

Implementación de Algoritmos Básicos de Búsqueda para la Solución de Problemas en Common Lisp

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

Resumen: En este artículo se presenta la implementación en Common Lisp de cuatro algoritmos de búsqueda para la solución de problemas, a saber, breadth-first, depth-first, best-first y steepest-ascent hill-climbing. La finalidad de este trabajo es presentar a través de un problema específico la manera en que se pueden resolver los problemas basados en inteligencia artificial con la definición de espacios del problema, búsquedas y estrategias.

Palabras clave: Algoritmos de búsqueda, resolución de problemas, algoritmos en common lisp.

1. Introducción

La importancia de definir espacios es vital para la resolución de problemas, esto incluye especificar el estado inicial, las reglas que definen los movimientos legales y la posición objetivo. En general se debe [1]: Primero, definir un espacio del problema que contiene todas las posibles configuraciones de los objetos relevantes; segundo, especificar uno o más estados de inicio que describen situaciones posibles en que el proceso debe empezar; tercero, especificar uno a mas estados que pueden ser aceptables como soluciones al problema, conocidos como estados objetivo; y cuarto, especificar un conjunto de reglas que describen las acciones disponibles.

A continuación se mencionarán los diferentes algoritmos, luego en el caso de estudio se definirá un problema específico para implementar los algoritmos en el

lenguaje Common Lisp y se hará una breve discusión de resultados.

2. Algoritmos de búsqueda

Los cuatro algoritmos de búsqueda que se implementan en este trabajo se mencionan a continuación, los dos primeros son algoritmos no informados, es decir, la búsqueda se hace a ciegas, mientras que los últimos dos son informados y se necesita de cierta heurística para implementarlos:

- Breadth-First: Conocido como búsqueda primero en anchura, es una estrategia [2] sencilla en la que se expande primero el nodo raíz, a continuación se expanden todos los sucesores del nodo raíz, y así sucesivamente nivel por nivel. Generalmente la implementación de este algoritmo se hace por medio de una cola FIFO first-in first-out.

- Depth-First Backtracking: Este algoritmo en su forma sencilla siempre expande el nodo más profundo en la frontera actual del árbol de búsqueda [2]. En contraste con la búsqueda primero en anchura, este algoritmo se implementa por medio de cola LIFO last-in first-out. La variante de backtracking utiliza menos memoria, ya que esta variación recuerda que sucesor se expande y se puede optimizar a través de verificaciones para lograr el estado objetivo.
- Best-First: Conocido como algoritmo de búsqueda primero el mejor, este algoritmo trata de expandir el nodo más cercano al objetivo, notando que probablemente conduzca rápidamente a una solución [2]. Dicho algoritmo se conoce como un algoritmo voraz, ya que trata de encontrar rápidamente la solución, aunque hay ciertos problemas que convierten la voracidad en una desventaja, por ejemplo, no seleccionar el camino correcto de solución. Hay varias formas de implementación, una es similar a la búsqueda por profundidad, en la que se selecciona el siguiente sucesor mediante cierta heurística pero se debe guardar los estados visitados para no visitarlos de nuevo, así también, para encontrar la solución en caso de no tener éxito con el estado seleccionado; otra implementación es realizar una función de ordenamiento para tener a la izquierda o frontera los mejores estados, luego se realiza una búsqueda por profundidad normal, ya que el árbol está ordenado implícitamente, lo que hace una implementación más fácil y eficiente en cuanto a la búsqueda.
- Steepest-Ascent Hill-Climbing: En su forma sencilla la búsqueda de ascensión de colinas es simplemente un bucle que continuamente se mueve en dirección del valor creciente [2].

Similar a la búsqueda primero el mejor, este es un algoritmo voraz, debido a que trata de encontrar la solución en el menor número de pasos posibles.

