

Comparación de Heurísticas para la Solución del Problema 8-puzzle mediante el Algoritmo A* en Common Lisp

Luis Fernando Espino Barrios
Instituto Tecnológico de Costa Rica
luisespino@yahoo.com
Octubre 2009

Resumen: Este artículo trata acerca de la comparación de dos diferentes heurísticas que se pueden utilizar para el problema del 8-puzzle, estas dos heurísticas son bien conocidas; una es el número de casillas fuera de lugar y la otra es la llamada distancia de Manhattan que calcula la sumatoria de las distancias en número de movimientos de casillas fuera de lugar, ambas están basadas en el nodo objetivo. Dicha comparación puede servir para decir que heurística utilizar en la solución de este tipo de problemas.

Palabras clave: Algoritmo A*, 8-puzzle, algoritmos de búsqueda, resolución de problemas, algoritmos en common lisp.

1. Introducción

El algoritmo a utilizar en este artículo es el llamado A* [1] o A-estrella, llamado así por ser un algoritmo de búsqueda admisible, pertenece al grupo de los algoritmos de búsquedas informadas, la idea principal del algoritmo es proporcionar información para tomar decisiones respecto a la expansión de nodos; un algoritmo eficiente debe poder evaluar los nodos disponibles para determinar cual expandir.

Este algoritmo evalúa los nodos combinando dos costos [2] [3], el costo de venir desde el nodo inicial y el costo de ir al nodo objetivo:

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

Donde f es el costo más barato estimado de la solución a través del nodo n . En este algoritmo, se deben utilizar heurísticas admisibles, es decir, que no sobreestimen el costo de alcanzar el nodo objetivo.

2. 8-puzzle

8-puzzle es una versión del conjunto de juegos n-puzzle, siendo un problema clásico para modelar algoritmos heurísticos.

1	2	3
4	5	6
7	8	

Figura 1: Objetivo del 8-puzzle

La anterior figura es uno de los objetivos del juego, puede haber otras configuraciones acerca del orden de los números; luego se desordena y la idea básica es mover las casillas para alcanzar de nuevo el objetivo.

3. Implementación

Para la implementación se utilizó una computadora tipo laptop, Dell Studio, procesador Intel Centrino con 4 GB de RAM.

Se probó en Windows Vista y en Ubuntu 9.04, con las versiones de GNU Common Lisp (gcl) v.2.6.7. [4] [5]

El estado o nodo objetivo es el mismo descrito en la Figura 1, el cual, se representa de la siguiente manera:

```
(defvar goal
  (make-array '(3 3)
    :initial-contents
    '((1 2 3) (4 5 6) (7 8 0))))
```

En este caso se utilizó un arreglo para representar el objetivo, debido a la fácil manipulación de los datos por medio de sus coordenadas y a través de la función *aref*.

Para desordenar el tablero se utilizó la función:

```
(defun random-moves(state)
  (let ((m (random 4)))
```

Donde *m* es un número aleatorio de 1 a 4, luego se ejecuta el movimiento seleccionado:

```
(cond
  ((= m 0) (move-up state))
  ((= m 1) (move-down state))
  ((= m 2) (move-right state))
  ((= m 3) (move-left state))))
```

Estas funciones describen los posibles movimientos, si el movimiento es posible devuelve un nuevo estado, de lo contrario devuelve *nil*.

Una de las principales funciones que se utiliza sobre arreglos es la función de búsqueda de elemento:

```
(defun get-place(element state)
  (loop for row from 0 to 2
    thereis (loop for col from 0
      to 2
        thereis (when (eq (aref
          state row col) element)
          (list row col))))))
```

La idea es recorrer el arreglo desde sus dos coordenadas y verificar si el elemento es igual al de la posición examinada, *thereis* sirve para cuando en el *loop* ya se encontró el elemento buscado y realiza la salida.

Un nodo generado se compone de tres partes: la primera es el estado en sí, la segunda es el costo del inicio a *n*, y la tercera es el costo heurístico hacia el objetivo.

El primer nodo se construye de la siguiente manera:

```
(list start 0 (heuristic start))
```

La función de heurística se tratará en la siguiente sección.

El algoritmo se implementó por medio del algoritmo *best-first*, realizando previamente una función de ordenamiento [6].

```
(defun successor-order (queue)
  (sort queue
    (lambda (x y)
      (< (+ (second x) (third
        x)) (+ (second y) (third
        y))))))
```

La anterior función realiza un *sort* por medio de una función *lambda* que recibe dos valores, se ordena verificando si la suma de *g(n)* y *h(n)* del primer elemento es menor que la suma del segundo

4. Heurísticas utilizadas

Se realizaron dos tipos de heurísticas, la primera es sumar cuantas casillas están fuera de lugar:

5	2	6
4	1	7
	3	8

Figura 2: Cierta instancia del 8-puzzle

En el Figura 2, el costo calculado con esta heurística es de 6, debido a que solamente los número 2 y 4 están en su posición.

