Estructuras de Datos y Algoritmos I - Trabajo Práctico Final

Bertoni Juan Ignacio

4 de septiembre de 2024

Estructura del proyecto

Tiene dos carpetas, **parte1** y **parte2**, respectivamente según el enunciado.

- Ambos tienen la carpeta **estr**, donde se guardan los archivos que contienen las implementaciones de las estructuras de datos utilizadas en esa parte.
- Ambos tienen la carpeta **ejemplos**, con ejemplos de mapas en el formato que el programa acepta. Una aclaración importante es que no se leeran mapas con un N o M superior al valor `LONGITUD_MAX_LINEA`, pudiendo modificarse de ser necesario

Parte 1

Idea

El algoritmo que se me ocurrió consiste en:

En base a la posición actual del robot y a la meta, distinguir cuales son las direcciones más "preferidas" para que el robot se mueva. E.g. si el robot se halla en $(0,1)$ y la meta está en $(6,5)$, las direcciones preferibles serán *abajo* y *derecha*, o si el robot está en (5, 1) y la meta está en (9, 1), la dirección preferida será solamente *abajo*.

Después, escoger de manera aleatoria alguna de las direcciones preferidas: es decir, verificar si un movimiento en esa dirección es válido y además que la celda destino no haya sido visitada con anterioridad. (1)

Si ninguna de las direcciones verifica (1), probar con la otra dirección; la que no es ni favorita, pero tampoco es la menos favorable (la menos favorable es la opuesta a la dirección de donde vinimos, ya que se retrocede sobre sus pasos). Si dicha dirección también falla en cumplir (1), entonces el robot volverá sobre sus pasos (backtracking).

El backtracking consiste en:

- Retroceder 1 paso.
- Chequear por casillas no visitadas, preferentemente en dirección a la meta (vuelve a escoger según el mismo criterio de antes)
- Si se encuentra casilla, moverse hacia ella. Caso contrario, seguirá retrocediendo.

De esta manera, el robot sabe reconocer cuando "se queda encerrado" en un sector del mapa, y busca la salida a medida que va retrocediendo, hasta que eventualmente sale del lugar donde se "encerró".

Decidí este enfoque ya que me pareció la opción más rápida para mapas sin tanta complejidad. En efecto, el robot simplemente va desplazandose por el mapa teniendo en cuenta en qué dirección moverse la próxima vez, teniendo en cuenta la información recabada hasta ese momento. Estas consultas que realiza el robot en cada vuelta son rápidas gracias al tiempo promedio de búsqueda `O(1)` de una tabla hash.

Debido a la elección "random" que debe hacer a veces el robot, los resultados de los recorridos pueden diferir, donde en el mejor caso encuentra la salida sin chocarse demasiado, y en el peor caso explora muchos lugares que le parecían "factibles", hasta que descubrió (chocándose) que había una pared ahí que impedía el paso.

Estructuras utilizadas

Tabla hash

Para guardar las casillas *NodoMapa* que va recorriendo el robot se me ocurrió utilizar un `Tabla Hash`, cuyo método para resolver colisiones es encadenamiento. Me pareció la opción más adecuada debido a que mi algoritmo requiere muchas consultas de si una casilla ya fue visitada con anterioridad, entonces puedo aprovechar el tiempo de búsqueda promedio `O(1)`. En cuanto al método para resolver colisiones, me pareció más adecuado

el encadenamiento ya que reduce las probabilidades de colisiones (a que si se usara direccionamiento abierto), y reduciendo las colisiones, las búsquedas en listas enlazadas de k elementos, con k constante pequeña, no son costosas. Se podría asegurar la no-ocurrencia de colisiones si el tamaño inicial de la tabla fuera $N \cdot M$, donde N y M indican el largo y ancho del mapa respectivamente. No obstante, la consigna aclara que el robot desconoce las dimensiones del mapa.

Por lo cual, se me ocurrió proponer un tamaño inicial de $p(2 \cdot |(i_2 - i_1)| \cdot |j_2 - j_1|)$, donde $p : \mathbb{N} \to \mathbb{N}$ denota a la función que busca el primo más cercano (por arriba) el número dado, mientras que $2 \cdot |(i_2 - i_1)| \cdot |j_2 - j_1|$ es el doble de la distancia de Manhattan entre el punto de partida y llegada del robot. Es decir, se hace el doble de espacio del "rectangulo" cuyos vértices opuestos son (i_1, j_1) e (i_2, j_2). El doble para prever situaciones en las que el robot se salga de dicha área determinada por el rectángulo. En cuanto al tamaño primo, es útil para minimizar la formación de **clusters**, ya que de lo contrario muchos elementos podrían caer en la misma casilla solo porque al ser hasheados, los resultados sean comunes divisores del tamaño de la tabla.