En su variación Steepest-Ascent considera todos los movimientos desde el estado actual y con cierta heurística selecciona el mejor como el siguiente estado [1]. La implementación es similar a la búsqueda primero el mejor, con la diferencia que un sucesor se selecciona y los demás son rechazados y nunca serán reconsiderados.

Como es de notar, en ciertos casos existe la posibilidad de que este algoritmo no encuentre solución.

3. Caso de estudio

Se realizó la implementación para un problema específico, el cual, se detalla a continuación.

Un robot debe ejecutar seis pruebas en una competencia, a saber, patinaje artístico, confección de muñecos de nieve, decoración de trineos, pesca, karaoke y 5000 metros en la nieve, en cada prueba se adjunta cierto puntaje.

Estas pruebas pueden repetirse con el afán de mejorar el puntaje, pero en cada repetición se descarga la batería del robot, la batería tiene una capacidad finita de 30 Kwatts, por lo que se debe planificar una buena estrategia.

El objetivo es realizar las seis pruebas, que alcance la batería y lograr un puntaje superior a 4000 puntos para garantizar un éxito en las pruebas.

A continuación se muestra la Tabla 1, que detalla cuantos intentos puede realizar el robot por prueba, el puntaje acumulado y cuanta energía necesita por intento.

| Prueba | Kw | Puntos Por Intento | | | | |
|------------|----|--------------------|-----|------|-----|-----|
| | | 1 | 2 | 3 | 4 | 5 |
| Patinaje | 2 | 525 | 600 | 675 | 750 | 825 |
| Muñecos | 3 | 300 | 600 | 900 | | |
| Decoración | 3 | 450 | 600 | 750 | 900 | |
| Pesca | 4 | 150 | 600 | 1350 | | |
| Karaoke | 3 | 525 | 750 | | | |
| Metros | 4 | 600 | | | | |

Tabla 1. Datos por pruebas individuales

4. Implementación

La implementación se realizó en el lenguaje Common Lisp, este es un dialecto de Lisp, siendo un sucesor de MacLisp influenciado por ZetaLisp [3]. Steele también menciona ciertos objetivos que Common Lisp está destinado a satisfacer, a saber, que sea común, portable, consistente, expresivo, compatible, eficiente, potente y estable.

Una de las principales cuestiones que se atribuyen a Lisp y cualquier variación de dicho lenguaje es la recursividad y el manejo de las listas de manera implícita [4].

Con respecto a la computadora utilizada fue una laptop Dell Inspiron 1501 con procesador AMD Sempron Procesador 3600+ 2.00 GHz con 2 GB de RAM, se realizaron las pruebas en dos sistemas operativos distintos, uno fue Windows Vista y el otro Ubuntu 9.04, en ambos sistemas operativos se ejecutaron los algoritmos en GNU Common Lisp (gcl) v.2.6.7 .

Continuando con la implementación en sí, tal como se vio en la sección 1, se debe tener un estado inicial, un estado objetivo y una serie de reglas de producción de sucesores, que se detallan a continuación.

El estado inicial tiene cuatro elementos, el primer elemento indica la energía utilizada, el segundo elemento indica el puntaje obtenido, el tercer elemento indica el número de prueba, y el último elemento indica el número de intentos de la prueba indicada. Inicialmente el estado tiene sus valores en cero.

```
(defvar start '(0 0 0 0))
```

El estado objetivo debe cumplir con que el primer valor que representa la energía tiene que ser menor o igual que 30, el segundo valor o el puntaje debe ser mayor o igual que 4000 puntos y el tercer elemento debe garantizar que se realizaron las seis pruebas.

```
(defun goal (state) ; goal state
  (and
    (<= (first state) 30)
    (>= (second state) 4000)
    (eq (third state) 6) ))
```

Las reglas de producción deben generar los sucesores de un estado dado, estas reglas están basadas con la Tabla 1, y es relevante mencionar que las pruebas se deben realizar en orden y la repetición de pruebas debe hacerse antes de pasar a la siguiente prueba. Por lo que el robot debe realizar inicialmente la primera prueba, luego puede repetir dicha prueba o bien pasar a la segunda prueba, pero ya no podrá regresar a la anterior prueba.

