Laboratorio 1 de Programación 3

Alejandro Mujica

- **Lee completamente este documento antes de comenzar a trabajar o formular preguntas.**
- **Este trabajo es estrictamente individual.**
- **Esta asignación es evaluada y la calificación que obtengas cuenta para tu calificación definitiva.**
- **Fechas válidas de entrega: desde el 25/10/2018 hasta el 31/10/2018.**
- **El rango de fechas de entrega no será cambiado bajo ninguna circunstancia. Toma tus precauciones al respecto.**
- **Tu entrega consiste de un archivo llamado `solver.H`, Por favor, al inicio de este archivo, en comentario, pon tu número de cédula, nombre y apellido.**

1. Introducción

El fin de este laboratorio es desarrollar destrezas algorítmicas con matrices y listas.

Para solucionarlo debes estar familiarizado con las clases vector, list, pair, tuple y mt19937 de la Standard Template Library (STL) de `C++`.

1.1. El juego Buscaminas

El Buscaminas es un juego diseñado para un solo jugador el cual consiste en despejar un campo de minas sin detonar alguna.

Las reglas del juego son las siguientes:

1. El campo minado se modela con una rejilla regular de dos dimensiones comúnmente.
2. Algunas casillas de la matriz contienen minas y otras no.
3. Las minas están distribuidas uniformemente en el tablero. Es decir, cualquier casilla tiene la misma probabilidad de contener una mina.

4. Una casilla no minada, puede contener un número que indica cuántas minas tiene alrededor. Por ejemplo: Si tiene un número 3, entonces significa que de las 8 casillas que tiene alrededor[1], 3 de éstas contienen minas y las otras 5 están despejadas.

5. Si se descubre una casilla con mina, se pierde la partida.

6. Se puede colocar una marca en alguna casilla tapada de la cual se sospeche que esté minada.

7. Existen 4 modos clásicos de juego:

   - Nivel principiante: 8×8 casillas y 10 minas.
   - Nivel intermedio: 16×16 casillas y 40 minas.
   - Nivel experto: 16×30 casillas y 99 minas.
   - Nivel personalizado: Cualquier tamaño de rejilla y cualquier cantidad de minas.

1.2. Cómo se juega

Para resolver el tablero se deben observar las casillas para determinar cuáles tienen minas y cuáles no. Cuando se identifican casillas que contienen minas, entonces lo ideal es marcarla. Una vez que están marcadas es más fácil ver cuáles no tienen minas y luego se procede a destaparlas.

Existen estados del tablero para los cuales no es posible saber cuáles casillas están minadas y cuáles no. Comúnmente se elige al azar. A veces cuando se está comenzando a jugar se tiende a elegir la primera casilla tapada de izquierda a derecha y de arriba a abajo, hay quienes estilan elegir las esquinas en esos casos. En otras ocasiones se estila ver en las zonas donde puede que hayan menos minas según los números en las casillas destapadas.

