Universidad del Valle de Guatemala
Proyecto # 1: Primera Fase
Miércoles 4 de Agosto de 2016
Algoritmos y Estructura de Datos
Douglas Barrios

José Javier Jo, Carnet: 14343
Jonnathan Juarez, Carnet: 15377
Diego Castañeda, Carnet: 15151

Proyecto # 1: Laberinto
Primera Fase

Algoritmos

Existen una cantidad amplia de algoritmos creados con el fin de que un objeto pueda escapar de un laberinto eventualmente. Entre los encontrados para la investigación están:

Algoritmo Aleatorio del Ratón (Random Mouse Algorithm): Un algoritmo cuya característica es la de tomar decisiones sin procedencia y al azar. De esta manera, el objeto seguirá un camino recto hasta encontrarse con una decisión, la cual tomará aleatoriamente y el cual llegará, eventualmente y con un tiempo indefinido a la meta. Debido a que puede llevarse un tiempo indefinido para recorrer el laberinto, no se utilizara para este proyecto.

Algoritmo Recursivo: Tiene la capacidad de llevar al sujeto hasta el final desde un punto X y Y. Si el punto inicial es X y Y y no está en una pared, manda a llamarse a sí mismo para sus adyacencias y, de esa manera, encontrar el camino hacia la salida. No se utilizara debido a que se desconoce el punto de partida y el punto de salida, haciendo imposible el utilizar este método.

Algoritmo de Rastreo del Laberinto (Maze-Routing Algorithm): Este método utiliza tres parámetros de entrada para poder resolver el problema del laberinto. El primero es el origen o punto de partida, luego el destino a donde se desea llegar y finalmente la posición donde se encuentra actualmente. En este algoritmo se hace una línea imaginaria entre la posición actual y la posición destino, en donde se determina a qué dirección ir dependiendo de que movimiento hará la distancia entre la posición actual y la destino aún menor.

Algoritmo de Seguidor de Muros (Wall follower): Este algoritmo es llamado como seguidor de muros debido a que utiliza la regla de “la mano izquierda ó derecha” ya que utiliza cualquiera de los dos muros laterales del laberinto. Para ello el laberinto debe contar con paredes conectadas entre sí y posteriormente se debe ir analizando si se tiene contacto con el muro. La solución del laberinto es garantizada que no se perderá y que encontrara la salida (si es que existe) sino existiera la salida significa que es un ciclo que por ende tendríamos que regresar al inicio o entrada del laberinto.

Algoritmo de Llenado sin salida (Dead end filling): Esta basado mas que todo en el relleno de puntos muertos ó callejones sin salida detectados en el laberinto. El método consiste en identificar todos los callejones sin salida del laberinto y rellenar cada uno de ellos dejando expuesto el primer cruce y así sucesivamente hasta encontrar una salida

concreta. Este tipo de algoritmo también es funcional si se tiene un laberinto con más de una salida ya que expondrá todas las posibles salidas. No lo utilizaremos ya que tratará de recorrer todo el laberinto para ver el mejor camino a utilizar, mientras que nosotros queremos encontrar el mejor camino, ahorrandonos tiempo en lugares que no necesitamos revisar.

Algoritmo de ruta más corta (Shortest path algorithm): Este tipo de algoritmo es útil cuando se tiene una cantidad grande de salidas en un laberinto utilizando la teoría de grafos. El método consiste en hacer una búsqueda de amplitud utilizando una cola para ver que células son crecientes desde la distancia recorrida desde el inicio hasta encontrar la salida. Las células realizan un seguimiento de su distancia desde el inicio y cuando encuentran el lugar final siguen el camino de las células que reportaron menor distancia recorrida desde la entrada.

Algoritmo a implementar (Tremaux):

¿ Por qué ?

Se decidió implementar este algoritmo debido a que es un método bastante eficiente en la resolución de laberintos. Este algoritmo está basado en trazados de líneas en el suelo ó en el recorrido hecho por el robot en nuestro caso, garantizando que funcione para todo tipo de laberintos con espacios definidos. Debido al trazado de líneas a través del recorrido un camino puede o no ser recorrido y ser marcado una o dos veces.

Además de esta decisión, se tomó en cuenta la complejidad del tiempo a invertir con este algoritmo. Después de hacer un par de investigaciones, se pudo concluir que el algoritmo de Tremaux está basado en una técnica de nombre Depth-First Search o bien la búsqueda de profundidad (DFS). Esta se basa en árboles binarios con la misma temática que Tremaux, no volviendo al mismo lugar dos veces, sino solo visitando nodos conectados entre ellos una única vez. Se encontró que la función que domina este algoritmo, su O grande, es O(n + m). Esto nos causó una seguridad, ya que se comporta como una función lineal, no siendo perfecto al ser constante, ni más eficiente al ser logarítmica pero la escogimos por su falta de comportamiento polinómico con potencias diferentes a 1. Esto nos determinó que crecerá el tiempo a medida que crezcan las posibilidades de los nodos.

