Tarea 5 - Programación y algoritmos 2

Erick Salvador Alvarez Valencia

CIMAT A.C.,
erick.alvarez@cimat.mx

Resumen En la presente tarea se describirá el código implementado para el problema llamado *Robots on ice* el cuál se explicará a detalle en la siguiente sección, de igual manera se mostrará el algoritmo utilizado para atacar este problema y los resultados obtenidos.

1. Descripción del problema

Dada una matriz de m filas y n, un punto de inicio y un punto final, se quiere saber cuántos caminos existen entre dichos puntos y pasando exactamente una vez por todos los demás, además se dan tres grids especiales llamados *Checkpoints* en los cuales se debe pasar en un cierto momento del trayecto, más específicamente, se debe pasar por el primer grid en $\frac{1}{4}$ del trayecto, por el segundo en $\frac{1}{2}$ del trayecto y por el tercero en $\frac{3}{4}$ del trayecto.

Las restricciones del problema son:

1. Tiempo límite: 9s
2. $2 \leq m, n \leq 8$ (número de filas y columnas)
3. $0 \leq r_i < m$ (Posición en filas de los checkpoints)
4. $0 \leq c_i < n$ (Posición en columnas de los checkpoints)

2. Solución implementada

Una forma de atacar este problema es por medio de una búsqueda completa (backtracking), si analizamos las restricciones podemos ver que en el peor de los casos se tiene una matriz de $8x8$ por lo cual si generamos todos los caminos verificando los 4 vecinos adyacentes por casilla tenemos una solución de $O(64^4)$ lo cual es muy probable que no entre en el tiempo que da el problema así que hay que recurrir a la técnica de *Poda* la cual elimina caminos en los que se sabe que no se llegará a una solución. El libro de *Competitive Programming* sugiere varias formas de ir podando la búsqueda y por lo mismo se reduce mucho la complejidad del algoritmo, a continuación se verán dichas formas:

1. Si la casilla actual está fuera de los límites de la matriz (Caso trivial).
2. Si la casilla actual es un *Checkpoint* pero aún no era tiempo de pasar por él.

3. Si es tiempo de pasar por el siguiente *Checkpoint* pero se atravesó por otra casilla distinta.
4. Si desde la posición actual no se podrá llegar al siguiente *Checkpoint* antes de que sea tiempo de pasar por él (En este caso se usa la distancia de Manhattan para verificar).
5. Si hay casillas a las que ya no se podrá llegar porque se tendría que pasar por segunda vez a través de otras casillas (El robot se bloqueó).

Como se puede observar, el hecho de que el existan los *Checkpoints* en el problema reduce mucho el espacio de búsqueda y hace que el algoritmo pueda entrar en tiempo. En general el algoritmo queda de la siguiente manera:

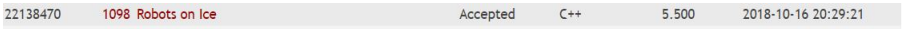
Algorithm 1 Búsqueda completa.

```
1: procedure CompleteSearch(Mat, y, x, m, n, &noPaths, deep, nextCP, cpR, cpC, cpTime)
2:     if y = 0 and x = 1 then
3:         if deep = m * n then
4:             noPaths ← noPaths + 1
5:         return
6:     hit ← False
7:     if nextCP < 3 and y = cpR[nextCP] and x = cpC[nextCP] then
8:         if deep < cpTime[nextCP] then      ▷ Pisó un CheckPoint antes de tiempo
9:             return
10:        else
11:            hit ← True
12:    if nextCP < 3 hit = False and deep = cpTime[nextCP] then      ▷ No pisó el CheckPoint en el momento exacto
13:        return
14:    if nextCP < 3 then
15:        md ← manhattan(y, x, cpR[nextCP], cpC[nextCP])
16:        if deep + md > cpTime[nextCP] then ▷ No alcanzará a pisar el siguiente CheckPoint a tiempo
17:            return
18:    if isBlocked(y, x) then                                        ▷ Se bloqueó
19:        return
20:    for (xi, yi) ← casilla adyacente a (x, y) do
21:        if checkBoundaries(xi, yi) = False or Mat[yi][xi] = False then
22:            continue
23:        Mat[yi][xi] = True
24:        if hit = True then
25:            CompleteSearch(Mat, yi, xi, m, n, noPaths, deep + 1, nextCP + 1, cpR, cpC, cpTime)
26:        else
27:            CompleteSearch(Mat, yi, xi, m, n, noPaths, deep + 1, nextCP, cpR, cpC, cpTime)
28:        Mat[yi][xi] = False
```

El algoritmo anterior ejecuta la búsqueda completa aplicando los criterios anteriormente mencionados, el número de caminos exitosos es almacenado en la variable *noPaths* y por lo mismo esta se pasa por referencia.
En el algoritmo anterior, para poder verificar si en un estado hay casillas a las que no se pueden llegar una manera es hacer un DFS/BFS partiendo desde la casilla final hacia todas las demás, si al hacer la búsqueda hay alguna casilla a la que no se accedió y esta casilla no pertenece al camino que se lleva hasta el momento entonces dicho camino ya no es viable.

3. Resultados

Ejecutando el algoritmo anterior se obtuvo un resultado exitoso por parte del juez aunque el tiempo de ejecución no fue tan bueno (5.5s) como se muestra en la siguiente Figura.



(a) Figura 1. Envío 1 al juez UVA del problema *Robots on ice*.

Esto se debe a que en cada paso se hace un BFS y posteriormente se verifica si alguna casilla no pudo ser alcanzada por el mismo, esto tiene un costo $O(nm)$ y en parte es lo que alenta más a la solución, por lo cual se sustituyó por un pequeño algoritmo que ayuda a verificar esto anterior en $O(1)$. Dicho algoritmo toma el punto actual y verifica a todos sus 8 vecinos en sentido horario empezando por el que está en la parte superior izquierda, posteriormente cuenta el número de cambios que hay en este recorrido, un cambio es el hecho de que un vecino no se accesible y el siguiente si o viceversa, si al final el número de cambios es mayor que tres entonces no hay bloqueos existentes, por lo que con esto se puede evitar el hacer un DFS por cada paso de la búsqueda y el tiempo de ejecución baja mucho como se puede ver en la siguiente Figura.

(b) Figura 2. Envío 2 al juez UVA del problema *Robots on ice*.

Al final se logró con esta mejora un tiempo de ejecución de 0.35 s.