Facultad de Ciencias Exactas Ingeniería y Agrimensura

Trabajo Practico Final

Alumno:
Lucas Nicolas Wasserstrom

4 de septiembre de 2024

1. Ejercicio 1

1.1. Funcionamiento y diseño

El archivo principal de mi programa es **TPFinal1.c**, luego también tengo implementadas las siguientes librerías:

- `TPFinal1`: Archivo principal, en el que se analiza el archivo y se llaman las funciones para el funcionamiento general del programa `MazeInfo`.
- `LibreriaRobot`: Tiene las funciones y estructuras del robot, para inicializarlo, manejar su información y moverse.
- `LibreriaLaberinto`: Están las estructuras del laberinto y para representar posiciones, así como funciones para crear el laberinto y verificar si posiciones son validas.
- `glist`: Implementación de glists.
- `pilas`: Implementación de pilas.
- `LibreriaHash`: Implementación de tablas hash.

Para usar el ejercicio 1 desde su carpeta, primero usar **make** para compilar el programa y después **./robot (Nombre de archivo)**

1.2. Estructuras

Utilizo una estructura **Pos2D** para representar posiciones en 2 dimensiones, la cual consta de 2 valores tipo int para coordenada x e y. Mi programa utiliza estas coordenadas de forma invertida, por como cree el array bidimensional para representar el laberinto, ya que al leer el archivo iba primero viendo las filas, así que el primer campo representa el eje y, y el segundo el eje x.

La estructura de laberinto es solo un array bidimensional con el alto y ancho del laberinto.

Para el **robot**, este tiene una referencia a su posición y su destino, como **Pos2D**. Luego utilizo una tabla hash para representar su información del entorno, de forma que guardo las casillas que ya visitó. Mi tabla utiliza listas de rebalse debido a su fácil implementación, no necesito mucho mas de la tabla pues no voy a eliminar elementos, con una función hash definida por $31 * x + y \ mod \ Tabla.Longitud$. Si se tratan de agregar elementos de forma que el factor de balance de la tabla es mayor a 3, entonces redimensiono (Multiplicando el tamaño por 4, para así pasar a tener un factor de 0.755) para mantener la eficiencia.

1.3. Algoritmos

El algoritmo del primer ejercicio implementa un estilo de `random inteligente` en el que mi robot se trata de mover siempre hacia la salida, almacenando el movimiento contrario en una pila, y si tiene que elegir entre casillas igual de importantes, elige una aleatoria. Este marca casillas en las que ya estuvo, guardándolas en la tabla hash, y las

trata como obstáculos. El robot se movera de esta forma hasta que se encuentra en un camino sin salida, y ahí es donde hace uso de su pila para hacer backtracking.

Esto garantiza que el robot llegue a la salida, pues cada vez que encuentra una `bifurcación` el robot va a seguir por un camino hasta la salida o callejón sin salida, si se encuentra el segundo, va a hacer uso de la pila hasta poder moverse de nuevo, lo que lo lleva a la ultima bifurcación visitada, y así el robot va a visitar todas las bifurcaciones en el peor caso, por lo que si el laberinto tiene salida significa que la va encontrar.

Para definir que casilla es mas preferible sobre otras, se podría decir que hago uso de Distancia Manhattan ($|x1 - x2| + |y1 + y2|$) ya que trato de acercar las coordenadas x e y de mi robot a las de la salida preferentemente.

El robot funciona mejor cuando no hay grandes caminos que no lleven a nada, o espacios grandes en los que no este la salida, ya que si esto pasa el robot se ve obligado a visitar la totalidad del espacio hasta poder hacer backtracking.

1.4. Proceso de desarrollo

Mi idea desde el comienzo era que el robot trate de ir siempre hacia la salida, cuando encuentra un pasadizo lo siga hasta quedarse sin salida y entonces backtrackear hasta poder moverse otra vez. Es lo que termine haciendo, el robot trata como obstáculo los lugares a los que se movió y los obstáculos en si, y vuelve a explorar ramas inexploradas cuando esta atrapado.

