Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos
Segundo semestre 2017

Tarea 3

Fecha de entrega: Miércoles 1 de Noviembre a las 23:59

Objetivos

- Resolver un problema de búsqueda dentro del grafo implícito de los estados de un problema.
- Diseñar una función de hash acorde a un problema e implementar una tabla de hash.

Introducción

¿Recuerdas los puzzles con los que jugabas cuando pequeño?. Uno de los puzzles más comunes y bastante complicados para aquella edad era el *sliding puzzle*. En este formato de puzzle la imagen original es dividida en $m \times n$ *celdas* y una de ellas es removida de modo que las demás se puedan desplazar para ordenar y reordenar las piezas de modo de reconstruir la imagen original.

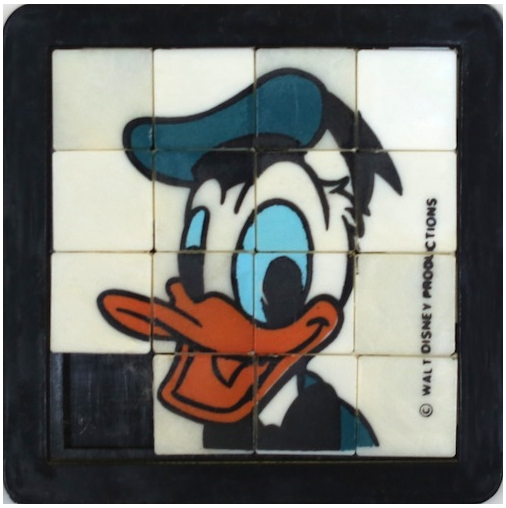


Figura 1: Sliding puzzle de 4×4

En esta tarea se resolverá un caso un poco distinto pero basado en el mismo tipo de tablero, llamado Flip Puzzle. Imagina que en vez de remover una de las piezas del puzzle de $h \times w$, las piezas son desordenadas mediante la inversión de filas y columnas, y para ordenarlas solo se te permite realizar ese tipo de movimientos. ¿Cómo lo harías?

Problema: Flip puzzle

Formalmente, tanto el sliding puzzle como el flip puzzle son en realidad una cuadrícula de $h \times w$ con números. El objetivo del puzzle es lograr hacer que cada celda de la cuadrícula vuelva a tener el valor que tenía originalmente en el puzzle ordenado. Estos valores van de 0 a $hw - 1$, como se muestra en la Figura 2

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figura 2: El tablero resuelto, S_f

Los movimientos sobre el tablero corresponden a inversiones de una fila o una columna específica, los que llamaremos *flips*. Al aplicar flip en una columna o fila, se intercambia el orden de los elementos dejándolos en el orden inverso. Los abreviaremos como sigue:

- Flip de la fila `y` $\implies R_y$
- Flip de la columna `x` $\implies C_x$

Estos se pueden ver como **funciones** que se aplican sobre el tablero: al aplicar una de ellas, se obtiene un nuevo tablero con una distinta configuración. La Figura 3 ilustra esto con el tablero de la Figura 2.

0	1	14	3
4	5	10	7
8	9	6	11
12	13	2	15

(a) $C_2(S_f)$

0	1	14	3
7	10	5	4
8	9	6	11
12	13	2	15

(b) $R_1(C_2(S_f))$

Figura 3: Ejemplos de flip

Formalmente, nos referimos a estas funciones como las **operaciones** que llevan de un **estado** del problema a otro. Nótese que para toda operación P, $P(P(S)) = S$, es decir, cada operación es su propia inversa.

Deberás escribir un programa en C que dado una matriz desordenada, encuentre la cantidad mínima de operaciones de flip que la deja ordenada. Para esto, se espera que ejecutes **BFS** sobre el **espacio de estados** del problema.

Espacio de Estados

Para problemas como este, se llama **estado** del problema a una configuración específica del tablero.

Se puede definir un grafo $G(V, E)$ de la siguiente manera:

- V son todos los estados distintos del problema.
- $(u, v) \in E$ si y sólo si existe una operación P tal que $P(u) = v$

Llamamos a G el **espacio de estados** del problema.

Como podrás imaginar, $|V|$ es enormemente grande, así que no es posible generarlo completo.

Para ejecutar BFS sobre él, lo que debes hacer es ir generando uno a uno los estados a medida que los encuentras, y llevar un registro de los estados que ya has visto antes, para no abrir dos veces un mismo estado. Para esto necesitarás un **diccionario**, y se recomienda que sea una **tabla de hash**.

Tu recibes el **estado inicial** del problema, y tu objetivo es convertirlo en S_f, el tablero resuelto.

Librería: Estados + Operaciones

Para que puedas llegar y ponerte a programar, se ha preparado una librería que maneja la lógica de los estados y sus operaciones. Esta está en `Programa/src/state/state.h`

Sus tipos son los siguientes:

- `State`: es una matriz de filas de $h \times w$. Puedes accederla con `s[row][col]`, para `State s`.
- `Operation`: contiene información del tipo de la operación (R o C), y el índice sobre el que actúa.

