Universidad Autónoma

Inteligencia Artificial

Práctica 2

Memoria

Autores:
Adrián Fernández
Santiago González-Carvajal

Pareja 7
Grupo 2302

27 de marzo de 2018

Índice

1. Modelización del problema

1.1. Heurística

Batería de pruebas

```
» (f-h-galaxy 'Sirtis *sensors*) --> 0
» (f-h-galaxy 'Avalon *sensors*) --> 15
» (f-h-galaxy 'Earth *sensors*)  --> NIL
```

Pseudocódigo

```
Input:  state (estado del problema),
        sensors (lista con tuplas (estado, estimación del coste))
Output: estimación del coste del estado 'state',
        NIL si no se ha encontrado

Process:
f-h-galaxy (state, sensors):
    for sens in sensors:
        if state == state(sens):
            return cost(sens)
    return NIL
```

Código

```
(defun f-h-galaxy (state sensors)
  (unless (null sensors)
    (let* ((sensor (first sensors))
           (planet (first sensor))
           (cost (second sensor)))
      (if (equal state planet)
          cost
        (f-h-galaxy state (rest sensors))))))
```

Comentarios

Esta función consiste en una recursión simple sobre una lista para encontrar un elemento.

1.2. Operadores

Batería de pruebas

```
» (navigate-white-hole 'Uranus *white-holes*)
  --> NIL
» (navigate-worm-hole 'Uranus *worm-holes* *planets-forbidden*)
  --> NIL

» (navigate-white-hole 'Kentares *white-holes*)
  --> (#S(ACTION :NAME NAVIGATE-WHITE-HOLE
                 :ORIGIN KENTARES
                 :FINAL AVALON
                 :COST 3)
       #S(ACTION :NAME NAVIGATE-WHITE-HOLE
                 :ORIGIN KENTARES
                 :FINAL KATRIL
                 :COST 10)
       #S(ACTION :NAME NAVIGATE-WHITE-HOLE
                 :ORIGIN KENTARES
                 :FINAL PROSERPINA
                 :COST 7))

» (navigate-worm-hole 'Mallory *worm-holes* *planets-forbidden*)
  --> (#S(ACTION :NAME NAVIGATE-WORM-HOLE
                 :ORIGIN MALLORY
                 :FINAL KATRIL
                 :COST 5)
       #S(ACTION :NAME NAVIGATE-WORM-HOLE
                 :ORIGIN MALLORY
                 :FINAL PROSERPINA
                 :COST 11))
```

Pseudocódigo

```
Input:  state (estado del problema),
        holes (enlaces del probema),
        forbidden (enlaces no permitidos),
        action-name (nombre de la acción a crear)
```

```
Output: lista de acciones posibles

Process:
navigate (state, holes, forbidden, action-name):
    actions <-- empty_list()
    for link in possible_links(state, holes, forbidden):
        act <-- make_action(action-name,
                            origin(link),
                            final(link),
                            cost(link))
        add(actions, act)
    return actions
```

Código

```
(defun find-lnks (state links forbidden)
  (unless (null links)
    (let ((link (first links))
          (found-links (find-lnks state (rest links) forbidden)))
      (if (and (eql state (first link))
               (not (member (second link) forbidden :test #'eql)))
          (cons (first links)
                found-links)
        found-links))))


(defun navigate (state holes forbidden funct-name)
  (let ((actions (find-lnks state holes forbidden)))
    (mapcar #'(lambda (act) (make-action :name funct-name
                                         :origin (first act)
                                         :final (second act)
                                         :cost (third act)))
      actions)))


(defun navigate-white-hole (state white-holes)
  (navigate state white-holes '() 'navigate-white-hole))
```

```
(defun navigate-worm-hole (state worm-holes planets-forbidden)
  (navigate state worm-holes planets-forbidden 'navigate-worm-hole))
```

Comentarios

Hemos creado una función más general, llamada *navigate*, que funciona como una interfaz para las funciones *navigate-white-hole* y *navigate-worm-hole*. Esta función recibe un estado, los enlaces posibles, los prohibidos y el nombre que queremos dar a la acción y genera todas las acciones posibles desde ese estado.
La función *navigate* utiliza a su vez la salida de la función *find-lnks*, que selecciona los enlaces posibles con origen en un estado dado.

