Asignatura	Datos del alumno	Fecha
Razonamiento y planificación automática	Bernal Castillo Aldo Alberto Calderón Zetter María Inés Domínguez Espinoza Edgar Uriel	11 de abril de 2022

Actividad grupal: Resolución de un problema mediante búsqueda heurística

Introducción

Una observación común sobre los problemas de búsqueda en un grafo es que coinciden con la localización de la ruta más corta. Los algoritmos que buscan en grafos representan al mismo como una matriz o una estructura (arbórea) implícita de nodos generada de forma iterativa y expandida según el algoritmo implementado.

Los problemas más prominentes sobre la ruta más corta son: *Single-source shortest path* en el cual se busca un camino que minimiza la suma de los pesos de los bordes constituyentes; y *All pairs shortest paths* que busca caminos para cada dos vértices. Los algoritmos más importantes para estos problemas son: 1) Bellman-Ford, que soluciona Single-source shortest path incluso hay costes negativos; 2) Floyd-Warshall, soluciona All pairs shortest paths; 3) Dijkstra, soluciona Single-source shortest path siempre que los costes sean mayores a cero; y 4) A*, soluciona Single-source shortest path con costes no negativos. (Edelkamp & Schroedl, 2011, p. 47)

Particularmente, A* realiza una búsqueda heurística, característica de la cual carecen otros algoritmos. La heurística se usa para estimar la distancia restante aún sin haber explorado una ruta hasta el objetivo. En este sentido es posible diferenciar entre dos tipos de algoritmos: aquellos informados, como el A* y aquellos no informados como los otros listados en el párrafo anterior. (Edelkamp & Schroedl, 2011, p. 47)

En este documento se muestra una implementación del algoritmo A* para solucionar el problema de un robot que puede moverse en un mapa cuadricular en cuatro direcciones, dos verticales y dos horizontales y debe mover tres cargamentos a lugares específicos.

Sobre el algoritmo A*

El algoritmo A* es considerado comúnmente como una mejora de Dijkstra (cap. Domkin, 2021, p. 7). La estimación heurística más común del algoritmo es la mostrada en la ecuación .

$$f(u) = g(u) + h(u) \tag{1}$$

Dentro de la función $g(u)$ calcula el coste del actual camino, mientras que $h(u)$ es la estimación heurística del coste restante hasta el objetivo, por lo tanto $f(u)$ es la aproximación total del coste de la ruta. (Edelkamp & Schroedl, 2011, p. 69)

Existen variantes sobre A*, constantemente se busca la optimización del algoritmo para un mejor desempeño computacional. La implementación más básica se muestra en el algoritmo 1 (Edelkamp & Schroedl, 2011, p. 70).

Asignatura	Datos del alumno	Fecha
Razonamiento y planificación automática	Bernal Castillo Aldo Alberto Calderón Zetter María Inés Domínguez Espinoza Edgar Uriel	11 de abril de 2022

Algorithm 1 Algoritmo A*

```
Cerrado ← ∅                                          ▷ Inicialización
Abierto ← s                                          ▷ Inserta s en la búsqueda
f(s) ← h(s)                                          ▷ Inicializa estimado
while Abierto ≠ ∅ do                                 ▷ Mientras haya nodos donde buscar
    Elimina u con f(u) mínimo de Abierto             ▷ Selección de nodo a expandir
    Inserta u en Cerrado                             ▷ Actualiza lista de nodos expandidos
    if Meta(u) then                                  ▷ Si llego a la meta
      return Camino(u)                               ▷ Muestra la solución
    else if Sucesor(u) ← (u) then                    ▷ La expansión produce un conjunto sucesor
        for v ∋ Sucesor(u) do                        ▷ Para todo sucesor v de u
            Mejor(u, v)                              ▷ Llamada de subrutina
        end for
    end if
end while
      return ∅                                       ▷ Sin solución
```

Implementación

Para implementar el algoritmo A* se usará la familia de lenguajes Lisp (Code, 2022), por lo que se solicitarán los siguientes requisitos:

- Un intérprete de Lisp compatible con Quicklisp, por ejemplo: SBCL o CLisp.
- Quicklisp.
- Un editor de texto que facilite la interacción con el REPL de Lisp, por ejemplo: GNU Emacs.

