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

Tarea 2

IIC2133 - Estructuras de Datos y Algoritmos

Primer semestre, 2018

Entrega código: 12 de octubre
Entrega informe: 14 de octubre

Objetivos

- Modelar un problema de satisfacción de restricciones de manera formal.
- Implementar este modelo para resolverlo con backtracking.
- Proponer optimizaciones al modelo en forma de podas o heurísticas.

Introducción

Como alumno de ingeniería se te asigna en tu malla un ramo de ingeniería eléctrica en que se estudian las interacciones de imanes en dispositivos electrónicos. En el ramo tienes que hacer un laboratorio en el cual debes poner imanes en una caja rectangular cumpliendo ciertas restricciones con las cargas y las polaridades. Inmediatamente te das cuenta de que no es un problema sencillo y decides tratar de solucionarlo de manera computacional.

Problema: Imanes y cargas

Tienes una grilla de $N \times M$ cuyas celdas están conectadas de a pares (Figura 1). La idea es que insertes imanes en las celdas conectadas dejando el lado positivo en una y el lado negativo en la otra. Pero no es suficiente solo agregar los imanes, debes poner en las filas y columnas indicadas una cantidad específica de polos positivos y polos negativos. Además, si pones dos polos iguales en celdas aledañas, estos imanes saltan y no se quedan en la posición inicial.

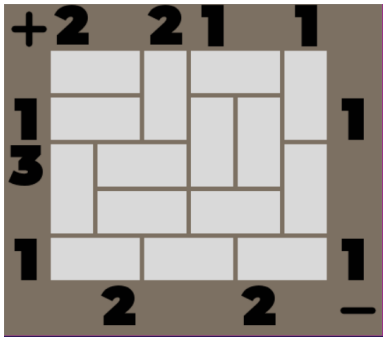


Figura 1: Grilla y sus restricciones

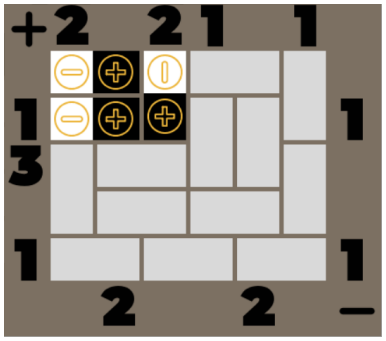


Figura 2: Imanes mal puestos

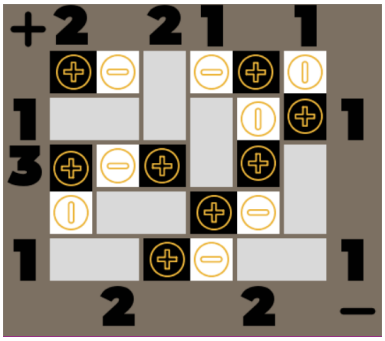


Figura 3: Problema resuelto

Asignación & Restricciones

Cada par de celdas puede tener un imán puesto en los 2 sentidos o puede quedar vacío. No hay un límite de imanes pero debes cumplir con las restricciones siguientes:

Vecinos

Las celdas conectadas deben tener alguna de las siguientes combinaciones:

- Positivo-Negativo
- Negativo-Positivo
- Sin carga-Sin carga

Carga

Algunas filas y columnas tienen un valor que indica el número de cargas positivas (arriba y a la izquierda) o negativas (abajo y a la derecha) que debe tener. Por ejemplo en el figura 1 la primera columna debe tener exactamente 2 cargas positivas. En esa misma columna no existe una restricción para el número de cargas negativas.

Polaridad

Nunca deben haber dos celdas aledañas con carga con el mismo signo. Esto solo aplica para vecinos en la dirección vertical y horizontal, y no aplica en las diagonales.

