


ENUNCIADO DE TAREA 2.

	<p>Universidad de Costa Rica Escuela de Ciencias de la Computación e Informática Semestre I - 2021 Curso CI-0113 - Programación II Profesor: Edgar Casasola Murillo</p>
---	--

Tarea programada II

<https://git.ucr.ac.cr/ci0113/tarea2>

Forma de entrega: Grupos (3)	Fecha de entrega: Jueves 8 de junio - 11:55 p.m.
-------------------------------------	---

Forma de entrega: Subir a plataforma.ecci.ucr.ac.cr según se indica en el enlace respectivo. Solo tiene que subir la tarea 1 integrante de cada grupo. Deben entregar documentación interna y externa, en formato PDF.

Descripción del problema

Se realizará un Solucionador Genérico capaz de resolver problemas solucionables. Cada estudiante diseñará e integrará una implementación de Solucionador y una implementación de Problema original. Además, cada equipo deberá programar cualquier .cpp que consideren necesario para la funcionalidad de la aplicación. Para esta tarea deberán de utilizar como base los archivos aportados por el profesor.

Clases

Un **Problema** solucionable debe contar con:

- Un Estado inicial
- Un método para detectar cuando se ha llegado a la solución.
- Un método para calcular una Heurística: este método hará un estimado de qué tan cerca se está dado un estado actual de llegar al Estado meta. Mide la distancia entre el estado actual y el estado meta. No todos los problemas pueden tener esta medida por lo que en caso de ser este el caso, el método retornará 1 para indicar que es desconocido. Esta heurística puede ser utilizada por un solucionador para resolver el problema.
- Un método para pedirle a un estado que retorne la lista de estados posibles que se generan inmediatamente después de él
- Lista getSiguientes(Estado *) Retorna los siguientes estados

Por ejemplo, para problemas como el del 8-puzzle (**ESTO ES SOLO UN EJEMPLO**) (consiste en una matriz de números que deben ser ordenados, moviéndose solo si hay un espacio en blanco en su cercanía. Más información del ejemplo: https://en.wikipedia.org/wiki/15_puzzle). El método getSiguientes() será

que dado una matriz de números, devolverá una lista de estados con matrices que representan todos los estados a los que se llega después de realizar un movimiento válido.

Un **Estado** debe contar con:

- El Estado debe ser amigo del Problema específico. Dado que **el Problema es el que debe retornar con el método getSiguientes(...)** los siguientes estados posibles, no el Estado.
- Sobrecarga del operador de >>: este método permitirá que cada estado pueda ser cargado con sus respectivos datos.
- Sobrecarga del operador de <<: este método permitirá que se pueda imprimir un Estado.
- Sobrecarga del operador == : se utilizará para comparar Estados.
- Sobrecarga del operador !=

Recordar que cada problema sabe a cuál estado debe de llegar, por ende, el Estado utilizará esta información para indicar que el problema de cierto tipo fue resuelto.

Por último, se tendrá un **Solucionador** que recibe desde su **Problema** el conjunto de estados resultante del método que recibe un Estado y se encarga de expandir ese Estado y retornar una lista de estados siguientes utilizando el método (Lista * getSiguientes(...) que existen en todos los problemas)

y según el tipo de solucionador utilizará una técnica para decidir cuál de estos estados recibidos expandir. En otras palabras, decide cómo “explorarlos”.

Un **Solucionador** debe contar con:

- Un método solucione(Problema *) y retorna una instancia de Solucion

Utilización de Fábricas

Se utilizará un patrón del modelo fábrica abstracta y producto abstracto, donde una fábrica puede generar dos tipos de **Producto: Problemas y Solucionadores**. La fábrica será la clase padre de la cual deberán heredar las clases hijas según su función, además de indicar los métodos abstractos que cada hijo implementará posteriormente. Más específicamente, la fábrica deberá tener un método **producir()**, el cual es un método virtual puro, que devuelve un puntero a un Producto, en este caso puede ser un Solucionador o un Problema. La Fábrica tendrá un método para preguntarle si produce un “nombre de producto” particular (int produce(char * nombreProducto))

Se contará con un **Registro**, el cual contiene vector con una fábrica en cada posición. Cuando una fábrica es creada, se inscribe en una posición del registro. Puesto que cada fábrica identifica con el nombre de su **Producto**, el Registro se utiliza para comparar este nombre contra los nombres reconocidos por todas las fábricas inscritas en él,

para determinar cuál es la fábrica asociada a ese Producto. Al terminar el programa, el registro destruye las fábricas inscritas.