En este documento se harán anotaciones mínimas debido a que Lisp es un lenguaje autodocumentado y al leer el código fuente, adjunto a este documento, se podrá leer de forma directa la documentación del mismo.

El código inicia con la preparación del entorno: se instalan las dependencias necesarias y se crea un paquete para el correcto procesamiento del programa.

```
(eval-when (:load-toplevel :compile-toplevel :execute)
  (ql:quickload '("pileup" "iterate")))
(defpackage :a*-search (:use :common-lisp :pileup :iterate))
(in-package :a*-search)
```

Asignatura	Datos del alumno	Fecha
Razonamiento y planificación automática	Bernal Castillo Aldo Alberto Calderón Zetter María Inés Domínguez Espinoza Edgar Uriel	11 de abril de 2022

Continúa el código con la definición de variables a utilizar. Se puede señalar que cada una de las variables son modificables y permitirían adaptar el programa a cualquier problema similar, es decir, que tenga los mismos elementos a considerar.

```
(defvar *size* 4 "Esta será el tamaño del área, el valor es el lado de
   un cuadrado.")
(defvar *barriers* '((0 . 1) (1 . 1)) "Esta variable es una lista de
   cons (X . Y). La posición 0 es la esquina inferior izquierda.")
(defvar *barrier-cost* 100 "No es posible evitar que  se inspeccione
   una pared, pero se eleva el coste, así se impide su elección.")
(defvar *directions* '((0 . -1) (0 . 1) (1 . 0) (-1 . 0)) "Agregar aquí
    los posibles movimientos posibles, de 1 a 8 elementos.")
(defvar *load* 0 "Si *LOAD* es 0 no hay carga sobre el robot")
```

Ahora se define la estructura del nodo, misma que al tener un conjunto de nodos formará un grafo. Note que el coste por nodo es cero, por tanto, solo los movimientos suman al modelo matemático.

```
(defstruct (node (:constructor node))
  "Estructura de datos para un nodo de un grafo."
  (pos (cons 0 0) :type cons)
  (path nil)
  (cost 0 :type fixnum)
  (f-value 0 :type fixnum))
```

Poder imprimir el grafo en forma de mapa ayudará a comprender mejor el resultado del programa, incluso si es un mapa sencillo creado solo por elementos de texto. Es importante destacar que para esta implementación, los nodos se numeran igual que en un plano cartesiano, es decir, el nodo (0.0) está en la esquina inferior izquierda. Por lo tanto se considera que el mapa se imprimirá en espejo horizontal respecto al documento de requerimientos, esto es así porque se usaron expresiones S para representar coordenadas, no un sistema matricial común.

```
(defun print-path (path start end &optional (barriers *barriers*)
                   &aux (size (+ 2 *size*)))
  "Imprime el área de trabajo.  Identifica PATH por medio de un punto
     (.),  START con una r, END
con una m y las paredes con una X. Cualquier otro elemento quedará en
   blanco."
  (format t "~v@{~A~:*~}~%" size "-") ; Borde superior
  ; Área disponible, imprime línea a línea
```

```
    (iter (for y from (1- *size*) downto 0)
          (format t "|") ; Margen izquierdo
          ; Columnas
          (iter (for x from 0 below *size*)
                (format t "~A"
                        (cond ((member (cons y x) barriers :test #'equal)
                               "X")
                              ((equal (cons y x) start) "r")
                              ((equal (cons y x) end) "m")
                              ((Member (cons y x) path :test #'equal) "."
                               )
                              (t " "))))
          (format t "|~%")) ; Margen derecho
    (format t "~v@{~A~:*~}~%" size "-") ; Borde inferior
    (iter
      (for position in path)
      (format t "(~D,~D)" (car position) (cdr position))
      (finally (terpri))))
```

También es necesario crear una serie de funciones secundarias para realizar tareas comunes. Si se imprimiera el resultado de cada una de ellas se obtendría la información más detallada del proceso realizado por el algoritmo, sin embargo, se ha decidido para esta implementación solo imprimir la lista abierta, es decir, las posibilidades de movimiento desde un nodo actual hacia la meta.

```
(defun valid-position-p (position)
  "Regresa T si POSITION es un punto válido en el mapa."
  (let ((x (car position))
        (y (cdr position))
        (max (1- *size*)))
    (and (<= 0 x max)
         (<= 0 y max))))
(defun move (position direction)
  "Regresa un nuevo punto cuando se mueve POSITION en una DIRECTION.
   Asume posiciones válidas."
  (let ((x (car position))
        (y (cdr position))
        (dx (car direction))
        (dy (cdr direction)))
    (format t "Posición posible: (~D . ~D)~%" (+ x dx) (+ y dy))
    (cons (+ x dx) (+ y dy))))
```