1.3. Test objetivo

Batería de pruebas

```
» (defparameter node-01
    (make-node :state 'Avalon))
» (defparameter node-02
    (make-node :state 'Kentares :parent node-01))
» (defparameter node-03
    (make-node :state 'Katril :parent node-02))
» (defparameter node-04
    (make-node :state 'Kentares :parent node-03))

» (f-goal-test-galaxy node-01 '(kentares urano) '(Avalon Katril)) --> NIL
» (f-goal-test-galaxy node-02 '(kentares urano) '(Avalon Katril)) --> NIL
» (f-goal-test-galaxy node-03 '(kentares urano) '(Avalon Katril)) --> NIL
» (f-goal-test-galaxy node-04 '(kentares urano) '(Avalon Katril)) --> T
```

Pseudocódigo

```
Input:  node (nodo a evaluar),
        planets-destination (lista de planetas destino),
        planets-mandatory (lista de planetas obligatorios)
Output: T si el nodo es meta y pasa por los planetas mandatorios,
        NIL en caso contrario
```

```
Process:
f-goal-test-galaxy (node, planets-destination, planets-mandatory):
    return and(check_mandatory(node, planets-mandatory)
               member(node, planets-destination))
```

Código

```
(defun check-mandatory (node mandatory)
  (unless (null node)
    (let* ((parent (node-parent node))
           (state (find (node-state node)
                        mandatory
                        :test #'eql))
           (new-mandatory (remove state mandatory)))
      (if (null mandatory)
          T
        (or (check-mandatory (node-parent node) new-mandatory)
            (null new-mandatory))))))


(defun f-goal-test-galaxy (node planets-destination planets-mandatory)
  (and (check-mandatory node planets-mandatory)
       (not (null (member (node-state node)
                          planets-destination
                          :test #'eql)))))
```

Comentarios

Hemos construido un función auxiliar, *check-mandatory*, que devuelve una lista con los planetas mandatorios que faltan por visitar desde un nodo. Así, *f-goal-test-galaxy* comprueba que un nodo esté contenido en la lista de metas y que la función anterior evaluada en ese nodo devuelva una lista vacía.

1.4. Predicado de igualdad

Batería de pruebas

```
» (f-search-state-equal-galaxy node-01 node-01) --> T
```

```
» (f-search-state-equal-galaxy node-01 node-02) --> NIL
» (f-search-state-equal-galaxy node-02 node-04) --> T

» (f-search-state-equal-galaxy node-01 node-01 '(Avalon Katril)) --> T
» (f-search-state-equal-galaxy node-01 node-02 '(Avalon Katril)) --> NIL
» (f-search-state-equal-galaxy node-02 node-04 '(Avalon Katril)) --> NIL
```

Pseudocódigo

```
Input:  node1 (primer nodo a evaluar),
        node2 (segundo nodo a evaluar),
        planets-mandatory (lista de planetas obligatorios)
Output: T si los nodos son el mismo planeta y si la lista de
        caminos por visitar de ambos coincide,
        NIL en caso contrario

Process:
f-search-state-equal-galaxy (node1, node2, planets-mandatory):
    return and(state(node1) == state(node2)
               check_mandatory(node1) == check_mandatory(node2))
```

Código

```
(defun f-search-state-equal-galaxy (node-1 node-2
                                    &optional planets-mandatory)
  (and (equal (node-state node-1)
              (node-state node-2))
       (equal (check-mandatory node-1 planets-mandatory)
              (check-mandatory node-2 planets-mandatory))))
```

Comentarios

Para comprobar que ambos nodos son iguales, hemos comprobado que su etiqueta y lista de planetas por visitar sean las mismas. Para ello, hemos utilizado la función *check-mandatory*.

