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

IIC2133 — Estructuras de Datos y Algoritmos
2019 - 2

Tarea 2

Fecha de entrega código: Domingo 13 de octubre
Fecha de entrega informe: Domingo 13 de octubre

Objetivos

- Implementar distintas variedades de tablas de hash
- Experimentar con parámetros de la tabla de hash
- Analizar eficiencia de manera empírica y sacar conclusiones a partir de esto

Introducción

Baba Is You es un juego de puzle publicado en marzo de 2019 (link). El juego consiste en resolver puzles donde los distintos objetos tienen propiedades que permiten interactuar con ellos, por ejemplo `ROCK IS PUSH` indica que las rocas pueden ser empujadas, `FLAG IS WIN` indica que si el jugador toca la bandera el puzle está resuelto. Lo que lo diferencia de otros juegos de puzle es que las reglas pueden ser modificadas por el jugador en la mitad del nivel. De esta manera se puede lograr llegar a la meta cuando con las reglas originales no se podría.

Para esta tarea trabajarás con una versión simplificada del juego donde las reglas son más acotadas y no se pueden modificar en la mitad de un nivel. Harás tu código a partir de un programa que resuelve los puzles pero que necesita optimizaciones. En particular desarrollarás una tabla de hash junto con una función de hash especializada para resolver este problema específico para optimizar el programa original. Una vez hecho el programa tendrás que experimentar con distintos parámetros y modalidades de la tabla de hash para determinar cuál es la configuración que resuelve el programa de la manera más eficiente.

Baba Is You simplificado

Cada nivel del juego consiste en un tablero de $M \times N$ celdas con distintos objetos junto con un set de reglas. Estas reglas indican cómo los distintos objetos interactúan entre sí y cuál es el objetivo del nivel. Los distintos objetos que pueden haber en el nivel son:

Figura 1: Objetos

Sobre estos objetos se aplican las reglas que definen su comportamiento. Las reglas son: `YOU`, `STOP`, `PUSH`, `OPEN`, `SHUT`, `WIN`, `DEFEAT`, `MELT`, `HOT`. Cada objeto puede tener cualquier regla, no tener ninguna o tener varias. También varios objetos pueden tener las mismas reglas lo que hace que se comporten de la misma manera. A continuación se explica como funcionan las reglas:

- `YOU`: Todo lo que tenga la regla `YOU` es controlado por el jugador, es decir, se puede mover usando las direcciones `UP`, `RIGHT`, `DOWN`, `LEFT`.
- `STOP`: Todo lo que tenga la regla `STOP` no puede ser traspasado por ningún objeto. En otras palabras actúa como una barrera por la que no se puede mover ningún objeto.
- `PUSH`: Si un objeto es `PUSH`, este puede ser empujado para moverlo. Si un objeto es a la vez `PUSH` y `STOP`, prevalece el `STOP`.
- `SHUT` y `OPEN`: Si un objeto de tipo `SHUT` toca un objeto de tipo `OPEN`, ambos desaparecen (por ejemplo una llave y una puerta).
- `MELT` y `HOT`: Si Un objeto de tipo `MELT` toca a un objeto del tipo `HOT`, se derrite y queda solo el objeto de tipo `HOT`.
- `WIN`: Si un objeto de tipo `YOU` toca a un objeto de tipo `WIN`, el nivel está resuelto.
- `DEFEAT`: Si un objeto de tipo `YOU` toca un objeto de tipo `DEFEAT`, desaparece dejando solo el objeto de tipo `DEFEAT`. En el caso de que un objeto de tipo `YOU` toque a la vez un objeto de tipo `DEFEAT` y uno de tipo `WIN`, se prioriza el `DEFEAT` por lo que no se resuelve el nivel.

Un ejemplo de un nivel es el siguiente:

Figura 2: Nivel de ejemplo. La secuencia óptima de movimientos que lleva a ganar el nivel es `RIGHT, RIGHT, RIGHT, DOWN, RIGHT, UP, RIGHT, DOWN, DOWN, DOWN, DOWN, DOWN, RIGHT`.

El objetivo del jugador es encontrar una secuencia de movimientos que haga ganar el nivel. Este es un tipo de problemas común en computación llamado problema de búqueda en el espacio de estados.

Qué está hecho y qué hay que hacer

Como se mencionó antes ya está hecha parte del código y tu deber es optimizarlo. En esta sección se detalla cuales son las tareas a realizar y se explica qué está implementado y de que forma.