Main

El main fue diseñado y construido en clase y al igual que los .h abstractos **no podrá modificarse**. En este se creará el Estado (se escoge uno de la Fábrica de estados) y pasa al Solucionador escogido el problema.

Los parámetros de entrada son:

- Nombre del problema
- Nombre del solucionador

Documentación Interna

Debe venir en cada archivo .cpp . Para cada método, debe venir explicado en un comentario su: *función, parámetros, retorno*.

Documentación Externa

Debe iniciar con una copia del enunciado de la tarea para que se sepa que hace esa tarea. Expliquen cómo solucionaron el problema (diagramas de clase pueden ser útiles aquí). Hagan un análisis de su trabajo, cosas que se pueden mejorar, puntos donde el programa puede fallar para saber si ustedes lo detectaron antes y están conscientes del problema. Por ejemplo alguna validación de entrada de datos, que se acaba el archivo en forma abrupta etc.

Es conveniente capturar pantallas y mostrar casos de funcionamiento del programa, además de cómo se compila y ejecuta. Es como un manual para que alguien vea como funciona , y que efectivamente funciona. Para eso capturan pantallas y las van explicando paso a paso

La tarea debe entregarse en Mediación Virtual, no es suficiente con tener el código en Gitlab. Entregar una tarea por equipo.

Tarea Programada 2

Descripción breve: Este proyecto consiste en una serie de “problemas” y otra de “solucionadores”, con el objetivo de que cada solucionador pueda seguir una serie de pasos para llegar al estado meta de un problema, y que este se muestre al usuario. Se programaron los problemas de Ball Sort, Torres de Hanoi y Lights Out.

Autores:

- Fabián Orozco Chaves (B95690)
- Sofía Velásquez Shum (C08395)
- Carolina Zamora Nasrallah (C08633)

Compilación

Usando g++:

Utilizar el siguiente comando:

```
g++ -o solucionar *.cpp
```

Usando Makefile:

Utilizar el comando `make` o el comando `make solucionar` para compilar todos los archivos.

Ejecución

Utilizar el comando `./solucionar` seguido de los parámetros de entrada en orden: nombre del problema y nombre del solucionador. Ejemplo:

```
./solucionar Problema Solucionador
```

Para los problemas, los nombres registrados del equipo son:

- ProblemaFabian
- ProblemaSofia
- ProblemaCarolina

Para los solucionadores, los nombres registrados del equipo son:

- SolucionadorFabian
- SolucionadorSofia
- SolucionadorCarolina

Guía de uso

Problema Fabián (Bubble Sort):

La implementación del problema de Fabián trata un puzzle conocido como Ball Sort que tiene como objetivo clasificar las bolas de colores en los tubos hasta que todas las bolas del mismo color permanezcan en el mismo tubo.

En la implementación particular de esta tarea se utiliza una matriz de tamaño fijo 4x6, donde sus columnas representan los tubos/stacks y en cada fila se posa una bola

tratada como caracter que representa la inicial de cada color en inglés (Red,Green,Blue,Yellow) o un punto que denota que el espacio está vacío. Por ejemplo:

R		G		R		G		.		.
B		B		Y		R		.		.
G		R		B		G		.		.
B		Y		Y		Y		.		.

Posible estado inicial

Reglas:

- Máximo 4 bolas por stack simultáneamente.
- Solo se puede mover la bola superior de cada stack. Una a la vez.
- Solo puede ponerse sobre otra bola del mismo color o en un stack vacío.