En el principio se elige aleatoriamente la dirección a donde dirigirse, al llegar a un cruce que no se ha visitado antes deberá elegir de nuevo aleatoriamente un camino que no esté marcado para inspeccionar, si el camino elegido no tiene salida se deberá regresar al punto donde existió otro cruce trazando por segunda vez la línea y anulando el espacio muerto. Si en dado caso no existiera una salida el método regresaría por la línea trazada desde la entrada.

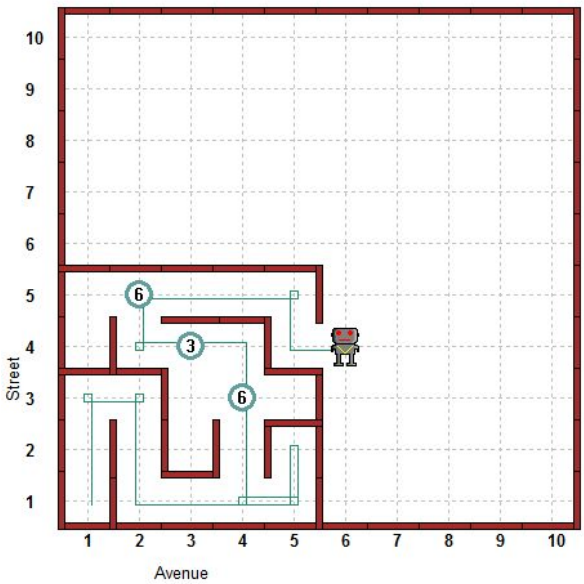
Pruebas y Tiempos de Recreación Virtual

Para esta sección se utilizó el programa RURPLE, en el cual se crearon 2 mapas de laberintos junto con uno que el programa trae dentro de sus bibliotecas. Por cada uno de los

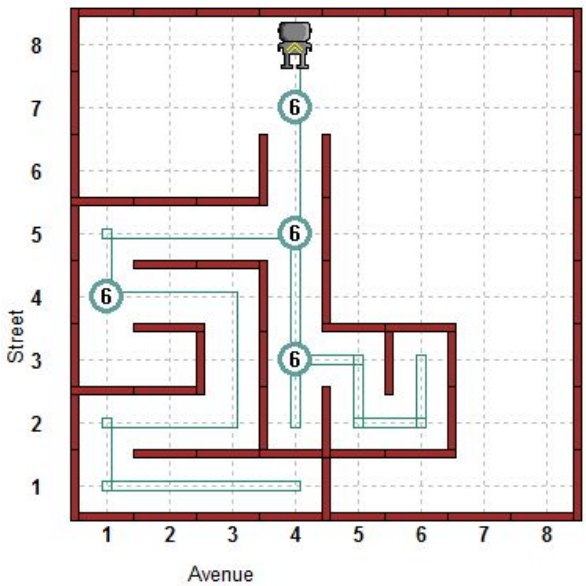
laberintos se corrió dos veces el programa para obtener resultados con una mejor tendencia a la realidad y esto fue lo que se obtuvo.

***IMPORTANTE**

Tener en cuenta, si se quieren recrear en su computador, se necesita que el usuario, en la opcion de la barra de herramientas de RURPLE, de nombre “give beepers to robot”, le de beepers al robot, con el fin que funcione el programa correctamente.

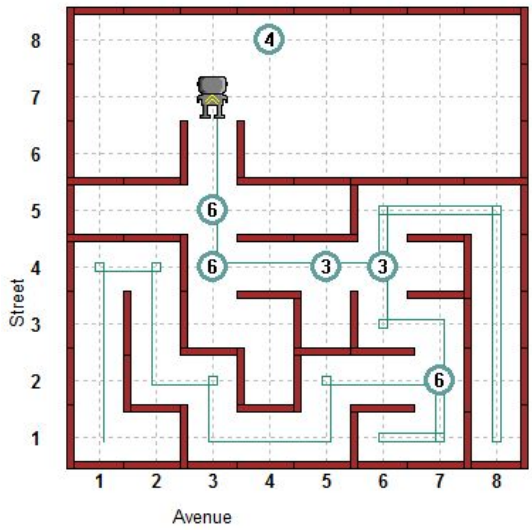


Mundo: maze1.wld, extraido de librerias de RURPLE
Corrida 1: 2:50 segundos Corrida 2: 2:56 segundos



