Inteligencia Artificial

PL1

Tema: Búsqueda no informada e informada.

Integrantes:

Ana Cortés Cercadillo

Carlos Javier Hellín Asensio

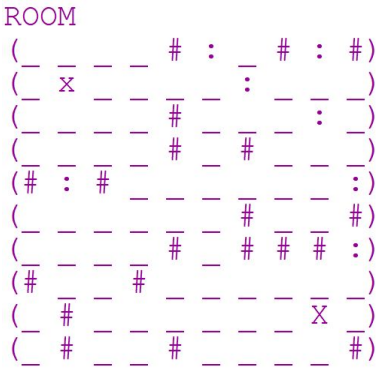
Daniel Ferreiro Rodríguez

Curso: 2020-2021

Índice

Enunciado

En un videojuego 2D de tipo roguelike, se generan las mazmorras proceduralmente mediante un sencillo algoritmo. Este algoritmo genera una rejilla de NxM celdas, representadas con '_', coloca en ella un cierto número de paredes que bloquean el paso, representadas con '#', situadas en posiciones aleatorias y finalmente coloca, en las posiciones que aún estén libres, un cierto número de puertas, representadas con ':'. La celda (0,0) se corresponde con la esquina superior izquierda, mientras que la celda (N-1, M1) se corresponde con la esquina inferior derecha. La mazmorra tiene siempre una casilla de salida, representada con 'x' situada en la posición (1,1), y una casilla de meta, representada con 'X' y situada en la posición (N-2, M-2). El algoritmo se asegura de que la salida y la meta siempre estén libres.



Ejemplo de una de las mazmorras generadas.

Retos realizados

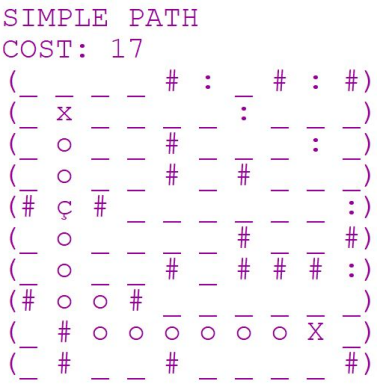
Los 4 retos se han desarrollado **completamente** ajustándose a las especificaciones pedidas en esta práctica.

¿Cómo abordamos el problema?

1. Leímos la práctica en repetidas ocasiones para tener claro lo que se nos pedía.
2. Antes de programar nada, estudiamos a fondo el funcionamiento en el papel de los algoritmos de búsqueda no informada (anchura, profundidad, coste uniforme, bidireccional) e informada el de A*.
3. Escogimos la **búsqueda en anchura** y el **A*** para el **reto 2** y **3**, respectivamente porque entendimos que eran los algoritmos más óptimos para los retos en cuestión.
4. Puesta en marcha. ¡**A programar**!

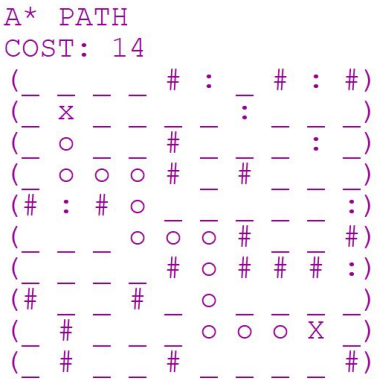
Algoritmos empleados

Búsqueda en anchura:
Completitud: El algoritmo siempre encuentra una solución de haberla.
Optimalidad: La solución obtenida es óptima respecto al número de pasos desde la raíz, si los operadores de búsqueda tienen coste constante, el coste de la solución sería el óptimo.



Ejemplo de camino obtenido con la búsqueda en anchura.

A*:
Ya que consiste en intentar combinar las ventajas de la búsqueda desinformada de coste uniforme (completitud y optimalidad) con las de la búsqueda heurística primero el mejor (eficiencia, por la poda del árbol de búsqueda). Nuestro objetivo no es solo llegar lo más rápidamente a la solución, sino encontrar la de menor coste, tendremos que tener en cuenta el coste de todo el camino y no solo el camino por recorrer.