Tiene las siguientes constantes como variables globales:

- `height`: alto de la cuadrícula, h
- `width`: ancho de la cuadrícula, w
- `operations`: un arreglo con las $h + w$ operaciones posibles.
- `op_count`: $h + w$, la cantidad de operaciones posibles.

Básicamente, `operations` $= [\ R_0,\ \cdots\ ,\ R_{h-1},\ C_0,\ \cdots\ ,\ C_{w-1}\ ]$, lo que te permite iterar sobre este arreglo para generar todos los hijos de un nodo en el grafo.

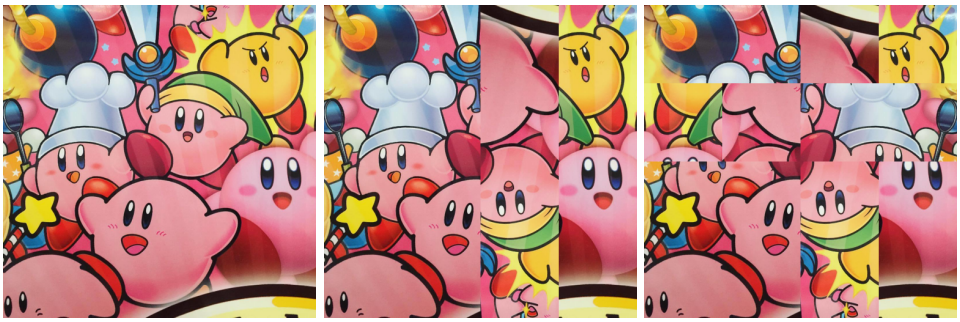
Algunas funciones son las siguientes:

- `State state_init(filename)`: Lee el puzzle a partir de un archivo, prepara las variables globales, y abre el watcher.
- `State state_next(state, op)`: obtiene el estado producto de aplicar una operación.
- `bool state_is_solution(state)`: Entrega true si y solo si el estado es solución

Asegúrate de revisar las demás funciones en `Programa/src/state/state.h`.

Interfaz: Watcher

Se ha preparado una interfaz gráfica que permite la visualización de tanto los estados como las operaciones:



(a) S_f (b) $C_2(S_f)$ (c) $R_1(C_2(S_f))$

Figura 4: Ejemplos de flip en la interfaz

Está definida en `Programa/src/watcher/watcher.h`. Para controlarla, se proveen las siguientes funciones:

- `watcher_open(img_file, height, width)`: Abre una ventana para mostrar una imagen sobre el tablero de las dimensiones dadas.
- `watcher_flip_row(row)`: Visualiza la inversión de la fila especificada.
- `watcher_flip_col(col)`: Visualiza la inversión de la columna especificada.
- `watcher_snapshot(filename)`: Genera una imagen PNG con el contenido actual de la ventana.
- `watcher_close()`: Cierra y libera los recursos de la ventana.

También puedes hacer uso directamente de `operation_watch(operation)` para visualizar esa operación.

Input

Tu tarea debe recibir el siguiente input:

```
./solver <puzzle.txt>
```

Donde `<puzzle.txt>` es el archivo con la descripción del puzzle y posee la siguiente estructura:

- La primera línea contiene la ruta de la imagen correspondiente al puzzle.
- La segunda línea contiene el alto y el ancho de la imagen, `H W`
- La siguientes `H` líneas contienen `W` números separados por espacios, correspondientes al estado inicial del puzzle.

Output

El output de tu programa es mostrar correctamente la imagen en su estado resuelto. Se medirá la cantidad de operaciones que hayas enviado a la interfaz, por lo que cuida de hacer solo las necesarias al entregar tu tarea.

Evaluación

Tu código será evaluado con diferentes tests y se evaluará configuración de la matriz de la interfaz luego de cerrar el watcher. Se medirá la cantidad de operaciones realizada para asegurar que sea mínima.

En esta tarea no habrá informe, por lo tanto la nota de tu tarea va a corresponder:

- 50% a que tu código resuelva el problema correctamente.
- 50% a que la cantidad de pasos sea mínima.

Tendrás 10 segundos para cada test. Si tu programa no responde en ese tiempo, será cortado y recibirás 0 puntos en ese test.

Entrega

- **Lugar:** GIT - Repositorio asignado (asegúrate de seguir la estructura inicial de éste).
  - En la carpeta *Programa* debe encontrarse el código.
  - Se recogerá el estado en la rama *master*.
- **Hora:** 1 minuto pasadas las 23:59 del día de la entrega.
- Se espera que el código compile con `make` dentro de la carpeta *Programa* y genere un ejecutable de nombre `solver` en esa misma carpeta.
- No se permitirán entregas atrasadas.

Bonus

- **Manejo de memoria perfecto (+5%):**
  Se aplicará este bonus si **valgrind** reporta en tu código 0 leaks y 0 errores de memoria, considerando que tu programa haga lo que tiene que hacer.
- **Gotta go fast (+20%):**
  Este bonus se aplicará si tu tarea es capaz de resolver tests más difíciles. Para esto, se espera que potencies tu implementación de **BFS** agregándole heurísticas, convirtiéndolo efectivamente en $\mathbf{A}^\star$