Mundo: MapTest_1.wld, creado por Jonnathan Juarez

Corrida 1: 4:03 segundos Corrida 2: 4:71 segundos



Mundo: MapTest_2.wld, creado por Jonnathan Juarez
Corrida 1: 4:15 segundos Corrida 2: 3:96 segundos

Explicación general del algoritmo solución:

Se le llama algoritmo solución al encargado de solucionar un problema en específico. Nuestro algoritmo se encargará de permitir la movilidad del robot a través del laberinto y así encontrar una solución (salida). Debido a un conjunto de funciones el robot logrará avanzar una distancia en específico analizando que no tenga ningun tipo de obstruccion su camino. Cuando logre avanzar (si bien no tiene algún tipo de obstáculo por delante), hará un análisis frontal y lateral de su entorno. Con esto analiza que hacer siempre colocando como prioridad los movimientos hacia adelante, luego girar a la izquierda y como última opción realizar un giro a la derecha. Si se quedara atorado en algún lugar donde no pueda avanzar hacia el frente o sus laterales, retrocede hasta el último punto donde analizo que existía otro camino anteriormente.

Seguidamente les daremos una breve explicación acerca de los métodos a utilizar para la resolución del laberinto.

Variables:

- **int frente:** Utilizada como bandera para analizar los objetos frente al robot determinando si hay espacio libre para poder caminar. Si el valor resultante fuese 1 es porque si existe espacio suficiente para que el robot avance, si el valor fuese 0 significa que el robot tiene un obstáculo enfrente posterior no podría avanzar.
- **int R:** Utilizada como bandera para analizar la parte lateral derecha del robot determinando si hay espacio libre para caminar. Si el valor es 1 significa que hay espacio suficiente para que el robot avance, si el valor fuese 0 significa que el robot tiene un muro u obstáculo posterior no podría avanzar.

- **int L:** Utilizada como bandera para analizar la parte lateral izquierda del robot determinando si hay espacio libre para caminar. Si el valor es 1 significa que hay espacio suficiente para que el robot avance, si el valor fuese 0 significa que el robot tiene un muro u obstáculo posterior no podría avanzar.
- **int cont:** Utilizada para determinar cuántas series de pasos se realizan antes de haber encontrado un obstáculo. Es utilizada para la reversa ó regreso al checkpoint donde se determino existía otro giro.

Stacks:

- **stackL:** Almacena un 1 en el stack si identifica un espacio libre a la izquierda, de lo contrario guarda un 0. Este stack y el stackR serán utilizados al momento de que se encuentre un bloque sin salida y realizar un tracking al checkpoint más cercano donde se determinó la existencia de otro giro.
- **stackR:** Almacena un 1 en el stack si identifica un espacio libre a la derecha, de lo contrario guarda un 0. Este stack y el stackL serán utilizados al momento de que se encuentre un bloque sin salida y realizar un tracking al checkpoint más cercano donde se determinó la existencia de otro giro.
- **stackRoad:** Almacena un -1 en el stack si en algún punto del camino cruzó a la izquierda y un -2 si cruzo a la derecha. Principalmente almacena la distancia recorrida entre bloques (almacenando un número positivo) esta distancia se actualiza cada vez que llega a un callejón sin salida. En general el stackRoad ayuda a la hora de querer regresar al checkpoint donde se registro otro cruce.

Funciones:

- void left()
    - Propósito: hace girar a la izquierda 90° sobre su eje.
    - Algoritmo narrativo:
        - Mueve el robot a la izquierda.
- void right()
    - Propósito: hace girar a la derecha 90° sobre su eje.
    - Algoritmo narrativo:
        - Mueve el robot a la derecha.
- void checking()
    - Propósito: Analiza el frente y sus laterales para verificar si existen obstáculos en la posición donde se encuentre y asi darle una persepcion del camino al robot. Primero revisa la derecha luego la izquierda y por último el frente para darle prioridad al frente.
    - Algoritmo narrativo:
        - Girar al robot a la izquierda
        - Si existe un obstáculo a la izquierda
            - L=1
        - Si no
            - L=0
        - Girar el robot 180° (a la derecha)
        - Si existe un obstaculo a la derecha
            - D=1
        - Si no