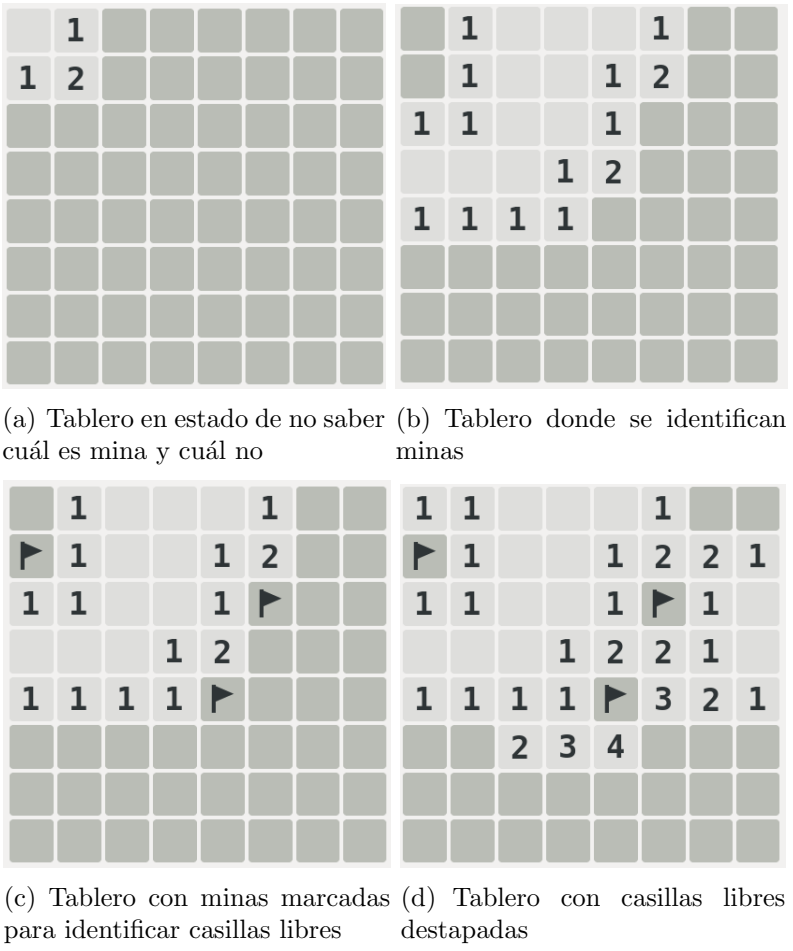
Considere la figura 1. En la figura 1-a se observa un tablero en un estado para el cual es imposible saber dónde están las minas. Pues en la casilla $(0, 1)$ tengo un número 1, lo cual me indica que esta casilla tiene una mina alrededor, pero tiene dos casillas tapadas y no hay forma de saber cuál tiene la mina. Análogamente ocurre con la casilla $(1, 0)$. Si vemos la casilla $(1, 1)$ sucede algo similar. Ésta nos indica que tiene 2 minas alrededor, sin embargo, tiene 5 casillas tapadas contiguas.

En la figura 1-b tenemos otro escenario. Si observamos la casilla $(0, 1)$ estamos en un caso similar al anterior; igualmente sucede con $(1, 1)$. Sin embargo, si observamos las casillas $(2, 0)$ o $(2, 1)$ (cualquiera de ellas basta) nos damos cuenta de que nos señalan una sola mina alrededor y solamente hay una casilla tapada alrededor, por lo que sabemos que necesariamente esa casilla es la mina. La misma lógica es aplicada para identificar las casillas $(2, 5)$ y $(4, 4)$ como minadas.

Una vez identificadas, las marco como minadas (figura 1-c) para poder identificar las casillas no minadas con más facilidad. Una vez que marcamos la casilla $(1, 0)$

[1]En caso de no ser un extremo o esquina. Las esquinas solamente tienen 3 casillas alrededor y extremos tienen solamente 5.

como minada, volvemos a mirar la casilla $(0, 1)$. Ésta indica que tiene una mina alrededor y tiene dos casillas tapadas alrededor. Pero cómo ya tenemos la certeza de que la única mina contigua está en $(1, 0)$, entonces concluimos que la casilla $(0, 0)$ no está minada y la podemos destapar con toda seguridad. El mismo razonamiento es aplicado en las otras marcas que se colocaron. La figura 1-d muestra el mismo tablero luego de haber destapado las casillas de las cuáles se está seguro y lo deja en un estado para el cual es necesario colocar nuevas marcas a casillas minadas.



(a) Tablero en estado de no saber cuál es mina y cuál no

(b) Tablero donde se identifican minas

(c) Tablero con minas marcadas para identificar casillas libres

(d) Tablero con casillas libres destapadas

Figura 1: Capturas del juego gráfico.