2. Formalización del problema

2.1. Representación LISP del problema

Código

```
(defparameter *galaxy-M35*
  (make-problem
   :states               *planets*
   :initial-state        *planet-origin*
   :f-h                  #'(lambda (node)
                             (f-h-galaxy node *sensors*))
   :f-goal-test          #'(lambda (node)
                             (f-goal-test-galaxy node *planets-destination*
                                                 *planets-mandatory*))
   :f-search-state-equal #'(lambda (node-1 node-2)
                             (f-search-state-equal-galaxy (node-1
                                                           node-2
                                                           *planets-mandatory*)))
   :operators            (list #'(lambda (node) (navigate-white-hole
                                                 node
                                                 *white-holes*))
                               #'(lambda (node) (navigate-worm-hole
                                                 node
                                                 *worm-holes*
                                                 *planets-forbidden*)))))
```

Comentarios

Elegimos como función h *f-h-galaxy*, y como operadores *navigate-white-hole* y *navigate-worm-hole*.

2.2. Expandir nodos

Batería de pruebas

```
» (defparameter node-00
     (make-node :state 'Proserpina :depth 12 :g 10 :f 20) )
```

```
» (expand-node node-00 *galaxy-M35*)
  --> (#S(NODE :STATE AVALON
               :PARENT #S(NODE :STATE PROSERPINA
                               :PARENT NIL
                               :ACTION NIL
                               :DEPTH 12
                               :G 10
                               :H 0
                               :F 20)
               :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE
                                 :ORIGIN PROSERPINA
                                 :FINAL AVALON
                                 :COST 8.6)
               :DEPTH 13   :G 18.6  :H 15  :F 33.6)
       #S(NODE :STATE DAVION
               :PARENT #S(NODE :STATE PROSERPINA
                               :PARENT NIL
                               :ACTION NIL
                               :DEPTH 12
                               :G 10
                               :H 0
                               :F 20)
               :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE
                                 :ORIGIN PROSERPINA
                                 :FINAL DAVION
                                 :COST 5)
               :DEPTH 13   :G 15    :H 5   :F 20)
       #S(NODE :STATE MALLORY
               :PARENT #S(NODE :STATE PROSERPINA
                               :PARENT NIL
                               :ACTION NIL
                               :DEPTH 12
                               :G 10
                               :H 0
                               :F 20)
               :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE
                                 :ORIGIN PROSERPINA
                                 :FINAL MALLORY
```

```
                          :COST 15)
        :DEPTH 13   :G 25    :H 12  :F 37)
#S(NODE :STATE SIRTIS
        :PARENT #S(NODE :STATE PROSERPINA
                        :PARENT NIL
                        :ACTION NIL
                        :DEPTH 12
                        :G 10
                        :H 0
                        :F 20)
        :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE
                          :ORIGIN PROSERPINA
                          :FINAL SIRTIS
                          :COST 12)
        :DEPTH 13   :G 22    :H 0   :F 22)
#S(NODE :STATE KENTARES
        :PARENT #S(NODE :STATE PROSERPINA
                        :PARENT NIL
                        :ACTION NIL
                        :DEPTH 12
                        :G 10
                        :H 0
                        :F 20)
        :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE
                          :ORIGIN PROSERPINA
                          :FINAL KENTARES
                          :COST 12)
        :DEPTH 13   :G 22    :H 14  :F 36)
#S(NODE :STATE MALLORY
        :PARENT #S(NODE :STATE PROSERPINA
                        :PARENT NIL
                        :ACTION NIL
                        :DEPTH 12
                        :G 10
                        :H 0
                        :F 20)
        :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE
                          :ORIGIN PROSERPINA
```

```
                                :FINAL MALLORY
                                :COST 11)
                :DEPTH 13   :G 21    :H 12  :F 33)
        #S(NODE :STATE SIRTIS
                :PARENT #S(NODE :STATE PROSERPINA
                                :PARENT NIL
                                :ACTION NIL
                                :DEPTH 12
                                :G 10
                                :H 0
                                :F 20)
                :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE
                                :ORIGIN PROSERPINA
                                :FINAL SIRTIS
                                :COST 9)
                :DEPTH 13   :G 19    :H 0   :F 19))
```

Pseudocódigo

```
Input:  node (nodo a evaluar),
        problem (problema actual)
Output: lista de nodos accesibles desde 'node'

Process:
expand-node (node, problem):
    hfun <-- f-h(problem)
    nodes <-- empty_list()
    for oper in operators(problem):
        for act in oper(node):
            g <-- g(node) + cost(act)
            h <-- hfun(final(act))
            node <-- make_node(final(act),
                               node,
                               act,
                               depth(node) + 1,
                               g,
                               h,
                               g + h)
```

```
            add(nodes, node)
    return nodes
```