            - D=0
        - Girar el robot a la derecha (hacia el frente)
        - Si hay obstaculo al frente
            - frente=1
        - Si no
            - frente=0
- void step()
    - Proposito: Dar un paso hacia adelante con el robot en una serie de pasos analizando si hay espacio disponible enfrente en cada paso.
    - Algoritmo narrativo:
        - Crear variable int local step e inicializar su valor en 0.
        - Ciclo while
            - Sumar a step el valor 5
            - Mover el robot 5 espacios en ambas ruedas.
            - Sumar a variable global cont el valor 5
            - Analizar distancia hacia el frente.
                - x<7 fin del ciclo
                - x>7 continuar con el ciclo
            - Comparar la variable step con el valor 40
                - x<40 seguir dentro del ciclo
                - x>40 salir del ciclo
- void result(*stackR,*stackL,*stackRoad)
    - Proposito: Analiza la ultima lectura que realizo el sensor, en base a eso realiza diferente acciones. Para esto, hace una sumatoria de las tres variables, frente, L, R, y de acuerdo a este resultado sabe que ocurrio en el recorrido. Si la suma es mayor a 1, sabra que hay una bifurcacion a la que puede regresar si se traba mas adelante. Si la suma es 1, y frente es 0, se ira a la derecha o a izquierda dependiendo si esta libre. Si la suma es 0, llama al backtracking debido a que no tiene mas caminos a donde ir. Tambien almacenara los stacks correspondientes la informacion necesaria.
    - Algoritmo narrativo:
        - Inicializa la variable local int resultado con el valor 0
        - Guarda en resultado el resultado de la suma de R+L+frente
        - Si resultado>1 (hay bifurcacion)
        - Pushea L a stackL
        - Si frente=0
            - Pushea cont a stackRoad
            - Establece cont=0
            - Pushea cont a stackRoad
            - Pushea -2 a stackRoad
            - Pushea 0 a stackR}
        - Si no
            - Pushea cont a stackRoad
            - Establece stackRoad=0
            - Pushea cont a stackRoad
            - Pushea R a stackR

    - Si frente=0 y resultado=1 (solo existe un camino libre, diferente a adelante)
        - si L=1
            - Pushea cont a stackRoad
            - Pushea -1 a stackRoad
            - Establece cont=0
        - Si R=1
            - Pushea cont a stackRoad
            - Pushea -2 a stackRoad
            - Establece cont=0
    - Si resultado=0 (no existen caminos libres, regresa)
        - Pushea cont a stackRoad
        - Establece cont=0
        - Llama a backtracking
- void backtracking(*stackR, *stackL, *stackRoad)
    - Proposito: Revisar stackCamino y realizar las operaciones para regresar al robor hasta el ultimo checkpoint encontrado. En general cuando el robot encuentre un tope retorna sobre su propio camino en reversa. Esto significa que al momento de cruzar deberá hacerlo en dirección contraria de cuando lo hizo de frente.
    - Algoritmo narrativo
        - Crear variable valor(valores extraidos de stackRoad)
        - Ciclo While
            - Si la cima del stack es un numero positivo
                - Guardar la cima en valor
                - Extraer la cima (pop)
                - Indicar al robor que camine(valor*-1)unidades, esto significa que regresara la misma cantidad de pasos que avanzó.
                - Si la cima del stack es igual a -1 (deberá cruzar a la derecha porque antes lo hizo a la izquierda)
                    - Extraer la cima
                    - Llamar a la funcion R()
                - Si la cima stack es igual a -2 (deberá cruzar a la izquierda porque antes lo hizo a la derecha)
                    - Extraer la cima
                    - Llamar a la funcion L()
- void checkStacks(*stackD, *stackL, *stackRoad)
    - Proposito: Analizar los stacks luego del backtracking para regresar el robot en la posicion correspondiente.
    - Algoritmo narrativo:
        - Realizar Comparacion entre stackR=1 y stackL=0
            - True: Hacer Pop a la cima del stackRoad
            - True: Girar el robot hacia la derecha
            - True: Ingresar al stackRoad (push) -2
            - True: Llamar a la funcion step()
        - Realizar Comparacion entre stackR=0 y stackL=1

                - True: Hacer Pop a la cima del stackRoad
                - True: Girar el robot hacia la izquierda
                - True: Ingresar al stackRoad (push) -1
                - True: Llamar a la funcion step()
            - Realizar Comparacion entre stackR=1 y stackL=1
                - True: Hacer Pop a la cima del stackRoad
                - True: Girar el robot hacia la derecha
                - True: Ingresar al stackRoad (push) -2
                - True: Llamar a la funcion step()
- Main:
    - Proposito: En el se llamara a todas las funciones por tiempo indefinido hasta que se logre salir del laberinto. El robot siempre priorizara caminar hacia el frente, luego girar a la derecha y posteriormente izquierda.
    - Algoritmo narrativo:
        - Se instancian los stackD, stackL y stackRoad
        - Ciclo while infinito
            - Llama a la funcion checking()
            - Llama a la funcion result(stackD, stackL, stackRoad)
            - Si frente=1
                - Llama a step()
            - Si no., si R=1
                - Llama a R()
                - Llama a step()
            - Si no, si L=1
                - Llama a L()
                - Llama a step()
        - Regresa a paso 2