Ejemplo de camino obtenido con el algoritmo A*.

Ideas descartadas

En el reto 3 implementamos el algoritmo A* y pensamos utilizar esa misma función para resolver el reto 2, tan solo teníamos que crear un nuevo parámetro que fuera la heurística, pasándole el valor 0 si era el reto 2 y el resultado de calcular la heurística Manhattan para el reto 3. Ventajas: sencillez, claridad y se lograría el mismo objetivo con la mitad de esfuerzo.
Problema: Por recomendación del profesor, dicha idea fue desestimada ya que se quería que los retos 2 y 3 se resolvieran con 2 algoritmos diferentes.

Algoritmos descartados

Los algoritmos de búsqueda sin información descartados:

1. **Búsqueda en profundidad**: Fue descartada porque no es completa ni óptima.
2. **Búsqueda en profundidad iterativa:** es equivalente a la búsqueda en anchura, pero usa mucha menos memoria; en cada iteración, visita los nodos del árbol de búsqueda en el mismo orden que una búsqueda en profundidad, pero el orden en el que los nodos son visitados finalmente se corresponde con la búsqueda en anchura.
   Esta opción la descartamos porque añadía más complejidad a nuestro código y no mejoraría mucho ni el tiempo ni el rendimiento. Además, nuestra intención era hacer funciones recursivas siempre que se pudiera.
3. **Búsqueda de coste uniforme**: Al leer el enunciado de la práctica, por primera vez, pensamos en implementar el algoritmo de búsqueda de coste uniforme, básicamente por las pistas que nos daba el texto, pero nos dimos cuenta que el coste a utilizar era constante siendo equivalente a hacer una búsqueda anchura.
4. **Búsqueda bidireccional**: Se explorará al mismo tiempo el estado inicial hacia la meta y viceversa (usando la búsqueda en anchura) hasta que se encuentren en un estado común. Desventaja al tener que comprobar en cada paso si se ha encontrado dicho estado.
   Ello ocurrirá necesariamente si por lo menos una de las dos búsquedas se efectúa en anchura. Optamos por la opción más sencilla, implementar una búsqueda en anchura en vez de dos.

Problemas encontrados

Reto 1

Problema: la condición de parada y los valores i,j de nuestra función recursiva de generar habitación, no sabíamos cuándo sumar o no en las llamadas recursivas.
Solución:
Comprobar si i fuera igual al número de filas (condición de parada).
Cuando j llegara a (columna - 1) la llamada recursiva sería:
i+1
j = 0
y si no se llegara a cumplir la condición sería i y el otro sería j+1.

Reto 2

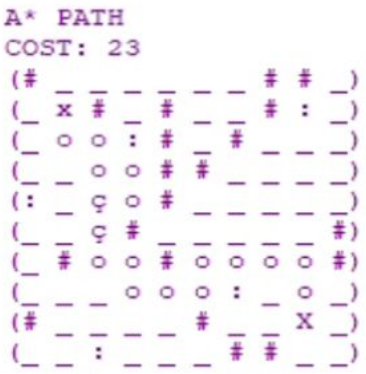
Problema: a la hora crear la lista de sucesores se añade los hijos después de comprobar si es un sucesor válido provocando que haya voids en la lista. Queríamos evitar esos voids.
Solución: Hacer la comprobación de si es válido antes de añadirlo a la lista, usando filter.

Reto 3

Problema: Tuvimos problemas para hacer el A*.

Solución: Decidimos hacer primero el coste uniforme para luego añadirle la función heurística y así reducir la complejidad del problema.

Problema: con la función backtracking del principio (backtracking-antiguo), vimos que funciona bien para anchura pero cuando hay costes como el A* daba problemas. El camino solución que se terminaba pintando en la consola no era correcto.



