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

Tarea 3

IIC2133 - Estructuras de Datos y Algoritmos

Primer semestre, 2017

Entrega: Lunes 12 de Junio

Objetivos

- Optimizar un problema mediante el uso de tablas de hash
- Diseñar multiples funciones de hash ad-hoc a un problema y comparar su desempeño
- Analizar el impacto de una tabla de hash en un algoritmo de búsqueda (en el espacio de estados)

Problema

En esta tarea hay dos problemas: resolver un problema de búsqueda e implementar un diccionario de tal manera de hacer eficiente la búsqueda. Para beneficio de los estudiantes, el problema de búsqueda ya está resuelto. Su tarea consiste en implementar el diccionario, que en este caso se restringirá a una tabla de hash, y analizar su eficiencia.

Problema de Búsqueda

Se tiene una grilla rectangular de $n \times m$ bloques de colores como se muestra en la figura 1. Los valores de n y m están sujetos a las restricciones $0 \leq n < 64$, $0 \leq m < 64$, y a lo más pueden haber 8 colores distintos, es decir, en cada celda hay un número entre 0 y 7 inclusive.

Figura 1: Ejemplo de una grilla.

Se puede modificar el estado de esta grilla mediante las siguientes operaciones:

- `R i`: Desplazamiento de la i-ésima fila hacia la derecha.
- `L i`: Desplazamiento de la i-ésima fila hacia la izquierda.
- `U j`: Desplazamiento de la j-ésima columna hacia arriba.
- `D j`: Desplazamiento de la j-ésima columna hacia abajo.

Al desplazar una fila o una columna, el bloque que sale del tablero aparece por el otro lado. El objetivo del problema es encontrar la secuencia de operaciones que lleva de un estado a otro. Se puede apreciar esto en la figura 2.

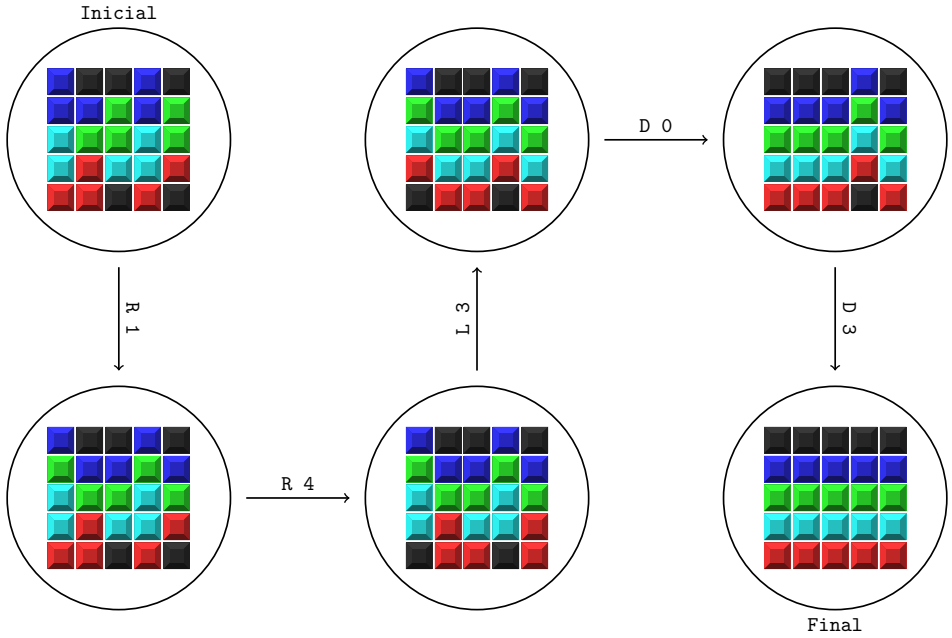

Figura 2: Secuencia de pasos que lleva del estado inicial al estado final

El problema de búsqueda consiste en determinar la secuencia de movimientos más corta que lleva desde un estado inicial a un estado final.

Diccionarios

Como podrás notar, en el problema anterior es muy fácil tener estados repetidos. En la figura 3 se puede apreciar cómo a partir de un estado se pueden generar 12 estados sucesores, pero solo 5 de éstos estados son únicos. En particular, hay un estado que se repite 8 veces. Los hijos de estos a su vez generan aún más estados repetidos.