1.3. Representación del campo minado

En esta práctica representaremos un campo minado mediante una matriz. Esta matriz se encuentra en una clase llamada `Board` implementada en los archivos `board.H` y `board.C`.

Una casilla de la matriz contiene un booleano que indica si está minada o no. Además, cada casilla tiene un estado: Destapada, Tapada, Marcada. Estos estados se definen con un tipo enumerado como se muestra a continuación:

```
enum class StatusValue
```

```
{
  UNCOVERED, COVERED, FLAG
};
```

Cada casilla del tablero es referenciada mediante un par ordenado (i, j) en el cual i indica el número de fila y j el número de columa. Para un tablero de $n \times m$ (n filas y m columnas), las filas van desde 0 hasta $n - 1$ y las columnas desde 0 hasta $m - 1$.

1.4. Vista del tablero en pantalla

Un tablero se visualiza en la pantalla mediante caracteres ASCII. Se muestran en forma de tabla de la siguiente manera:

```
----------
| 112F3F*|
| 1F22F4?|
| 111223?|
|    1F2?|
|    12??|
|     2??|
|11   2??|
|F1   1??|
----------
```

El tablero visualizado tiene las siguientes características:

- Está configurado en modo principiante.
- Un signo de interrogación cerrando (?) indica una casilla tapada.
- Una letra F (mayúscula) indica una casilla marcada como minada.
- Un número del 1 al 8 indica la cantidad de minas que rodean la casilla destapada.
- El caracter espacio (‘ ’) indica una casilla destapada sin minas alrededor.
- El asterisco indica una casilla destapada con mina.

1.5. Representación en archivo texto

Los tableros pueden ser cargados desde un archivo de texto. Cada línea representa una fila de la matriz. En cada línea una casilla es representada por dos números consecutivos, el primer número indica el estado de la casilla y el segundo si está minado o no. El tablero visto anteriormente se representa en archivo como sigue:

```
0 0 0 0 0 0 0 0 2 1 0 0 2 1 0 1
0 0 0 0 2 1 0 0 0 0 2 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 2 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0
2 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0
```

1.6. Los archivos fuente

1.6.1. `board.H` y `board.C`

Contienen la definición e implementación de la clase `Board`, el tipo que modela un campo minado el cual vas a resolver.

Los tipos definidos en esta clase son los siguientes:

- ```
  enum class StatusValue
  {
    UNCOVERED, COVERED, FLAG, NUM_STATUS_VALUES
  };
  ```

  Estado de una casilla en el tablero (previamente mencionado).

- ```
  using Box = pair<StatusValue, bool>;
  ```

  Tipo que representa una casilla en la matriz, la cual tiene un estado y el indicador de si hay mina o no.

Las operaciones de esta clase son las siguientes:

1. ```
   Board(size_t h, size_t w, size_t m);
   ```

   Prepara un tablero de h filas, w columnas y m minas.

2. ```
   void generate_random(mt19937 &);
   ```

   Construye un tablero al azar sujeto a la cantidad de filas, columnas y minas que se hayan elegido en el constructor.

3. ```
   size_t rows();
   ```

   Retorna la cantidad de filas del tablero.

4. ```
   size_t cols();
   ```

   Retorna la cantidad de columnas del tablero.

5. ```
   size_t get_num_flags();
   ```

   Retorna la cantidad de marcas que se han colocado a casillas.
   ```

6. `size_t get_num_mines();`

   Retorna la cantidad total de minas en el tablero.

7. `bool is_uncovered_mine(size_t i, size_t j);`

   Indica si la casilla (i, j) está destapada y además contiene una mina.

8. `size_t get_num_mines_around(size_t i, size_t j);`

   Retorna la cantidad de minas que hay alrededor de una casilla. Esta operación solamente retornará un valor válido si la casilla está destapada. En caso de que no esté tapada retornará la constante `Board::UNKNOWN_VALUE`.

9. `StatusValue get_status(size_t i, size_t j);`

   Retorna el estado de la casilla (i, j).

