - Criptoaritmética



126

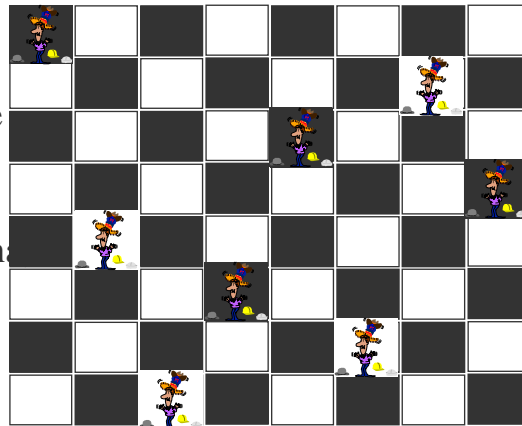
Exemplo de Problemas de Satisfação de Restrições - Criptoaritmética



Exemplo de Problemas de Satisfação de Restrições (Cont.)

♛ 8 Rainhas:

- colocar 8 rainhas no tabuleiro de xadrez de modo que nenhuma rainha ataque outra rainha (na mesma linha, coluna ou diagonal).



128

Análise Meios-Fins

- Busca para frente (em direção à meta) e para trás (da meta para o estado corrente) ao mesmo tempo.
- Isola uma diferença entre o estado corrente e o estado meta - encontra um operador que reduz a diferença.
- É importante resolver primeiramente as diferenças maiores
- Motivada pela forma com que alguns problemas são resolvidos por pessoas.

129

Análise Meios-Fins – Algoritmo GPS

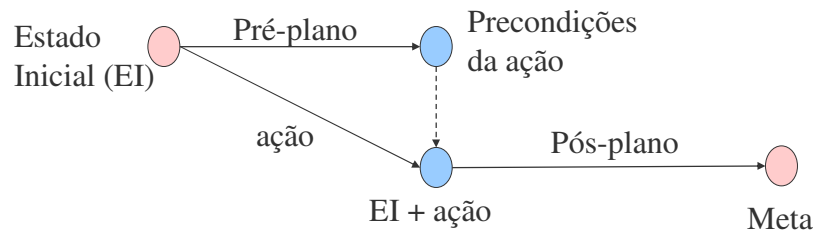
GPS(*estado-inicial*, *meta*)

1. Se *meta* \subseteq *estado-inicial*, então retorne *true*
2. Selecione uma diferença *d* entre *meta* e o *estado-inicial*
3. Selecione um operador *O* que reduz a diferença *d*
4. Se nenhum outro operador então *falhe*
5. *Estado* = **GPS**(*estado-inicial*, *precondições*(*O*))
6. Se *Estado*, então retorne **GPS**(**apply**(*O*, *estado-inicial*), *meta*)

130

Análise Meios-Fins - Esquema

- 🐼 Encontre uma ação útil (que reduza a diferença com a meta)

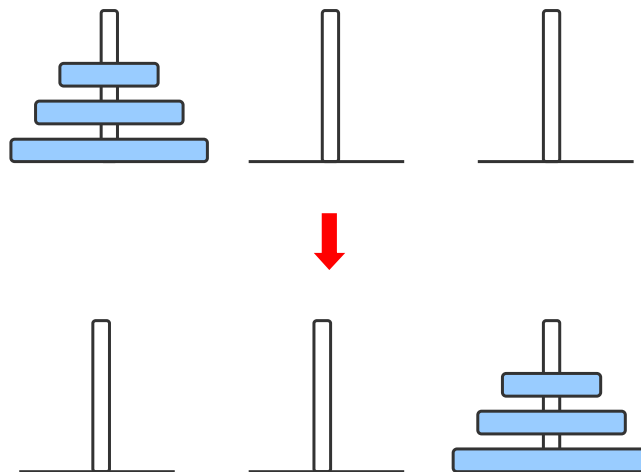


- 🐼 Crie dois novos subproblemas:

- tornar verdadeiras as precondições da ação a partir de EI
- encontrar a meta a partir do resultado da ação sobre EI

131

Exemplo de análise meios-fins: as Torres de Hanoi



132

Solução em Prolog

```

hanoi(1,Start,End,_) :-
    write('move disc from '),
    write(Start), write(' to '),
    write(End), nl.

hanoi(N,Start,End,Aux) :-
    N>1, M is N-1,
    hanoi(M,Start,Aux,End),
    write('move disc from '), write(Start),
    write(' to '), write(End),nl,
    hanoi(M,Aux,End,Start).

```

Se existe apenas um disco,
Então mova-o para o pino final

Se existe mais de um disco,
Então

mova todos os N-1 pinos de cima da
esquerda para o meio com o auxílio do
pino direito

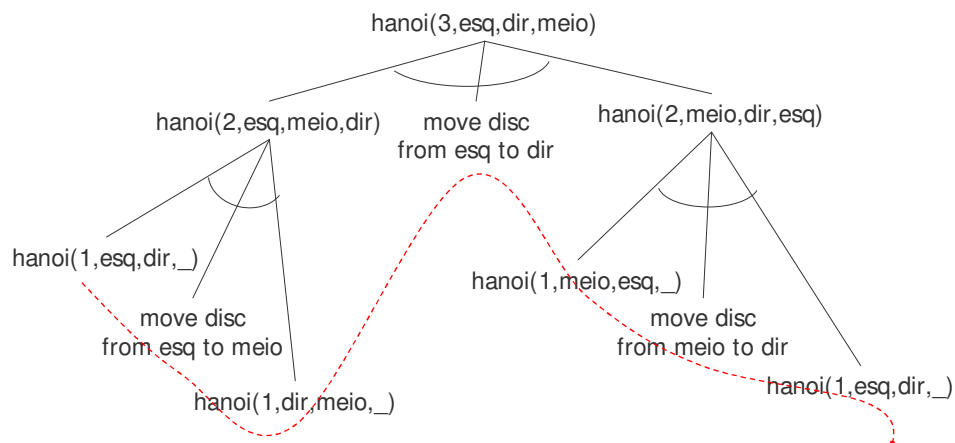
mova o pino de cima para o pino direito

mova todos os N-1 pinos do meio para a
direita com o auxílio do pino esquerdo

Questionamento: ?- hanoi(3,esquerda,direita,meio).

133

Execução do Programa Torres de Hanoi



134