Resolução de 8-Puzzle com A* em LISP

Léo Willian Kölln

13 de Agosto de 2006

Curso de Ciências da Computação
Programação Funcional - INE5363

INE - Departamento de Informática e Estatística
CTC - Centro Tecnológico
UFSC - Universidade Federal de Santa Catarina

1 O Algoritmo A*

Em Ciências da Computação, A* (pronunciado "A Estrela") é um algoritmo de procura em grafos que encontra um caminho de um dado nó inicial até outro nó de objetivo (ou algum teste de sucesso). Ele emprega a "heuristica de estimativa"que classifica cada nó por uma estimativa da melhor rota atravez de tal nó. Ele visita os nós na ordem desta estimativa heuristica. O algoritmo A* também é um exemplo de procura pelo primeiro melhor resultado.

O algoritmo foi primeiramente definido em 1968 por Peter Hart, Nils Nilsson e Bertram Raphael. Em seu artigo ele foi chamado algoritmo A.

2 8-Puzzle

O 8-puzzle é tipo de "quebra-cabeças N", ou "N-Puzzle".

É um quebra-cabeça de movimento que consiste em uma grade de quadrados numerados onde um deles é vazio, ou não existe, com algo escrito sobre ele. O quebra-cabeça começa embaralhado. Se a grade é 3x3, o quebra-cabeça é chamado de "8-puzzle"ou "9-puzzle". Se a grade for 4x4 será "15-puzzle"ou "16-puzzle", e assim por diante. O objetivo do quebra-cabeça é desembaralhar as peças fazendo apenas movimentos de "deslizar"quadrados para o espaço em branco, fazendo com que apareça outro espaço em branco na posição da peça que foi movimentada.

O "n-puzzle"é um problema clássico de modelagem de algoritmos heurísticos. Geralmente as heurísticas usadas para este problema incluem contar o número das peças faltantes e então encontrar a soma das Distâncias Manhattam entre cada bloco e sua posição ideal (o objetivo). Note que alguns casos são admissíveis, por exemplo, nunca sobre-estimam o numero de movimentos restantes, o que garante otimização para certos algoritmos de procura, como o A*.

Algumas posições iniciais do quebra-cabeça são impossíveis de se resolver, não importa quantos movimentos sejam feitos.

3 Resolvendo o problema em LISP

3.1 Linha de pensamento para a implementação

- Usaremos estruturas para representar os estados do jogo, cada estado será um nó do grafo.
- Iremos manter listas ordenadas dos nós abertos pelo valor de f do estado

- Devemos lembrar que LISP usa ponteiros. Assim sendo, se queremos uma nova lista precisamos explicitamente criar ela os elementos para os quais ela aponta. Para facilitar este processo é que estaremos usando estruturas. Que são mais fáceis de referenciar e instanciar.