Modelación

Toda la lógica del juego incluidos los movimientos, interacciones y condiciones de victoria están implementados de manera eficiente en el código base de la tarea. En particular la parte de la modelación que te interesa está en el archivo `board/board.h`. En el se encuentran las estructuras de datos usadas y la descripción de las distintas funciones aplicadas a los tableros. En particular te interesa:

- `board_print`: Imprime el tablero dado. Útil para debuguear.
- `board_compare`: Compara 2 tableros y retorna True si son iguales.
- `board_destroy`: Libera la memoria del tablero dado.

El tablero está programado con una matriz que almacena listas ligadas que contienen los distintos elementos de las celdas. Esto permite tener varios elementos en la misma celda de un tablero. En el archivo `board/object.h` se encuentran los `enum` que definen los objetos y las propiedades que pueden tener.

BFS

BFS es un algoritmo que permite encontrar la secuencia de pasos óptima para resolver el problema. Para esto explora todos los posibles estados del problema desde los más cercanos al estado inicial hasta los más lejanos. Este algoritmo está ya implementado correctamente en el código base de la tarea para resolver los distintos niveles.

BFS usa dos estructuras de datos para su funcionamiento: Una cola donde se almacenan los tableros que no se han explorado y una tabla de hash que permite determinar si un estado ya fue descubierto. Esta tabla permite ahorrar operaciones descartando los tableros que ya fueron explorados por el algoritmo. La cola está implementada con una lista ligada, lo que permite que crezca indefinidamente y permite hacer sus operaciones en $O(1)$. Sin embargo la tabla de hash no está implementada de manera eficiente y es aquí donde debes meter la mano para optimizar el programa.

Interfaz gráfica

Para poder visualizar el problema y hacerlo interesante se creó una interfaz gráfica que puede mostrar los tableros. En el código base se usa para visualizar la solución del problema pero puedes usarla para visualizar los tableros que quieras. Esta interfaz utiliza la librería GTK que se puede instalar en Linux, OS y Windows Linux Subsystem. Para esto sigue las instrucciones disponibles en la wiki del curso en el siguiente link). Si tienes cualquier problema con la interfaz menciónalo en el siguiente ISSUE) y trataremos de solucionarlo lo antes posible.

La interfaz gráfica puede ser usada ejecutando el programa con el parámetro `-w` al final del comando. Esta mostrará los pasos ejecutados por el programa desde el estado inicial hasta el final. La interfaz se usa desde el código a través de 4 funciones:

- `watcher_open(int height, int width)`: Abre la ventana con las dimensiones del tablero.
- `watcher_draw_board(Board* board)`: Muestra el tablero dado en la ventana.
- `watcher_snapshot(char* filename)`: Guarda en la dirección dada un archivo .PNG con el contenido que hay en la interfaz actualmente.
- `watcher_close()`: Cierra la interfaz y libera todos los recursos que usa.

Tabla de hash

Primera parte: Implementación

Como se mencionó en la sección anterior, la tabla de hash implementada es ineficiente por lo que debe ser remplazada por una programada por ti. Esta tabla implementada se encuentra en `search/table.h` y `search/table.c` y es simplemente un arreglo con un tamaño inicial muy grande que se recorre completo para insertar o buscar los elementos. Para mejorar la calidad de la tabla de hash debes hacer lo siguiente:

- Implementa *Zobrist hashing* con números aleatorios de tipo `uint64_t`. Para esto se te da la función `get_random()` del archivo `random/random.h`.
- Crear una tabla de hash con **direccionamiento abierto** que almacene en cada celda un par (tablero, hash) donde el hash es un número calculado a partir del tablero usando la función de hash hecha por ti. Esta tabla debe tener un factor de carga máximo y una función `rehash` que la haga crecer cuando se llene. La tabla de hash será utilizada mediante las siguientes funciones:
  - `Table* table_init(uint8_t height, uint8_t width)`: Crea ua tabla de hash vacía que almacena tableros de dimensiones $height \times width$
  - `bool table_insert(Table* table, Board* board)`: Revisa si el tablero `board` ya existe en la tabla. Si existe, retorna `false` y sino, lo inserta y retorna `true`.
  - `void table_destroy(Table* table)`: Libera la memoria usada por la tabla y de todos los tableros almacenados en ella.

  Además se espera que tu tabla tenga una factor de carga máximo y llame a la función `rehash` cuando se pase este valor. Es importante que funcione tal como se acaba de describir ya que BFS usa las funciones de la tabla asumiendo que funcionan de esta manera.