Una de las reglas se representa cuando el valor de la pruebas es igual a uno, si es así el robot puede realizar la segunda prueba o bien repetir 4 veces más la misma prueba.

```
(cond
  ((and (eq (third state) 1) (< (fourth state) 5))
   (list (+ (first state) 2) (+ (second state) 75) 1
         (+ (fourth state) 1))))
```

Para implementar los algoritmos de búsqueda, en el caso de anchura primero, se debe crear una lista y para el primer elemento se debe calcular los sucesores y se deben colocar al inicio de la lista, luego se debe continuar con los demás elementos para calcular los sucesores de cada nivel por completo.

En el caso del algoritmo de búsqueda por profundidad primero, se deben calcular los sucesores del primer elemento y se envía recursivamente la lista, calculando siempre el primer elemento para expandir.

Dado de esa forma, cada rama puede llegar a un estado u hoja no solución, el algoritmo recursivamente regresa a sus predecesores para probar otra alternativa, sin embargo, para hacer eficiente el algoritmo se hace la variación de backtracking con la verificación que si un nodo ya agotó su energía sería por de más que continuara, por lo que se debe abortar el procedimiento hasta el nivel superior, esta instrucción se da así:

```
((goal state) (list state))
(> (car state) 26) nil)
```

La primera instrucción verifica si el estado actual es el objetivo, si es así, devuelve el estado, sino viene la verificación de backtracking, como es un estado no objetivo se supone que al menos debe terminar la última prueba, la cual necesita de 4 Kw, por lo que si en el primer elemento del estado, o sea, la energía es mayor de 26 ya no podrá completar la última prueba y esa rama debe descartarse por lo que devuelve el valor nil.

Para el algoritmo primero el mejor, se optó por utilizar el mismo depth-first pero debe realizarse previamente un ordenamiento de los sucesores con cierta heurística:

```
(defun best-sort (queue)
  (sort queue
    #'(lambda (x y)
      (> (/ (second x) (first x))
        (/ (second y) (first y))))))
```

Esta función ordena la lista a través de una función temporal lambda tomando como heurística la razón del puntaje entre la energía, para obtener al mejor sucesor.

Por último el algoritmo de búsqueda steepest-ascent hill-climbing varía con el best-first por el ordenamiento, ya que hill-climbing produce solamente un sucesor y descarta todos los demás, por lo que con la función car se descartan los otros sucesores.

```
((let ((children
  (list
    (car
      (best-sort (successor state))))))
```

5. Resultados

Los resultados de las pruebas se realizaron con base a los nodos que son creados por cada algoritmo, se muestran en la Tabla 2.

Los primeros tres algoritmos consiguieron la solución con algunas diferencias entre la generación de nodos, pero las tres encontraron la única solución que es:

```
( PATINAJE MUNECO MUNECO DECORACION
PESCA PESCA PESCA KARAOKE METROS)
```

| Algoritmo | Nodos creados | Nodo Objetivo | Energía Utilizada | Puntaje |
|---------------|---------------|---------------|-------------------|---------|
| breadth-first | 431 | 288 | 30 | 2050 |
| depth-first | 343 | 340 | 30 | 2050 |
| best-first | 93 | 89 | 30 | 2050 |
| hill-climbing | 10 | 10 | - | - |

Tabla 2. Resultados por algoritmo

Cabe destacar que el algoritmo hill-climbing no encontró solución, solamente creó 10 nodos y al robot se le acabó la energía, y como la variación del algoritmo selecciona al mejor estado pero descarta los demás, no hay forma de retroceder en la selección realizada.

Tomando en cuenta solo los tres primeros algoritmos, el algoritmo óptimo en cuanto a la minimización del espacio por nodos es el algoritmo best-first, como se muestra en la Figura 1.