La función se representa de la siguiente manera:

```
(loop for num from 1 to 8
  sum (if (equal
    (get-place num state)
    (get-place num goal)) 0 1)
```

De modo que se comparará cada número *num* de 1 a 8 del estado actual contra la posición del estado objetivo, si los elementos son igual recibe un valor de 0 y si no recibe un valor de 1, por lo que si todos los elementos están fuera de lugar, la heurística calcula un costo de 8.

La segunda heurística es la llamada distancia de Manhattan, la cual, estima el costo de colocar cada casilla fuera de lugar en su lugar original,

1	2	3
4	5	8
6	7	

Figura 3: Cierta Instancia de 8-puzzle

En el ejemplo anterior los elementos fuera de lugar son 6, 7 y 8. El valor 6 necesita de de tres movimiento de lugar para estar en la posición objetivo, el valor 7 solo necesita un movimiento y el valor 8 necesita de dos movimientos, por lo que la heurística calcula un costo de 6.

5. Resultados de las heurísticas

La simulación se realizó con un serie de 10 ejecuciones con un movimiento inicial de 10 pasos para desordenar, los resultados promedio de los nodos calculados de las 10 ejecuciones se muestra en la siguiente figura.

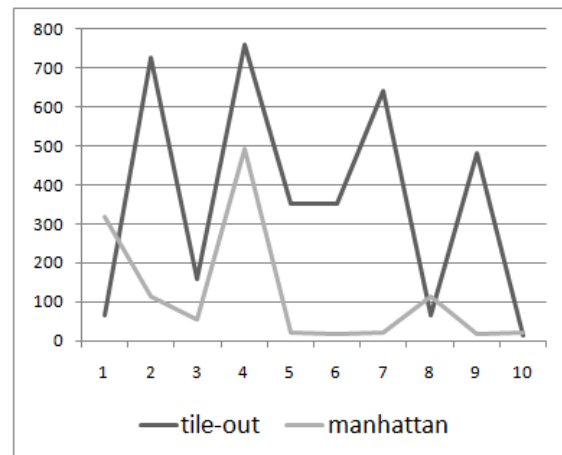


Figura 4: Resultado de la simulación

Los resultados de la gráfica dicen que la heurística de casillas fuera de lugar tiene:

Promedio de nodos abiertos: 361
 Promedio de nodo objetivo: 354
 Promedio de pasos al objetivo: 10

Estos valores dicen que se calcularon 361 sucesores, de los cuales el número 354 fue el objetivo, los pasos desde el inicio al objetivo fueron 10, los mismos pasos que se utilizaron para desordenar el 8-puzzle.

Con respecto a la heurística de distancia de Manhattan tiene:

Promedio de nodos abiertos: 118
 Promedio de nodo objetivo: 109
 Promedio de pasos al objetivo: 9.5

Lo que nos dice que se calcularon 118 sucesores, de los cuales el número 109 fue el objetivo, los pasos desde el inicio al objetivo fueron de 9.5, es decir, hubieron ciertas ejecuciones que lograron mejorar el número de pasos que se llevó para desordenar el 8-puzzle.

Con respecto al número de pasos que el algoritmo calcula para llegar a la solución, generalmente, es el mismo número de pasos para desordenar inicialmente el juego. Esto debido a que la forma de desordenar recuerda el último movimiento e impide regresar en esa dirección, evitando movimientos, por ejemplo, arriba-abajo o derecha-izquierda.

Minimización del número de pasos

En raras ocasiones se logró conseguir que el algoritmo minimice ese número de pasos, uno de los casos se menciona a continuación. La función desordenar realizó los siguientes 12 movimientos:

```
((1 2 3) (4 5 6) (7 8 0)) 'inicial'
((1 2 3) (4 5 0) (7 8 6))
((1 2 3) (4 0 5) (7 8 6))
((1 2 3) (4 8 5) (7 0 6))
((1 2 3) (4 8 5) (7 6 0))
((1 2 3) (4 8 0) (7 6 5))
((1 2 3) (4 0 8) (7 6 5))
((1 2 3) (4 6 8) (7 0 5))
((1 2 3) (4 6 8) (7 5 0))
((1 2 3) (4 6 0) (7 5 8))
((1 2 3) (4 0 6) (7 5 8))
((1 0 3) (4 2 6) (7 5 8))
((0 1 3) (4 2 6) (7 5 8))
```

Y el estado inicial quedo de la siguiente manera:

	1	3
4	2	6
7	5	8

Figura 5: Estado inicial

Es notable que la razón por la que el algoritmo consiguió minimizar el número de pasos fue porque se hicieron movimientos en ciclos, y de esta manera se minimiza el número de pasos, debido a que el algoritmo por medio de la heurística estima el mejor movimiento.

