Informe de la Práctica 5: Uso de mapas para planning & driving

Grupo: 01
Nombre y apellidos: Daniel Ruskov Vangelov
Nombre y apellidos: Sergio Hurtado Solorzano
Fecha: 20/12/2020

1. Código de los programas

a. **Proyecto NetBeans exportado en .zip:**
https://drive.google.com/file/d/1UtNYnFS7o4TRat6BvzXqUPZq2Br6uA-l/view?usp=sharing

b. **Enlace al código base del algoritmo BFS adaptado a nuestro problema:**
https://www.geeksforgeeks.org/print-paths-given-source-destination-using-bfs/

2. Descripción detallada de la solución diseñada

a. Simulación y control remoto simultáneos

A continuación se muestra la función dedicada a manejar el robot físicamente mediante lógica de estados (maquina de estados) parecida a la de la práctica 4.

Establece conexión serie con el iRobot para el guiado mediante marcas por el laberinto o rejilla. El parámetro estado_actual indica la acción a hacer (GIRAR o AVANZAR). Los parámetros dirobj y diract son necesarios solo si la acción es GIRAR, dando a conocer la orientación actual del robot y la orientación objetivo (±90º).

No se muestra el código de ControlRobot.cpp/h por su extenso código, pero cabe mencionar que el avance y/o giro se hacen con la ayuda de los cuatro sensores cliff.

Los dos sensores laterales sirven para detectar que el robot ha llegado al siguiente nodo cuando se avanza por el mapa, después de lo cual se desplaza una distancia mínima hasta quedar centrado en él, mientras los sensores frontales sirven para enderezar el iRobot en su recorrido.

En el acto de giro se usan los cuatro sensores hasta detectar que la línea perteneciente a la dirección objetivo queda entre los dos sensores frontales.

Finalmente, finaliza la ejecución llegando al estado SALIR, lo cual genera que la condición del while sea falsa y termina la función.

```
//=============================================================================================
void control_robot(char estado_actual, char dirobj, char diract) {
   ControlRobot robot;
   robot.inicializacion(estado_actual, dirobj, diract);
   while (!robot.condicionSalida()) {
       robot.leerSensores();
       robot.logicaEstados();
       robot.moverActuadores();
       robot.imprimirInfo();
   }
   robot.finalizacion();
}
//=============================================================================================
```

En el siguiente trozo de código se muestra como se ha logrado que el robot se oriente por el laberinto o rejilla según la tecla pulsada (flechas arriba, abajo izda o dcha). Se obtiene la orientación actual y según la dirección objetivo (flecha pulsada) se llama a la función anterior control_robot(...) para que el robot se reoriente. Así mismo se simula el movimiento y se vuelve a imprimir el mapa en la Salida Estándar (SE) del programa, con lo cual en todo momento habrá una correspondencia. Esto es útil si necesitamos controlar el robot sin poder verlo de forma directa.

```
//=============================================================================================
maze->getPosRobot(&x, &y, &orientacion);
switch (tecla) {
   case 65: // Flecha arriba
       if (orientacion == NORTE) {
           cout << "El robot ya se encuentra orientado hacia el Norte\r" << endl;
       } else {
           cout << "Robot orientado al Norte\r" << endl;
           if (orientacion == SUR) {
               control_robot(POSICIONAR, OESTE, orientacion);
               control_robot(POSICIONAR, NORTE, OESTE);
           } else {
               control_robot(POSICIONAR, NORTE, orientacion);
           }
           maze->orientarRobotNorte();
           mapa_modificado = true;
       }
       break;
   // cases 66, 67, and 68 with the same idea (...)
}
//=============================================================================================
```

A continuación, es la misma idea que antes, pero una vez pulsada la barra espaciadora. Esto hace que el robot avance en la dirección en la que está orientado siempre que haya camino posible. Se sigue la simultaneidad con la simulación y se imprime el mapa por la SE cada vez que haya un cambio en la posición del robot entre los diferentes nodos del laberinto (se omiten capturas de pantalla para no alargar el informe más de lo necesario con cosas que se saben y centrarse en lo aportado al proyecto).

```
//=========================================================================================
maze->getPosRobot(&x, &y, &orientacion);
switch (orientacion) {
   case NORTE:
       if (maze->esNorteLibre()) {
           control_robot(AVANZAR, NORTE, NORTE);
           maze->avanzaNorte();
           mapa_modificado = true;
       } else {
           cout << "No existe un camino al Norte \r" << endl;
       }
       break;
   // cases SUR, ESTE and OESTE with the same idea (...)
}
//=========================================================================================
```

Cabe mencionar, que al haber usado los sensores de forma adecuada, este programa es completamente capaz de recorrer cualquier mapa ortogonal con las mismas características sin importar cómo de largas son las aristas o el número de nodos que tiene.

b. Path planning y driving

i. Path planning

El código de la función a continuación es la adaptación del algoritmo BFS seleccionado en el enlace del apartado uno de este informe. Pertenece al fichero Laberinto.cpp.

