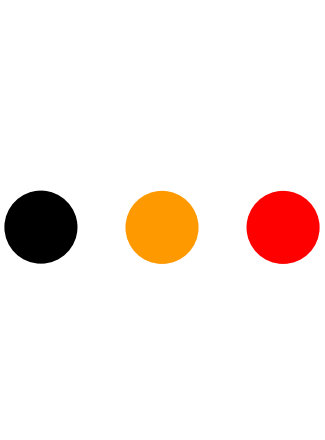


Refinamento das Estratégias de Busca

Profa. Josiane M. P. Ferreira

Texto base:

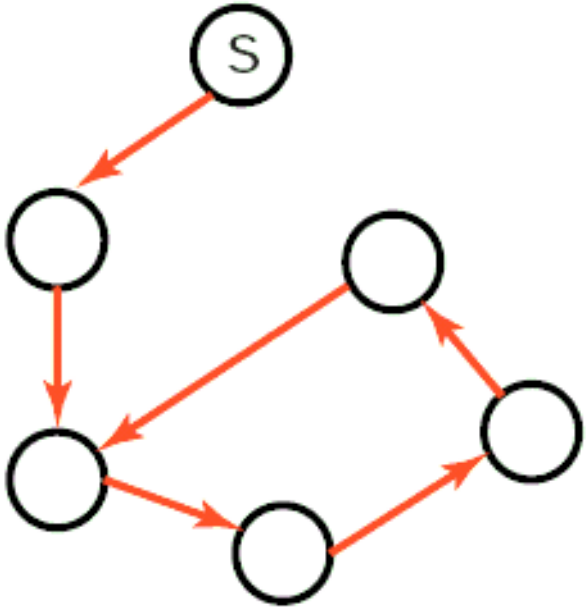
David Poole, Alan Mackworth e Randy Goebel - "*Computational Intelligence – A logical approach*"
Stuart Russel e Peter Norving - "Inteligência Artificial"

junho/2007

Resumo das estratégias de busca

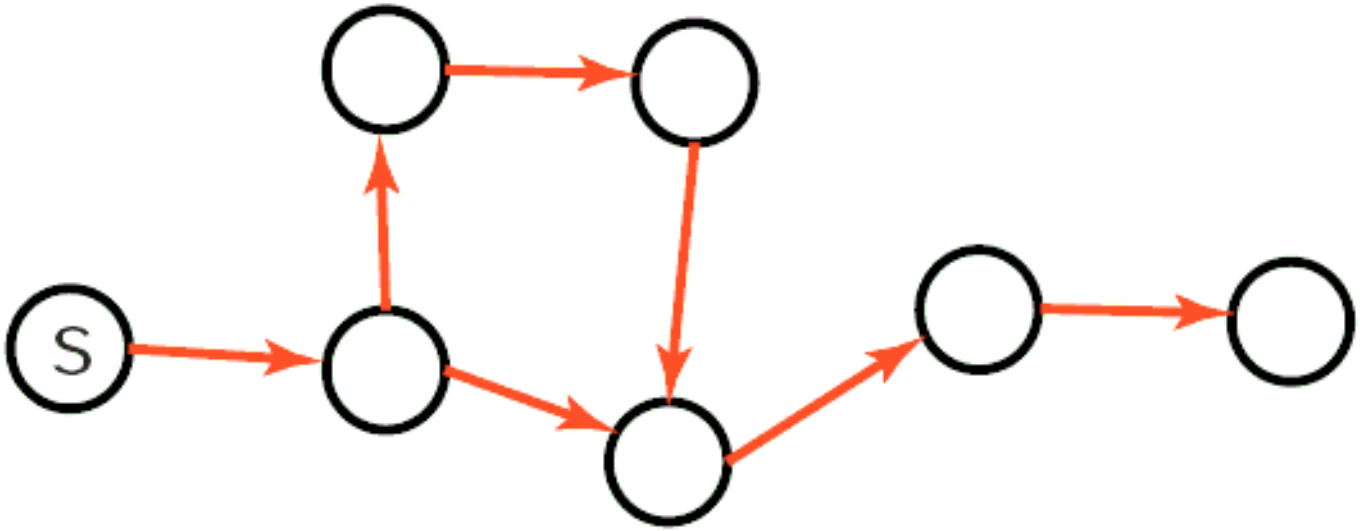
Estratégia	***Seleção da fronteira***	***Completa?***	***Espaço***
Profundidade	último nó adicionado	não	linear
Largura	primeiro nó adicionado	sim	exponencial
Heurística em profundidade primeiro	minimo local h(n)	não	linear
Melhor primeiro	mínimo global h(n)	não	exponencial
Menor custo primeiro	minimo g(n)	sim	exponencial
A*	minimo f(n) = h(n) + g(n)	sim	exponencial