Los pasos para llegar a la solución fueron 4:

```
((0 1 3) (4 2 6) (7 5 8)) 'inicial'
((1 0 3) (4 2 6) (7 5 8))
((1 2 3) (4 0 6) (7 5 8))
((1 2 3) (4 5 6) (7 0 8))
((1 2 3) (4 5 6) (7 8 0))
```

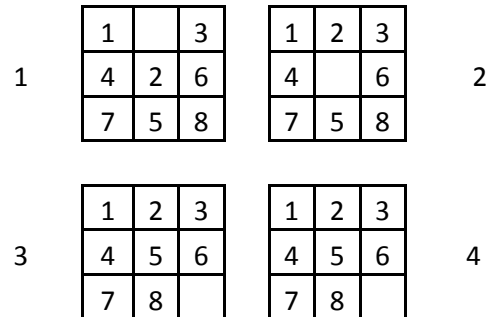


Figura 6: Solución para este caso.

¿Porqué la heurística de distancia de Manhattan tuvo mejor éxito?

La razón principal es la precisión de la estimación del número de movimientos, mientras que la otra heurística solo nos informa cuantos están fuera de lugar, no nos indica que sucesor está más cerca de la solución. Por ejemplo, el siguiente estado inicial tiene varios sucesores, dos ellos son:

4	1	8
2		8
7	6	5

4	1	3
2	8	
7	6	5

4		3
2	1	8
7	6	5

Figura 7: Dos sucesores para el estado inicial.

El sucesor de la izquierda es la solución más eficiente del problema, sin embargo, la heurística de casillas fuera de lugar les dá un valor de 2 a cada sucesor y tendríamos que escoger uno al azar, mientras que la distancia de Manhattan le dio al de la izquierda un valor de 9 y a la derecha 11, esa heurística es más precisa en cuanto a selección.

6. Conclusiones

Se implementó al algoritmo A* en Common Lisp para la solución del problema de 8-puzzle con dos tipos de heurísticas, la primera fue la de casillas fuera de lugar y la segunda fue la distancia de Manhattan.

Con base a los resultados, la heurística distancia de Manhattan es más eficiente que la de casillas fuera de lugar, debido a que logró expandir menor cantidad de nodos, por lo tanto, es más rápida la ejecución y además logró, en ciertos casos, minimizar el número de pasos comparado con los utilizados para desordenar inicialmente el 8-puzzle.

7. Referencias

- [1] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *IEEE Transactions of Systems Science and Cybernetics*, vol. SSC-4 No.2, United States of America, 1968, pp. 100-107.
- [2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. United States of America: Pearson Education, Inc., 2003.
- [3] E. Rich and K. Knight, *Artificial Intelligence*, 2nd ed. United States of America: McGraw-Hill, Inc., 1991.
- [4] G. Steele, *Common Lisp: The Language*, 2nd ed. United States of America: Digital Equipment Corporation, 1990.
- [5] D. Friedman and M. Felleisen, *The Little Schemer*, 4th ed., M. I. o. Technology, Ed. United States of America, 1996.
- [6] L. Espino, "Implementación de Algoritmos Básicos de Búsqueda para la Solución de Problemas en Common Lisp," Instituto Tecnológico de Costa Rica, 2009.