Cosas con las que tuve problemas fue en como representar la información que el robot va obteniendo. Al comienzo usaba una lista bidimensional con los tamaños del laberinto pero al final vi que eso no era posible, pues el robot desconocía las dimensiones del laberinto. Pensé entonces en formar una caja entre el inicio y la salida y que esas sean mis dimensiones de mi array, entonces cuando se quiera mover afuera de esta caja, (Sin contar el borde superior y el borde izquierdo, pues conozco la posición en 0,0 al saber la del robot y entonces estos bordes) la expando. Las consideraciones que tuve sobre esto fueron:

- `Agregar filas`: Al ser la primera componente de mi array la columna, agregar filas implicaría hacer realloc en todas las columnas, lo que no era eficiente y era engorroso.
- `Desperdicio del fácil acceso`: A pesar de que acceder a una casilla de mi información era O(1) con este método, me di cuenta que mi robot solo accede a la información de casillas adyacentes, así que esta ventaja no me servía.

Luego de analizarlo pensé en realizar una especie de lista enlazada pero con punteros a las cuatro direcciones. Inicializar y actualizar esto me termino pareciendo mas complicado que lo que use al final y no le di muchas vueltas.

Al final termine usando tablas hash, las cuales me trajeron las siguientes ventajas:

- `Dinámico`: Puedo almacenar las casillas en la tabla y agregar espacio cuando mi factor de carga sobrepasa un valor dado, lo cual me pareció mas eficiente que los métodos para redimensionar anteriores, además de que estos eran dependientes de a donde se movía el robot, mientras que este solo depende de la cantidad de casillas almacenadas.

- `Saber donde no me puedo mover por pertenencia`: Mi robot solo se mueve hacia casillas que no estén en la tabla hash, y voy metiendo estas en la tabla al descubrirlas, ya sea porque me moví a ellas o porque me choque.

En cuanto al algoritmo, como la idea de este era simple, no hubo problemas para implementarlo. Cuando estaba probando cual era la mejor forma de decidir entre las casillas que tenía para elegir, termine usando un método aleatorio pues el robot no tiene suficiente información como para saber realmente que casilla es mas preferible, este es prácticamente ciego.

2. Ejercicio 2

2.1. Funcionamiento y diseño

El archivo principal de mi programa es **TPFinal2.c**, así como también esta el archivo proveído por la cátedra **sensor.c**, además las librerías que implemente para la resolución son:

- `LibreriaRobot`: En esta están implementadas todas las funciones para llevar a cabo el algoritmo así como las estructuras para resolver el problema. Luego estas funciones se usan en TPFinal2.c hasta que el robot encuentre la salida.

- `heaps`: Implementación de bheaps, que uso como colas de prioridad. Una cosa especial de estos es que sus elementos son Celdas, en las que almaceno por una parte el dato y por otra la llave. Para operaciones como actualizar y eliminar busco por dato, mientras que el orden esta definido por la llave.

Para correr el ejercicio 2, desde su carpeta usar **make**. Una vez hecho esto ya se puede usar el programa con **./correr.sh (Nombre de archivo)**

2.2. Estructuras

Para la representación de mis datos en el heap, uso una estructura de Celda, la cual posee los campos de llave y dato.

Como en el ejercicio 1, también hago uso de la estructura **Pos2D**, de la misma forma que antes poniendo primero eje y y después x, para mantener la convención anterior. Tengo la estructura casillaInfo, que tienen todas mis casillas, para acceder rápidamente al valor g, rhs y estado de mis casillas, entro en mas detalle en esto en la parte de algoritmos.

Para este ejercicio la estructura del robot es mas compleja, como sé las dimensiones del mapa puedo representarlo con un array bidimensional de información de casillas. Tengo una cola de prioridad en la que almaceno casillas necesarias para recalcular el camino eficientemente, así como también para expandir en orden en ese momento. Luego tengo los valores de: El rango del sensor, que es un estimado que tiene el robot del rango, así como también el k_m, que es un modificador de llave, usado para que al recalcular el camino desde otro lado las llaves de datos en mi cola se mantengan coherentes.

