Trabajo Práctico Estructructras de Datos y Algoritmos I

Universidad Nacional de Rosario (UNR)

Carrera: Licenciado en Ciencias de la Computación

Autor: Ignacio Ortego

Ver consigna.

Actividad 1

Instrucciones para ejecutar

Utilizando un archivo compatible con el formato estipulado en la consigna, el programa se ejecuta de la siguiente forma

```
make
./programa ./ruta_archivo.txt
```

Estructuras

El programa está formado por dos estructuras que interactúan entre si. En primer lugar, tenemos a la estructura `Mapa`

```
typedef struct {
  char** mat;
  int N;
  int M;
} _Mapa;

typedef _Mapa* Mapa;
```

En la misma se representa el universo en el que se moverá el robot (una matriz de tamaño `N * M`), con los obstáculos ya colocados.

Por otro lado está la estructura `Robot`

```
typedef struct {
  Punto pos;
  Punto obj;
  Pila camino;
  TablaHash visitados;
} _Robot;

typedef _Robot* Robot;
```

Esta tiene todo lo necesario para el funcionamiento del robot, incluyendo las coordenadas de su posición actual, el lugar donde debe llegar y valores adicionales para su algoritmo.

Ambas estructuras dependen de una estructura auxiliar llamada `Info`, la cual se encarga de leer el archivo de entrada e inicializar ambas estructuras antes mencionadas. De esta forma, luego de ser inicializadas, `Robot` y `Mapa` no se conectan directamente. De esta forma, el robot no tiene conocimiento de las dimensiones ni de la ubicación de los obstáculos en el mapa.

Información recolectada

Durante su trayecto, es útil que el robot recuerde cosas para poder ser utilizadas a su favor más adelante. Para el algotirmo implementado, es importante tener un registro del camino recorrido y de los lugares visitados.

Para almacenar el camino recorrido y usarlo para hacer backtracking, el robot usa una `Pila` de direcciones (`LEFT`, `RIGHT` ,`UP` ,`DOWN`). Luego como cada movimiento está anclado a una dirección, la misma es apilada cada vez que el robot se mueve y es desapilada al hacer backtracking, leyendo la última dirección y moviéndose a la opuesta.

Luego para recordar las casillas previamente visitadas, el robot utiliza una `TablaHash` de puntos (coordenadas enteras `x` e `y`). Cada vez que se mueve a una casilla nueva, la almacena en la Tabla. Luego, la Tabla es consultada cuando se busca moverse a casillas no visitadas anteriormente.

Algoritmo

Para llegar al objetivo, el robot supone la mejor situación posible, es decir la ausencia total de obstaculos. Luego para evitar recorrer un camino inviable dos veces, las casillas ocupadas se consideran obstáculos.

Con la información de su propias coordenadas y las de su objetivo, la función `camino_corto` mueve al robot hasta llegar al objetivo o en su defecto, hasta chocarse. En caso de no poder moverse por estar atorado, la función devuelve 0. Si se realizó algún movimiento, la función devuelve 1.

En caso de chocarse o de no poder moverse, el robot llama a la función `buscar_no_visitados`, la cual intenta mover al robot a una casilla adyacente previamente no visitada. En caso de que sea posible, la función mueve al robot y devuelve 1. Luego desde esta nueva posición se llama a `camino_corto` hasta que se llegue al objetivo.

Si el robot no se pudo mover a una casilla adyacente, `buscar_no_visitados` devuelve 0, y esto implica que el robot está atorado o rodeado de obstaculos/casillas ya visitadas. En este caso, se hace uso de la Pila de movimientos para retroceder hasta que la función `camino_corto` devuelva un valor positivo y pueda continuar el trayecto del robot, siendo este el único momento en el que las casillas previamente visitadas pueden ser recorridas por el robot.

Actividad 2

Instrucciones para ejecutar

Utilizando un archivo compatible con el formato estipulado en la consigna, el programa se ejecuta de la siguiente forma

```
./correr.sh ./ruta_archivo.txt
```

Primer idea

Mi primer acercamiento al problema fue hacer algo similar al algoritmo del primer problema, donde el robot usaria su posicion relativa al objetivo para, luego de probablemente chocarse y hacer backtracking, pueda llegar. Luego de implementarlo rapidamente, ya que la logica estaba practicamente hecha, me di cuenta que no estaba usando toda la informacion que tenia disponible, como por ejemplo conocer las dimensiones del mapa desde el primer momento, o ver `D` casillas adelante con el sensor, ya que solo estaba verificando las adyacencias del robot.