3.2 O Código

```
; Primeiramente definimos como serão os estados.
; Cada estado possui um estado de tabuleiro, os valores f g e h além do caminho utilizado
(defstruct estado
(tabuleiro '(1 2 3 4 b 5 6 7 8))
(caminho nil)
(f 0)
(h 0) ; Distância manhattan até a resolução
(g 0)
)

; Cálculo de distância manhattan
(defun manhattan (tabuleiro)
(let
(
(objetivo '(1 2 3 4 b 5 6 7 8))
(sum 0)
(xpos1 nil)
(ypos1 nil)
(xpos2 nil)
(ypos2 nil)
)
(dolist (x tabuleiro sum) ; Para cada peça
(setf xpos1 (mod (position x tabuleiro) 3)) ; Calculamos a distância em coluna
(setf ypos1 (truncate (/ (position x tabuleiro) 3))) ; E a distância em linha
(setf xpos2 (mod (position x objetivo) 3))
(setf ypos2 (truncate (/ (position x objetivo) 3)))
(setf sum (+ sum (+ (abs (- xpos1 xpos2)) (abs (- ypos1 ypos2)))))
)
)
)

; Ordenação por f
(defun sort-by-f (l)
(sort (copy-seq l) #'< :key 'estado-f)
)

; Usamos esta função para expandir um estado em estados filhos
(defun expandir (momento)
(let
(
(filho nil)
(crianca nil)
(oposto nil)
```

```
)
; Executamos cada movimento possível
(dolist (movimento '(cima baixo esquerda direita) crianca)
; Setamos o movimento oposto de cada movimento
(cond
((equal movimento 'cima) (setf oposto 'baixo))
((equal movimento 'baixo) (setf oposto 'cima))
((equal movimento 'esquerda) (setf oposto 'direita))
((equal movimento 'direita) (setf oposto 'esquerda))
)
; Prevenimos a execução de um ciclo duplo
(unless
(equal (first (last (estado-caminho momento))) oposto)
(setf filho (movimentaPeca momento movimento))
)
(when filho
(setf (estado-g filho) (+ (estado-g momento) 1))
(setf (estado-h filho) (manhattan (estado-tabuleiro filho)))
(setf (estado-f filho) (+ (estado-g filho) (estado-h filho)))
(setf (estado-caminho filho) (append (estado-caminho momento) (list movimento)))
(setf crianca (cons filho crianca))
; E só então setamos o filho para nulo novamente
(setf filho nil)
)
)
)
)

; Verifica validade de um movimento
(defun validade (blank direcao)
(cond
((equal direcao 'cima) (when (> blank 2) (- blank 3)))
((equal direcao 'baixo) (when (< blank 6) (+ blank 3)))
((equal direcao 'esquerda)
(when
(and
(not (equal blank 0))
(not (equal blank 3))
(not (equal blank 6))
)
(- blank 1)
)
)
((equal direcao 'direita)
(when
(and
(not (equal blank 2))
(not (equal blank 5))
(not (equal blank 8))
)
```

```
(+ blank 1)
)
)
)
)

; Movimenta a peca
(defun movimentaPeca (momento direcao)
(let*
(
(blank (position 'b (estado-tabuleiro momento)))
(novomomento (make-estado))
(tpos (validade blank direcao))
(tval nil)
)
(setf tpos (validade blank direcao))
(if tpos
(progn
(setf tval (nth tpos (estado-tabuleiro momento)))
(setf (estado-tabuleiro novomomento) (copy-list (estado-tabuleiro momento)))
(setf (nth tpos (estado-tabuleiro novomomento)) 'b)
(setf (nth blank (estado-tabuleiro novomomento)) tval)
novomomento
)
nil
)
)
)

; Adicionamos um novo movimento a árvore de possibilidades
(defun novoMovimento (estado aberto)
(let ((filho nil))
(progn
; Seta filho como a expansão do estado atual
(setf filho (expandir estado))
; Ordenamos por f e então adicionamos aos estados abertos
(setf aberto (sort-by-f (append aberto filho)))
)
)
)

; Verificação de chegada ao objetivo
(defun testaObjetivo (momento)
; Apenas verifica se é igual ao tabuleiro que consideramos o objetivo
(equal (estado-tabuleiro momento) '(1 2 3 4 b 5 6 7 8))
)

; Execução principal do algoritmo
(defun executaAlgoritmoResolucao (aberto raiz)
(setq contagem 0) ; Iniciamos contagem com 0
```

```
(do ((estado (first aberto) (first aberto)))
; Verificamos o estado
(
(or
(null estado)
(and estado (testaObjetivo estado))
(> contagem 1000)
)
; Se a contagem for maior que o permitiro, terminamos o programa sem resolução
(if (> contagem 1000)
(print "Exedido o limite de iteracoes")
(final estado raiz contagem)
)
)
; Executamos os passos
(setf aberto (rest aberto))
; Incrementamos contagem
(setf contagem (+ contagem 1))
; Adicionamos aos nós abertos mais um movimento
(setf aberto (novoMovimento estado aberto))
)
)

; Faz as saídas na tela
(defun final (momento raiz contagem)
(let
((ans nil))
(if (eq momento nil)
(format t "Não foi encontrada uma resolução.~%") ; Caso não encontre Solução
(progn ; Se encontrar solução, mostrar as informações
(terpri)
(terpri)
(format t "Solucao encontrada em ~d nos!~%" contagem)
(exibeSolucao raiz momento)
)
)
)
)

; Função de início
(defun resolver (inicio)
(let
(
(raiz nil) ; A raiz é inicialmente nula
(abertos nil) ; Assim como os nós em aberto
)
; Setamos a raiz como o estado inicial baseado no início informado
(setf raiz
(make-estado :tabuleiro inicio
:g 0
```

```
:h (manhattan inicio)
:f 0
:caminho nil
)
)
; Setamos f de raiz como sendo a soma dos valores g e h
(setf (estado-f raiz) (+ (estado-g raiz) (estado-h raiz)))
; Adicionamos aos nós abertos a raiz
(setf abertos (list raiz))
; Executamos o algoritmo de resolução
(executaAlgoritmoResolucao abertos raiz)
)
)

(defun exibeSolucao (raiz momento)
(format t "A resolucao tem ~a passos: " (length (estado-caminho momento)))
(dolist (i (estado-caminho momento))
(format t "-> ~a " i)
)
(terpri)
)
```

3.2.1 Exemplo de Execução

Ao executar o seguinte código:

```
(resolver '(3 6 8 1 2 4 7 b 5))
```

Teremos o seguinte resultado:

```
Solucao encontrada em 279 nos!
A resolucao tem 19 passos: -> CIMA -> CIMA -> ESQUERDA -> BAIXO -> DIREITA -> BAIXO ->
DIREITA -> CIMA -> CIMA -> ESQUERDA -> BAIXO -> BAIXO -> DIREITA -> CIMA-> ESQUERDA ->
BAIXO -> ESQUERDA -> CIMA -> DIREITA
```

Referências

[1] **Tips for building an A* solution for the Eight Puzzle** - http://morden.csee.usf.edu/cap5625/hints8puz

[2] Wikipedia, **A* search algorithm** - http://en.wikipedia.org/wiki/A%2A

[3] Wikipedia, **N-puzzle** - http://en.wikipedia.org/wiki/15-puzzle