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

IIC2133 — Estructuras de Datos y Algoritmos
2020 - 2

# Tarea 3

**Fecha de entrega código e informe:** Viernes 13 de Noviembre a las 23:59

## Objetivos

- Modelar un problema y encontrar una solución usando backtracking
- Analizar distintas heurísticas y podas en la resolución de un problema

## Introducción

Destrozado por un inimaginable sentimiento de inferioridad, y completamente humillado por inteligencias artificiales definitivamente más *cool* que él, Guido tuvo una crisis de identidad la semana pasada.

Digitales pensamientos pasaron por la mente del otrora ser biológico. Quizás Deep Blue algún día venció a RoboKasparov –Slimossum razonó– pero ya poco queda de ese anticuado ser. IBM Watson dejó la vida pública tras aquel escándalo mediático de *RoboGate*, y ya no hablamos de AlphaGo. El camino a la fama necesariamente pasaba por su habilidad para resolver juegos de mesa, y el mundo necesitaba una nueva estrella.

Inspirado por sus provincianos encuentros con campesinos espaciales y lunas cuánticas, Guido ideó un exuberante juego en el cual distintos observadores colapsan funciones de onda de variados cuerpos celestes. La idea parecía perfecta y digna del más profundo estudio ludológico. Sin embargo, tras una rápida búsqueda en *Spoogle*, sus más profundos miedos se revelaron. ¡Oh no!, el juego ya había sido creado.

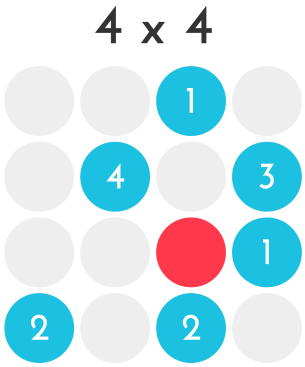
Decepcionado por sus continuos fracasos, el androide no tuvo más opción que la resignación. Quizás no pudo crear un juego perfecto, mas quedaba la opción de ser el mejor jugador de este. Guido no te ha contactado, está muy destrozado para eso. Sin embargo, aún puedes ayudarlo a construir sus rutinas de *backtracking*, convertirlo en un famoso jugador, y evitar una nueva crisis de identidad.

## Problema: 0h n0! I can't believe it's n0t trees!

El problema consiste en encontrar una solución válida a `0h n0!` [1], un juego de tablero de $n \times n$ con celdas azules y rojas.

Al comenzar el juego, algunas celdas ya están pintadas. Las celdas azules contienen un número, al que llamaremos `grado`: cada una debe poder ver un número `grado` de celdas azules. Una celda es visible por otra si se encuentran en la misma fila o columna, y no hay una celda roja entre ellas. Una celda no se ve a sí misma.

El objetivo del juego es pintar con el color correcto cada una de las celdas vacías con el fin de cumplir con las restricciones anteriores. Las celdas de la configuración inicial—tanto rojas como azules numeradas—están fijas y no pueden cambiar de color ni de `grado`.




Configuración inicial


Una posible solución


Cuantas celdas ve cada celda azul

Deberás escribir un programa en **C** que dado un tablero inicial, pueda encontrar una solución para el juego cumpliendo con las reglas anteriormente descritas. Para esta labor, se recomienda **fuertemente** la utilización de *backtracking*. A continuación se propone una modelación posible del problema, pero eres libre de utilizar otra a conveniencia.

## Modelación Propuesta

Con relación a las variables y sus dominios:

1. El color de cada celda vacía del tablero corresponde a una variable
2. Cada variable puede ser asignada de color azul o rojo

Por otra parte, las restricciones:

1. Cada celda del tablero debe tener un color asignado
2. Las celdas numeradas deben ver una cantidad de celdas azules igual a su $g$

[1] Se recomienda probar el juego para familiarizarse con el problema. Puedes encontrarlo en https://0hn0.com/ o descargarlo gratuitamente para plataformas *mobile*

# Ejecución

Tu programa se debe compilar con el comando `make` y debe generar un binario de nombre `ohno` que se ejecuta con el siguiente comando:

```
./ohno input output
```

Donde `input` es la ruta a un archivo con el tablero inicial, y `output` es la ruta al archivo de output donde se escribirá la solución. En esta ocasión, se les entregará el repositorio de la tarea que contará con el código de `I/O`, interfaz gráfica, `Makefile` y el archivo `main.c`.

## Input

El input está estructurado como sigue:

- Una linea que describe el ancho y alto $n$ del tablero
- $n$ líneas que describen el estado inicial de cada fila del tablero, en donde -1 indica que una celda es rojo y 0 que está vacío. Cualquier otro valor positivo $\alpha$ representa una celda azul de grado $\alpha$.