Código

```
(defun expand-node (node problem)
  (mapcan #'(lambda (oper)
              (mapcar #'(lambda (act)
                          (let ((gfun (+ (node-g node)
                                         (action-cost act)))
                                (hfun (funcall (problem-f-h problem)
                                               (action-final act))))
                            (make-node :state (action-final act)
                                       :parent node
                                       :action act
                                       :depth (+ (node-depth node) 1)
                                       :g gfun
                                       :h hfun
                                       :f (+ gfun hfun))))
                        (funcall oper (node-state node))))
    (problem-operators problem)))
```

Comentarios

Esta función crea un nodo para cada acción posible desde un nodo en un problema, lo cual se consigue explorando las acciones posibles de cada operador del problema en el nodo dado.
Todos los campos del nuevo nodo se pueden obtener con los campos del nodo predecesor y los campos de la acción que lleva de uno a otro.

2.3. Gestión de nodos

Batería de pruebas

```
» (insert-nodes-strategy '(4 8 6 2)
                         '(1 3 5 7)
                         (make-strategy :name 'simple
                                        :node-compare-p #'<))
  --> (1 2 3 4 5 6 7 8)
```

```
» (defparameter *uniform-cost*
    (make-strategy :name 'uniform-cost
                   :node-compare-p #'(lambda (n1 n2)
                                       (<= (node-g n1)
                                           (node-g n2)))))

» (defparameter lst-nodes-00
    (expand-node node-00 *galaxy-M35*))

» (insert-nodes-strategy (list node-00 node-01 node-02)
                         lst-nodes-00
                         *uniform-cost*)
  --> (#S(NODE :STATE AVALON :PARENT NIL :ACTION NIL :DEPTH 0 :G ...)
       #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G ...)
       #S(NODE :STATE AVALON
               :PARENT #S(NODE
                          :STATE PROSERPINA
                          :PARENT NIL
                          :ACTION NIL
                          :DEPTH 12
                          :G ...)
               :ACTION #S(ACTION
                          :NAME NAVIGATE-WHITE-HOLE
                          :ORIGIN PROSERPINA
                          :FINAL AVALON
                          :COST 8.6)
               :DEPTH 13
               :G ...)
       #S(NODE :STATE DAVION
               :PARENT #S(NODE
                          :STATE PROSERPINA
                          :PARENT NIL
                          :ACTION NIL
                          :DEPTH 12
                          :G  ...)
               :ACTION #S(ACTION
```

```
                    :NAME NAVIGATE-WHITE-HOLE
                    :ORIGIN PROSERPINA
                    :FINAL DAVION
                    :COST 5)
          :DEPTH 13
          :G ...)
#S(NODE :STATE MALLORY
          :PARENT #S(NODE
                    :STATE PROSERPINA
                    :PARENT NIL
                    :ACTION NIL
                    :DEPTH 12
                    :G ...)
          :ACTION #S(ACTION
                    :NAME NAVIGATE-WHITE-HOLE
                    :ORIGIN PROSERPINA
                    :FINAL MALLORY
                    :COST 15)
          :DEPTH 13
          :G ...)
#S(NODE :STATE SIRTIS
          :PARENT #S(NODE
                    :STATE PROSERPINA
                    :PARENT NIL
                    :ACTION NIL
                    :DEPTH 12
                    :G ...)
          :ACTION #S(ACTION
                    :NAME NAVIGATE-WHITE-HOLE
                    :ORIGIN PROSERPINA
                    :FINAL SIRTIS
                    :COST 12)
          :DEPTH 13
          :G ...)
#S(NODE :STATE KENTARES
          :PARENT #S(NODE
                    :STATE PROSERPINA
                    :PARENT NIL
```

```
                    :ACTION NIL
                    :DEPTH 12
                    :G ...)
          :ACTION #S(ACTION
                    :NAME NAVIGATE-WORM-HOLE
                    :ORIGIN PROSERPINA
                    :FINAL KENTARES
                    :COST 12)
          :DEPTH 13
          :G ...)
#S(NODE :STATE MALLORY
          :PARENT #S(NODE
                    :STATE PROSERPINA
                    :PARENT NIL
                    :ACTION NIL
                    :DEPTH 12
                    :G ...)
          :ACTION #S(ACTION
                    :NAME NAVIGATE-WORM-HOLE
                    :ORIGIN PROSERPINA
                    :FINAL MALLORY
                    :COST 11)
          :DEPTH 13
          :G ...)
#S(NODE :STATE SIRTIS
          :PARENT #S(NODE
                    :STATE PROSERPINA
                    :PARENT NIL
                    :ACTION NIL
                    :DEPTH 12
                    :G ...)
          :ACTION #S(ACTION
                    :NAME NAVIGATE-WORM-HOLE
                    :ORIGIN PROSERPINA
                    :FINAL SIRTIS
                    :COST 9)
          :DEPTH 13
          :G ...)
```

```
#S(NODE :STATE KENTARES :PARENT NIL :ACTION NIL :DEPTH 2 :G ...))
```