Asignatura	Datos del alumno	Fecha
Razonamiento y planificación automática	Bernal Castillo Aldo Alberto Calderón Zetter María Inés Domínguez Espinoza Edgar Uriel	11 de abril de 2022

```
(defun next-positions (current-position)
  "Regresa una lista con los posibles posiciones siguientes."
  (remove-if-not #'valid-position-p
                 (mapcar (lambda (d) (move current-position d)) *
                    directions*)))
```

Se procede a la implementación de la heurística con la distancia de Manhattan. Esta heurística puede ser reemplazada con otras similares, como la distancia euclidiana.

```
(defun distance (current-position goal)
  "Calcula la distancia Manhattan existente desde CURRENT-POSITION
     hasta GOAL."
  (+ (abs (- (car goal) (car current-position)))
     (abs (- (cdr goal) (cdr current-position)))))
```

Ahora es posible escribir el código del algoritmo principal. Es importante notar que esta función recibe como parte de sus argumentos otras dos funciones: la heurística y las posiciones siguientes. Al hacer esto, se facilita la modificación del programa para su adaptación a otros problemas.

```
(defun a* (start goal heuristics next &optional (information 0))
  "Calcula la ruta más corta de START a GOAL usando HEURISTICS. Genera
     la lista de caminos usando NEXT. Si INFORMATION es 1 se
     imprimirán detalles de cada iteración."
  (let ((visited (make-hash-table :test #'equalp))) ; Crea la lista
     cerrada. Nodos visitados
    (flet ((pick-next-node (queue)
             ; Obtiene el primer elemento que forma la cola
             (heap-pop queue))
           (expand-node (node queue)
             ; Expande los nodos de posible avance y los agrega a la
                cola si no han
             ; sido visitados.
             (iter
               (with costs = (node-cost node))
               (for position in (funcall next (node-pos node)))
               (for cost = (1+ costs))
               (for f-value = (+ cost (funcall heuristics position goal
                  )
                                 (if (member position *barriers* :test
                                    #'equal)
```

```
                                    100
                                    0)))
                ; Revisa si el nodo ha sido visitado
                (unless (gethash position visited)
                ; Agrega el nodo a la cola
                (heap-insert
                 (node :pos position :path (cons position (node-path
                    node))
                       :cost cost :f-value f-value)
                 queue)))))
        ; La búsqueda algoritmica
        (iter
          ;; Crea la cola
          (with queue = (make-heap #'<= :name "queue" :size 1000 :key #'
             node-f-value))
          (with initial-cost = (funcall heuristics start goal))
          (initially (heap-insert (node :pos start :path (list start) :
             cost 0
                                        :f-value initial-cost)
                                queue))
          (for counter from 1)
          (for current-node = (pick-next-node queue))
          (for current-position = (node-pos current-node))
          ; Imprime información sobre la iteración
          (when (and (not (zerop information))
                     (zerop (mod counter information)))
            (format t "Nodo ~D, tamaño de la lista abierta: ~D, coste
               actual: ~D~%"
                    counter (heap-count queue)
                    (node-cost current-node)))
          ; Si la posición actual no es GOAL continua
          (until (equalp current-position goal))
          ; Agrega el nodo actual a la lista de visitados
          (setf (gethash current-node visited) t)
          ; Expande el nodo actual
          (expand-node current-node queue)
          (finally (return (values (nreverse (node-path current-node))
                                   (node-cost current-node)
                                   counter)))))))
```

Asignatura	Datos del alumno	Fecha
Razonamiento y planificación automática	Bernal Castillo Aldo Alberto Calderón Zetter María Inés Domínguez Espinoza Edgar Uriel	11 de abril de 2022