Solución

Tú debes implementar una solución escrita en C que resuelva este problema usando backtracking. Para esto es conveniente seguir la modelación de satisfacción de restricciones mostrada a continuación:

Un problema de satisfacción de restricciones se define como $P = (X, D, R)$, donde

- X es un conjunto de variables.
- D es un conjunto con dominios para las variables de X.
- R es un conjunto de restricciones aplicadas a un grupo de variables.

Por ejemplo en la figura 4 tenemos el conjunto de variables $X = \{X_{0,0}, X_{0,1}, X_{0,2}, X_{1,0}, X_{1,1}, X_{1,2}\}$, donde cada $X_{r,c}$ representa la celda en la posición $row = r, col = c$. El domino de cada variable corresponde al conjunto $D_{r,c} = \{-1, 0, 1\}$, donde -1 es un polo de carga negativa, 1 es de carga positiva, y 0 es sin carga. Finalmente las restricciones se pueden representar como funciones que retornan **True** o **False**. Por ejemplo la restricción que dice que las celdas $X_{0,0}$ y $X_{0,1}$ no pueden ser de la misma carga se puede expresar como:

$$R(X_{0,0}, X_{0,1}) = \begin{cases} false & \text{Si tienen cargas distintas a 0 iguales} \\ true & \text{En el caso contrario} \end{cases}$$

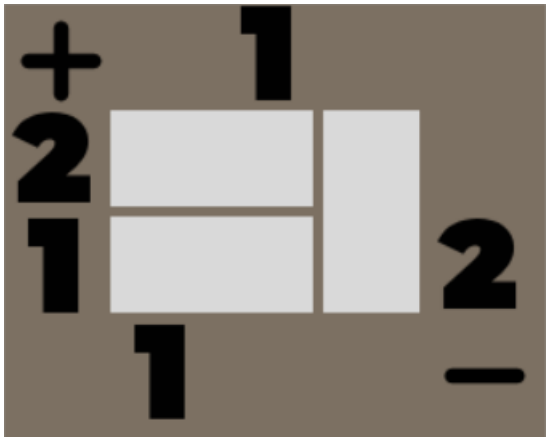


Figura 4: Ejemplo de modelación

Esta modelación formal permite escribir una solución del problema de la siguiente forma:

Algorithm 1 Backtracking

```
1: procedure BACKTRACK(X,D,R)
2:     if X = ∅ then return true
3:     x ← alguna variable de X
4:     for v ∈ D_x: do
5:         if x = v viola R then continue
6:         x ← v
7:         if Backtrack(X − x, D, R) then Return True
8:         x ← ∅
       return false
```

El algoritmo anterior permite resolver el problema si existe una solución, y retorna `false` si no existe una solución. El problema es que en el peor caso prueba todas las combinaciones de asignaciones de variables, es por esto que hay que tratar de hacerlo lo más eficiente posible. Para esto es altamente recomendable que las operaciones de las líneas 3, 5, 6 y 8 funcionen en $O(1)$ cada una. Haciendo esto tu tarea debería resolver todos los test en el tiempo dado de sobra sin necesidad de optimizaciones.

Optimizaciones

Si bien no es necesario implementar optimizaciones si hacemos nuestro algoritmo eficiente, es interesante analizar ciertos tipos de optimizaciones aplicables a backtracking.

Podas

Podar la ejecución de backtracking quiere decir descartar combinaciones que no llegarán a resolver el problema independiente de que no estén incumpliendo alguna restricción actualmente. Esto se puede hacer cuando alguna restricción R que valúa n variables no puede ser evaluada ya que algunas variables no tienen valor aún. Por ejemplo:

$$R(x_1, x_2, ..., x_n) = \begin{cases} false & \text{Si } \sum_{k=1}^{n} x_k > 10 \\ true & \text{En el caso contrario} \end{cases}$$

Asumiendo que todas las variables de x_1 a x_n toman tienen dominio $\{0, 1\}$. Puedo darme cuenta que no voy a cumplir la restricción cuando ya tengo evaluadas algunas variables a pesar de que no se el valor de todas. En este caso puedo considerar este estado de la ejecución como irresoluble y deshacer la última ejecución.
Sabiendo esto, no es difícil pensar en podas que apliquen a las distintas restricciones del problema.

