

# **Estratégias de Busca**

# Estratégia ....

- Maior problema da área de Busca/Procura tem a ver com determinação da melhor estratégia a aplicar na solução de dado problema
- Estratégias são avaliadas de acordo com quatro critérios:
  - (i) Completude (completeness) – estará garantido que a estratégia encontrará solução, se uma existir?
  - (ii) Complexidade de Tempo (Time Complexity) – quanto tempo leva a encontrar a solução?

# Estratégia....

- (iii) Complexidade de Memória (space complexity) – quanta memória será necessária para realizar a Busca?
  
- (iv) Optimalidade – será que a estratégia encontra solução de alta qualidade, na presença de várias soluções?

# Busca não Informada

- Busca não informada baseia-se na ausência de info sobre o número de passos a dar assim como do custo do percurso/path cost desde o estado corrente até ao estado Objectivo/Goal
- Distingue sómente estado goal do estado não goal
- Estratégias que usam alguma info para guiar o processo de Busca denominam-se de estratégias de busca informada ou estratégias de busca com heurística

# Breadth-First Search/Busca em Amplitude

- Expande a raiz primeiro
- Expanda os nós gerados pela raiz
- De seguida, os seus sucessores são também expandidos, e assim por diante ...

# Breadth-First Search/Busca em Amplitude

- Breadth-first pode ser implementado chamando o algoritmo geral de busca com a função de **fila (queue)** que coloca os novos estados gerados no fim do **Queue**, após os anteriormente gerados

Function Breadth-First-Search(problema) return Solução  
ou Falhanço

Return General-Search(problema, Enqueue-at-End)

# Breadth-First Search/Busca em Amplitude

- BrFS é sistemático, pois que considera todos os percursos de tamanho (profundidade) 1 primeiro, seguido dos de tamanho 2, etc.
- Encontra solução sempre que existir. Se forem várias, encontra a mais à esquerda primeiro

# Breadth-First Search/Busca em Amplitude

- Com relação aos quatro critérios, o BrFS:
  - É completo
  - É óptimo, desde que path cost/custo de percurso seja função não decrescente da profundidade do node/nó
  - Time e memory complexity é problemático
- **Factor de ramificação, LEMBRAM-SE??....**
- Requisitos de memória são grande problema comparativamente ao tempo



# Busca de Custo Uniforme

- Modifica o algoritmo de BrFS, expandindo sempre o node que tiver menor custo
- Custo medido pelo path cost  $g(n)$  em vez do node mais profundo
- Busca em amplitude é variante da Busca uniforme com custo  $g(n) = \text{profundidade}(n)$

# Busca de Custo Uniforme

- Quando o custo não decresce ao longo do percurso, Busca uniforme encontra a menos cara solução dada pela formula:  $g(\text{Sucessor}(n)) \geq g(n)$  para todo o node  $n$
- Restrição de percurso não decrescente faz sentido se o custo de percurso do node é tomado como soma do custo dos operadores que realizam/produzem o path/percurso

# Busca de Custo Uniforme

- Se todos os operadores têm custo não negativo, então o custo do percurso poderá não decrescer nunca conforme o vamos percorrendo
- A busca de custo uniforme poderá assim determinar o menos caro path sem explorar toda a árvore
- Se um dos operadores tiver valor negativo de custo, só uma busca exaustiva trará solução óptima

# Depth-First Search/Busca em Profundidade

- Expande nós que se encontram no nível mais baixo da árvore
- Se a busca depara-se com (dead end) node que não é o goal, o qual não permite expansão; voltamos e expandimos os nodes do nível mais alto
- Requisitos de memo muito modestos. Armazena sómente único percurso desde a raiz até nó abaixo, em conjunto com os nós primos/adjacentes não expandidos para cada node do percurso/path

# Depth-First Search/Busca em Profundidade

- Para espaço de estados com factor de ramificação  $b$  e profundidade máxima de  $m$ , DFS necessita de armazenar  $bm$  nodes, em contraste com BrFs que exigirá  $b^d$ , onde  $d$  – nível de profundidade do goal mais próximo
- Para problemas com muitas soluções DFS pode ser mais rápido que BrFS, pois que tem maior possibilidade de alcançar solução depois de explorar pequena porção do espaço de Busca

