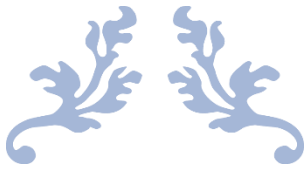


MISIONEROS Y CANÍBALES

Trabajo sobre el problema Teórico de “Misioneros y Caníbales”

Autor1: Diego-Édgar Gracia Peña
Correo: diego.gracia@edu.uah.es
Autor2: David Ariza Asperilla
Correo: david.ariza@edu.uah.es
Carrera: 3º de Ingeniera Informática UAH.
Asignatura: Inteligencia Artificial.

Contenido

MC1 – 3 Misioneros, 3 Caníbales y una barca de 2 plazas

Se consideran las operaciones de paso de una Orilla a otra:

- O1: va un caníbal
- O2: van un misionero
- O3: van 2 misioneros
- O4: van 2 caníbales
- O5: van 1 misionero y 1 caníbal

Contando con estas operaciones, se introduce en los abiertos el nodo inicial, se extrae sus posibles sucesores que cumplan con la condición de que no haya más Caníbales que misioneros en la misma orilla (si la barca está en una orilla, se considera también parte de esa orilla).

Con ello obtenemos los nodos abiertos, que serán ordenados respecto al coste calculado que tenga cada uno en cada iteración de la búsqueda nodal.

El coste de los nodos se obtiene tras obtener estos, y el cálculo sería tal que:

$$((Nº\ Misioneros\ Orilla\ Origen + Nº\ Caníbales\ Orilla\ Origen) * 2 - Valor\ Barca) + Profundidad\ Nodo$$

Se suma el cálculo heurístico y el coste (profundidad del nodo) en llagar a él.

Cuando encontramos el nodo meta, se terminará la búsqueda.

```
#lang racket


(define (sucesores num_op nodo_actual)

  (let* (;Declaramos las variables para las operaciones de trasicion que son
posibles.

         (op_transicion (list (list 0 1 1  0 -1)  ; o1 = 1 canibal se monta
en la barca.

                              (list 1 0 1 -1  0)  ; o2 = 1 misionero se monta
en la barca.

                              (list 2 0 1 -2  0)  ; o3 = 2 misioneros se
montan en la barca.

                              (list 0 2 1  0 -2)  ; o4 = 2 canibales se
montan en la barca.

                              (list 1 1 1 -1 -1)  ; o5 = 1 misionero y 1
canibal se montan en la barca.

                         ))

         (new_sucesor   (list null ))
```

```
        )
    ;Declaramos las expresiones a usar.
    (cond
      [(< num_op (length op_transicion))
         (set! new_sucesor (map (lambda (l1 l2) (+ l1 l2)) (list-ref
op_transicion num_op) nodo_actual))
         (printf " - Betta Sucesor ida a Y: ~v \n" new_sucesor)
         (if (and (>= (list-ref new_sucesor 4) 0) (>= (list-ref new_sucesor
3) 0) (>= (list-ref new_sucesor 0) 0) (>= (list-ref new_sucesor 1) 0) (equal?
(apply + new_sucesor) 7))
              (cond
                [(and (equal? num_op 0) (or (>= (list-ref new_sucesor 0) (+
(list-ref new_sucesor 1)) 1) (equal? (list-ref new_sucesor 0) 0)))
                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))
                 ]
                [(and (equal? num_op 3) (or (>= (list-ref new_sucesor 0) (+
(list-ref new_sucesor 1)) 2) (equal? (list-ref new_sucesor 0) 0)))
                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))
                 ]
                [(and (equal? num_op 1) (and (or (>= (list-ref new_sucesor 3)
(+ (list-ref new_sucesor 4)) 1) (equal? (list-ref new_sucesor 3) 1)) (>=
(list-ref new_sucesor 0) (- (list-ref new_sucesor 1)) 1)))
                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))
                 ]
                [(and (equal? num_op 2) (and (or (>= (list-ref new_sucesor 3)
(+ (list-ref new_sucesor 4)) 2) (equal? (list-ref new_sucesor 3) 2)) (>=
(list-ref new_sucesor 0) (- (list-ref new_sucesor 1)) 2)))
                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))
                 ]
                [(and (equal? num_op 4) (>= (list-ref new_sucesor 0) (list-
ref new_sucesor 1)))
                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))
                 ]
                [else (sucesores (+ num_op 1) nodo_actual)]
              )
```

```
            (sucesores (+ num_op 1) nodo_actual)
          )
      ]
      [else empty] ; Finaliza la construcion de los sucesores.
    )
  )
)


; Calculamos su coste para su ordenacion.
(define (fun n coste)
  (reverse (cons (+ (- (* 2 (+ (list-ref n 3) (list-ref n 4))) (list-ref n
2)) coste) (list n)))
)


(require dyoo-while-loop)
(define (MC1)
  (let* ( ; Declaracion de variables.
         (actual    (list null)                    )  ; Lista de nodos
actuales el camino designado.
         (sucesor   (list null)                    )  ; Lista de sucesores
         (abiertos  (list (list (list 0 0 1 3 3) 0)) )  ; Lista de abiertos
(My, Cy, B, Mx, Cx).
         (meta      (list 3 3 0 0 0)               )  ; Definimos el estao
meta del problema.
        )
    (println "/*                    -----> El formato de los nodos es: <-----
*/")
    (println "/*(Misioneros Destino, canibales Destino, Situacion barca,
Misioneros Origen, canibales Origen)*/")
    (println "")
    (while (and (not (equal? meta (car actual))) (not (empty? abiertos)))  ;
Bucle de busqueda en el arbol.
        (set! actual   (append (list (map (lambda (l1 l2) (- l1 l2)) (caar
abiertos) (list 0 0 1 0 0))) (remove '() actual))) ; Estraemos el elmento de
la lista de actuales l.
        (cond
          [(equal? meta (car actual))
```

```
              (printf "\n/*                      -----> Solucion MC1: <-----
*/\n")

              (printf
"/*****************************************************************/\n")

              (printf " - Numero de movientos: ~s \n - Camino seguido: ~s"
(length actual) (reverse actual))

          ]

          [else

             (printf "\n/*              -----> Nodo Actual: ~v: <-----
*/ \n" (car actual))

             (printf
"/*****************************************************************/\n")

             (printf " Actuales: ~v \n" actual)

             (set! sucesor  (reverse (map (lambda (s1) (fun s1 (length
actual))) (sucesores 0 (car actual)))))

             (printf " Sucesores: ~v \n" sucesor)

             (set! abiertos (sort (append (cdr abiertos) sucesor) #:key last
<))

             (printf " Abiertos: ~v \n" abiertos)

          ]

        )

    )

  )

)
```