Dadas las coordenadas, empezando desde el origen recorre la matriz de adyacencia en busca de camino al destino con la ayuda de una lista ligada y un vector de punteros a nodos - los propios de la matriz de adyacencia. De esta forma se ahorra espacio al no hacer copias de los nodos. Este algoritmo evita los nodos ya visitados para no entrar en ciclos.

El bucle de búsqueda finaliza al encontrar un camino o cuando se recorre todo el grafo y no se haya encontrado camino posible.

Después se crea la lista doblemente ligada de struct camino, que recorre el vector con la ruta solución del algoritmo, haciendo que los punteros apuntan a los nodos de la matriz de adyacencia, y de nuevo ahorrar espacio al no crear una copia de los mismos. Finalmente se devuelve el puntero al primer elemento de la lista de camino solucion.

```
//=========================================================================================
/**
*   Método que obtiene el camino entre un nodo de entrada y un nodo de salida.
*       int x_orig: posición origen del robot en la Matriz del laberinto
```

```
*       int y_orig: posición origen del robot en la Matriz del laberinto
*       int x_dest: posición destino del robot en la Matriz del laberinto
*       int y_dest: posición destino del robot en la Matriz del laberinto
*       return camino*: Apuntador a una estructura tipo camino con la solucion
*/
camino*  Laberinto::encontrarCamino(int x_orig, int y_orig, int x_dest, int y_dest) {
   struct camino* solucion  = (struct camino *) malloc (sizeof(camino));
   struct camino* aux;
   int i, j, v = dim_x*dim_y, camino_encontrado = 0, size;
   struct nodo *last;
   struct nodo *origen = &matriz[x_orig][y_orig];
   struct nodo *destino = &matriz[x_dest][y_dest];
   queue<vector<struct nodo *>> q;
   vector<struct nodo *> path;
   path.push_back(origen);
   q.push(path);
   while (!q.empty() && camino_encontrado == 0) {
       path = q.front();
       q.pop();
       last = path[path.size() - 1];
       if (last->x == destino->x && last->y == destino->y) {
           printpath(path);
           camino_encontrado = 1;
       } else {
           if (last->Norte != NULL && isNotVisited(last->Norte, path)) {
               vector<struct nodo *> newpath(path);
               newpath.push_back(last->Norte);
               q.push(newpath);
           }
           if (last->Sur != NULL && isNotVisited(last->Sur, path)) {
               vector<struct nodo *> newpath(path);
               newpath.push_back(last->Sur);
               q.push(newpath);
           }
           if (last->Este != NULL && isNotVisited(last->Este, path)) {
               vector<struct nodo *> newpath(path);
               newpath.push_back(last->Este);
               q.push(newpath);
           }
           if (last->Oeste != NULL && isNotVisited(last->Oeste, path)) {
               vector< struct nodo *> newpath(path);
               newpath.push_back(last->Oeste);
               q.push(newpath);
           }
       }
   }
   size = path.size();
       cout << size << "nodos a recorrer \r" << endl;
       solucion->nodo_actual = path[0];
       solucion->anterior = NULL;
       solucion->siguiente = NULL;
       aux = solucion;
   for (i = 1; i < size; i++) {
```

```
            aux->siguiente = new camino();
            aux->siguiente->nodo_actual = path[i];
            aux->siguiente->siguiente = NULL;
            aux->siguiente->anterior = aux;
            aux = aux->siguiente;
    }
    imprimirCamino(solucion);
    return solucion;
}
//============================================================================
```

ii. Driving

En este apartado se expone el código que llama a la función anterior dadas unas coordenadas de origen y destino y según el camino solución devuelto ejecuta un algoritmo para recorrerlo en la simulación y en el mapa físico simultáneamente.