El input para el tablero ejemplo se vería de la siguiente manera:

```
4
0  0   1   0
0  4   0   3
0  0  -1   1
2  0   2   0
```

## Output

Representa el estado final del tablero:

```
-1  -1   1  -1
-1   4   3   3
-1   2  -1   1
 2   4   2  -1
```

### Código Base

Como equipo de ayudantes hemos preparado código para ayudarte con la visualización del problema, lo cual puede ser muy útil para verificar que tu algoritmo funcione adecuadamente. Este código se encuentra incluido como código base en tu repositorio. A continuación se detallan los módulos que podrás encontrar en la carpeta **src/**

- **src/visualizer_core/**: módulo a cargo de la visualización del problema. Utiliza GTK+3, al igual que el de la Tarea 1 [2]. Si no lograste hacerlo funcionar contacta a los ayudantes.
- **src/visualizer/**: API de **visualizer_core** para usarlo desde tu programa. Trae definido además un enum llamado `Color` que representa el color de una celda. Eres libre de usar este enum en tu solución.
  - `void visualizer_open(int height, int width);` abre una ventana para una matriz de las dimensiones especificadas.
  - `void visualizer_set_cell_color(int row, int col, Color color);` indica el color de la celda en la fila y columna especificada.
  - `void visualizer_set_cell_degree(int row, int col, int number);` indica el grado de la celda en la fila y columna especificada.
  - `void visualizer_set_cell_highlight(int row, int col, bool highlight);` activa o desactiva que la interfaz destaque la celda en la fila y columna especificada. Útil para confirmar que estás revisando las celdas que quieres revisar.
  - `void visualizer_redraw();` actualiza el contenido de la ventana volviendo a dibujar todo. Las demás funciones solo modifican su estado interno, **debes llamar esta función para que esos cambios se hagan efectivos en la ventana.**
  - `void visualizer_close();` cierra la ventana y sus recursos asociados.
- **src/0hn0/**: módulo correspondiente a tu programa. El código base trae la lectura del archivo y ejemplos de como visualizarlo usando la API de visualización.

## Análisis

Deberás escribir un informe de análisis del problema donde trates los siguientes puntos:

- Presentar una forma de modelar el problema sustancialmente distinta a la propuesta en el enunciado.
- Escoger una de estas dos modelaciones y proponer para ella dos mejoras: podas y/o heurísticas. Deberás describir la utilidad de cada mejora, así como justificar usando datos empíricos.

[2] Guía de instalación

# Evaluación

La nota de tu tarea se separa en 2 secciones: nota de código y de informe. A continuación se detalla cada uno de estas secciones.

- 70 % a la nota de tu código: que retorne el `output` correcto. Para esto, tu programa se ejecutará con archivos de input de creciente dificultad, los cuales deberán terminar en menos de 10 segundos. La solución a cada test pueden no ser única, por lo que sólo de comprobará que tu output cumpla con las restricciones del problema. Esta sección se subdescompone como sigue:
  - 50 % a los tests de dificultad `easy`
  - 25 % a los tests de dificultad `normal`
  - 25 % a los tests de dificultad `hard`

  Para los tests `normal` y `hard` se recomienda implementar una modelación eficiente con podas y/o heurísticas. Los tests `normal` y `hard` serán subidos el fin de semana de la fecha de publicación de la tarea.
- 30 % a la nota del informe, que debe contener tu análisis.

# Entrega

**Código:** Git – Repositorio asignado. Se revisará el último `commit` en `main` a más tardar el día de entrega a las 23:59, hora de Chile continental.

**Informe:** Siding – En el cuestionario correspondiente en formato PDF. Sigue las instrucciones dadas. El cuestionario se puede responder a más tardar el día de entrega a las 23:59, hora de Chile continental.

# Bonus

Los bonus solo aplican si la nota base correspondiente es superior o igual a 4.0.

## Manejo de memoria perfecto: +5 % a la nota de código

Recibirás este bonus si `valgrind` reporta que tu programa no tiene ni leaks ni errores de memoria en los tests que logres resolver.

## `Grand Gu1d0nament`: 5–10 décimas a la nota final

Se abre el `Grand Gu1d0nament`, donde podrás obtener gloria y fortuna [3]. *Todos* quienes completen el conjunto opcional de tests `Lunatic`—que pondrá a prueba a los más valientes y osados—obtendrán un ticket para participar, y 5 décimas como premio inicial.

Ya en competencia, se otorgará un bonus proporcional extra de hasta 5 décimas a la nota final de la tarea a las soluciones más rápidas. Para esto se utilizará una escala lineal desde el primer hasta el último lugar.

Alternativamente, aquellas soluciones que no logren entrar a la competencia obtendrán un bonus proporcional a la cantidad de tests `Lunatic` resueltos, de hasta 5 décimas.

[3] **fortuna** f. décimas