Pseudocódigo

```
Input:  nodes (lista de nodos a insertar),
        lst-nodes (lista ordenada de nodos),
        strategy (estrategia de inserción)
Output: unión de ambas listas ordenadas según 'strategy'

Process:
insert-nodes-strategy (nodes, lst-nodes, strategy):
    node <-- extract(nodes)
    new-lst <-- insert-node-strategy(node, new-lst, strategy)
    if len(nodes) == 1:
        return new-lst
    else:
        insert-nodes-strategy(nodes, new-lst, strategy)
```

Código

```
(defun insert-node-strategy (node lst-nodes strategy)
  (if (null lst-nodes)
      (list node)
    (let* ((sorted-node (first lst-nodes))
           (res (funcall (strategy-node-compare-p strategy)
                         node
                         sorted-node)))
      (if res
          (cons node
                lst-nodes)
        (cons sorted-node
              (insert-node-strategy node
                                    (rest lst-nodes)
                                    strategy))))))


(defun insert-nodes-strategy (nodes lst-nodes strategy)
  (unless (null nodes)
```

```
    (let* ((node (first nodes))
           (new-lst-nodes (insert-node-strategy node
                                                lst-nodes
                                                strategy)))
      (if (null (second nodes))
          new-lst-nodes
        (insert-nodes-strategy (rest nodes)
                               new-lst-nodes
                               strategy)))))
```

Comentarios

Para implementar *insert-nodes-strategy*, hemos construido la función auxiliar *insert-node-strategy*, que inserta un nodo en la posición correspondiente de la lista ordenada según la estrategia de *insert-nodes-strategy*.
Esta función auxiliar itera sobre la lista dada hasta que encuentra la posición del nodo o hasta que termina la lista, en tal caso el nodo es insertado al final.
La función original consiste en una serie de llamadas a *insert-node-strategy* con cada nodo de la lista y ampliando a su vez la lista en la que se insertan los nodos con cada llamda.

3. Búsqueda

3.1. Estrategia para la búsqueda A*

Código

```
(defun node-f-<= (node-1 node-2)
  (<= (node-f node-1)
      (node-f node-2)))

(defparameter *A-star*
  (make-strategy
   :name 'A-star
   :node-compare-p #'node-f-<=))
```

Comentarios

La diferencia entre la búsqueda de coste uniforme y la búsqueda A*, es que la segunda tiene en cuenta una estimación (heurística) del coste hasta la meta, además del coste de las acciones realizadas.