La función hash que elegí forma usa **combinación de bits** para asignar pares clave-valor, en particular:

```
h(x, y) = (y << 16) ^ x
```

Lo que hace es aplicar un *shift* hacia la izquierda en 16 bits al número y en binario. Al resultado de eso, se le aplica un **XOR** con el valor de x en binario.

Se usa un **XOR** y no cualquier otra compuerta binaria ya que la tabla de verdad del **XOR** distribuye las probabilidades de 0/1 de manera mas uniforme (mientras que en un **AND** la aparicion de un 0 es la mas probable, asimismo en un **OR** la aparicion de un 1 es más probable).

El factor de carga $\alpha = \frac{\text{elementos}}{\text{tam tabla}}$ fue puesto en $0,75$. Cuando la tabla sobrepasa dicho umbral, se realiza un rehasheo de la tabla. (tanto el umbral, la función hash como el tamaño de la tabla fueron pensados para que las operaciones de rehasheo sean las mínimas, y en los mapas menos complejos ni haga falta)

Pila

Para guardar los nodos por los que se va construyendo el camino final, decidí usar una `Pila` (en particular, implementada con listas enlazadas generales). La misma va registrando todo nodo que el robot va recorriendo, en donde debido a la propiedad *LIFO* (Last-In-First-Out) de las pilas, en el tope se encuentra el ultimo nodo visitado, despues del tope se halla el segundo nodo visitado, ..., etc). Entonces, si el robot debiera retroceder sobre sus pasos n veces, bastaría con remover el tope de la pila n veces. Obs. que cuando el robot ya haya llegado a la meta, el contenido de la pila no necesariamente tendrá los nodos exactos que recorrió el robot en su camino (sólo en los casos donde no haya sido necesario remover ninguna vez el tope de la pila, i.e. que no se haya recurrido nunca al backtracking). Es por ello que el recorrido final del robot no puede ser guardado acá, y en su lugar simplemente se usa un arreglo de caracteres dinámico: `char* rastro`.

Criterio de igualdad de nodos NodoMapa

Simplemente se trata de una igualdad de pares ordenados:

$(a_1, a_2) = (b_1, b_2) \iff a_1 = b_1 \wedge a_2 = b_2$

Operación que sólo se utiliza para buscar en las listas enlazadas que corresponde al i-ésimo casillero de la tabla hash. Como las listas no son ordenadas, no es necesario que la relación entre nodos NodoMapa sea de órden.

Ideas descartadas

AVL para nodos visitados

Un primer enfoque que se me había ocurrido era guardar los nodos visitados en un `AVL`. De esta manera, al ser una estructura de datos dinámica, en lugar de asignar memoria (inicialmente) de $p(2 \cdot |(i_2 - i_1)| \cdot |j_2 - j_1|)$, un valor que puede llegar a ser grande en cuanto más alejado esté el punto de partida respecto al de llegada,

solamente se asignaría memoria para k nodos, donde k es el numero de nodos del mapa por los cuales estuvo el robot. Finalmente me decidí por las tablas hash, debido a que el tiempo de búsqueda en un AVL en promedio es `O(log n)` (siendo n la cantidad de nodos del árbol), en contraste al promedio `O(1)` de una tabla hash.

Compilación

```
make
```

O alternativamente:

```
gcc main.c robot_utils.c estr/pila.c estr/glist.c estr/tablahash.c -Wall -Wextra -std=c99 -g -o parte1
```

Ejecución

```
./parte1 ejemplos/ejemplo.txt
```

Parte 2

Idea

En esta parte el robot cuenta con un sensor de longitud k y se puede asumir que ahora conoce las dimensiones N y M del mapa. El objetivo es que el robot llegue a la meta aprovechando el sensor, pero siendo el uso del mismo mínimo (pues usa mucha batería).

Vale aclarar que si bien el robot no conoce la longitud de su propio sensor, la puede ir "deduciendo" a medida que lo va utilizando. Empieza asumiendo que su sensor posee la longitud mínima, 1, y cuando usa el scanner y en alguna direccion reconoce una pared a distancia k, con $k > 2$, entonces sabe que su sensor puede ver hasta por lo menos $k-1$ casilleros de distancia. De esta manera, eventualmente termina deduciendo la longitud, o un valor aproximado de la misma.