2.3. Algoritmos

El algoritmo que use para mi robot en este ejercicio el **D* Lite**, decidí este pues el robot tiene que resolver un mapa con obstáculos sin tener información de antemano, por lo que con D* Lite recalcular el camino al encontrarse un obstáculo es mucho mas rápido que para algoritmos como A* o Dijkstra. Siempre que la alteración que recibió el camino no requiera recalcularlo totalmente, D* Lite sera mas eficiente.

Voy a explicar como funciona el algoritmo pero aun así refiero al final el paper de donde saque la información.

Algunas variables importantes que se ven en mi programa son:

- `g`: Variable individual de una casilla. Representa un estimado que se tiene en base a la información actual de la distancia de la casilla a la salida.
- `rhs`: Variable individual de una casilla. Es una versión mas volátil de g basada en el g de casillas adyacentes. En esta se mantiene la información de búsqueda mas actualizada, es como un adelanto del valor que va a tener g.
- `Estado`: El estado de una casilla puede ser `DESCUBIERTO, SIN DESCUBRIR u OBSTACULO`
- `Costo`: Esto es mas una función que devuelve el costo de moverse entre 2 casillas adyacentes. Si es posible moverse el coste sera de 1, en caso de que sea un obstáculo asignamos el coste como ∞.
- `km`: Modificador de llave, variable del robot. Se usa para mantener una cota inferior en las llaves, cada vez que el robot se mueve y tiene que actualizar el camino esta variable se actualiza y se añade a la llave de las casillas en la cola. Así, casillas calculadas desde otra posición se mantienen consistentes, pues la llave esta basada en la heurística.
- `Llave de los elementos en la cola`: Esta consta de 2 campos y se obtiene de la siguiente formula, sea s la casilla:

  $Llave = [min(g(s), rhs(s)) + heuristica(s, destino) + km, min(g(s), rhs(s))]$. El segundo campo esta hecho para desempatar al tener mismo primer campo.
- `Heuristica`: Utilizo Distancia Manhattan como en el ejercicio anterior para calcular la heurística, pues no hay movimientos diagonales.

Así como también algunos términos asociados a las casillas:

- `Localmente consistente`: Si la casilla tiene $g = rhs$. Si una casilla cumple esta propiedad luego de calcular el camino, esto implica que esta actualizada con la información actual.
- `Alta consistencia local`: Si la casilla tiene $g > rhs$. Esto significa la casilla esta desactualizada, y se le puede asignar un nuevo valor asociado a un camino mas eficiente.
- `Baja consistencia local`: Si la casilla tiene $g < rhs$. Esto implica que la casilla esta desactualizada, y el camino por ella quedo invalidado y hay que calcular de nuevo partiendo de un $g = \infty$.

Ahora vamos sobre el funcionamiento general del algoritmo, el cual sigue el siguiente orden:

1. Inicializar todos los nodos con un valor de g y rhs infinitos (Todavía no se expandieron) excepto la salida que parte con un valor de $rhs = 0$ y se almacena en la cola.

2. Expandir casillas en orden de menor llave en la cola hasta que el inicio sea consistente y se haya expandido (Si esto ya se da, no se hace nada).

3. Tratar de moverse a la mejor casilla disponible, si esta descubierta me muevo y repito este paso, sino hago uso del sensor (Y actualiza, si es posible, el rango de sensor conocido por el robot) y vamos al siguiente paso.

4. Si se encontraron obstáculos, se actualiza el valor de km, y se actualizan los valores de las casillas en las que aparecieron estos obstáculos.

5. Se vuelve al paso 2

En el primer paso básicamente dejamos un entorno como para que al llegar por primera vez al segundo paso, se realiza una búsqueda como lo haría A*, solo que desde la salida al dejar solo está en la cola, vamos expandiendo desde ahí hasta encontrar la posición del robot y formar un camino de casillas consistentes de ahí a la salida.