10. `void flag(size_t i, size_t j);`

    Si la casilla (i, j) está tapada, le coloca la marca identificadora de mina. Si la casilla está marcada le quita la marca. Si está destapada, entonces la operación no tienen ningún efecto.

11. `void discover(size_t i, size_t j);`

    Destapa la casilla (i, j) si está tapada sin marca. Si la casilla destapada no tiene minas alrededor, destapa todas las casillas contiguas recursivamente.

12. `bool are_uncovered_all();`

    Indica si ya se destaparon (o no) todas las casillas que no cotienen mina.

13. `void save(ofstream &);`

    Escribe el tablero a archivo.

14. `void load(ifstream &);`

    Carga un tablero a archivo. Para usar esta operación, debes garantizar que el tablero esté construido con la cantidad de filas, columnas y minas que se encuentren en el archivo.

¡No modifiques estos archivos!

1.6.2. `solver.H`

Contiene las funciones que debes programar con el fin de completar la práctica. La explicación de cada función se encuentra en la siguiente sección. Además, se te provee un archivo main.C con un demo para visualizar tu solución.

2. Laboratorio

Para la solución del buscaminas, se requiere interacción entre 3 acciones básicas las cuales son: colocar marcas cuando se tiene seguridad de cuáles casillas están minadas, destapar las casillas para las cuales se tiene certeza de que están limpias y elegir una casilla cuando se encuentre en un estado para el cual no se puedan identificar casillas minadas o no minadas.

Todas estas rutinas se apoyan en otras que deberás programar. A continuación se coloca la interfaz de cada una de las rutinas y la debida explicación a cada una de éstas.

2.1. Posiciones tapadas alrededor de una casilla

Programa

```
PositionSet get_covered_positions_around(const Board & board,
 const Position & p)
```

La cual retorna el conjunto de casillas que rodean a (i, j) que estén tapadas sin marca.

2.2. Posiciones no destapadas alrededor de una casilla

Programa

```
PositionSet get_not_uncovered_positions_around(const Board & board,
            const Position & p)
```

La cual retorna el conjunto de casillas que rodean a (i, j) que estén tapadas o con marca.

2.3. Casillas destapadas últiles

Cómo se puede apreciar en la explicación de cómo se juega, nos concentramos en algunas casillas que estaban destapadas. Pero no era cualquier casilla la elegida, pues nos interesa que la casilla (o casillas) elegidas me ayuden a decidir sobre las que están tapadas. Definimos como una casilla destapada útil a aquella casilla que esté destapada y que, además, contenga al menos una casilla tapada (sin marca) adyacente.

Programa

```
PositionSet get_util_uncovered(const Board & board)
```

La cual debe retornar el conjunto de casillas útiles en todo el tablero.

2.4. Primera con menos minas

Cuando no se sabe cuáles casillas están minadas y cuáles no, es útil mirar aquellas casillas que tengan menos minas alrededor en comparación con la cantidad de casillas tapadas, pues mientras más casillas tapadas tenga alrededor y menos minas hayan alderedor, la probabilidad de que una de las casillas tenga mina disminuye. Por ejemplo, si una casilla marca 1 mina alrededor y tiene dos casillas tapadas adyacentes, la probabilidad de que cada una sea mina es de 0,5. Si la cantidad de casillas tapadas adyacentes es mayor, entonces la probabilidad disminuye.

Para una casilla $c = (i_c, j_c)$, sean $n = get\_num\_mines\_around(i_c, j_c)$ y $\mathcal{L}$ el conjunto de casillas adyacentes a c tapadas sin marca, entonces definimos $diff = |\mathcal{L}| - n$.