Segunda parte: Experimentos

Ahora que tienes una tabla de hash funcional y con una buena función de hash queremos hacer ciertos experimentos para probar la calidad de esta. Para esto trabajarás con una mala función de hash `bad_hash` para así compararla con `Zobrist`. Para esto probarás los distintos parámetros de una tabla de hash usando bad_hash para ver si compensa la calidad de la función. Los experimentos se detallan a continuación:

1. **Factor de carga máximo:** Usando `bad_hash` en tu tabla prueba usando siguientes valores de factor de carga máximo: 10%, 30%, 50%, 70% y 90%. Para el resto de los experimento ocupa el mejor valor obtenido de factor de carga máximo.
2. **Redistribución de la función de hash:** Usando `bad_hash` como función de hash prueba el método de la división y el método de la multiplicación para transformar el valor de la función de hash a una posición en la tabla. Además prueba 2 de los 3 métodos de sondeo vistos en clases (lineal, cuadrático y hashing doble) y compara los resultados para las 4 combinaciones.

Evaluación

En esta tarea no habrá nota de código ya que lo más importante es lograr hacer un análisis de lo implementado. Sin embargo es necesario usar código propio para crear el informe por lo que igualmente se debe subir el código a tu repositorio asignado para la tarea.

Informe

En en el informe estarán todos los resultados y análisis de los experimentos anteriores. Para comparar los resultados de las configuraciones tienes que crear 2 tipos de gráficos:

1. **Distribución en tabla:** Debes mostrar como distribuyen las búsquedas en la tabla de hash, para esto debes crear un arreglo de contadores del tamaño de la tabla que partan en 1 y sumar 1 cada vez que revisas una celda. Si se hace un *rehash* de la tabla simplemente borra los contadores anteriores y cuenta las búsquedas en la tabla nueva. No es necesario hacer este gráfico para todos los tests en cada experimento, basta con un test por modalidad o valor en cada experimento. Este gráfico es fácil de generar en `Excel` una vez que tienes los contadores y queda como:

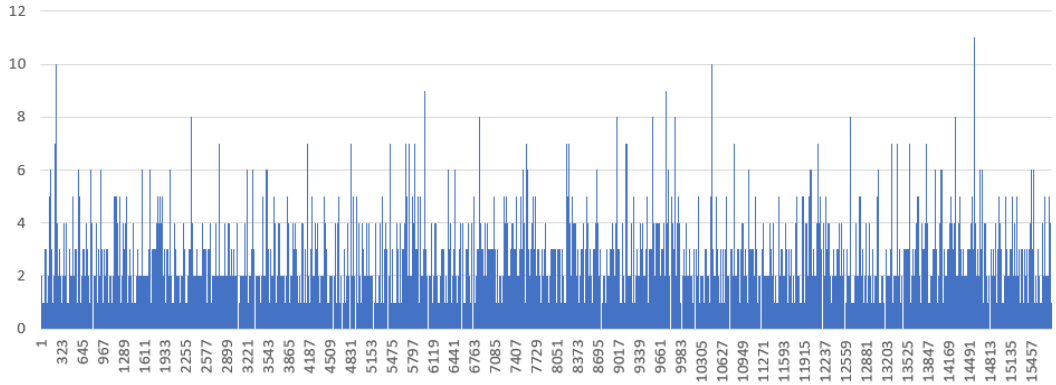

Figura 3: Histograma de accesos a celdas

2. **Tendencias en tests:** Debes mostrar como afectan los parámetros o métodos modificados en la experimentos graficando el tiempo que demoran en ejecutarse con respecto al número de tableros totales generados por el programa (el contador almacenado en la tabla al final del programa). Para esto debes medir los tiempos con una cantidad sustantiva de tests usando las distintas modalidades a probar. Un ejemplo del gráfico descrito se ve a continuación:

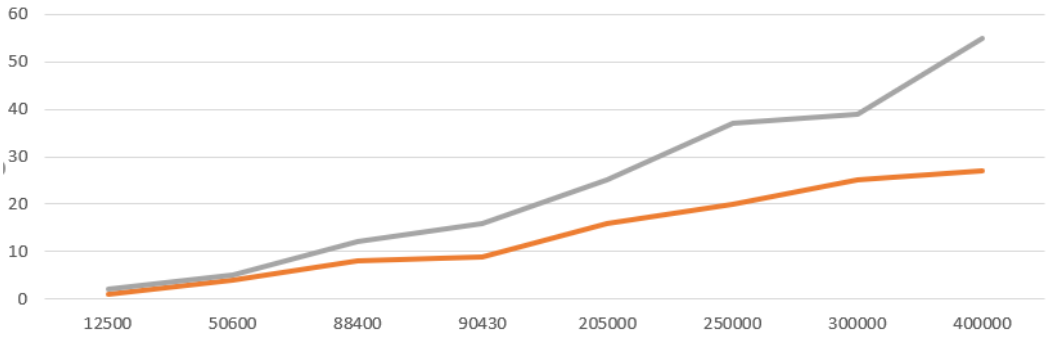

Figura 4: Gráfico de comparación entre dos métodos

   Es razonable usar un tiempo límite de 120 segundos para los tests para no alargar innecesariamente los experimentos.