Apéndice

```
;;; Artificial Intelligence, implementation of a-star search 8puzzle
;;; Author: Luis Fernando Espino Barrios
;;; Date: October 2009

(defvar goal
  (make-array '(3 3)
    :initial-contents '((1 2 3)(4 5 6)(7 8 0))))

(defparameter moves 12)

(defparameter last-move 4)

(defparameter depth 0)

(defparameter open '())

(defparameter path '())

(defun random-moves(state)
  (let ((m (random 4))) (setf last-move-tmp last-move) ;(print last-move) (print m)
    (or
      (and
        (or
          (and (= last-move 0) (= m 1))
          (and (= last-move 1) (= m 0))
          (and (= last-move 2) (= m 3))
          (and (= last-move 3) (= m 2)))
        ;(print 'repeated) (random-moves state))
        (random-moves state))
      (and
        (setf last-move m)
        (cond
          ((= m 0) (move-up state))
          ((= m 1) (move-down state))
          ((= m 2) (move-right state))
          ((= m 3) (move-left state))))
      (and
        (setf last-move last-move-tmp) ;(print 'illegal) (random-moves state) )))
    (random-moves state) )))

(defun successor (state) ; successors generation function
  (let ((up (move-up (first state)))
        (do (move-down (first state)))
        (ri (move-right (first state)))
        (le (move-left (first state))))
    (successor-order
      (successor-clean
        (list ;production rules
          (if up
            (if (eq nil (is-member up open))
              (list up (+ (second state) 1) (heuristic up))))
          (if ri
            (if (eq nil (is-member ri open))
              (list ri (+ (second state) 1) (heuristic ri))))
          (if do
            (if (eq nil (is-member do open))
              (list do (+ (second state) 1) (heuristic do))))
          (if le
            (if (eq nil (is-member le open))
              (list le (+ (second state) 1) (heuristic le))))))))) )
```

```

(defun successor-order (queue)
  (sort queue
    (lambda (x y)
      (< (+ (second x) (third x)) (+ (second y) (third y))))))

(defun is-member (x y)
  (cond
    ((null y) nil)
    ((equalp x (car y)) t)
    (T (is-member x (cdr y)))))

(defun successor-clean (queue) ; remove null states
  (cond
    ((null queue) nil)
    ((null (first queue)) (successor-clean (cdr queue)))
    ((cons (first queue) (successor-clean (cdr queue)))))

(defun move-up(state)
  (let* ((zero-pos (get-place 0 state))
        (row (first zero-pos))
        (col (second zero-pos))
        (state-tmp (duplicate state)))
    (when (> row 0)
      (setf (aref state-tmp row col) (aref state-tmp (- row 1) col))
      (setf (aref state-tmp (- row 1) col) 0)
      state-tmp)))

(defun move-down(state)
  (let* ((zero-pos (get-place 0 state))
        (row (first zero-pos))
        (col (second zero-pos))
        (state-tmp (duplicate state)))
    (when (< row 2)
      (setf (aref state-tmp row col) (aref state-tmp (+ row 1) col))
      (setf (aref state-tmp (+ row 1) col) 0)
      state-tmp)))

(defun move-right(state)
  (let* ((zero-pos (get-place 0 state))
        (row (first zero-pos))
        (col (second zero-pos))
        (state-tmp (duplicate state)))
    (when (< col 2)
      (setf (aref state-tmp row col) (aref state-tmp row (+ col 1)))
      (setf (aref state-tmp row (+ col 1)) 0)
      state-tmp)))

(defun move-left(state)
  (let* ((zero-pos (get-place 0 state))
        (row (first zero-pos))
        (col (second zero-pos))
        (state-tmp (duplicate state)))
    (when (> col 0)
      (setf (aref state-tmp row col) (aref state-tmp row (- col 1)))
      (setf (aref state-tmp row (- col 1)) 0)
      state-tmp)))

(defun get-place(element state)
  (loop for row from 0 to 2
    thereis (loop for col from 0 to 2
      thereis (when (eq (aref state row col) element)
        (list row col))))))

(defun duplicate(state)
  (let ((new-state (make-array '(3 3))))
    (loop for row from 0 to 2
      do (loop for col from 0 to 2
        do (setf (aref new-state row col) (aref state row col) )))
    new-state))

(defun heuristic(state)
  (loop for num from 1 to 8
    sum (if (equal (get-place num state) (get-place num goal)) 0 1)
    ; sum ((lambda (x y)
    ;   (+ (abs (- (car x) (car y)))
    ;     (abs (- (cadr x) (cadr y)))))
    ; (get-place num state) (get-place num goal)))
    ; ))

(defun path(queue)
  (loop for x in queue
    do (print (car x))))

(defun astar ()
  (setf nodes 0)
  (setf branches 0)
  (setf depth 0)
  (setf open '())
  (setf result (as1 (list start 0 (heuristic start))))
  (format t "~%Created nodes: ~D~%" (+ branches 1))
  (format t "Goal node: ~D~%" nodes)
  (format t "Steps: ~D~%" (- (list-length result) 1))
  (path result))

(defun as1 (state)
  (setf nodes (+ 1 nodes))
  (push (car state) open)
  (cond
    ((equalp goal (first state)) (list state) )
    ((> depth (- moves 1)) (and (pop open) nil))
    ((let ((children (successor state))) ; pick the best
      (setf branches (+ (length children) branches))
      (setf depth (+ depth 1))
      (let ((result (as2 children )))
        (and result ( cons state result ))))))))

(defun as2 (queue)
  (cond
    ((null queue) (and (pop open) (setf depth (- depth 1))
      nil))
    ((as1 (first queue)))
    ((as2 (cdr queue)))))

; execution

(format t "~%RANDOM MOVES: ~D~%" moves)

(print goal)

(defparameter start
  ((lambda (state)
    (loop repeat moves
      do (setf state (random-moves state))
        (print state) ;(print (get-tile-out state)) (print (get-
distance state))
        finally (return state))) goal))

(format t "~%~%A* SOLUTION:~%" )

(astar)

```