Programa

```
pair<bool, Position> get_first(const Board & board, size_t diff);
```

La cual debe retornar un par ordenado donde el segundo elemento es la primera posición (recorriendo el tablero de izquierda a derecha y de arriba a abajo) que cumpla con `diff`. El primer elemento del par ordenado es `true` si se consigue una posición que cumpla la condición. En caso de no conseguir ninguna, entonces debe retornar (`false`, `Position()`).

2.5. Todas las casillas tapadas en el tablero

Programa

```
PositionSet get_covered_positions(const Board & board);
```

La cual debe retornar un conjunto con todas las casillas tapadas sin marca en el tablero.

2.6. Selección de una casilla tapada al azar

Programa

```
pair<bool, Position> select_random_covered(const Board & board, mt19937 & g);
```

La cual debe retornar un par ordenado donde el segundo elemento es la posición de una casilla seleccionada al azar entre todas las casillas tapadas sin marca. El primer elemento del par ordenado es `true` si se consigue una posición. En caso de no conseguir ninguna, entonces debe retornar (`false`, `Position()`).

2.7. Seleccionar una posición

Cuando el tablero se quede en un estado en el cual no se pueda saber cuáles casillas están minadas y cuáles no, es necesario elegir una posición para destapar. Con el fin de no dejarle todo al azar y poder tener resultados reproducibles, sigamos la siguiente heurística para la elección de una casilla:

```
- Revise la esquina nor-oeste (0, 0), si está tapada sin marca,
 retórnela.
- Sino, entonces revise la esquina nor-este (0, m - 1), si está
 tapada sin marca, retórnela.
- Sino, entonces revise la esquina sur-oeste (n - 1, 0), si está
 tapada sin marca, retórnela.
- Sino, entonces revise la esquina sur-este (n - 1, m - 1), si está
 tapada sin marca, retórnela.
/* En este punto quiere decir que las esquinas están destapadas
o tienen marca.
*/
- Busque la primera casilla con diff = 4 (get_first).
- Si encontró una, entonces obtenga el conjunto de sus casillas
adyacentes y retorne una al azar.
- Si no la encontró, entonces busque una con diff = 3.
- Si encontró una, entonces obtenga el conjunto de sus casillas
adyacentes y retorne una al azar.
- Si no la encontró, entonces busque una con diff = 2
- Si encontró una, entonces obtenga el conjunto de sus casillas
adyacentes y retorne una al azar.
- Si llega hasta aquí, entonces retorne una casilla tapada
 seleccionada al azar.
```

Entonces programe:

```
pair<bool, Position> get_a_position(const Board & board, mt19937 g)
```

La cual debe instrumentar la heurística definida y retornar un par ordenado análogo a los de las subsecciones anteriores.

2.8. Obtención de casillas para marcar

Programa

```
PositionSet get_for_flags(const Board & board)
```

La cual debe retornar todas las casillas del tablero que deban ser marcadas en un momento dado por tener la seguridad de que tienen minas.

2.9. Obtención de casillas para destapar

Programa

```
PositionSet get_for_discover(const Board & board)
```

La cual debe retornar todas las casillas del tablero que deban ser destapadas en un momento dado por tener la seguridad de que no tienen minas.

2.10. Solucionar

Con lo anterior, ya tenemos la maquinaria suficiente para resolver un tablero (o no). Lo que falta ahora es utilizar todas las rutinas previas para programar la solución. Nótese que esta es la única rutina que recibe el tablero por referencia. Ésta será invocada pasando un tablero como parámetro y al finalizar éste debe estar en un estado de solución. Es decir, o están destapadas todas las casillas limpias y marcadas las minadas o el tablero explotó. Por lo cual debería quedar en el estado anterior al cual se descubrió la mina y la casilla que contiene la mina descubierta, debe estar destapada.

Programa

```
void solve(Board & board, mt19937 & g);
```

3. Evaluación

La fecha de entrega de este trabajo es desde el 25/10/2018 hasta el 31/10/2018.