MC2 – 4 Misioneros, 4 Caníbales y una barca de 2 plazas

Se ha realizado el cálculo teórico del problema con 4 misioneros, 4 caníbales y una barca de 2 plazas, resultando en todas las búsquedas de nodos un estado cíclico, ya que se alcanza de forma irremediable nodos en los que hay más caníbales que misioneros en la misma orilla.

MC3 – 5 Misioneros, 5 Caníbales y una barca de 3 plazas

Al igual que en MC1, se determina el conjunto de Operaciones y se especifica las restricciones de estas:

- O1: va un caníbal
- O2: va un misionero
- O3: van 2 misioneros
- O4: van 2 caníbales
- O5: van 1 misionero y caníbal
- O6: van 3 caníbales
- O7: van 3 misioneros
- O8: van 1 misionero y 2 caníbales

- O9: van 2 misioneros y 1 caníbal

En cuanto a restricciones, se añade la restricción de que no puede haber más caníbales que misioneros en la barca, descartando automáticamente la operación O8.

Por el resto es igual en cálculo y método a MC1.

```
#lang racket

(define (sucesores num_op nodo_actual)

  (let* (;Declaramos las variables para las operaciones de transicion que son
posibles.

         (op_transicion (list (list 0 1 1  0 -1)  ; o0 = 1 canibal se monata
en la barca.

                              (list 1 0 1 -1  0)  ; o1 = 1 misionero se monta
en la barca.

                              (list 2 0 1 -2  0)  ; o2 = 2 misioneros se
montan en la barca.

                              (list 0 2 1  0 -2)  ; o3 = 2 canibales se
montan en la barca.

                              (list 1 1 1 -1 -1)  ; o4 = 1 misionero y 1
canibal se montan en la barca.

                              (list 3 0 1 -3  0)  ; o5 = 3 misioneros se
montan en la barca.

                              (list 0 3 1  0 -3)  ; o6 = 3 canibales se
montan en la barca.

                              (list 1 2 1 -1 -2)  ; o7 = 1 misionero y 2
canibal se montan en la barca.

                              (list 2 1 1 -2 -1)  ; o8 = 2 misionero y 1
canibal se montan en la barca.

                           ))

         (new_sucesor   (list null ))

        )

    ;Declaramos las expresiones a usar.

    (cond

      [(< num_op (length op_transicion))

         (set! new_sucesor (map (lambda (l1 l2) (+ l1 l2)) (list-ref
op_transicion num_op) nodo_actual))

         (printf " - Betta Sucesor ida a Y: ~v \n" new_sucesor)

         (if (and (>= (list-ref new_sucesor 4) 0) (>= (list-ref new_sucesor
3) 0) (>= (list-ref new_sucesor 0) 0) (>= (list-ref new_sucesor 1) 0) (equal?
(apply + new_sucesor) 11))

             (cond
```

```
                [(and (equal? num_op 0) (or (>= (list-ref new_sucesor 0) (+
(list-ref new_sucesor 1)) 1) (equal? (list-ref new_sucesor 0) 0)))

                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))

                 ]

                [(and (equal? num_op 1) (and (or (>= (list-ref new_sucesor 3)
(+ (list-ref new_sucesor 4)) 1) (equal? (list-ref new_sucesor 3) 1)) (>=
(list-ref new_sucesor 0) (- (list-ref new_sucesor 1)) 1)))

                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))

                 ]

                [(and (equal? num_op 2) (and (or (>= (list-ref new_sucesor 3)
(+ (list-ref new_sucesor 4)) 2) (equal? (list-ref new_sucesor 3) 2)) (>=
(list-ref new_sucesor 0) (- (list-ref new_sucesor 1)) 2)))

                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))

                 ]

                [(and (equal? num_op 3) (or (>= (list-ref new_sucesor 0) (+
(list-ref new_sucesor 1)) 2) (equal? (list-ref new_sucesor 0) 0)))

                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))