Checando de ciclos



- Podemos podar um caminho que termina em um nó que já faz parte do caminho
  - Esta poda não pode remover uma solução ótima
  - Normalmente leva tempo linear no tamanho do caminho

Podando múltiplos caminhos



- Podemos podar uma caminho para um nó *n* quando a busca já encontrou um caminho para *n*
  - Deveremos guardar todos os nós para os quais a busca já encontrou um caminho (lista de nós fechados)

Podando múltiplos caminhos e a solução ótima

- E se um caminho para *n* encontrado depois for menor do que o primeiro caminho encontrado para *n*?
  - Podemos remover todos os caminhos da fronteira que usam um caminho maior até *n*
  - Podemos modificar o segmento inicial dos caminhos da fronteira para utilizar o caminho menor
  - Podemos assegurar que isso não aconteça tendo certeza que o caminho mais curto para um nó é encontrado primeiro

Podando múltiplos caminhos e A*

- Usando A* é possível que quando um nó seja selecionado pela primeira vez, o caminho associado a ele não seja o menor caminho
- Para que o primeiro caminho encontrado para um nó *n*, seja o caminho mais curto, a heurística utilizada deve ser monotônica
  - Uma função heurística *h* satisfaz a restrição de monotonicidade se |h(*n*′) – h(*n*)| <= d(*n*′, *n*) para todo arco <m, n>
- Desta forma, os valores de *f* na fronteira serão monotonicamente não decrescentes
- A* que utiliza uma *h* monotõnica e poda de múltiplos caminhos sempre encontrará o menor caminho para o objetivo

Quando usar o quê

- Poda de múltiplos caminhos engloba a checagem de ciclos
  - Pode ser feito em tempo constante se todos os nós encontrados estiverem armazenados
- Poda de múltiplos caminhos
  - Métodos em largura primeiro – quando temos que armazenar virtualmente todos os nós considerados
- Checagem de ciclos
  - Métodos em profundidade primeiro – quando armazenamos somente o caminho para o qual estamos realizando a busca

Busca em profundidade iterativa

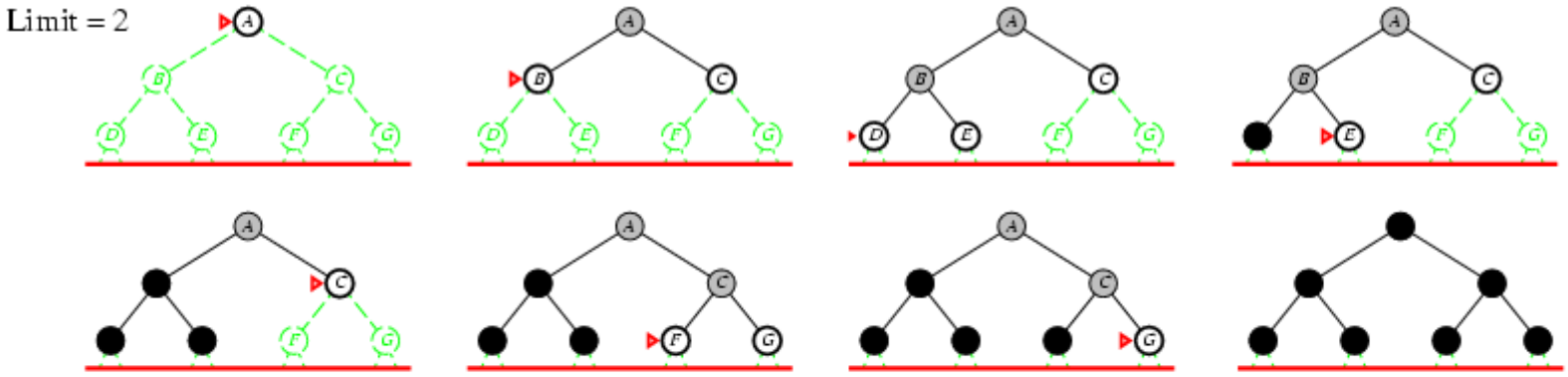
- Até agora, todas as estratégias de busca que garantem encontrar uma solução usam espaço exponencial
- Idéia: Recomputar os nós ao invés de armazená-los
- Procurar por caminho de profundidade zero, depois profundidade 1, depois profundidade 2, depois profundidade 3, etc
- Precisamos de um algoritmo de busca em profundidade limitada
- Se o caminho não é encontrado na profundidade B, então procure por um caminho na profundidade B + 1
  - Aumentando o limite de profundidade quando a busca falha não-naturalmente (limite de profundidade foi alcançado)

