

Estratégias informadas de Busca

March 3, 2016

Busca de Soluções: Métodos Informados

- Utilizam conhecimento específico do problema para encontrar a solução
- algoritmo geral de busca somente permite introduzir conhecimento na função de enfileiramento
- em métodos informados normalmente utiliza-se uma **função de avaliação** que descreve a prioridade com que um nó deve ser expandido
- Algoritmos **best-first search**: “melhor” nó deve ser expandido primeiro

Algoritmo Best-First Search

```
function BEST_FIRST_SEARCH(problem, EVAL_FN)
    return solucao
    QUEUEING_FN := funcao que ordena os nos
                  de acordo com EVAL_FN
    return GENERAL_SEARCH(problem, QUEUEING_FN)
```

Algoritmo Best-First Search

- melhor nó: o mais próximo do objetivo/estado final
- melhor nó: aquele que está no caminho de menor custo
- custo para atingir o estado final a partir de um determinado nó pode ser estimado, mas não determinado exatamente
- função que estima custos: **heurística**

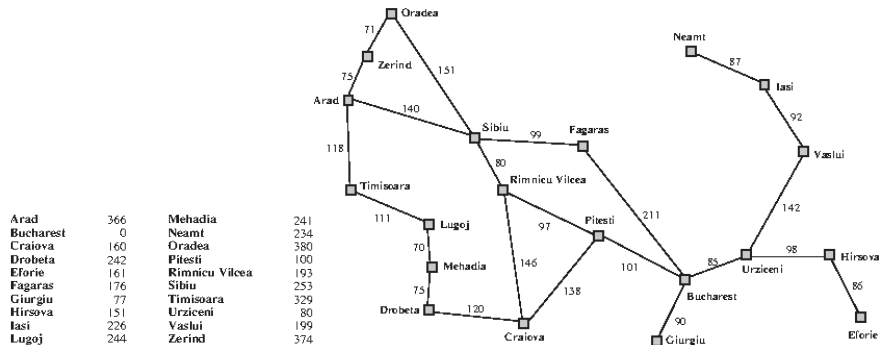
Best-First Search: Estratégia Gulosa

- tenta minimizar custo estimado para chegar à solução
- o nó que está **aparentemente** mais próximo do objetivo é expandido primeiro
- $h(n)$: custo estimado do caminho mínimo entre o estado corrente e o objetivo

```
function GREEDY_SEARCH(problem) return solucao ou falha
    return BEST_FIRST_SEARCH(problem,h)
```

- se n é o estado final, então $h(n) = 0$.

Estratégia gulosa: Exemplo



Arestas no grafo mostram o custo do caminho real entre duas cidades.

A tabela da esquerda mostra a distância em linha reta entre cada cidade e a cidade destino (Bucharest).

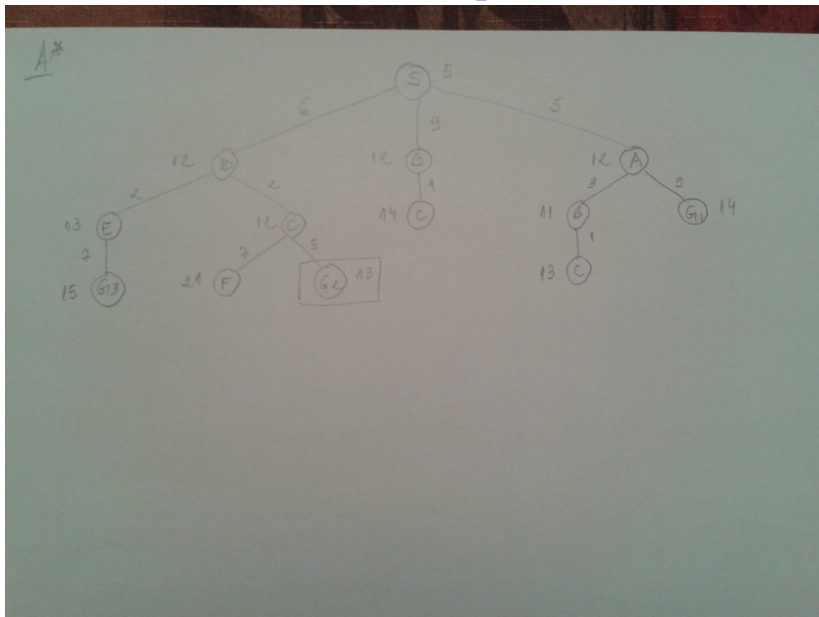
Estratégia gulosa: Análise

- similar à BP porque vai sempre na mesma direção num caminho da árvore para procurar a solução, **mas pode mudar de direção** dependendo do custo dos nós ainda não explorados da árvore de procura
- na presença de “dead-ends” pode ter que escolher caminho de maior custo
- não é ótima
- é incompleta (por *default* não verifica nós repetidos no caminho)
- complexidade temporal: $O(b^m)$
- complexidade espacial: $O(b^m)$
- qualidade de h e tipo de problema podem ajudar a diminuir complexidades temporal e espacial