3.2. Función de búsqueda

Batería de pruebas

```
» (defun a-star-search (problem)
    (graph-search problem
                  *A-star*))


» (graph-search *galaxy-M35* *A-star*)
  --> #S(NODE :STATE SIRTIS
              :PARENT #S(NODE :STATE PROSERPINA
                              :PARENT #S(NODE :STATE DAVION
                                              :PARENT
      #S(NODE :STATE KATRIL
              :PARENT #S(NODE :STATE MALLORY
                              :PARENT NIL
                              ...)))))

» (a-star-search *galaxy-M35*)
  --> #S(NODE :STATE SIRTIS
              :PARENT #S(NODE :STATE PROSERPINA
                              :PARENT #S(NODE :STATE DAVION
                                              :PARENT
      #S(NODE :STATE KATRIL
              :PARENT #S(NODE :STATE MALLORY
                              :PARENT NIL
                              ...)))))
```

Pseudocódigo

```
Input:  problem (problema a resolver),
        strategy (estrategia para resolver el problema)
```

```
Output: nodo meta del problema,
        NIL si no encuentra la meta

Process:
graph-search-rec (open-nodes closed-nodes problem strategy):
    if open-nodes == NIL:
        return NIL
    node <-- pop(open-nodes)
    if test(node, problem):
        return node
    closed-node <-- find(node, closed-nodes)
    if closed-node == NIL:
        sons <-- expand-node(node, problema)
        new-open <-- insert-nodes-strategy(sons, open-nodes, strategy)
        push(node, closed-nodes)
        graph-search-rec(new-open, closed-nodes, problem, strategy)
    else:
        graph-search-rec(open-nodes, closed-nodes, problem, strategy)
```

Código

```
(defun graph-search-rec (open-nodes closed-nodes problem strategy)
  (unless (null open-nodes)
    (let* ((node (first open-nodes))
           (new-lst (rest open-nodes))
           (test (funcall (problem-f-goal-test problem) node)))
      (if test
          node
        (let ((closed-node (find node
                                 closed-nodes
                                 :test (problem-f-search-state-equal problem))))
          (if (null closed-node)
              (graph-search-rec (insert-nodes-strategy (expand-node node
                                                                    problem)
                                                       new-lst
                                                       strategy)
                                (cons node
                                      closed-nodes)
```

```
                              problem
                              strategy)
            (graph-search-rec new-lst
                              closed-nodes
                              problem
                              strategy)))))))


(defun graph-search (problem strategy)
  (let* ((name (problem-initial-state problem))
         (hfun (funcall (problem-f-h problem) name))
         (node-ini (make-node :state name
                              :parent NIL
                              :action NIL
                              :h hfun
                              :f hfun))
         (open-nodes (list node-ini))
         (closed-nodes ()))
    (graph-search-rec open-nodes closed-nodes problem strategy)))
```

Comentarios

En la función a implementar, *graph-search*, creamos el nodo inicial a partir de los datos del problema y llamamos a la función recursiva, *graph-search-rec*, que realiza la búsqueda en grafo.
En *graph-search-rec*, comprobamos si el primer nodo de la lista-abierta es meta, si es así, devolvemos ese nodo. Si no es meta y no pertenece a la lista-cerada, lo exploramos e insertamos sus hijos en la lista-abierta.
Cabe destacar que en este problema en concreto, la condición de pertenencia a la lista-cerrada se comprueba con la etiqueta y el camino recorrido hasta llegar al nodo.

3.3. Camino seguido y secuencia de acciones

Batería de pruebas

```
» (solution-path nil) --> NIL
» (solution-path (a-star-search *galaxy-M35*))
```

```
  --> (MALLORY KATRIL DAVION PROSERPINA SIRTIS)

» (action-sequence (a-star-search *galaxy-M35*))
  --> (#S(ACTION :NAME NAVIGATE-WORM-HOLE
                 :ORIGIN MALLORY
                 :FINAL KATRIL
                 :COST 5)
       #S(ACTION :NAME NAVIGATE-WORM-HOLE
                 :ORIGIN KATRIL
                 :FINAL DAVION
                 :COST 5)
       #S(ACTION :NAME NAVIGATE-WHITE-HOLE
                 :ORIGIN DAVION
                 :FINAL PROSERPINA
                 :COST 5)
       #S(ACTION :NAME NAVIGATE-WORM-HOLE
                 :ORIGIN PROSERPINA
                 :FINAL SIRTIS
                 :COST 9))
```