                 ]

                [(and (equal? num_op 4) (>= (list-ref new_sucesor 0) (list-
ref new_sucesor 1)))

                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))

                 ]

                [(and (equal? num_op 5) (and (or (>= (list-ref new_sucesor 3)
(+ (list-ref new_sucesor 4)) 3) (equal? (list-ref new_sucesor 3) 3)) (>=
(list-ref new_sucesor 0) (- (list-ref new_sucesor 1)) 3)))

                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))

                 ]

                [(and (equal? num_op 6) (or (>= (list-ref new_sucesor 0) (+
(list-ref new_sucesor 1)) 3) (equal? (list-ref new_sucesor 0) 0)))

                 (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))

                 ]

                [(and (equal? num_op 7)  (>= (list-ref new_sucesor 0) (list-
ref new_sucesor 1)) (or (>= (list-ref new_sucesor 0) (+ (list-ref new_sucesor
1)) 2) (equal? (list-ref new_sucesor 0) 0)))
```

```
                (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))

                ]

               [(and (equal? num_op 8)  (>= (list-ref new_sucesor 0) (list-
ref new_sucesor 1)) (or (>= (list-ref new_sucesor 3) (+ (list-ref new_sucesor
4)) 2) (equal? (list-ref new_sucesor 3) 2)) (>= (list-ref new_sucesor 0) (-
(list-ref new_sucesor 1)) 2))

                (append (sucesores (+ num_op 1) nodo_actual) (cons
new_sucesor '()))

                ]

               [else (sucesores (+ num_op 1) nodo_actual)]

            )

            (sucesores (+ num_op 1) nodo_actual)

         )

      ]

      [else empty] ; Finaliza la construcion de los sucesores.

    )

  )

)


; Calculamos su coste para su ordenacion.

(define (fun n coste)

  (reverse (cons (+ (- (* 2 (+ (list-ref n 3) (list-ref n 4))) (* (list-ref n
2)) 2) coste) (list n)))

)


(require dyoo-while-loop)

(define (MC3)

  (let* ( ; Declaracion de variables.

         (actual    (list null)                      )  ; Lista de nodos
actuales el camino designado.

         (sucesor   (list null)                      )  ; Lista de sucesores

         (abiertos  (list (list (list 0 0 1 5 5) 0)) )  ; Lista de abiertos
(My, Cy, B, Mx, Cx).

         (meta      (list 5 5 0 0 0)                 )  ; Definimos el estado
meta del problema.

        )
```

```
    (while (and (not (equal? meta (car actual))) (not (empty? abiertos)))  ;
Bucle de busqueda en el arbol.

        (set! actual   (append (list (map (lambda (l1 l2) (- l1 l2)) (caar
abiertos) (list 0 0 1 0 0))) (remove '() actual))) ; Extraemos el elmento de
la lista de actuales l.

        (cond

          [(equal? meta (car actual))

              (printf "\n/*                      -----> Solucion MC3: <-----
*/\n")

              (printf
"/*****************************************************************/\n")

              (printf " - Numero de moviento: ~s \n - Camino seguido: ~s"
(length actual) (reverse actual))

          ]

          [else

             (printf "\n/*              -----> Nodo Actual: ~v: <-----
*/ \n" (car actual))

             (printf
"/*****************************************************************/\n")

             (printf " Actuales: ~v \n" actual)

             (set! sucesor  (reverse (map (lambda (s1) (fun s1 (length
actual))) (sucesores 0 (car actual)))))

             (printf " Sucesores: ~v \n" sucesor)

             (set! abiertos (sort (append (cdr abiertos) sucesor) #:key last
<))

             (printf " Abiertos: ~v \n" abiertos)

          ]

        )

    )

  )

)
```