Best-First Search: A^*

- minimização do custo total do caminho
- busca gulosa diminui o custo estimado para atingir a solução, $h(n)$, mas não é completa nem ótima
- busca de custo uniforme minimiza o custo do caminho da raiz até o nó corrente, $g(n)$. É ótima e completa, mas muito ineficiente
- estratégia melhor: combinação de $h(n)$ com $g(n)$
- $f(n) = g(n) + h(n)$
- é completa e ótima com uma restrição na função h : nunca super-estimar o custo real da melhor solução
- neste caso, h é dita **admissível**
- se h é admissível, $f(n)$ também é admissível.

A*: Exemplo



Best-First Search: A^*

```
function A*_SEARCH(problem) return solucao ou falha  
    return BEST_FIRST_SEARCH(problem,g+h)
```

- características:
 - ▶ f nunca decresce. Isto deveria acontecer com todas as funções que usam heurísticas admissíveis
 - ▶ $\Rightarrow f$ é monótona

A^*

- se f não for monótona, pode-se fazer uma correção para restaurar a monotonicidade
- equação do máximo dos caminhos:
 $f(n') = \max(f(n), g(n') + h(n'))$ com n pai de n'
- esta equação deve ser sempre verificada para garantir monotonicidade de f .

A^* : Prova de Otimalidade

- Seja G uma solução ótima com custo f^*
- Seja $G2$ uma solução sub-ótima, isto é, um estado final com $g(G2) > f^*$ ($h(G2) = 0$)
- Hipótese: A^* seleciona $G2$ da fila. Como $G2$ é um estado final, a busca termina com solução sub-ótima. Provaremos que isto não é possível.
- Prova:
 - ▶ Considere um nó folha n (ainda não expandido) que está no caminho da solução G .
 - ▶ h é admissível, logo $f^* \geq f(n)$ (1)
 - ▶ Se n **não** foi escolhido para ser expandido no caminho de $G2$, foi porque $f(n) \geq f(G2)$
 - ▶ donde: $f^* \geq f(G2)$
 - ▶ Como $G2$ é um estado final, $h(G2) = 0$, logo $f(G2) = g(G2)$
 - ▶ Logo, $f^* \geq g(G2)$ o que contradiz a hipótese.

A^* : Análise

- A^* é completa somente para grafos com fator de ramificação finito
- Complexidade espacial: número de nós expandidos para chegar a um estado final cresce exponencialmente com o tamanho da entrada
- Entretanto, crescimento exponencial não ocorre se o erro na função heurística não crescer mais rápido do que o logaritmo do custo do caminho real:
 $|h(n) - h^*(n)| \leq O(\log(h^*(n)))$, onde $h^*(n)$ é o custo real entre n e o estado final.
- Na prática: crescimento exponencial
- Problema maior: complexidade espacial
- Nenhum outro algoritmo ótimo garante expandir menos nós do que o A^* .

Heurísticas

- Ex: jogo dos oito
 - ▶ h_1 = número de peças na posição errada
 - ▶ h_2 = soma das distâncias das peças às suas posições originais (city block distance ou Manhattan distance)
- devemos dar preferência a uma das duas?

Inventando Heurísticas

- automaticamente: se a definição do problema puder ser descrita em linguagem formal e se os operadores puderem ser “relaxados” removendo condições. Ex:
 - ▶ uma peça pode mover de A para B se A for adjacente a B
 - ▶ uma peça pode mover de A para B se B for o espaço em branco
 - ▶ uma peça pode mover de A para B
- Ex: programa ABSOLVER:
 - ▶ heurística nova para o jogo dos oito melhor que qq heurística existente
 - ▶ encontrou a primeira heurística útil para o cubo mágico

Inventando Heurísticas

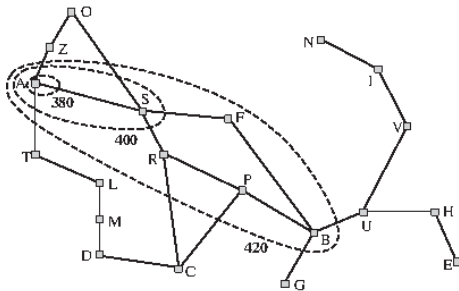
- treinando através de exemplos
- Ex: coletar estatísticas de 100 configurações aleatórias do jogo dos oito.
- Podemos constatar que para 90% das configurações de entrada, a distância real para o nó final é 18, com $h_2(n) = 14$.
- Podemos usar este valor em novas rodadas toda vez que $h_2(n) = 14$.
- qtde menor de nós expandidos, mas função pode deixar de ser admissível.