Busca em profundidade iterativa

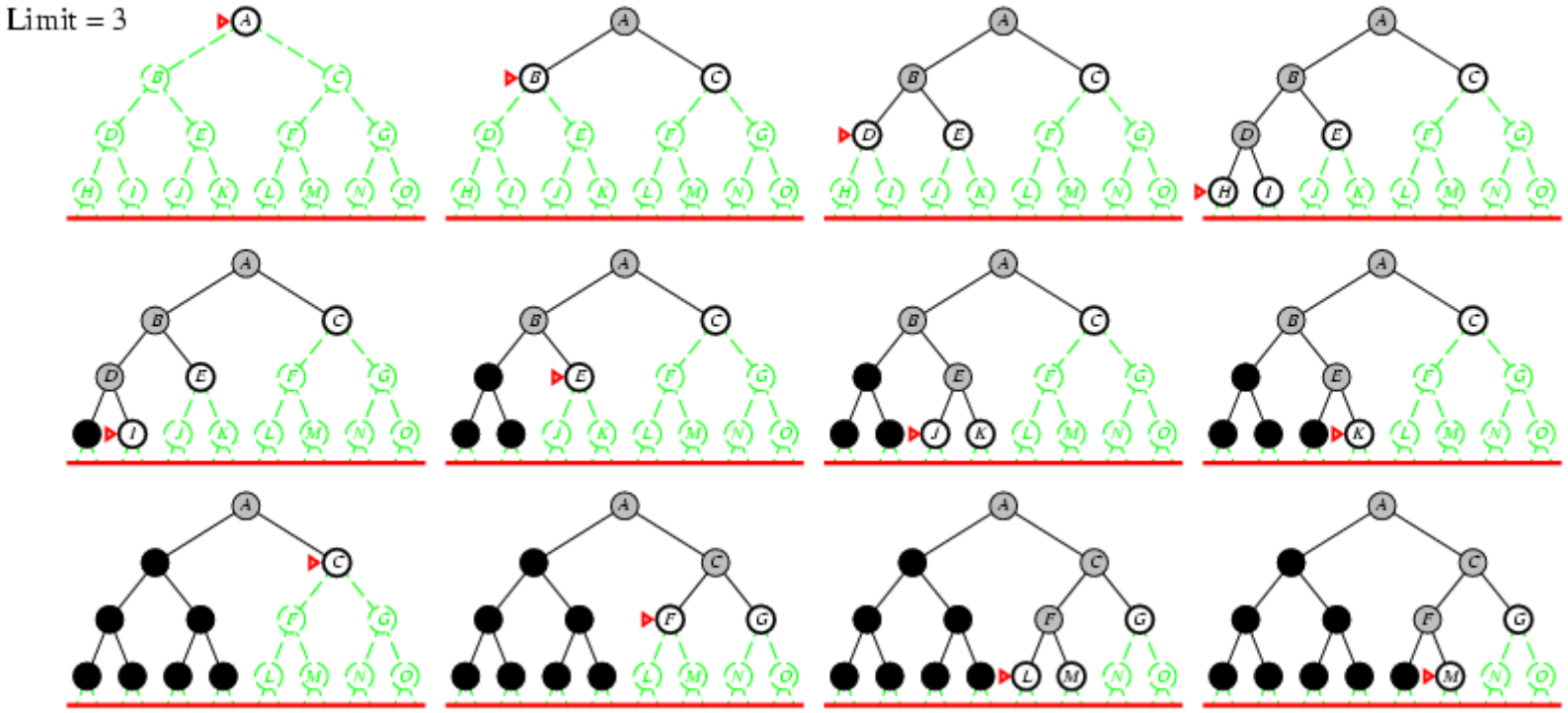
Limit = 0

Busca em profundidade iterativa

Busca em profundidade iterativa



Busca em profundidade iterativa



Apesar de parecer nem um pouco eficiente, os nós que são “recomputados” várias vezes são os nós dos níveis mais altos, que normalmente são em menor quantidade

Direção da Busca

- A definição de busca é simétrica: encontrar caminhos do nó inicial até o nó objetivo ou do nó objetivo até o nó inicial
- **Fator de ramificação para frente**:Número de arcos que saem de um nó
- **Fator de ramificação para trás**:Número de arcos que entram em um nó
- Devemos usar busca para frente se o fator de ramificação para frente for menor do que o fator de ramificação para trás, e vice-versa
- Algumas vezes quando o grafo é dinamicamente construído, pode ser que não possamos construí-lo de trás para frente

Busca Bidirecional

- Podemos buscar do objetivo para trás e do nó inicial para frentes simultaneamente
- Pode resultar em economia de espaço e tempo
  - $2b^{k/2} << b^k$
- O principal problema é ter certeza que as fronteiras se encontram
  - Isso deve ser uma operação pouco custosa para não tornar a busca inviável
- Nem todas as estratégias de busca funcionam
  - A busca em profundidade nas duas direções pode ser desastrosa
  - A busca em largura garante que as fronteiras se encontram
  - Normalmente é utilizada busca em largura em pelo menos uma das direções