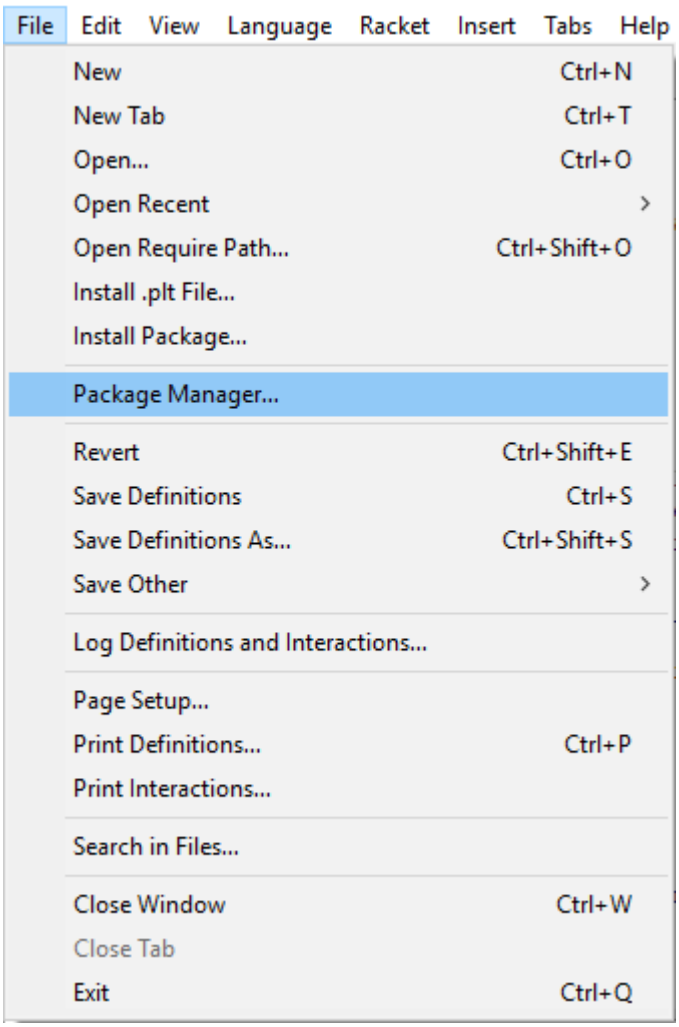
Tutorial de Instalación de las librerías necesarias

En caso de que le falten las librerías necesarias para la ejecución del código, el propio DrRacket proporciona las librerías usadas en el código.