Busca com Memória Limitada

- *IDA**: busca A^* em profundidade iterativa.
- *RBFs*: Recursive-bounded best-first search.

Busca com Memória Limitada: IDA^*

- IDA^* : tenta encontrar soluções iterativamente variando o valor da função de custo.



- procura solução com custo f . Se não encontrar, retorna novo f ($f1$) e continua procurando solução com custo $f1$, e assim por diante.
- IDA^* é completa e ótima da mesma forma que A^* , mas como a busca é feita em profundidade utiliza menos espaço, que é proporcional ao tamanho do caminho mais longo explorado.

Busca com Memória Limitada: *IDA**

```
function IDA*(problem) return solution
  root <- MAKE_NODE(INITIAL_STATE[problem])
  f_limit <- fcost(root)
  loop
    solution, f_limit <- DFS_CONTOUR(root, f_limit)
    if solution is non-null then return solution
    if f_limit = infinito then return failure
  end
```

Busca com Memória Limitada: *IDA**

```
function DFS_CONTOUR(node,f_limit) return solution
                                e uma nova funcao de custo
next_f <- infinito
if fcost(node) > f_limit then
    return null,fcost(node)
if GOAL_TEST[problem](STATE(node)) then
    return node,f_limit
for each node S in successors(node) do
    solution,new_f <- DFS_CONTOUR(S,f_limit)
    if solution is non-null then
        return solution,f_limit
    next_f <- MIN(next_f,new_f)
end
return null,next_f
```

Busca com Memória Limitada: IDA^* , Análise

- complexidade espacial: na maioria dos casos no. de nós armazenados $b \times d$.
- no pior caso: $\approx \frac{bf^*}{\delta}$, onde f^* custo da solução ótima e δ custo da operação de valor mínimo.
- em geral: IDA^* passa por 2 ou 3 iterações
- eficiência similar a do A^*
- overhead pode ser menor porque nós não precisam ser inseridos na lista em ordem.

Busca com Memória Limitada: *SMA**

- Simplified Memory-Bounded A^*
- Propriedades:
 - ▶ utiliza somente a memória disponível
 - ▶ evita estados repetidos se memória permitir
 - ▶ é completa se memória suficiente para armazenar o caminho da solução menos profunda
 - ▶ é ótima se memória suficiente para armazenar caminho da solução ótima

Busca com Memória Limitada: *SMA**

```

function SMA*(problem) return solucao
  Queue <- MAKE_QUEUE(MAKE_NODE(
    INITIAL_STATE[problem]))
  loop
    if EMPTY?(Queue) then return failure
    n <- no mais profundo de menor custo de Queue
    if GOAL_TEST(n) then return solucao
    s <- NEXT_SUCCESOR(n)
    if not GOAL_TEST(s) e s em nivel maximo then
      f(s) <- infinito
    else
      f(s) <- max(f(n),g(s)+h(s))
    if todos os sucessores de n foram gerados
      atualiza fcost de n e de todos os
      ancestrais, se necessario
    if SUCCESSORS(n) todos em memoria then
      remove n da Queue
    if memory is full then
      rem. no mais raso e de > custo de Queue
      remover este no da lista de suc. do pai
      inserir o pai em Queue, se necessario
    inserir s em Queue
  end

```

Busca com Memória Limitada: RBFS

```

function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
  return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE),  $\infty$ )

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors  $\leftarrow$  []
  for each action in problem.ACTIONS(node.STATE) do
    add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure,  $\infty$ 
  for each si in successors do /* update f with value from previous search, if any */
    s.f  $\leftarrow$  max(s.g + s.h, node.f)
  loop do
    best  $\leftarrow$  the lowest f-value node in successors
    if best.f > f_limit then return failure, best.f
    alternative  $\leftarrow$  the second-lowest f-value among successors
    result, best.f  $\leftarrow$  RBFS(problem, best, min(f_limit, alternative))
    if result  $\neq$  failure then return result

```

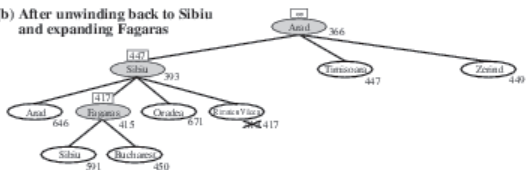
Figure 3.26 The algorithm for recursive best-first search.

Busca com Memória Limitada: RBFS

(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



(c) After switching back to Rimnicu Vilcea and expanding Pitesti