Tienes permitido enviar tu práctica máximo una vez por día. Es decir, desde el día de inicio hasta el día final de la práctica tienes disponibles 5 intentos. Estos no son acumulativos. Si llegas a los 5 días de finalizar la práctica, cada día que no envíes será un intento que desperdiciarás. Entonces te quedarían solamente 4 intentos disponibles.

Para evaluarte debes enviar el archivo `solver.H` a la dirección:

`alejandro.j.mujic4@gmail.com`

El "subject" debe ser **exactamente** el texto **"PR3-LAB-01"** sin las comillas. Si fallas con el subject entonces probablemente tu trabajo no será evaluado, pues el mecanismo automatizado de filtrado no podrá detectar tu trabajo. No comprimas el archivo y no envíes nada adicional a este.

El único contenido que debe aparecer en el correo es tu número de cédula y tu nombre separados por espacio como se muestra en el siguiente ejemplo:

```
V01XXXXXXX Alejandro Mujica
```

Por favor, no uses otros medios para enviar la solución ni escribas otros comentarios adicionales en el email.

Atención: si tu programa no compila, entonces el evaluador no compila. Si una de tus rutinas se cae, entonces el evaluador se cae. Si una de tus rutinas cae en un lazo infinito o demora demasiado, entonces el evaluador cae en un lazo infinito o se demora demasiado. Por esa razón, en todos estos casos será imposible darte una nota, lo que simplemente se traduce en que tienes cero en el intento en el cual ocurra una de las circunstancias mencionadas.

No envíes programas que no compilan o se caen. Hacen perder los tiempos de red, de cpu, el tuyo y el mío. Si estás al tanto de que una rutina se te cae y no la puedes corregir, entonces trata de aislar la falla. Si no logras implementar una rutina, entonces haz que dé un valor de retorno el cual, aunque estará incorrecto y no se te evaluará, le permitirá al evaluador proseguir con otras rutinas. De este modo no tumbarás al evaluador y eventualmente éste podría darte nota para algunos casos (o todos si tienes suerte). Si no logras aislar una falla, entonces deja la rutina tal como te fue dada, pero asegúrate de que dé un valor de retorno. De este modo, otras rutinas podrán ser evaluadas.

La primera indicación dada en este documento fue que lo leyeras completamente antes de trabajar o formular preguntas. Cualquier falta a las normas de evaluación aquí mencionadas, serán consideradas como un "no leyó el documento", por lo tanto, no habrá manera de que tu práctica sea evaluada.

4. Recomendaciones

1. Haz un boceto de tu estructura de datos y cómo esperas utilizarla. Por cada rutina, plantea qué es lo que vas hacer y cómo vas a resolver el problema. Esta es una situación que amerita una estrategia de diseño y desarrollo.

2. La distribución de este trabajo contiene un pequeño test. No asumas que tu implementación es correcta por el hecho de pasar el test. Construye tus casos de prueba, verifica condiciones frontera y manejo de alta escala.

3. Este es un problema en el cual los refinamientos sucesivos son aconsejables. La recomendación general es que obtengas una correcta versión operativa lo más simplemente posible. Luego, si lo prefieres, puedes optar por mejorar el rendimiento.

4. Ten cuidado con el manejo de memoria. Asegúrate de no dejar "leaks" en caso de que utilices memoria dinámica. Todo `new` que hagas debe tener su `delete` en algún lugar. valgrind **y** DDD **son buenos amigos**.

5. Usa el grupo para plantear tus dudas de comprensión. Pero de ninguna manera compartas código, pues es considerado **plagio**. Tampoco hagas preguntas en privado, cualquier pregunta que tengas, es posible que otro también la tenga, si cada uno pregunta por separado, entonces tendré que dar varias veces la misma respuesta.

6. No incluyas headers en tu archivo `solver.H` (algo como `# include ...`), pues puedes hacer fallar la compilación. Si requieres un header especial, el cual piensas no estaría dentro del evaluador, exprésalo en el grupo para así incluirlo.