Hasta este punto ya sería posible utilizar el código escrito para resolver problemas Single-source shortest path. Más allá de lo anterior, el problema que se desea resolver requiere que un robot viaje de un punto inicial a recoger una carga para después llevarla a otro punto dentro del mapa. Es decir, cada trayecto del robot consiste en dos viajes y debe distinguir cuando lleva carga y cuando viaja solo, este comportamiento corresponde a esta última función.

```
(defun robot (start package goal &key (heuristics #'distance))
  "Define el movimiento de un robot el cual se ubica en START, se mueve
      y recoge un inventario en PACKAGE y lo lleva hasta GOAL."
  (multiple-value-bind (path cost steps)
      (a* start package heuristics #'next-positions 1)
    (format t "La ruta entre el punto inicial (r) al punto final (m) en
        ~D pasos con coste: ~D~%" steps cost)
    (print-path path start package))
  (setq *load* 1)
  (format t "Carga del robot: ~D~%" *load*)
  (multiple-value-bind (path cost steps)
      (a* package goal heuristics #'next-positions 1)
    (format t "La ruta entre el punto inicial (r) al punto final (m) en
        ~D pasos con coste: ~D~%" steps cost)
    (print-path path package goal))
  (setq *load* 0)
  (format t "Carga del robot: ~D~%" *load*))
```

Resultado

Se dividió el problema general en tres problemas menores:

Problema 1 El robot inicia en (2.2), recoge un inventario en (0.0) y lo lleva a (3.3).

Problema 2 El robot inicia en (2.2), recoge un inventario en (2.0) y lo lleva a (3.2).

Problema 3 El robot inicia en (2.2), recoge un inventario en (0.3) y lo lleva a (3.1).

Los códigos para ejecutar estas trayectorias se muestran a continuación.

```
(robot '(2 . 2) '(0 . 0) '(3 . 3))
(robot '(2 . 2) '(2 . 0) '(3 . 2))
(robot '(2 . 2) '(0 . 3) '(3 . 1))
```

El resultado es amplio y descriptivo, incluye el número de nodos que se han examinado, el número de nodos en la lista, el coste actual y los posibles movimientos que pueden seguir. Posteriormente, junto con el mapa que marca la ruta total, se mostrará el número de pasos realizados y el coste total de la operación. Finalmente, se muestran los nodos que componen la ruta y si el resultado de la operación de carga que lleva a cabo el robot[1]. Un ejemplo de esta información se muestra enseguida con el resultado del Problema 2.

```
A*-SEARCH[18]> Nodo 1, tamaño de la lista abierta: 0, coste actual: 0
Posición posible: (2 . 1)
Posición posible: (2 . 3)
Posición posible: (3 . 2)
Posición posible: (1 . 2)
Nodo 2, tamaño de la lista abierta: 3, coste actual: 1
Posición posible: (2 . 0)
Posición posible: (2 . 2)
Posición posible: (3 . 1)
Posición posible: (1 . 1)
Nodo 3, tamaño de la lista abierta: 6, coste actual: 2
La ruta entre el punto inicial (r) al punto final (m) en 3 pasos con coste: 2
------
|    |
|m.r |
| X  |
| X  |
------
(2,2)(2,1)(2,0)
Carga del robot: 1
Nodo 1, tamaño de la lista abierta: 0, coste actual: 0
Posición posible: (2 . -1)
Posición posible: (2 . 1)
Posición posible: (3 . 0)
Posición posible: (1 . 0)
Nodo 2, tamaño de la lista abierta: 2, coste actual: 1
Posición posible: (2 . 0)
Posición posible: (2 . 2)
Posición posible: (3 . 1)
Posición posible: (1 . 1)
Nodo 3, tamaño de la lista abierta: 5, coste actual: 1
Posición posible: (3 . -1)
Posición posible: (3 . 1)
Posición posible: (4 . 0)
Posición posible: (2 . 0)
Nodo 4, tamaño de la lista abierta: 6, coste actual: 2
Posición posible: (2 . 1)
Posición posible: (2 . 3)
Posición posible: (3 . 2)
Posición posible: (1 . 2)
Nodo 5, tamaño de la lista abierta: 9, coste actual: 2
Posición posible: (3 . 0)
Posición posible: (3 . 2)
Posición posible: (4 . 1)
```