# Depth-First Search/Busca em Profundidade

- DFS tem a desvantagem de tornar-se imóvel por ter entrado por caminho abaixo errado
- Incluindo até loops infinitos ou trazer solução superior ao óptimo
- Não é completo nem ótimo (pode não produzir solução ótima)
- NB: **Evite usar DFS para árvores infinitas ou com nível de profundidade aparentemente infinita**

# Depth-Limited Search/Busca em Profundidade Limitada

- Introduz um limite (Cut-Off) no nível de profundidade do percurso/path
- Implementado via algoritmo especial de depth-limited search ou usando DFS geral com operadores que controlam a profundidade
- É completo mas não é óptimal

# Depth-Limited Search/Busca em Profundidade Limitada

- Se escolher um limite bastante pequeno, então nunca será completo
- Time complexity:  $O(b^d)$
- Space complexity:  $O(bl)$
- Onde  $l$  – é o limite de profundidade



# Iterative Deeping Search (IDS)/Busca de Profundidade Iterativa

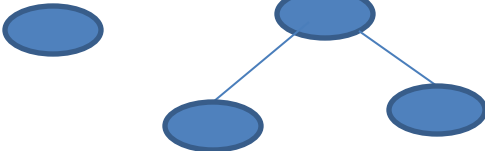
- A dificuldade com profundidade limitada está em determinar/encontrar um limite certo/apropriado
- IDS é uma estratégia que elimina este constrangimento através do teste de todos os possíveis limites de profundidade
- Primeiro profundidade 0, depois 1, a seguir 2 e por aí em diante

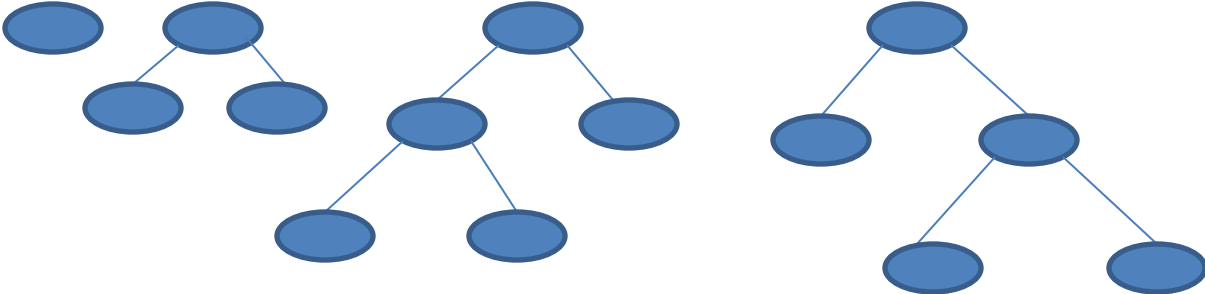
# Iterative Deeping Search (IDS)/Busca de Profundidade Iterativa

- IDS combina os beneficios de DFS e BrFS
- É óptimo e completo como BrFS com requisitos de memória modestos, do DFS
- Ordem expansão dos estados é similar ao BrFS, excepto que alguns estados são expandidos por diversas vezes.
- Numa árvore binária podemos ter:

# Iterative Deeping Search (IDS)/Busca de Profundidade Iterativa

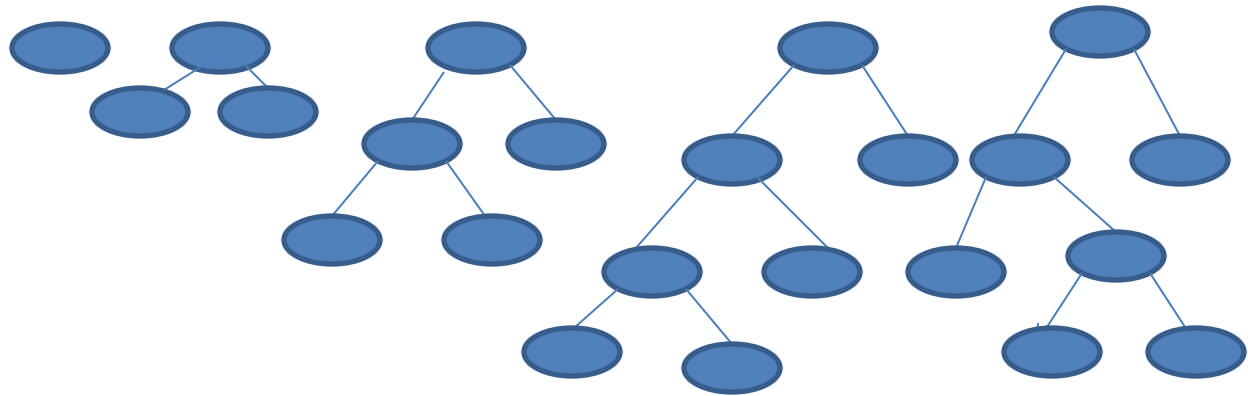
- Limite 0: 