En el segundo paso es donde se lleva a cabo mayor parte del algoritmo, que es donde calculamos el nuevo camino. Para esto vamos sacando la casilla al tope de la cola y trabajando con ella, cabe recalcar que las casillas en la cola son de baja o alta consistencia, pero no consistentes porque entonces eso significa que ya sabemos el valor que tienen dada nuestra información actual, o aun no llegamos a ellas y calcular en base a estas nos va a dar valores erróneos.

1. Primero vemos su llave, si esta no tiene el k_m actualizado, se actualiza para mantener un orden consistente con la posición en la cola, y no actualizar casillas en un orden incorrecto, lo que puede entorpecer el algoritmo.

2. Si la casilla sacada es de alta consistencia eso quiere decir que $g > rhs$ y entonces se encontró un mejor camino así que igualamos g a rhs, con lo que la hacemos consistente, y expandimos de forma que actualizamos los valores de casillas adyacentes.

3. Si es de baja consistencia, $g < rhs$ y entonces el camino ya no sirve por lo que se asigna ∞ a g y se expande, pero no solo a casillas adyacentes, sino que también se actualiza el valor de la casilla en si, pues ahora puede que sea de alta consistencia.

Las casillas analizadas por la función que calcula el camino van decreciendo con el tiempo, debido a que esta expande cada casilla máximo 2 veces, máximo 1 vez si la casilla es de alta consistencia local, y máximo 2 si es de baja consistencia local.

Ahora, muchas veces hago referencia a actualizar el valor de una casilla, con esto a lo que me refiero es cambiar el valor de rhs de forma que $rhs = min(g(s) + costo(s))\forall s \in Adyacencias$ y así ir formando el camino entre casillas adyacentes. Si la casilla se vuelve

consistente y estaba en la cola, la sacamos (Pues ya es candidata a casilla de camino) y en caso contrario le recalculamos la llave y la insertamos en la cola.

En el paso 3 y 4, cuando hacemos uso del sensor, al actualizar el k_m podemos mantener una consistencia sobre el valor de las llaves en la cola, al hacer que estas no se vean afectadas por de donde se calcularon.

Cuando uso el sensor, primero corroboro si recibí una lectura tal que si le resto 1, esta es mayor al rango de sensor estimado por el robot, si eso pasa lo actualizo. Una vez hecho esto, voy en cada dirección actualizando a las casillas y poniéndolas como DESCUBIERTAS, y si encuentro un obstáculo, lo marco como OBSTACULO (Para así cuando calcule el costo de moverse de/hacia ella me de ∞), y actualizo el valor de la casilla, pues ahora variara su consistencia.

Luego del paso 2 (Si el mapa tiene salida, sino el $g(posicionActual) = \infty$ y es imposible) el robot tiene un camino de casillas consistentes para seguir hasta llegar a la salida, el cual sigo en el paso 3.

A pesar de que este algoritmo realiza operaciones como eliminar y actualizar en una cola de prioridad, las cuales no son eficientes para este tipo de estructuras, tener la ventaja de poder recalcular pequeños cambios en el entorno ayuda mucho, y mas mientras mayores dimensiones tenga el mapa a resolver.

Doy el pseudocódigo del programa planteado al final.

2.4. Proceso de desarrollo

Al principio pensé en implementar un A* o el algoritmo de Dijkstra, pero no me parecía lo mejor tener que expandirme desde la posición del robot hasta encontrar la salida cada vez que encuentro un obstáculo, debido a la simplicidad del sensor. La máxima cantidad de obstáculos que puede encontrar mi sensor es de 4, y en mapas grandes esta cantidad me implica que voy a tener que recalcular el camino muchas veces, para cambios minúsculos en los que, por ejemplo, se pone un único obstáculo entre mi camino y la salida.

Para solucionar esto me puse a buscar y encontre el algoritmo de D* Lite, el cual al tener que recalcular el camino para cambios pequeños, en vez de recalcular todo el camino, solo tenia que hacer pequeños cambios en los alrededores del obstáculo si es posible reutilizar mi camino (Con esto me refiero a que no se forme una extensa pared entre mi camino y la salida, pues en ese caso recalcular es mas costos que en otros algoritmos).