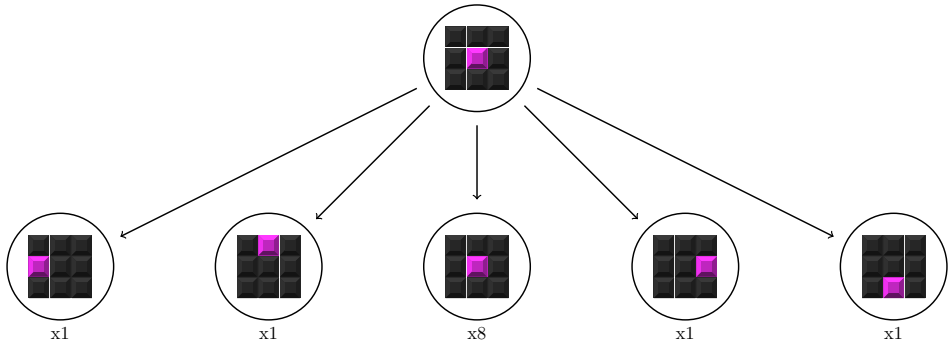

Figura 3: Repetición de estados sucesores.

Es muy probable que a partir de un estado se llegue a otros que ya han sido visitados. La figura 4 muestra todo el espacio de estados y transiciones entre éstos para la grilla mostrada en la figura 3.

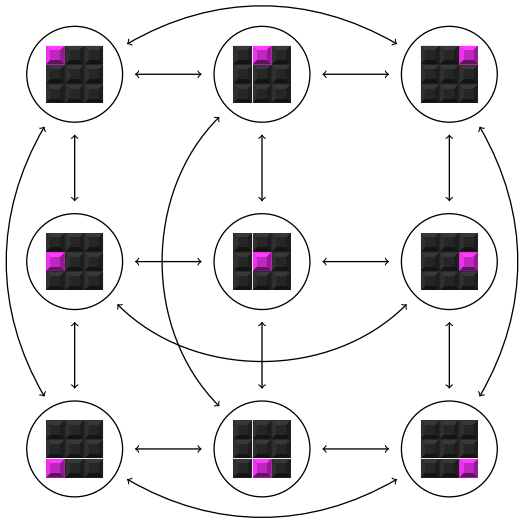

Figura 4: Espacio de estados de un puzzle.

Por lo tanto, lo ideal sería comparar el estado actual con los ya visitados para ver si se ha pasado antes por este estado, y no seguir explorando en esta dirección si éste es el caso.

Para poder hacer esto de manera eficiente lo que se hace es usar como diccionario una Tabla de Hash

Su tarea será diseñar funciones de hash ad-hoc al problema anteriormente descrito e implementar una tabla de hash que almacene los estados ya visitados. Específicamente, se requerirá:

1. Una tabla de hash. Ustedes eligen si el direccionamiento es abierto/cerrado, cuándo/cómo agrandar la tabla, etc.
2. Una función de hash incremental cuyo tiempo de ejecución sea proporcional al número de celdas cambiadas. Con incremental se refiere a que debe ser calculada a partir del valor de hash del estado anterior y la operación realizada. No es necesario que esta función cumpla ninguna propiedad especial además de distribuir de manera razonablemente uniforme.
3. Una función de hash perfecta, es decir, tal que dos estados son iguales si y solo si sus valores de hash son iguales. En otras palabras, la función debe ser inyectiva pero no necesariamente sobreyectiva. (Ver Anexo)

Tu objetivo es en base a evidencia empírica y analítica, decidir cual de las dos funciones de hash es mejor para la búsqueda. Para esto, se espera que elabores un informe detallado justificando tu decisión. En particular, se espera lo siguiente:

Análisis Teórico

En tu informe deberás, formalmente:

- Justificar el tipo de tabla de hash utilizada
- Explicar cada función de hash y calcular su complejidad en notación O.
- Determinar el recorrido de la función de hash perfecta.
- Demostrar la inyectividad de la función de hash perfecta.

Análisis Empírico

Además, en tu informe deberás comparar de manera práctica el desempeño de su tabla y funciones de hash. En particular, se espera que analices cómo varía el desempeño al variar los siguientes parámetros:

- Tamaño de la tabla de hash (numeros primos, compuestos, potencias de 2, etc.)
- Tamaño del puzzle
- Profundidad de la solución
- Factor de carga λ con el cual la tabla se amplía

Además, para cada función de hash se espera que analices:

- Tasa de colisiones
- Distribución final de los datos en la tabla (en forma de histograma)
- Costo de búsqueda (promedio y acumulado)
- Costo de inserción (promedio y acumulado)
- Costo de calcular el hash

Se espera que construyas gráficos y tablas para presentar de manera amigable todos los datos recolectados, y que luego **entregues explicaciones/conclusiones acerca del comportamiento del programa** al variar los distintos parámetros.