Esta función está incorporada en el código pero no la usamos, la hemos dejado porque al profesor le resultó interesante nuestra solución.

```
(define (backtrack-antiguo camino hijos cerrados)
  (let (
        [cerrado (car cerrados)]
       )
    (cond
      [(es-salida? cerrado) camino]
      [(list? (member cerrado hijos)) (backtrack-antiguo (append (list cerrado) camino) (sucesores cerrado) (remove cerrado cerrados))]
      [else (backtrack-antiguo camino hijos (cdr cerrados))]
    )
  )
)
```

Solución: Tener al padre junto al hijo en lista de ABIERTOS y CERRADOS. Con este último se hace el backtracking para obtener el camino.

Problema: No sabíamos cómo ir acumulando el valor del coste total necesario para calcular la prioridad en la cola de ABIERTOS.
Solución: Pensamos actualizar con el procedimiento map los costes, iniciando los nodos previamente con valor infinito pero descartamos esa idea. Finalmente, se suma el coste acumulado hasta el momento (coste del padre) con el coste de esa posición (1 ó 4) en el momento que se va a ordenar.

Reto 4

Problema con raco cover: Se marcaba todo en verde por lo que se obtenía un % de acierto mayor al esperado.
Solución: Comentar la llamada de (run).

Problema con test: La mazmorra se generaba aleatoriamente cada vez que ejecutamos el código y no nos dejaba realizar mejores tests.
Solución no encontrada: Intentamos predefinir una mazmorra que solo se tuviese en cuenta al hacer el “raco test” para realizar mayores y mejores tests pero no llegamos a conseguirlo con éxito por lo que hicimos los tests que admitían cualquier mazmorra aleatoria.

Explicación final/consideraciones

Conjunto de operaciones básicas que programamos antes de implementar los algoritmos:
La **celda** es una lista compuesta por dos valores, ejemplo '(1 2), el primer número es la fila y el segundo la columna.
(celda p) devuelve el elemento de la mazmorra que se encuentra en la posición especificada.
(es-camino? p) devuelve true si el elemento que se encuentra en esa celda es diferente a '#' (pared).
(es-salida? p) devuelve true si el elemento en la celda dada es igual a 'x'.
(es-meta? p) devuelve true si el elemento en la celda dada es igual a 'X'.
(en-lista? elemento lista) devolverá true si encuentra el elemento en la lista proporcionada y false en caso contrario. La lista está formada por parejas y busca si el elemento coincide con la primera posición de la pareja.
(izquierda p) devuelve la celda que está a la izquierda de la dada.
Coordenada de la nueva celda: El valor de la fila es igual y se le resta 1 al valor de la columna de la celda dada.
(derecha p) devuelve la celda que está a la derecha de la dada.
Coordenada de la nueva celda: El valor de la fila es igual y se le suma 1 al valor de la columna de la celda dada.
(arriba p) devuelve la celda que está arriba de la dada.
Coordenada de la nueva celda: Al valor de la fila se le resta 1 y el de la columna es igual al de la celda dada.
(abajo p) devuelve la celda que está a la abajo de la dada.
Coordenada de la nueva celda: Al valor de la fila se le suma 1 y el de columna es igual al de la celda dada.
(posicion-fila p) devuelve el valor de la fila de la posición dada. Retorna el primer valor de la pareja de números que ubican la celda en la mazmorra.
(posicion-columna p) devuelve el valor de la columna de la posición dada. Retorna el segundo valor de la pareja de números que ubican la celda en la mazmorra.

Ficheros entregados

El fichero zip de la entrega posee la siguiente distribución:

- ia_practica1.rkt: código fuente del programa y los test unitarios.
- Documentación.pdf: memoria del proyecto.
- Dos carpetas: coverage-conrun y coverage-sinrun.

Carpeta **coverage-conrun**: obtenemos mayor porcentaje, pero el resultado no es tan exacto porque incluye las funciones de imprimir por pantalla cuando no le hemos hecho tests a esas.
Carpeta **coverage-sinrun**: Obtuvimos menor porcentaje comparado con el coverage-conrun sólo tiene en cuenta las funciones que realizamos el test y obvia las de imprimir por pantalla. Las funciones de imprimir por consola se ponen en rojo, sino sería el 100%.