- Limite 1: 

- Limite 2: 

# Iterative Deeping Search (IDS)/Busca de Profundidade Iterativa

- Limite 3:



....

- Embora pareça perca de tempo e espaço pela expansão múltipla de estados, IDS em diferentes problemas tem tido bom performance, e as expansões são até poucas ...

# Bidirectional Search/Busca Bidirecional

- A ideia é fazer a busca simultaneamente, a frente desde o estado inicial e atrás a partir do goal, parando onde as duas buscas se encontram no meio
- Questões pertinentes:
  - (i) o que significa fazer a busca detras a partir do goal?
    - Define-se predecessor do node  $n$  como sendo todos os nós que têm  $n$  como sucessor

# Bidirectional Search/Busca Bidirecional

- Buscando para trás significa gerar predecessores de forma sucessiva partindo do goal node
- (ii) O conjunto dos predecessores dos sucessores são identicos quando todos os operadores forem reversíveis. Determinar sucessor é extremante dificil ...
- (iii) O que fazer quando existem diversos estados goal?
  - lista explicita de goals: use função predecessor para estados (conjunto deles)
  - descrição do goal: necessario descrição precisa do conjunto de estados que podem degenerar em goal
- (iv) Necessaria forma de verificação de cada novo node para determinar sua ocorrencia anterior em outra metade da árvore
- (v) Que tipo de busca ocorrerá em cada metade do problema?

# Comparando Estratégias

**Load File:**  
**Criterio.pdf**

# Como evitar repetição de Estados

- (i) Não regresse ao estado anterior ao corrente.  
Construa função de expansão que refute gerar qualquer sucessor que o mesmo node pai do estado corrente
- (ii) Não crie path/percursos com ciclos
- (iii) Não degenere nenhum estado/node que tenha sido anteriormente degenerado



# Métodos de Busca Informada

- No qual podemos observar como a *info* sobre o espaço dos estados pode evitar que os algoritmos executem o processo de Busca “às escuras ou às cegas” ..
- Onde colocar algum conhecimento no âmbito da busca?

# Métodos de Busca Informada

- Sómente no local/momento de decidir qual dos nodes a expandir
- Esta info usualmente providenciada pela função avaliativa, a qual devolve um número descrevendo as vantagens de expansão do node/nó respectivo

# Best-First Search (BFS)/Busca pelo Melhor Primeiro

- Quando os nodes estão ordenados de tal forma que o que tiver melhor valor da função avaliativa é expandido primeiro, resulta numa estratégia denominada de Best-First Search (BFS)/Busca pelo Melhor Primeiro

NB: Não expandimos realmente o melhor nó primeiro, Senão seria uma marcha directa para o goal. O que fazemos é expandir o nó que aparenta ser o melhor de acordo com a função avaliativa.