Finalmente, debes decidir cual de sus dos funciones de hash es la mejor y **dejar esa funcionando en tu código**.

Input y Output

Esta parte no debe ser implementada por ustedes, pero de todas maneras se describe para que tengan como saber si sus cambios en el código afectaron el funcionamiento de la búsqueda, dado que el usar un diccionario no debería afectar la solución en sí.

Output

El output del programa es un archivo llamado `output.txt` que sigue el siguiente formato:

1. Una línea con un número `n` correspondiente a la cantidad de operaciones que lleva del estado inicial al final
2. `n` líneas con cada una de las distintas operaciones en orden que resuelven el problema

Por ejemplo, la solución del puzzle de la figura 2:

```
5
R 1
R 4
L 3
D 0
D 3
```

Input

El programa se llama con

```
./untangle
```

y recibe directamente el puzzle desde la consola. Automáticamente se abrirá una interfaz gráfica GTK [1] para poder visualizar la resolución del puzzle. Esta interfaz puede suprimirse pasandole el flag `-s` a `untangle`, esto es, llamandolo como

```
./untangle -s.
```

Puzzles aleatorios

Para que puedan probar de manera apropiada sus funciones y tabla de hash, se pone a su disposición un generador de puzzles aleatorios. Se puede llamar de dos maneras distintas:

```
./generator [seed] [entropy] [h] [w]
./generator [seed] [entropy] -i [path.png]
```

donde

- `seed` es la semilla aleatoria
- `entropy` indica cuántas veces se hace una operación para desordenar el puzzle
- `h` indica el alto del puzzle
- `w` su ancho
- `path.png` es el path a una imagen en formato png a partir de la cual se construirá el puzzle. Esta imagen debe contener como máximo 8 colores.

El generador imprime el puzzle directamente en consola. Para guardar el puzzle en un archivo debe llamarse de la siguiente forma:

```
./generator [seed] [entropy] [h] [w] > [puzzle.txt]
```

donde `puzzle.txt` es el path al archivo donde quieren guardar el puzzle. Para luego pasarle este archivo como input al programa `untangle`, este se debe llamar con

```
./untangle < [path.txt].
```

Alternativamente, si quieren generar un puzzle y pasarselo directamente al programa (sin un archivo .txt intermedio) se debe ejecutar lo siguiente:

```
./generator [seed] [entropy] [h] [w] | ./untangle
```

Creación de tablas

Para hacer más rápido el testeo y la generación de datos para su informe también se les provee un script que automáticamente ejecuta el programa con varios puzzles e imprime el output en el archivo "tabla.csv"para poder abrirla directamente con Excel o un programa similar. Se llama[2] con

```
./tabular.sh [entropy] [h] [w] [l] [p]
```

donde `entropy`, `h` y `w` son los mismos parametros que se le pasaban al generador, `l` es el número de puzzles con que se correrá el programa y `p` es el header que irá en el archivo csv: los nombres de las columnas separados por coma.

[1] De la misma naturaleza que la de la T1, ver aquí

[2] Recordar correr el comando `chmod +x ./tabular.sh` antes para poder ejecutar directamente el script

Para que el output quede bien formateado el programa deberá imprimir una única línea con los distintos valores de interés separados por coma. Por ejemplo, si en el programa se mide el número de inserciones en la tabla y tiempo promedio por inserción, entonces se debe imprimir una única linea con el formato "[numero de inserciones], [tiempo promedio por insercion]". Si luego el programa se llamara con el comando `./tabular.sh 10 8 8 2 "numero de inserciones, tiempo promedio por inserción"`, el programa correrá 2 veces con puzzles random de 8x8 y se guardará el output de su programa como filas en el csv de la siguiente forma:

Número de inserciones	Tiempo promedio por inserción
x_1	y_1
x_2	y_2

Es decir, el programa imprime "x_i, y_i" en su i-ésima ejecución. Notar que el código base imprime algunos mensajes, pero esos no afectan a este script dado que se imprimen de otra manera.