Esto, desde mi punto de vista, quedaba perfecto para un robot que puede que solo encuentre un único obstáculo entre el camino y la salida en múltiples iteraciones. Incluso si tuviese que recalcular todo el camino, lo cual es malo con este algoritmo, sigue siendo mejor ya que para encontrarse en esta situación va a tener que ir encontrando los obstáculos de a poco y recalcular, y esos cálculos los hace de forma eficiente.

Empecé a resolver el trabajo practico teniendo ya en mente que iba a usar D* Lite, así que los problemas que fui encontrando tenían que ver con como funciona el algoritmo. Luego tuve otras complicaciones con mi implementación del heap, principalmente al eliminar elementos de este, pues como puedo eliminar en medio del heap, puede que al reemplazar por el ultimo elemento, este venga de otra rama, y me haga falta flotarlo en vez de hundirlo, cosa que no tuve prevista.

Por ultimo, tuve que buscar una solución a que el robot no sepa la longitud del sensor, el cual solucione haciendo que pueda estimar cuanto vale basándose en lo que iba devolviendo.

D* Lite Ejercicio 2

```
1: function CALCULAR_LLAVE(c)                                   ▷ c Es una casilla
2:     return [min(g(c), rhs(c)) + heuristica(c, c_actual) + k_m];   ▷ c_actual pos del robot
3: function INICIALIZAR()
4:     Cola = ∅;
5:     k_m = 0;
6:     ∀c ∈ C/rhs(c) = g(c) = ∞;
7:     rhs(c_destino) = 0;                                      ▷ c_destino es la pos de destino
8:     Cola.Insertar(c_destino, Calcular_llave(c_destino));
9: function ACTUALIZAR_CASILLA_VALORES(c)
10:    if c ≠ c_destino then rhs(c) = min_{c'∈Adyacencias(c)}(costo(c, c') + g(c'));
11:    if c ∈ Cola then Cola.Eliminar(c);
12:    if g(c) ≠ rhs(c) then Cola.Insertar(c, Calcular_llave(c));
13: function OBTENER_CAMINO()
14:    while Cola.MaxLlave() < Calcula_llave(c_actual) OR rhs(c_actual) ≠ g(c_actual) do
15:        k_anterior = Cola.MaxLlave();
16:        k_nueva = Calcular_llave(c);
17:        c = Cola.MaxDato();
18:        if k_anterior < k_nueva then                          ▷ Si llave desactualizada
19:            Cola.Actualizar(c, k_nueva);
20:        else if g(c) > rhs(c); then                           ▷ Alta consistencia local
21:            g(c) = rhs(c)
22:            ∀c' ∈ Adyacencias(c) : Actualizar_casilla_valores(c');
23:        else                                                  ▷ Baja consistencia local
24:            g(c) = ∞;
25:            ∀c' ∈ Adyacencias(c) ∪ {c} : Actualizar_casilla_valores(c');
26: function MAIN()
27:    c_ultima = c_actual;
28:    Inicializar();
29:    Obtener_camino();
30:    while c_actual ≠ c_destino do
31:        c_proxima = min_{c'∈Adyacencias(c)}(costo(c_actual, c'));
32:        if Estado(c_proxima) ≠ DESCUBIERTA then
33:            Usar sensor;
34:            if Detectados obstaculos then
35:                k_m = k_m + heuristica(c_ultima, c_actual);
36:                c_ultima = c_actual;                          ▷ Cambiamos referencia a ultima
37:                for all c_i ∈ Cambios do
38:                    Estado(c_i) = OBSTACULO;                  ▷ Actualiza el costo
39:                    Actualizar_casilla_valores(c_i);
40:                Obtener_camino();                             ▷ Recalculamos
41:        else
42:            c_actual = c_proxima                              ▷ Se mueve
```

Referencias

[1] Sven Koenig, Maxim Likhachev, *D* Lite*, `https://idm-lab.org/bib/abstracts/papers/aaai02b.pdf`

[2] MIT OpenCourseWare, *Advanced 1. Incremental Path Planning*, `https://www.youtube.com/watch?v=_4u9W1xOuts`