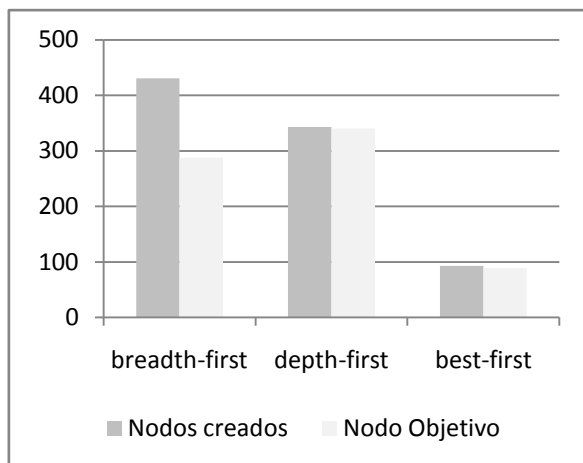


Figura 1. Comparación entre algoritmos por generación de nodos.

Otra comparación realizada se muestra en la Tabla 3, se comparó el tiempo de ejecución de los algoritmos, las instrucciones de Lisp utilizadas fueron “get-internal-run-time” y “get-internal-real-time”.

La diferencia entre las instrucciones utilizadas es que run-time es dependiente de la implementación, que toma métricas tales como el tiempo real, el tiempo de ejecución, los ciclos del CPU y otras cantidades. Mientras que real-time es relativa a un tiempo base arbitrario, y que toma en cuenta solamente el tiempo real entre llamadas [3].

| Algoritmo | Segundos | |
|---------------|------------------------|-----------------------|
| | get-internal-real-time | get-internal-run-time |
| breadth-first | 0.03 | 0.02 |
| depth-first | 0.02 | 0.01 |
| best-first | 0.01 | <0.01 |
| hill-climbing | <0.01 | <0.01 |

Tabla 3. Comparación de tiempo de ejecución entre algoritmos.

Como ya se mencionó para este problema particular el algoritmo de hill-climbing no encontró solución, por eso el corto tiempo de ejecución.

En comparativa entre los tres primeros tres algoritmos el algoritmo best-first tiene el menor tiempo promedio en ambas instrucciones de Lisp.

6. Conclusiones

Se implementaron los algoritmos de búsqueda breadth-first, depth-first, best-first y steepest-ascent hill-climbing en el lenguaje Common Lisp y se realizó dos comparaciones, una acerca del número de nodos que genera cada algoritmo y otra fue el tiempo de ejecución, en ambas pruebas se concluyó que el algoritmo de búsqueda best-first es el más eficiente en cuanto a minimización de espacio y de tiempo de ejecución.

Además, el algoritmo hill-climbing para este caso de estudio en particular no obtuvo ninguna solución, debido a que la variación del algoritmo sugiere la selección de un mejor sucesor y el descartar a los sucesores restantes impide algún tipo de retroceso al llegar a un estado dead-end.

7. Referencias

- [1] E. Rich and K. Knight, *Artificial Intelligence*, 2nd ed. United States of America: McGraw-Hill, Inc., 1991.
- [2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. United States of America: Pearson Education, Inc., 2003.
- [3] G. Steele, *Common Lisp: The Language*, 2nd ed. United States of America: Digital Equipment Corporation, 1990.
- [4] D. Friedman and M. Felleisen, *The Little Schemer*, 4th ed., M. I. o. Technology, Ed. United States of America, 1996.