# Best-First Search (BFS)/Busca pelo Melhor Primeiro

- Existe uma família de algoritmos de BFS, com diferentes funções avaliativas
- Procuram encontrar soluções baratas, usando medidas de estimativas do custo da solução os quais devem ser minimizados, i.e. minimal cost solution
- O custo do percurso até agora visto  $g(.)$ , não conduz a busca directa para o goal

# Best-First Search (BFS)/Busca pelo Melhor Primeiro

- A nova medida deve incorporar alguma estimativa do custo do percurso desde o estado corrente até ao mais próximo estado goal
- Duas formas possíveis:
  - (i) Expansão do node mais próximo do goal
  - (ii) Expansão do nó do percurso menos honoroso em direcção a solução

# Greedy Search

- A minimização do custo estimado para o alcance do goal, denomina-se: Greedy Search
- É a estratégia mais simples de BFS
- O nó com melhor possibilidade de alcançar o goal é expandido primeiro

# Greedy Search

- Para a maior parte dos problemas o custo para alcançar o goal a partir de um estado particular pode ser estimado, mas não pode ser determinado com exactidão
- A função que calcula este custo estimado denomina-se de função heurística, usualmente denotada por  $h$
- $h(n)$  é o custo estimado do mais barato percurso desde o estado do nó  $n$  até ao estado goal

# Greedy Search (GS)

- BFS que usa  $h$  para seleccionar o próximo nó para expansão – Greedy Search
- $h$  pode ser qualquer função com requisito de que  $h(n) = 0$  se  $n$  é o estado goal
- As funções heurísticas dependem do tipo de problema em mão – são específicas pra cada problema



# Greedy Search

- Para o exemplo de melhor rota (route-finding) partindo da vila de Moamba ... Chicualacuala, toma-se a distância directa entre dois pontos/vilas como heurística
- $h_{dd}(n)$  – distância directa entre  $n$  e a localização do goal
- Greedy search actua como DFS em termos de preferir seguir percurso único para o goal, mas consegue refazer-se ao alcançar “dead end”
- Não é óptimo nem completo

# Minimização do Custo total do path: A\*

## Search

- GS minimiza o custo estimado para o goal  $h(n)$ , diminuindo o custo da busca consideravelmente. Infelizmente não é óptimo nem completo
- A busca de custo uniforme, minimiza o custo do percurso  $g(n)$  sendo óptimo e completo, mas pode ser ineficiente devido ao formato de expansão dos nodes

# Minimização do Custo total do path: A\* Search

- A solução é combinar GS e a busca de custo uniforme, obtendo-se uma função avaliativa:

$$f(n) = g(n) + h(n)$$

- $g(n)$  – custo do percurso desde o estado inicial até ao nó  $n$
- $h(n)$  – custo estimado do mais barato percurso desde nó  $n$  até ao goal

# Minimização do Custo total do path: A\*

## Search

- $f(n)$  – custo estimado do percurso mais barato até a solução passando pelo node  $n$
- Sempre que procurarmos solução barata, a escolha razoável é tomar primeiro o nó com menor  $f$
- A\* é ótimo e completo, o que pode ser provado com pequenas restrições em  $h$ , a qual é escolhida para que nunca seja superior ao valor necessário ao alcance do goal

# Minimização do Custo total do path: A\* Search

- Este tipo de  $h$  denomina-se de heurística admissível
- Heurísticas admissíveis são por natureza óptimas, pois qe assumem que o custo da solução do problema é menor do que actualmente é
- Se  $h$  é admissível,  $f(n)$  nunca será superior ao custo actual da melhor solução através de  $n$

# Minimização do Custo total do path: $A^*$ Search

- BFS que utiliza  $f$  como função avaliativa e função admissível  $h$  denomina-se de  $A^*$
- Se para qualquer percurso desde a raiz  $f$ -cost nunca decresce (true pra todas heurísticas admissíveis), diz-se que heurística  $f$  exibe monotonocidade
- Se heurística não é monotónica, podemos com pequenas correcções torna-la monotónica

# Minimização do Custo total do path: A\* Search

- Consideremos dois nós  $n$  e  $n'$ , onde  $n$  é pai de  $n'$ . Seja que  $g(n)=3$  e  $h(n)=4$ , então  $f(n)=g(n) + h(n) = 3+4 = 7$ , i.e. o path real até a solução via  $n$  é pelo menos 7
- Suponhamos que  $g(n')=4$  e  $h(n')=2$ , então  $f(n')=g(n') + h(n') = 4+2 = 6$