Heurísticas

Una heurística es un método para determinar la mejor opción al momento de tener una decisión. Específicamente en backtracking una heurística se puede aplicar al elegir la variable de cada iteración (línea 3 del algoritmo) o al elegir el valor dado a la variable (línea 4 del algoritmo). Por ejemplo, yo puedo elegir primero las variables cuyo dominio está más acotado, ya sea porque su dominio estaba incialmente acotado o porque las restricciones lo acotaron mucho. De esta manera evito probar variables en las cuales se está adivinando mucho. Un ejemplo clásico de esto es el de decidir qué celdas rellenar al resolver un Sudoku. Por otro lado al elegir la celda a probar, podría ser más inteligente probar primero con algunos números que con otros determinando el estado del tablero.
A diferencia de las podas, las heurísticas muchas veces no aseguran que el algoritmo funcione más rápido en todos los casos ya que toman en cuenta las probabilidades de encontrar una solución.

Propagación

La propagación consiste en hacer varias operaciones (asignaciones de variables) en un paso, y en el caso de deshacer el paso, desasignar todas las variables asignadas con la propagación. Volviendo al caso del Sudoku, hay muchos casos en que agregar un número en una celda implica que otras celdas solo tengan una asignación posible, por lo que vale la pena asignarlas inmediatamente.

Informe y análisis

El informe se divide en 2 secciones:

1. **Análisis:** Debes mostrar a través de gráficos la complejidad de los distintos test con respecto a su tamaño. Específicamente debes hacer un gráfico donde el eje x corresponde a las dimensiones de la matriz $(N \cdot M)$ y el eje y corresponde al tiempo total que tomó tu programa en resolverlo. También debes crear otro gráfico con el mismo eje x, pero muestre en el eje y la cantidad de veces que se deshace una asignación (cantidad de veces que se ejecuta la línea 8 del algoritmo).
   Junto con los gráficos debes comentar la complejidad que crees que tiene el algoritmo viendo los datos.

2. **Optimizaciones:** Debes proponer una poda, una heurística y una manera de propagar en este problema explicando brevemente la forma de implementarlo y por qué se espera que esto mejore la eficiencia del algoritmo.

Input

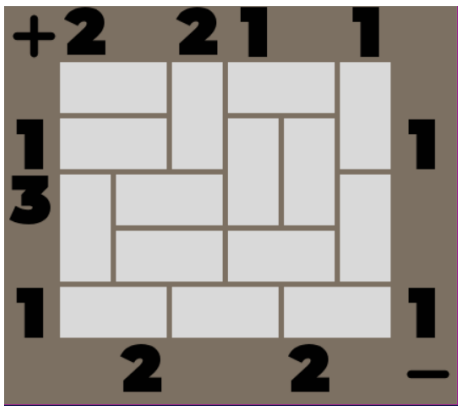
Tu programa debe recibir como input el path a un archivo de test el cual tiene el siguiente formato:

La primera línea tiene dos números enteros $w = width$ y $h = height$. La segunda y tercer línea contienen w números enteros, los cuales representan las restricciones de carga de las columnas. La cuarta y quinta línea contienen h números enteros cada una y representan las restricciones de carga de cada fila. Para todas las restricciones de carga $r = 0$ significa que no existe una restricción y $r > 0$ significa que deben haber r celdas en la fila o columna correspondiente con la carga correspondiente. Finalmente las últimas h líneas tienen w caracteres cada una mostrando el layout del tablero: $T = top$ corresponde a una celda que está conectada con su celda de abajo, $B = bottom$ corresponde a una celda que está conectada con su celda de arriba, $L = left$ y $R = right$ siguen la misma regla.