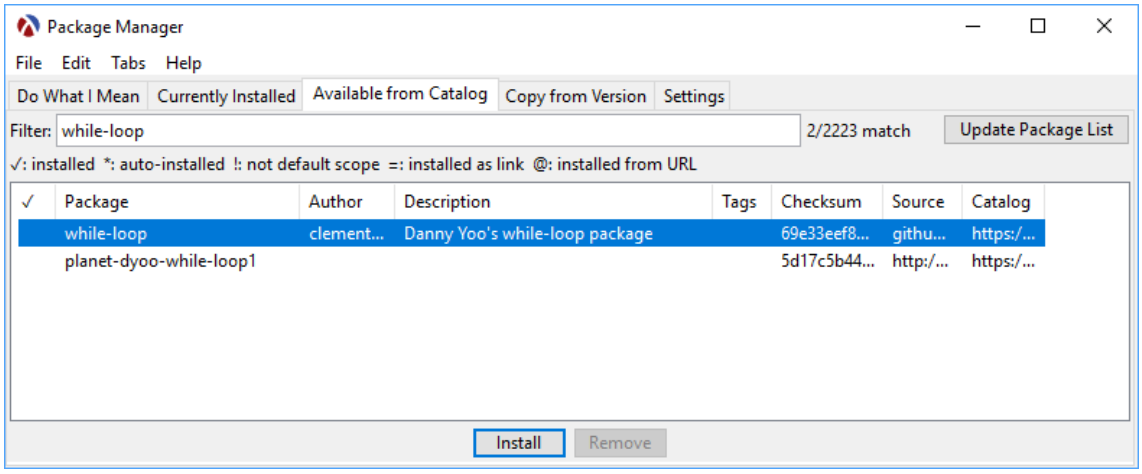
Si recibe este mensaje al ejecutar el código:

```
Welcome to DrRacket, version 6.12 [3m].
Language: racket, with debugging; memory limit: 128 MB.
 standard-module-name-resolver: collection not found
  for module path: dyoo-while-loop
  collection: "dyoo-while-loop"
  in collection directories:
   C:\Users\diego\AppData\Roaming\Racket\6.12\collects
   C:\Program Files\Racket\collects
   ... [165 additional linked and package directories] in: dyoo-while-loop
  no package suggestions are available [update catalog]
>
```

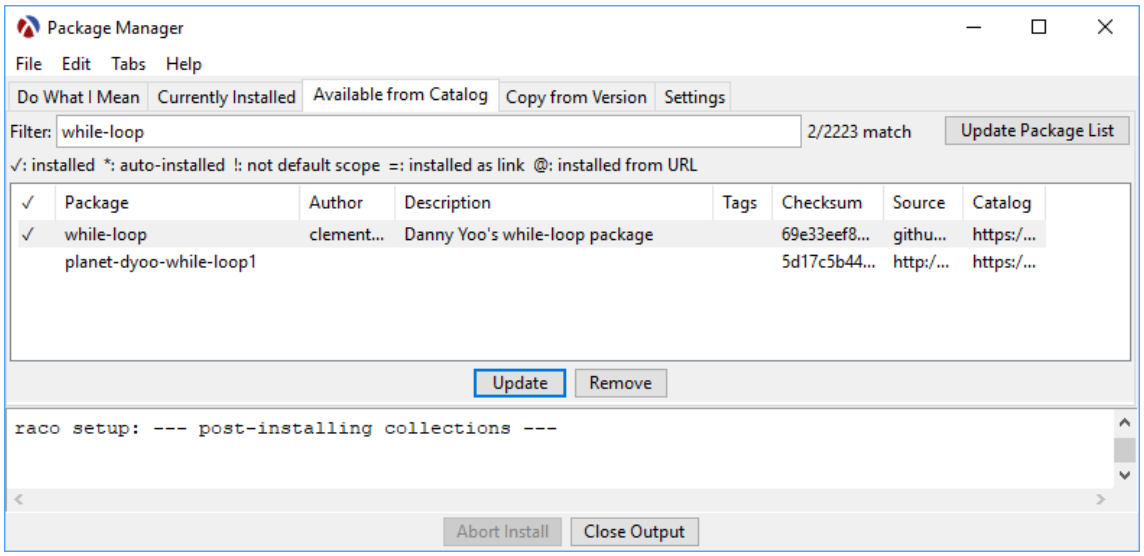
necesita tener instalada la librería “while-loop”, para ello, vaya a “Archivo” o “File” y seleccione “Package Manager…”:



Tras esto, se abrirá esta ventana. Seleccione la pestaña “Available from Catalog” e introduzca el Filter “while-loop”. A continuación, seleccione la opción de la imagen y pulse “Install”:



Tras esto el gestor instalará la librería “While-Loop”, terminando de la siguiente forma:



Cierre la ventana y ya tendrá instalada la librería necesaria. Vuelva a ejecutar el código.