Mi solución se basa en una versión adaptada del algoritmo de replanning `D* Lite (S. Koenig & M. Likhachev, 2002)` [1], el cual usa búsqueda incremental, o sea que utiliza toda la información recabada de las búsquedas anteriores. Primeramente, veamos la estructura usada para representar los nodos:

El robot se crea un mapa interno en el cual irá almacenando información recabada por el sensor e irá calculando la ruta óptima a seguir. El mapa, de dimensiones NxM se trata de un arreglo de tipo `Estado`, con campos:

- el valor g, el cual se define como: $g(s) := d(s, s_{meta})$, dado s un nodo del conjunto de estados, siendo s_{meta} el nodo al que el robot quiere llegar, y d en este caso es la *distancia de Manhattan* entre dos vertices en $\mathbb{N}^2$. Sólo se trabajará con esta distancia, debido a que la consigna indica que los movimientos del robot son únicamente en los 4 sentidos cardinales (L, D, U, R). Por otra parte, se consideraría la *distancia Euclideana* si se permitieran movimientos diagonales.
- el valor `rhs`, el cual se trata del valor g "esperado" del estado. Se dice así ya que es calculado en base a los vecinos del estado actual. Se define como: $rhs(s) := min_{u \in Ady(s)}(c(s,u) + g(u))$, es el valor g que "debería" tener el nodo, si suponemos que los costos de los nodos adyacentes cambiaron y existe una nueva ruta óptima. En dicho caso, el nodo es inconsistente.
- `TipoCasilla`, que se encarga de diferenciar si la celda es: desconocida, válida (no fue visitada, pero el sensor ya la detectó dentro de su rango alguna vez), si la celda ya fue visitada o si la celda se trata de un obstáculo.

[1] Fuente: `https://cdn.aaai.org/AAAI/2002/AAAI02-072.pdf`

Principales diferencias en esta implementación

- Como ya se mencionó, los movimientos del robot son sólo en los 4 puntos cardinales, de modo que es suficiente medir las distancias con la *distancia de Manhattan*.
- `D* Lite` está pensado para replanear en entornos dinámicos (lo cual aplica a nuestro caso, donde no se conoce de la existencia de un obstáculo hasta que no es detectado por el escáner), pero en particular también puede prever aparición de nuevos obstáculos sobre la marcha (e.g. que aparezca un enemigo en una casilla que ya fue recorrida, y que al querer volver por ahí no se pueda). Para ello, se usan grafos dirigidos, de modo que los valores g de s_i a s_j pueden diferir de los costos de s_j a s_i. En nuestro caso, será suficiente con verlo como un grafo no dirigido, lo cual permite simplificar el conjunto de *sucesores* y de *predecesores* de un estado s, en un único conjunto de estados *adyacentes*.

Dirección de búsqueda invertida

Para la expansión de los nodos, el algoritmo se comporta de manera similar a `A*`, pero con la dirección de búsqueda al revés. Es decir, en lugar de encontrar la mejor ruta desde el nodo inicial hasta el nodo final bajo la asunción de que no hay obstáculos de por medio (1), se encuentra la mejor ruta desde el final hasta el principio, donde el robot arranca (2).

Supongamos (1), entonces obs. que dados los nodos s_1, s_2, s_3, s_4 contiguos entre sí, con $c(s_1, s_2) = n$, $c(s_2, s_3) = m$, donde se denota con c a la función costo de movimiento entre dos nodos; n, $m \in \mathbb{N}^+$. Cuando el robot se mueve, el estado inicial s_{inicio} ahora pasa a ser la posición actual del robot, de modo que si fuera $s_{inicio} = s_1$, tendríamos que el costo de viajar de s_{inicio} a s_3 sería de $n+m$, mientras que si fuera $s_{inicio} = s_2$, el costo de viajar de s_{inicio} a s_3 sería de m, por lo cual los valores g no se preservarían entre cada movimiento del robot. Para solucionar esto, habría que recomputar los costos cada vez que el robot se mueve, lo cual se volvería costoso.