Un ejemplo de test es el siguiente:

```
6 5
2 0 2 1 0 1
0 2 0 0 2 0
0 1 3 0 1
0 1 0 0 1
L R T L R T
L R B T T B
T L R B B T
B L R L R B
L R L R L R
```

Este test representa al siguiente tablero:



Output e interfaz

Se provee una interfaz gráfica para resolver tu tarea, la cual puede ser utilizada con funciones hechas por los ayudantes. Esta interfaz tiene la finalidad de ayudarte a debuguear tu código de manera visual y además es el medio para dar el output de cada test.

El funcionamiento de debugueo de la interfaz funciona a través de operaciones que insertan y sacan imanes de las distintas celdas, pero la velocidad del programa hace que sea necesario agregar `sleeps` al código para ver cada uno de los pasos.

Al momento de finalizar tu programa la interfaz debe mostrar el estado resuelto del puzle. Ojo que debes eliminar toda línea de código que detenga al programa (i.e. `sleeps` o `getchars`) ya que el código es corregido de manera automática y debes terminar en un tiempo limitado.

Las funciones disponibles para usar con la interfaz son las siguientes:

- `watcher_load_layout(char* filename)`: Abre el archivo en la dirección `filename` y abre la ventana con el layout inicial del tablero.
- `watcher_set_magnet(int row, int col, bool vertical, bool positive)`: Recibe la posición de una celda de tipo *top* o de tipo $left$ e inserta un imán. Si el parámetro `vertical` es **true**, lo inserta de manera vertical, y si es **false**, lo inserta horizontal. El parámetro `positive` indica si la celda indicada es positiva o negativa.
- `watcher_clear_magnet(int row, int col, bool vertical)`: Recibe la posición de una celda de tipo *top* o $left$ y elimina el imán de la celda y su celda vecina (si no hay un iman, no hace nada). El parámetro `vertical` al igual que antes indica si se elimina un iman vertical u orizontal.
- `watcher_close()`: Cierra la ventana y libera los recursos utilizados por la interfaz. Es necesario que se llame a esta función al final de la ejecución del programa para poder corregir el estado final.

Si bien las funciones `watcher_clear_magnet(int row, int col, bool vertical)` y `watcher_set_magnet(int row, int col, bool vertical, bool positive)` deben recibir la posición de una celda de tipo *top* o $left$, la interfaz no va restringir que le entregues posiciones inválidas. Debes evitar usar mal la interfaz para que se pueda corregir correctamente. En el código base hay ejemplos para utilizar las distintas funciones de la interfaz.

No debes modificar nada del código fuera de la carpeta `Programa/src/solver` ya que puedes hacer que la corrección automática de tu código no funcione.

La interfaz gráfica funciona con la librería `GTK3`. Para hacer funcionar esta librería en tu computador puedes seguir la guía de `GTK3` disponible acá.

Evaluación

La nota de tu tarea estará descompuesta en dos partes:

- 50 % corresponde a que tu código pase los tests dentro del tiempo máximo.
- 50 % corresponde a la nota de tu informe.

Se probará tu programa con distintos test de dificultad creciente, los cuales deben ser resueltos en menos de 10 segundos. Pasado ese tiempo el programa será terminado y se asignarán 0 puntos en ese test. Todos los test tienen al menos una solución.

Entrega

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

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

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

Estar reglas ayudan a recolectar y corregir las tareas de forma automática, por lo que su incumplimiento implica un descuento en tu nota.

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

Bonus

Gotta go fast (+30 décimas a tu nota de código)

Se te asignarán 30 décimas a tu nota de código si este es capaz de resolver los tests de dificultad `Lunatic`[1]. Estos son tests superdifíciles, que requerirán que incluyas mejoras a tu solución. Utiliza podas, propagación y heurísticas de manera eficiente para poder resolverlos tan rápido como puedas. El bonus se te asignará según cuantos de estos tests consigas resolver.

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.

[1] Serán subidos en los próximos días