[1]Adjunto a este documento se encuentra el archivo `a-star.out`, un documento de texto plano que tiene la bitácora de ejecución completa de la consola de Common Lisp.

```
Posición posible: (2 . 1)
Nodo 6, tamaño de la lista abierta: 11, coste actual: 3
La ruta entre el punto inicial (r) al punto final (m) en 6 pasos con coste: 3
------
|  m |
|r.. |
| X  |
| X  |
------
(2,0)(2,1)(2,2)(3,2)
Carga del robot: 0
NIL
```

Como es posible observar, cada problema a su vez es dividido en dos subproblemas donde primero se encuentra una ruta para llegar al inventario (subproblema a) y posteriormente se encuentra otra ruta para llevar dicho inventario a su destino final (subproblema b). En general la solución genera los siguientes resultados:

Tabla 1: Resultados generales

Problema	Coste	Ruta
1a	4	(2,2)(2,1)(2,0)(1,0)(0,0)
1b	6	(0,0)(1,0)(2,0)(3,0)(3,1)(3,2)(3,3)
2a	2	(2,2)(2,1)(2,0)
2b	3	(2,0)(2,1)(2,2)(3,2)
3a	3	(2,2)(2,3)(1,3)(0,3)
3b	5	(0,3)(0,2)(1,2)(2,2)(3,2)(3,1)

Respecto a tomar los problemas como uno solo, se puede decir que el robot siempre debe estar en la posición (2.2) para recibir una nueva tarea. De ser así, la única consideración adicional a estos resultados es el orden de solución, el problema 2 debería solucionarse antes que el problema 1, pues el segundo inventario corta el paso al primer inventario.

Si los problemas son totalmente secuenciales, entonces el primer problema a resolver es el número dos. posteriormente no importará porque problema se opte porque el coste total de la solución será 24 para ambos casos posibles.

Conclusión

En este documento se implementó una solución al problema de la ruta más corta, mismo que ha sido de gran utilidad en la automatización de tareas. Particularmente se ha solicitado emular el comportamiento de un robot que mueve inventarios a través de un mapa cuadricular en el cual se tienen algunas barreras.

Para solucionar esta situación se investigó que el algoritmo A* se considera una mejora al clásico

algoritmo de Dijkstra, mejorándolo por medio de una heurística, es decir, estimar cuál puede ser el coste total del camino en caso de ser elegido.

Si bien el modelo general de la heurística es sencillo pues consiste tan solo de la suma de funciones, el algoritmo cuenta con un número importante de mejoras y hay numerosos pseudocódigos, algunos más complicados que otros. Tan solo en el material consultado pueden recogerse cuatro pseudocódigos y hasta treinta implementaciones que obedecen a problemas diferentes.

Otro punto negativo está en cómo implementar las posibles barreras que puede contener un mapa. El pseudocódigo no suele contemplar esos casos. La mayoría de las implementaciones simplemente manejan los mapas como un árbol, de esta manera las barreras simplemente son nodos no conectados. En este documento particular, se asumió que cada nodo agregaba un coste de cero, es decir, lo único que cuesta es el movimiento del robot, y solo los nodos que se identifican como una barrera tendrán un coste adicional de cien. La consecuencia es que cada ruta que contiene un nodo con barrera será muy costosa.

A pesar de lo anteriormente dicho, el algoritmo tiende a ser manejable incluso para seguirlo en papel siempre que el mapa sea pequeño debido a que, diferenciándose de Dijkstra, no tiene que recorrer cada nodo del mapa antes de tomar una decisión final. Gracias al uso de la heurística, el algoritmo reconoce que nodos son más prometedores y los inspecciona, omitiendo aquellos nodos menos probables. Además, ofrece la ventaja de estar incluido en muchas bibliotecas escritas en muchos lenguajes de programación.

Un trabajo prometedor sería mejorar el programa para que detectara barreras de forma dinámica, es decir, localizar en tiempo real inventarios u otros robots que estorben al movimiento.

Referencias

Code, R. (2022). A* search algorithm. En *Rosetta code*. `https://rosettacode.org/wiki/A*_search_algorithm`

Domkin, V. (2021). *Programming algorithms in lisp: Writing efficient programs with examples in ansi common lisp*. Apress.

Edelkamp, S., & Schroedl, S. (2011). *Heuristic search: Theory and applications* (p. 47). Elsevier.