Código base

Dentro de la carpeta **Programa/src** existen 5 carpetas con archivos:

- **beholder:** Archivos encargados de la visualización del puzzle.
- **generator:** Programa encargado de generar los puzzles a resolver.
- **puzzle:** Archivos relacionados a la lógica del puzzle y sus movimientos. Al final de los archivos `operation.h` y **puzzle.h** se encuentran algunas funciones que podrían ser útiles para el funcionamiento de su diccionario. Estas se pueden usar pero **NO SE PUEDEN MODIFICAR**.
- **random:** Una pequeña librería que podría ser útil para generar números aleatorios de manera simple.
- **untangle:** Archivos encargados de resolver el puzzle usando un algoritmo de búsqueda y un diccionario. Los componentes relevantes son:
  - **search/dictionary/dictionary.h:** Contiene la descripción de los elementos del diccionario que debes crear. En particular debes completar los structs `Cell` y `Dictionary` para que funcionen correctamente.
    La struct `Cell` almacena los datos que usa el algoritmo de búsqueda y están guardados como una struct llamada `Info` la cual **NO SE DEBE MODIFICAR**. Además de la `Info`, `Cell` contiene un bool llamado new_cell, el cual debe ser true al momento de crear una instancia de la struct `Cell`. Además de estos atributos, la struct `Cell` puede tener más datos si lo estimas conveniente.
    La struct `Dictionary` se debe encargar de almacenar las Cells de los estados ya explorados por la búsqueda. Para esto, se debe poder crear y modificar Cells.
  - **search/dictionary/dictionary.c:** Contiene los métodos relacionados con el diccionario. En particular hay 3 métodos que **DEBES** implementar:
    - **dictionary_init:** Inicializa el diccionario y todos sus recursos. El código base llama a este antes de comenzar con la búsqueda.
    - **dictionary_get_cell:** Este es el método más importante de la tarea. Está encargado de agregar nuevos estados al diccionario y de retornar las celdas con los estados guardados. Sus argumentos son: el diccionario `dict`, el estado del puzzle `state`, la celda del diccionario con el estado anterior de búsqueda `prev`, y la operación que creó el nuevo estado a partir del anterior `op`. Los últimos dos argumentos son de especial utilidad al momento de crear una fución de hash incremental. En caso de que el estado `state` dado en el método sea el estado inicial del puzzle, entonces el argumento `prev` es igual a `NULL`.
      La función `dictionary_get_cell` está encargada de dos cosas:
      - Si el estado `state` dado como input no está en el diccionario, entonces debe crear una nueva `Cell` para guardarlo. **OJO:** al crear la `Cell` nueva se debe inicializar con la variable `new_cell` = `true`. Luego se debe retornar la celda creada.
      - Si el estado `state` dado ya está en el diccionario, entonces simplemente debe retornarlo.
    - **dictionary_destroy:** Libera los recursos asociados al diccionario. Debe completar el método para que no hayan leaks de memoria. El código base llama a este método se llama al finalizar la búsqueda.

Eres es libre de hacer estos métodos como quieras pero deben recibir y retornar los parámetros que están especificados en el archivo `dictionary.h`.

- **Otros:** Eres es libre de crear tus propios métodos, structs, archivos y carpetas dentro de la carpeta **src/untangle/search/diccionary**, pero no puedes modificar nada que se encuentre fuera de ese espacio.

Evaluación

La nota de tu tarea está descompuesta en dos partes:

- 20 % corresponde a que tu código pase los tests dentro del tiempo máximo.
- 80 % corresponde a la nota de tu informe.

Su programa con la tabla de hash implementada y la función elegida deberá ser capaz de resolver los tests en un máximo de 10 segundos. Pasado ese tiempo el programa será terminado y se asignarán 0 puntos en ese test.

Entrega

Deberás entregar tu tarea en el repositorio que se te será asignado; asegúrate de seguir la estructura inicial de éste.

Se espera que el código compile con `make` dentro de la carpeta **Programa** y genere un ejecutable de nombre `untangle` en esa misma carpeta. **No modifiques código fuera de la carpeta *src/untangle/search/dictionary*.**

Se espera que dentro de la carpeta **Informe** entregues tu informe en formato *PDF*, con el nombre *Informe.pdf*

Cada regla tiene asociado un puntaje, asegúrate de cumplirlas para no perder puntos.

Se recogerá el estado de la rama `master` de tu repositorio, 1 minuto pasadas las 23:59 horas del día de entrega. Recuerda dejar ahí la versión final de tu tarea. No se permitirá entregar tareas atrasadas.

Bonus

Hash master (+25 % a la nota de la *Tarea*)

Deberás probar ambas funciones de hash en una tabla con direccionamiento abierto y en una tabla de hash con direccionamiento cerrado, y hacer el análisis para las 4 posibles combinaciones, de manera de poder decidir cual es el mejor tipo de tabla y funcion de hash para el problema.