# Minimização do Custo total do path: A\* Search

- Este é um exemplo claro de ausência de monotonocidade. Felizmente que qualquer path via  $n$  é também percurso via  $n$ , podemos considerar o valor 6 de insignificante pois que já sabemos que o custo real é pelo menos 7
- Sempre que geramos novo nó verificamos se seu f-cost é menor do que o do seu predecessor.
- Se for o caso, usamos o f-cost do seu node pai, i.e.  $f(n) = \max(f(n), g(n) + h(n))$  – equação do máximo percurso



# Minimização do Custo total do path: A\* Search

- Desta forma ignoramos valores insignificantes que possam ocorrer com heurísticas não monotónicas e  $f$  torna-se não decrescente

# Funções Heurísticas

- Até ao momento vimos um só exemplo de heurística: distância directa para o problema do caixeiro viajante
- O problema de 8-puzzle traz-nos outras questões. Em geral leva 20 passos para resolver dependendo do estado inicial. O factor de ramificação  $b = 3$
- Uma busca exaustiva até profundidade 20 conduzirá a observação de  $3^{20} = 3 * 5 + 10^9$  estados.

# Funções Heurísticas

- Se cuidarmos de não repetir estados, este número reduz drasticamente para  $9! = 362\,880$  arranjos diferentes dos nove quadradinhos, o que continua sendo bastante elevado
- A solução está em encontrar uma boa heurística. Dois candidatos:
  - $h_1$  = Nº de quadradinhos que se encontram em posições erradas
  - $h_2$  = soma das distancias dos quadradinhos para as correspondentes posições do goal. Quadradinhos só se movem na horizontal e na vertical. Isto é conhecido por City block ou Manhattan distance.  $h_1$  e  $h_2$  são admissíveis.

# Iterative Deeping A\*

- Vimos que IDS é técnica útil usada para reduzir os requisitos de memória
- Modificamos o IDS para usar f-cost em vez de limite de profundidade
- Cada iteração expande todos os nós alcançáveis de acordo com f-cost
- IDA\* é óptimo e completo, com requisitos de espaço de acordo com DFS

# SMA\* (Simplified Memory Bounded) Search

- O problema de IDA\* reside no facto de que entre as iterações retem sómente um único número, o f-cost limite corrente
- Assim não regista a sua história, levando a repetições
- IDA\* pode ser modificado para verificar no percurso corrente a ocorrência de estados repetidos, embora não possa evitar que estados repetidos ocorram em percursos alternativos

# SMA\* Search

- Simplified Memory-Bounded A\* (SMA\*) proposto para usar mais memória com objectivo de melhorar o processo de busca
- SMA\* tem as seguintes propriedades/características:
  - Utiliza memória a sua disposição
  - Evita estados repetidos enquanto memo permitir
  - É completo e óptimo se memória for suficiente para guardar o percurso considerado óptimo no momento. De contrário devolve a melhor solução alcançável com memo disponível
  - Quando memo é suficiente para árvore inteira, a busca é eficiente e óptima

# Algoritmos Iterativos Melhorados

- Baseados na ideia geral de que iniciamos com uma configuração completa e vamos modificando pra melhorar sua qualidade
- Dividem-se em duas grandes classes:
  - Hill-Climbing: tenta fazer modificações que melhorem o estado corrente
  - Simulated Annealing: produz alterações que tornam a situação pior, pelo menos temporariamente

# Hill-Climbing

- É um loop simples que continuamente se move na direcção de valores cescntes
- O algoritmo não mantém uma àrvore de Busca, o node regista sómente o estado e sua evolução – denotado por valor
- Quando existe mais do que um sucessor por escolher, a escolha é aleatória



# Hill-Climbing

- Cria dois problemas:
  - Local maxima: ponto que é o menos alto de entre os altos pontos do espaço de busca. O algoritmo é forçado a um exit
  - Plateaux: área do espaço de busca onde o valor da função avaliativa é constante. O processo de busca será aleatório a partir deste momento

# Simulated Annealing

- Baseia-se na ideia de que: em vez de iniciar de novo de forma aleatória quando cai em local maxima, a busca vai pra o fundo tentando escapar-se do local maxima
- Em vez de se tomar o melhor movimento, escolhe-se aleatoriamente
- Se o movimento melhora a situação, é executado. De contrário, o algoritmo realiza o movimento com base num valor provavel menor que 1 (um)
  - **(ver literatura para completar ambos....)**