En tu informe tiene que haber una sección por cada experimento y una sección para la versión con *Zobrist* y en cada sección debes crear ambos gráficos. Además de los gráficos debes explicar a partir de un análisis de los resultados qué valor o configuración es la mejor para la tabla y justificarlo. Finalmente compara la merjor versión de `bad_hash` con *Zobrist* y escribe tus conclusiones.

Bonus

Existen 3 bonus distintos en esta tarea, cada uno aumenta tu nota en un porcentaje. Puedes tener puntaje parcial en los bonus y no tener el porcentaje completo. Si los tienes todos juntos puedes tener una nota máxima de 8.05 en tu tarea. Los bonus se detallan a continuación:

- **Gráfico de llamadas** [7%]**:** Existe entre las opciones de valgrind una herramienta llamada `cachegrind`. Esta herramienta calcula el tiempo y el número de llamadas de cada función de un programa ejecutado con el comando `valgrind --tool=callgrind`. Estos datos pueden ser visualizados con el programa `kcachegrind` (se puede instalar en linux por consola usando `apt install kcachegrind`). El gráfico de llamadas se ve como:

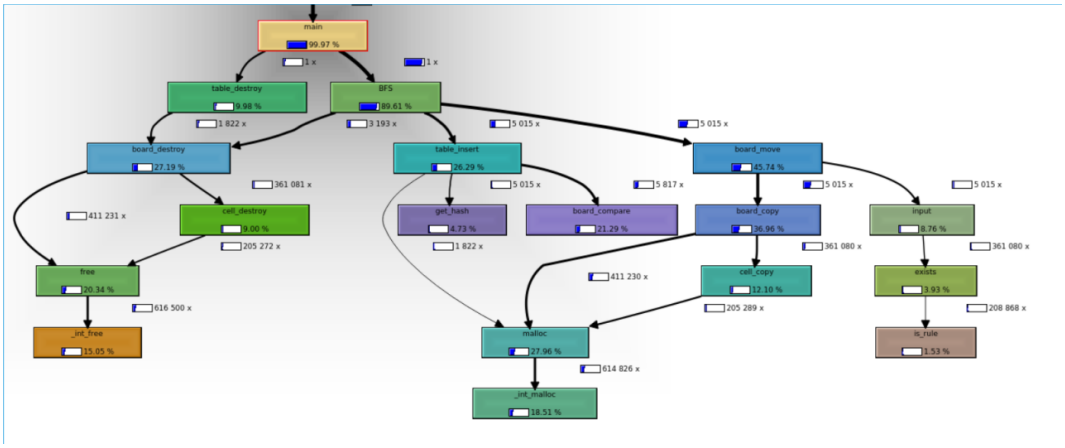

Figura 5: Gráfico de llamadas

  Esta herramienta es muy útil para ver qué partes de un programa son las que toman más tiempo y merecen la pena optimizar. Para obtener el bonus genera este gráfico por cada experimento de tu programa en cada una de las configuraciones con algún test a tu elección.

- **Orden y legibilidad** [4%]**:** Tendrás este bonus si tu informe se deja leer con facilidad y está en general ordenado. Este bonus queda a criterio personal del ayudante que corrige tu informe.

- **Crea tu propio test** [4%]**:** Crea un test interesante y súbelo a tu repositorio junto con el código en la carpeta tests. Además pon una imagen del test en el anexo del informe para saber que hiciste el bonus y revisar así tu test (te recomiendo usar la función `watcher_snapshot` de la interfaz para esto). Trata de que se pueda resolver en un tiempo razonable con BFS para poder probarlo. Para hacer un test debes crear un archivo con las reglas y otro con el tablero. Revisa el formato de los tests que ya vienen en el repositorio para que quede más claro. Los números correspondientes a los objetos están en el archivo `board/object.h`. Tendrás el bonus si tu test es creativo e interesante.

Entrega

- **Código:** GIT - Repositorio asignado. Si se crean varios repositorios cuando accedes al link para crear repositorios, trabaja en el que no tenga un número al final y borra el resto de los repositorios.

- **Informe:** SIDING - A través de un cuestionario.

Ambos se entregan a mas tardar el domingo 13 de octubre a las 23:59 hora de chile continental.