Apéndice

```
(defvar start '(0 0 0 0)) ; initial state

(defun goal (state) ; goal state
  (and (<= (first state) 30) (>= (second state) 4000) (eq (third state) 6)))

(defun successor (state) ; successors generation function
  (successor-clean (list ;production rules
    (cond ((eq (third state) 0) (list 2 525 1 1)))
    (cond ((and (eq (third state) 1) (< (fourth state) 5))
      (list (+ (first state) 2) (+ (second state) 75) 1
        (+ (fourth state) 1))))
    (cond ((eq (third state) 1) (list (+ (first state) 3)
      (+ (second state) 300) 2 1)))
    (cond ((and (eq (third state) 2) (< (fourth state) 3))
      (list (+ (first state) 3) (+ (second state) 300) 2
        (+ (fourth state) 1))))
    (cond ((eq (third state) 2) (list (+ (first state) 3)
      (+ (second state) 450) 3 1)))
    (cond ((and (eq (third state) 3) (< (fourth state) 4))
      (list (+ (first state) 3) (+ (second state) 150) 3
        (+ (fourth state) 1))))
    (cond ((eq (third state) 3) (list (+ (first state) 4)
      (+ (second state) 150) 4 1)))
    (cond ((and (eq (third state) 4) (eq (fourth state) 1))
      (list (+ (first state) 4) (+ (second state) 450) 4 2)))
    (cond ((and (eq (third state) 4) (eq (fourth state) 2))
      (list (+ (first state) 4) (+ (second state) 750) 4 3)))
    (cond ((eq (third state) 4) (list (+ (first state) 3)
      (+ (second state) 525) 5 1)))
    (cond ((and (eq (third state) 5) (eq (fourth state) 1))
      (list (+ (first state) 3) (+ (second state) 225) 5 2)))
    (cond ((eq (third state) 5) (list (+ (first state) 4)
      (+ (second state) 600) 6 1))))))

(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 show-path(queue) ; show the solution path
  (format t "Created nodes: ~D~%" (+ branches 1))
  (format t "Goal node: ~D~%" nodes)
  (format t "Energy used: ~A~%" (car (car (last queue))))
  (format t "Score: ~A~%" (cadr (car (last queue))))
  (setf path '()) (loop for x in (reverse (cdr queue)) do
    (push (cond
      ((eq (third x) 1) 'patinaje)
      ((eq (third x) 2) 'muneco)
      ((eq (third x) 3) 'decoracion)
      ((eq (third x) 4) 'pesca)
      ((eq (third x) 5) 'karaoke)
      ((eq (third x) 6) 'metros)) path)))

;;; breadth-first search

(defun breadth-first ()
  (setf nodes 0) (setf branches 0)
  (setf result (bfs (list (list start))))
  (show-path (reverse result)) path)

(defun bfs (queue)
  (setf nodes (+ 1 nodes))
  (cond ((null queue) nil)
    ((goal (caar queue)) (first queue))
    ((let ((children (successor (caar queue))))
      (setf branches (+ (length children) branches))
      (bfs (append
        (cdr queue)
        (mapcar #'(lambda (tmp)
          (cons tmp (first queue)))
          children))))))

;;; Depth-first search with backtracking

(defun depth-first ()
  (setf nodes 0) (setf branches 0) (setf result (depth1 start))
  (show-path result) path)

(defun depth1 (state)
  (setf nodes (+ 1 nodes))
  (cond ((goal state) (list state))
    ((> (car state) 26) nil) ;backtracking instruction
    ((let ((children (successor state)))
      (setf branches (+ (length children) branches))
      (let ((result (depth2 children)))
        (and result (cons state result))))))

(defun depth2 (queue)
  (cond ((null queue) nil)
    ((depth1 (first queue)) ((depth2 (cdr queue)))))

;;; Best-first search (solo código adicional al de breadth-first)

((let ((children (best-sort (successor state)))) ; pick the best

(defun best-sort (queue)
  (sort queue
    #'(lambda (x y)
      (> (/ (second x) (first x)) (/ (second y) (first y))))))

;;; Steepest-ascent hill-climbing search
;;; solo se muestra código adicional al de best-first

((let ((children (list (car (best-sort (successor state)))))) ; the one
```