Si optamos por (2), vemos que dado $s_{meta} = s_4$, tenemos que si $s_{inicio} = s_1$, $c(s_3, s_{meta}) = k$, y si $s_{inicio} = s_2$, se mantiene que $c(s_3, s_{meta}) = k$, de manera que nos ahorramos bastantes cómputos de valores g que no se ven influenciados por inconsistencias locales (es decir, cuando el robot se encuentre un obstáculo, sólo resolverá los costos para sus adyacencias, en lugar de recomputar todo el camino desde la posición actual hasta la meta de nuevo).

Es por ello que el valor g se define como la distancia del nodo actual a la salida, y la heuristica (o valor h), como la distancia del nodo actual al inicio del recorrido.

Estructuras utilizadas

El algoritmo hace uso de una `Cola de prioridad` para procesar a los nodos localmente inconsistentes [2] En mi caso, decidí implementar la cola de prioridad con un min-heap binario. La clave de prioridad resulta un tupla $k = [k_1, k_2] = [min((g(s), rhs(s)) + h(s, s_{inicio}); min((g(s), rhs(s)]$, donde la clave k_2 solo está para desempatar entre dos nodos que compartan prioridad k_1.

La utilidad de esta estructura es que manda a procesar a los nodos de mayor prioridad primero (esto es, los nodos con una clave k más baja), lo que ayuda a encontrar la ruta óptima de manera más rápida y por ende las replanificaciones son más breves, ya que sólo se encargan de resolver las porciones del grafo que han sido afectadas por los cambios, es decir los cambios de entorno "locales". A medida que va siendo necesario (e.g. obstáculos cerca de la meta), se van desencolando los nodos necesarios para llegar, pero se evita tener que actualizar todo nodo que no esté involucrado en la ruta óptima.

De manera simplificada, el funcionamiento del algoritmo es el siguiente:

[2] Dado un nodo s, se define al nodo como consistente si $g(s) = rhs(s)$, esto significa que el valor g esperado (en base a sus vecinos) es el óptimo. Si $g(s) > rhs(s)$ el nodo es sobreconsistente, es decir que el valor g actual es menor al valor g esperado en base a sus vecinos, por lo que es conveniente actualizar el valor g por este. Y si $g(s) < rhs(s)$, el nodo el subconsistente, y no es conveniente cambiar de valor g aún.

- Inicializar robot en (i_1, j_1). s es consistente, con $g(s) = rhs(s) = \infty \; \forall s$ estado.
- Realizar una propagación de nodos hasta que el nodo s_{inicio} sea consistente con sus adyacentes.
- Intentar moverse por la ruta óptima calculada, esto es: $s_{inicio} = min_{u \in Ady(s_{inicio})}(c(s_{inicio}, u) + g(u))$.
- Si el nodo es desconocido, tirar el sensor. Si la longitud actual del sensor es k, y en alguna dirección se registra un obstáculo a distancia l del robot, con $l - 1 > k$, entonces reasignar longitud del sensor a $l - 1$. Aparte, registrar obstáculos y casilleros válidos en el mapa interno del robot.
- Mientras el siguiente nodo de la ruta óptima sea una posición válida, seguir recorriendo y de encontrar un nodo desconocido, repetir el paso anterior.
- Si el siguiente nodo de la ruta es un obstáculo:
  - Procesar posibles cambios en la heurística (en efecto, el valor se modifica cada vez que el robot cambia de posicion) (*)
  - Actualizar los costos del nodo obstáculo y replanificar en base a este cambio
- Repetir, hasta que el robot llegue a (i_2, j_2)

(*) Obs que si notamos a s_{ant} como el nodo anterior por dónde estuvo el robot, y s_{inicio} es su nodo actual, tenemos que las heurísticas entre estos nodos difieren en $h(s_{ant}, s_{ini})$. Por lo cual, introducimos un desfasaje k_m (modificador de clave), de modo que cada vez que se agregue un nuevo nodo a la cola de prioridad, se deberá incrementar k_m en $h(s_{ant}, s_{ini})$. Asimismo, se deberá añadir el modificador a la definición de la clave del heap:

$$k = [k_1, k_2] = [min((g(s), rhs(s)) + h(s, s_{inicio}) + k_m;\ min((g(s), rhs(s)]$$

De esta manera, las prioridades de la cola de prioridad se mantienen actualizadas con respecto a los posibles cambios en el mapa.

Compilación

`make`

O alternativamente:

```
gcc main.c robot_utils.c estr/bheap.c -lm -Wall -Werror -std=c99 -o robot
```

Ejecución

```
./correr.sh ejemplos/ejemplo.txt
```