.		.		R		G		.		.
B		B		Y		R		.		.
G		R		B		G		.		.
B		Y		Y		Y		R		G

.		B		R		G		.		.
.		B		Y		R		.		.
G		R		B		G		.		.
B		Y		Y		Y		R		G

Mueve de columna 1 a columna 2

Solución:

- El problema será solucionado cuando los mismos caracteres estén en una sola columna. Por ejemplo:

B		.		.		Y		R		G
B		.		.		Y		R		G
B		.		.		Y		R		G
B		.		.		Y		R		G

Posible solución.

Problema Carolina (Lights Out):

El puzzle de Lights Out consiste en una cuadrícula (en este caso de tamaño 4x4) en la cual cada celda puede estar “encendida” (contiene un 1) o “apagada” (contiene un 0), y el objetivo es lograr obtener una cuadrícula en la que todas las celdas están apagadas, por medio del movimiento llamado “switch”. El movimiento switch se aplica a una celda de la cuadrícula, y consiste en invertir el estado de la celda elegida, así como las adyacentes, como se muestra en la figura:

0	0	0	0		0	0	0	0
0	0	0	1	switch (2,3)	0	0	0	0
0	0	1	1	→	0	0	0	0
0	0	0	1		0	0	0	0

Note que la celda a la que se aplica el switch se indica con el número de fila seguido del número de columna, donde ambos se empiezan a contar desde 0.

Para cargar un estado inicial, **se utiliza un archivo con el nombre “GridLightsOut.txt”**, en el cual se escribe la matriz 4x4 con el formato mostrado en la figura anterior. Por ejemplo:

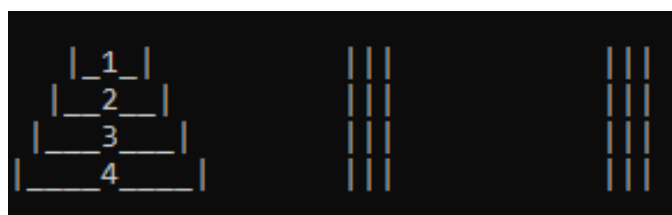
```
1 0 1 1
0 0 1 0
1 0 1 0
0 0 0 1
```

El programa primero intentará leer dicho archivo para solucionar el puzzle y, en caso de que el archivo no esté o tenga un formato equivocado, notifica al usuario y genera un puzzle aleatorio para resolver. Otro aspecto que hay que notar, es que en la solución se indica el movimiento que se realiza, seguido de la cuadrícula luego de aplicar dicho movimiento. Por ejemplo en consola se muestra lo siguiente en formato vertical (ordenado aquí horizontalmente):

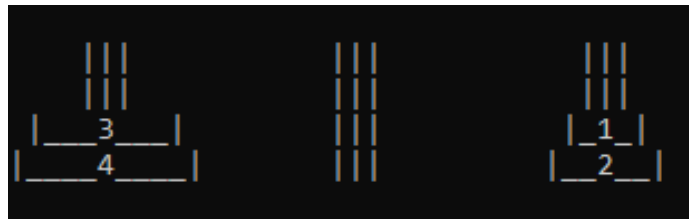
	Flip (1, 3):	Flip (2, 3):
0 0 0 1	0 0 0 0	0 0 0 0
0 0 1 0	0 0 0 1	0 0 0 0
0 0 1 0	0 0 1 1	0 0 0 0
0 0 0 1	0 0 0 1	0 0 0 0

Problema Sofía (Torres de Hanoi)

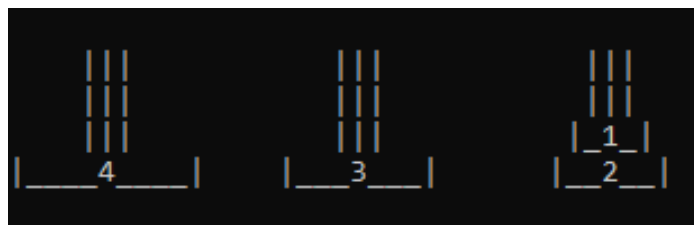
El problema Torres de Hanoi consiste en una serie de discos de diferentes tamaños que se pueden distribuir a través de 3 posiciones o ‘torres’. Estos pueden apilarse uno encima del otro únicamente si el disco de abajo tiene mayor magnitud del de arriba. El problema tiene una posición inicial constante con todos sus discos apilados en la primera torre:



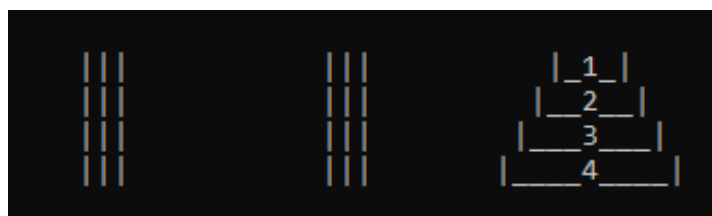
Para resolver el problema se puede mover un solo disco a la vez. Este tiene que estar en la parte de arriba de su torre. Por ejemplo, en el estado presentado abajo, los discos 'libres' (sin un disco encima de él) son los de magnitud 2 y 3.



En un movimiento se puede mover el disco a una sola torre diferente a su torre origen. El siguiente estado representa un movimiento válido desde el estado presentado arriba en el cual se mueve el disco de magnitud 3 de la primera torre a la segunda.



La solución de este problema consiste en ejecutar movimientos válidos hasta apilar todos los discos en la última torre. De modo que, al llegar al siguiente estado, se ha solucionado el problema.



Solución del problema (implementación de Solucionadores)

Solucionador de Fabián:

Este solucionador se apoya en una clase llamada **EstadoConRuta** que contiene un **estado derivado** y una lista nombrada **ruta** que representa el camino para llegar hasta él, es decir, los estados anteriores.

Su funcionamiento está basado en la búsqueda de la solución óptima del problema utilizando una lista denominada **frontera** en forma de cola donde se encuentran los **estados con ruta** que deben verificarse si son solución y además que faltan de expandirse para obtener los siguientes.

Explora por niveles, obteniendo los estados siguientes de cada estado, los verifica para saber si ya ha sido agregado anteriormente, esto lo hace mediante una lista de estados verificados denominada **visitados**, de no ser así, agrega uno por uno a la

lista de visitados y a la frontera para que sea evaluado (si es solución o no) y explorado futuramente.

Esa evaluación la realiza mediante un ciclo while que itera hasta encontrar una solución o hasta que todos los estados hayan sido explorados, en este caso, finaliza y la frontera quedará vacía por lo que retornará un 0.

Solucionador de Sofía:

El solucionador consiste en un método recursivo que, a partir de un estado dado, devuelve la variable(int) *cambiaRuta*. Esta determina si la serie de estados seguidos llega a la solución o a un estado no-solucionable. Una vez llamada la función, se tienen los tres siguientes casos:

- El estado en cuestión es la solución: *cambiarRuta* = 0 (termina el llamado recursivo).
- El estado ya ha sido visitado (está en la lista de visitados): *cambiarRuta* = 1.
- Cuando no es ninguno de los anteriores se ejecuta la siguiente serie de pasos:
 - Se agrega el estado al inicio de la lista de estados visitados
 - Se obtiene la lista de forma ordenada (por heurística) de los estados siguientes.
 - Se recorre la lista de siguientes dentro de un 'for loop' hasta que se obtenga *cambiarRuta* = 0 con el siguiente llamado recursivo:
 - *cambioRuta = funcionRecursiva(estadoSiguiente)*
- Al terminar cualquiera de las siguientes condiciones, si *cambioRuta* = 0, se agrega el estado al inicio de la lista de soluciones y devuelve la variable *cambioRuta*.