Segunda idea

Con la nueva informacion en mente, me planteé implementar un algoritmo basado en costos. Investigué alternativas como A* o Dijkstra, los cuales están centrados en mapas ya conocidos y este no era el caso.

Encontré que lo que mas se adaptaba a mapas dinámicos es D* Lite (de menor complejidad computacional que D*). El mismo era en mi opinión, la opción más óptima, pero encontré su implementación muy complicada y no llegaba a entender del todo su funcionamiento. También leí sobra LPA* que combina ideas de cosas sobre las que ya había investigado.

Con esta informacion, intente construir algo desde cero que pueda entender por mi cuenta.

Estructura

La estructura para este problema consiste de una combinación de las estrucucturas del problema anterior. En este caso el robot y el mapa interactúan constantemente, pero el mismo se considera vacío hasta encontrar obstáculos.

```
typedef struct {
  int N;
  int M;
  int D;
  char** mat;
  int** gScore;
  Punto robot;
  Punto objetivo;
  Cola cola;
  Arreglo camino;
} _Mapa;

typedef _Mapa* Mapa;
```

La matriz `gScore` y la `Cola` son usadas para el algoritmo propuesto, y el `Arreglo` es simplemente un array de `char` redimensionable para poder almacenar el camino total del robot hasta su objetivo.

Algoritmo final

Sabía que tenia que minimizar el uso de sensores para llegar al objetivo, por lo que utilizar toda la informacion posible de cada uso de sensor era crucial.

Siguiendo la idea de A*, agregue una matriz de costos de tamaño `N*M` a mi estructura `Mapa` y escribí la función `generar_g_score`, la cual utilizando una cola calcula la distancia del recorrido desde cada nodo hasta el objetivo.

Con la función para calcular costos hechas, faltaban algunas heuristicas y decisiones que tomar en cuanto al movimiento del robot y el uso de sensores.

Para la función principal `path_planning`, lo primero que hago es lanzar un sensor y generar la primer iteracion de la matriz `g_score`. Luego, mientras el robot no llegue al objetivo, el algoritmo buscará entre sus vecinos adyacentes al de menor costo y se moverá al mismo, hasta completar su recorrido.

Problemas

Al no saber la distancia exacta del sensor, al principio mi robot creaba paredes falsas al final del rayo, lo que evitaba la resolución de mapas muy simples como un mapa vacío. Para resolver esto, agregue a la estructura una variable `D` la cual indica la máxima distancia enviada por el sensor. Esta variable es inicializada en 1 y se

actualiza cuando sea necesario. De esta forma, el robot sólo coloca obstáculos donde está seguro de que los haya.

Otro problema fue el uso del sensor y el calculo de la matriz de costos, ya que ambos se ejecutaban en cada paso y debía encontrar una forma controlada para que estos solo se usen cuando sea necesario. Esto fue resuelto en las optimizaciones finales.

Optimizaciones

La primer optimización que realice fue para el cálculo de los costos. Para esto, detuve el proceso de cálculo de costos cuando el algoritmo encontraba el robot. Esto evita que el algoritmo calcule costos para celdas por las cuales el robot no tiene la necesidad de pasar para llegar al objetivo. Como la matriz es inicializada con `INT_MAX`, las celdas no calculadas serían consideradas como obstáculos y el robot no las toma en cuenta para su camino.

Luego para optimizar el uso de sensores, programé `path_planning` para que en caso de encontrarse con dos vecinos con el mismo costo, priorice aquellos ya conocidos. En caso de que el vecino con el menor costo sea desconocido, lanzo el sensor para verificar si me puedo mover al mismo.

Para no recalcular la matriz de costos en cada uso del sensor, hice que el sensor devolviera `1` en caso de encontrarse con un muro y `0` en el caso contrario. De forma que si el sensor no detecta ningún obstáculo, no hay necesidad de calcular una nueva ruta.

Finalmente, reemplace la cola de prioridad por una cola estándar. La razón es que, aunque la cola de prioridad sea más rapida en encontrar un camino, el mismo generalmente era diagonal, lo cual implica muchos más usos del sensor. Por ello, sacrifiqué calcular algunas celdas de más a cambio de reducir las llamadas al mismo.