Pseudocódigo

```
Input:  node (nodo a evaluar)
Output: camino hasta el nodo evaluado

Process:
solution-path (node):
    if node == NIL:
        return ()
    else:
        parent <-- parent(node)
        path <-- solution-path(parent)
        push(state(node), path)
        return path


Input:  node (nodo a evaluar)
Output: secuencia de acciones que llevan a ese nodo
```

```
Process:
action-sequence (node):
    if node == NIL:
        return ()
    else:
        parent <-- parent(node)
        actions <-- action-sequence(parent)
        push(action(node), actions)
        return actions
```

Código

```
(defun solution-path (node)
  (when node
    (append (solution-path (node-parent node))
            (list (node-state node)))))


(defun action-sequence (node)
  (when node
    (when (node-action node)
      (append (action-sequence (node-parent node))
              (list (node-action node))))))
```

Comentarios

Estas funciones se implementan mediante una recursión que extrae el campo necesario de cada nodo del camino y concatena los resultados en orden inverso.

3.4. Otras estrategias de búsqueda

Batería de pruebas

```
» (solution-path (graph-search *galaxy-M35* *depth-first*))
  --> (MALLORY PROSERPINA SIRTIS KATRIL SIRTIS)

» (solution-path (graph-search *galaxy-M35* *breadth-first*))
```

```
  --> (MALLORY KATRIL DAVION PROSERPINA SIRTIS)
```

Código

```
(defun depth-first-node-compare-p (node-1 node-2)
  T)

(defparameter *depth-first*
  (make-strategy
   :name 'depth-first
   :node-compare-p #'depth-first-node-compare-p))


(defun breadth-first-node-compare-p (node-1 node-2)
  NIL)

(defparameter *breadth-first*
  (make-strategy
   :name 'breadth-first
   :node-compare-p #'breadth-first-node-compare-p))
```

Comentarios

La búsqueda en anchura utiliza una cola para implementar la lista-abierta, mientras que la búsqueda en profundidad utiliza una pila. Esto de puede conseguir forzando a que la función *depth-first-node-compare-p* devuelva T, para que cada nodo descubierto se inserte al principio de la lista y forzando a que la función *breadth-first-node-compare-p* devuelva NIL, para que cada nodo descubierto se inserte al final de la lista.

4. Preguntas

¿Por qué se ha realizado este diseño para resolver el problema de búsqueda?

Porque es un diseño generalizado que nos permite resolver cualquier tipo de problema con cualquier estrategia que elijamos.

¿Qué ventajas aporta?

Aporta versatilidad, ya que permite resolver problemas con diferentes estrategias, sin tener que volver a codificar el algoritmo.

¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?

Para abstraer la estructura lo máximo posible, de manera que sea lo más general posible. También ayuda a simplificar la estructura y facilitar cambios futuros en dichas funciones.

Sabiendo que en cada nodo de búsqueda hay un campo "parent", que proporciona una referencia al nodo a partir del cual se ha generado el actual ¿es eficiente el uso de memoria?

Sí, ya que es la mejor manera de recuperar el camino hasta el nodo utilizando el menor espacio posible, porque sólo guardamos una referencia en cada nodo. Cualquier otro sistema para guardar caminos sería menos eficiente, ya que guardaríamos más referencias por nodo.

¿Cuál es la complejidad espacial del algoritmo implementado?

La complejidad espacial del algoritmo A* es exponencial, ya que descubre todos los nodos vecinos del nodo explorado antes de explorar el siguiente. Esto hace que el uso de memoria aumente de forma exponencial a medida que la meta se aleja del origen.

¿Cuál es la complejidad temporal del algoritmo?

La complejidad temporal del algoritmo A* es exponencial. Igual que con el uso de memoria, el método de exploración hace que el tiempo de ejecución aumente de manera exponencial.

Indicad qué partes del código se modificarían para limitar el número de veces que se puede utilizar la acción "navegar por agujeros de gusano" (bidireccionales).

Podríamos modificar la estructura de nodo para llevar la cuenta de los agujeros de gusano atravesados en un estado de búsqueda, lo cual sería una solución eficaz en cuanto a coste computacional. No obstante, esta solución sacrifica la generalidad del estado de búsqueda.
Otra opción sería calcular el número de agujeros de gusano atravesados accediendo a los antecesores de un nodo. Esta solución no pierde generalidad, pero requiere un coste computacional mayor.