La serie de llamados recursivos se detiene cuando se encuentra la solución y devuelve 0 a los llamados anteriores, agregando la ruta de estados que produjeron el resultado a la lista de solución. Y, se devuelve (cambia de ruta) al utilizar otro estado de la lista de estados siguientes para continuar el llamado recursivo si el estado actual ya ha sido visitado al igual que todos sus siguientes (y ninguno es la solución).

Solucionador de Carolina:

Se soluciona el problema de forma recursiva para cada estado. El método *resolver* recibe un puntero a Problema, un puntero al Estado actual, un puntero a Lista que corresponde a la ruta para llegar a actual (incluye a actual) y una Lista (que funciona como un conjunto ya que no tiene repeticiones) de los Estados ya visitados.

En el método:

Se recibe el Estado actual e_k , la ruta $R=[e_0, \dots, e_k]$ y los visitados $V=\{e_0, e_1, \dots, e_k\}$. Luego el booleano *resuelto* corresponde a si e_k es una solución. Si no es una solución, se obtienen los Estados siguientes de e_k $S=\{e_{k+1}, \dots, e_{k+m}\}$ (en caso contrario, el método no hace nada más y retorna true inmediatamente). De S se

excluyen aquellos Estados en V, es decir, ahora S es $S \setminus V$. Ahora, para cada estado e_{k+i} en S (de forma iterativa) se realiza el procedimiento:

1. Se agrega al final de R (ahora $R=[e_0, \dots, e_k, e_{k+i}]$) y se agrega a V ($V=\{e_0, \dots, e_k, e_{k+i}\}$)
2. Se asigna a *resuelto* el retorno de *resolver*, esta vez con *actual* = e_{k+i} , y con R y V modificados
3. Si *resuelto* es falso, se elimina el último elemento de R ($R=[e_0, \dots, e_k]$), pues ninguna ruta que inicie con la secuencia e_0, \dots, e_k, e_{k+i} conduce a una solución, y se continúa con los demás Estados de S de la misma forma (con la salvedad de que, al haber llamado recursivamente el método, es posible que alguno de los siguientes Estados haya sido visitado previamente y no se deba analizar de nuevo, es decir, se analiza solo si el elemento no está ya en V)
4. Si *resuelto* es verdadero, significa que se ha llegado a la solución por medio de una serie de estados (que ahora están ordenados en R ya que se ha llamado el método de forma recursiva), por lo que R debe permanecer como está, y no continúa con los otros elementos de S.

Inicialmente, se añade el estado inicial e_0 a V y R, luego se llama a *resolver*. Si retorna false, se elimina e_0 de R y se muestra en consola e_0 (de lo contrario el usuario no necesariamente puede saber cuál es el estado inicial que no posee solución). Si retorna true, no altera R.

Finalmente, si la Lista de solución (correspondiente a R) está vacía, crea un puntero a Solucion nulo, en caso contrario apunta a una nueva instancia de Solucion creada a partir de la Lista R. Este es el puntero a Solucion que retorna el método *solucione*.

Posibles mejoras

- En el caso del puzzle Ball Sort, se puede hacer más escalable permitiendo un mayor número de bolas y de tubos con la implementación de una matriz más dinámica.
- Para el problema de Lights Out, también se podría hacer escalable de acuerdo con lo que introduzca el usuario, sin embargo se implementó con un tamaño fijo para hacer más sencillo su uso, y para hallar la solución más rápidamente ya que el propósito es probar los solucionadores (cuadrículas más grandes dificultan mucho más el determinar que no hay solución, por ejemplo).
- El *SolucionadorSofia* y el *SolucionadorCarolina* no proporcionan las soluciones óptimas, y su cantidad de pasos se incrementa (en comparación a la cantidad óptima) con la magnitud del problema. Una posible mejora se puede dar al mejorar las condiciones que agregan los estados a la solución de modo que se obtenga la menor cantidad de pasos posible.

Análisis del trabajo

Se trabajó de forma individual para el diseño e implementación de cada problema al igual que los solucionadores. Sin embargo, se atendieron dudas de cualquier miembro en el proceso de desarrollo de manera grupal.