Se trata de un algoritmo (bucle while), que mientras existan nodos por recorrer, se orienta según donde esté situado el siguiente nodo respecto al actual y avanza con la ayuda de la función control_robot(...) anteriormente explicada.

```
//============================================================================
void prueba3(const char *path) {
    char orientacion;
    int x, y;
    Laberinto *maze;
    maze = new Laberinto(path);
    maze->imprimirLaberinto();
    struct camino * recorrido = maze->encontrarCamino(ox, oy, dx, dy);
    while (recorrido->siguiente != NULL) {
        maze->getPosRobot(&x, &y, &orientacion);
        if (recorrido->nodo_actual->x == recorrido->siguiente->nodo_actual->x) {
            if (recorrido->nodo_actual->y < recorrido->siguiente->nodo_actual->y) {
                if (orientacion != SUR) {
                    if (orientacion == NORTE) {
                        control_robot(POSICIONAR, ESTE, orientacion);
                        control_robot(POSICIONAR, SUR, ESTE);
                    } else {
                        control_robot(POSICIONAR, SUR, orientacion);
                    }
                    maze->orientarRobotSur();
                    maze->avanzaSur();
                }
            } else {
                if (orientacion != NORTE) {
                    if (orientacion == SUR) {
                        control_robot(POSICIONAR, OESTE, orientacion);
                        control_robot(POSICIONAR, NORTE, OESTE);
                    } else {
                        control_robot(POSICIONAR, NORTE, orientacion);
                    }
                    maze->orientarRobotNorte();
                    maze->avanzaNorte();
```

```
                }
            }
        } else {
            if (recorrido->nodo_actual->x < recorrido->siguiente->nodo_actual->x) {
                if (orientacion != ESTE) {
                    if (orientacion == OESTE) {
                        control_robot(POSICIONAR, SUR, orientacion);
                        control_robot(POSICIONAR, ESTE, SUR);
                    } else {
                        control_robot(POSICIONAR, ESTE, orientacion);
                    }
                    maze->orientarRobotEste();
                    maze->avanzaEste();
                }
            } else {
                if (orientacion != OESTE) {
                    if (orientacion == ESTE) {
                        control_robot(POSICIONAR, NORTE, orientacion);
                        control_robot(POSICIONAR, OESTE, NORTE);
                    } else {
                        control_robot(POSICIONAR, OESTE, orientacion);
                    }
                    maze->orientarRobotOeste();
                    maze->avanzaOeste();
                }
            }
        }
        control_robot(AVANZAR, NULL, NULL);
        maze->imprimirLaberinto();
        recorrido = recorrido->siguiente;
    }
}
//============================================================================================
```

c. Driving con evitación de obstáculos

En esta práctica por falta de tiempo no se ha implementado la funcionalidad de evitar obstáculos. Sin embargo, explicaremos los pasos a seguir para lograrlo.

En la función prueba3(...) mostrada en el apartado anterior se deberá controlar si el robot llega al siguiente nodo o no. En caso de no llegar, habría que modificar la estructura ***Laberinto::matriz* eliminando la arista que tiene el obstáculo y llamar de nuevo a la función de encontrarCamino(...) con las coordenadas actuales del robot como origen y las mismas coordenadas de destino.

Si existe camino se deberá seguir con la ejecución con el nuevo recorrido, y en caso de que no exista camino, se podrá dar una señal mediante los actuadores (ej leds o altavoces) y finalizar la ejecución.

En lo referente a la detección de obstáculos, habría que hacer modificaciones/ aportaciones en ControlRobot.cpp/h de forma que, usando los bumpers se detecte si hay obstáculos o no.

3. Preguntas

a. ¿qué algoritmo de búsqueda habéis empleado?

Finalmente hemos utilizado el algoritmo BFS (Breadth First Search, búsqueda en anchura) adaptando el código referenciado en el primer apartado de este informe a nuestro problema, utilizando la estructura *nodo* ***Laberinto::matriz* como lista de adyacencia, de forma que al encontrar el primer camino que conecte el punto de origen con el punto de destino, finalice.

b. ¿Por qué?

Nos ha parecido un algoritmo adecuado para nuestro problema, sin preocupación por su complejidad espacial y temporal, ya que las dimensiones de nuestro problema son mínimas. Además, siempre que existe solución, se encuentra en la menor profundidad posible y es fácil de adaptar a nuestro código y la estructura **Laberinto::matriz*.

Por último, tiene un plus, y es que se podría usar para obtener todos los caminos posibles de una sola ejecución inicial, y si se encuentran obstáculos no habría que volver a ejecutar el algoritmo de nuevo si se usa de forma adecuada. Esto es bueno si las dimensiones del problema son más grandes y el cálculo de los caminos es más costoso.