Debes analizar además que es lo que pasa si utilizas sobre las funciones de hash el método de la multiplicación y el método de hash universal (sumado a todo lo anterior).

Buen uso del espacio y del formato (+5 % a la nota de *Informe*)

La nota de tu informe aumentará en un 5 % si tu informe, a criterio del corrector, está bien presentado y usa el espacio y formato a favor de entregar la información.

Manejo de memoria perfecto (+5 % a la nota de *Código*)

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.

Anexo: Herramientas útiles

GMP: GNU Multiple Precision

Para la función de hash perfecta deberán manejar números muy grandes (de más de 64 bits). Para hacer esto se espera que usen la librería **GMP**. Esta librería sirve para manejar números arbitrariamente grandes. En particular les será util el tipo de datos `mpz_t`, utilizado para manejar números *enteros* de tamaño arbitrario. Pueden encontrar más información sobre esta librería en la página principal, y sobre `mpz_t` en particular en este link.

Para usar esta librería en su programa deberán instalarla escribiendo el siguiente comando en tu consola:

En **Ubuntu** y **Linux Subsystem for Windows (aka bash)**:

```
sudo apt-get install libgmp-dev
```

En **Mac OSX**:

```
brew install gmp
```

Luego en su código en `C` deberán incluir la librería con `#include <gmp.h>`.

Callgrind + Kcachegrind

Les puede resultar útil las herramientas Callgrind y Kcachegrind. Con ellas pueden generar el grafo de llamada de funciones en su programa y representar de manera visual el porcentaje de tiempo de ejecución utilizado por cada función, como se puede apreciar en la figura 5.

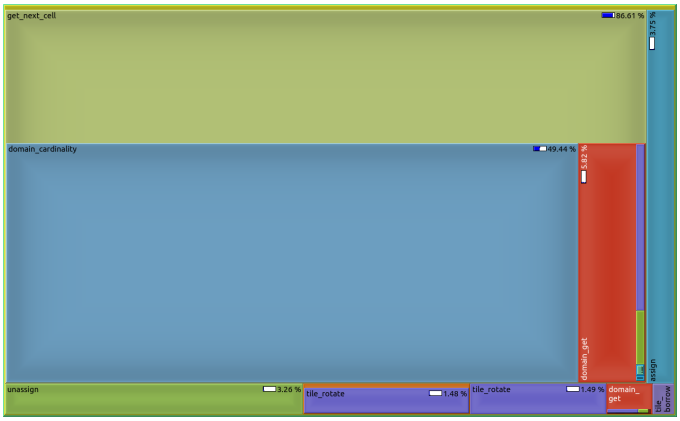

Figura 5: El area como costo de una función, en este caso, `solve` de la solución de la T1

Callgrind es una herramienta de Valgrind, pero Kcachegrind debe instalarse de manera aparte. Para usar Callgrind deben ejecutar

```
valgrind --tool=callgrind --callgrind-out-file=callgrind.dump [ejecutable]
```

Esto guarda en el archivo `callgrind.dump` la información de las llamadas a funciones en la ejecución del programa.

Luego uno puede abrir el archivo `callgrind.dump` con KCacheGrind como sigue:

```
kcachegrind callgrind.dump
```

Para obtener, entre otras cosas, gráficos como el de la figura 5.

De todas las estadísticas entregadas por KCacheGrind, el gráfico que muestra el tiempo utilizado por cada función les puede ser particularmente útil para analizar los tiempos de ejecución acumulados en su programa.

Librería time

Para medir los tiempos dentro de su programa (tiempo promedio de inserción a la tabla de hash, tiempo de búsqueda, etc.) no basta on ejecutar `time ./untangle`, ya que eso no les entrega cuanto tiempo se demora el programa en cada operación. Para hacer esto se espera que utilicen el módulo `time` dentro de la librería estándar de `C`.

A continuación pueden ver un ejemplo de uso de `time.h`:

```c
#include <time.h>

// Registramos el tiempo de CPU usado al comienzo de la operacion
clock_t start = clock();

// Operacion cara en tiempo
time_consuming_function();

// Registramos el tiempo de CPU usado al finalizar la operacion
clock_t end = clock();

// Calculamos la diferencia y pasamos el tiempo en clocks a tiempo en segundos
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
```

Básicamente, `clock()` entrega el tiempo de CPU usado por su programa y para convertirlo a segundos hay que dividirlo por `CLOCKS_PER_SEC`. Para medir el tiempo en un intervalo hay que tomar la diferencia de la función `clock` al comienzo y al final del intervalo.