Universidad Diego Portales

Facultad de Ingeniería

Tarea 1

Algoritmos Exactos y Metaheurística

Autores:
Pablo Moraga
Diego Vergara

Abril 2023

Índice

1. Introducción

En esta primera tarea del curso, se solicita la creación de un programa que solucione un problema, esto transformando el problema en un CSP y luego implementando dos algoritmos, uno de Forward-Checking y uno de Forward-Checking con heurística de variable. Con el objetivo de compararlos y determinar el más rápido y el porqué de su rapidez a partir del árbol de búsqueda y el tiempo de CPU.

2. Especificaciones

Para el desarrollo y evaluación de esta tarea se utilizaron dos computadores. Los cuales tienen las siguientes especificaciones:

- Computador 1
  - CPU: Ryzen 5, 4° Gen
  - RAM: 8 GB, 3200 Hz
- Computador 2
  - CPU: Intel Core i5-9300H
  - RAM: 12 GB, 2400 Hz

3. El problema

Se presenta el problema como una matriz de tamaño 10x10.

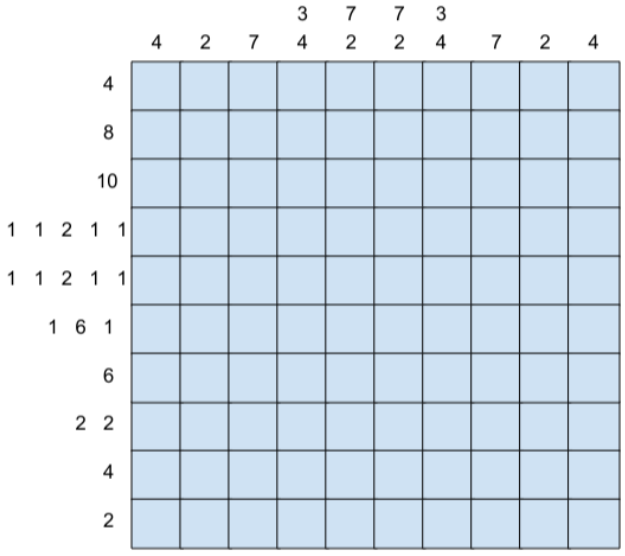

Figura 1: Matriz a rellenar

Las reglas del problema indican:

- La cantidad x señalada por el número de la fila/columna indica que se deben rellenar x ■ consecutivos.
- Si existen 2 o más números, además de la regla anterior deberá existir a lo menos un espacio libre entre ellos.

Para representar este problema de forma que sea posible cambiar las restricciones de este sin tener que modificar el código, se optó por escribir las cantidades en un archivo txt donde la primera linea sean las cantidades de las filas y la segunda linea, las cantidades de las columnas.

Considerando además que puede existir más de un número en las cantidades, se utilizó un para separar entre filas y columnas, y una ","para separar entre cantidades de una misma fila/columna. Con este formato el problema se representaría como:

```
4 2 7 3,4 7,2 7,2 3,4 7 2 4
4 8 10 1,1,2,1,1 1,1,2,1,1 1,6,1 6 2,2 4 2
```

4. Formulación del Problema

Para formular el problema como un CSP, primero se deberán definir las variables y sus restricciones, en este caso se tomaron las variables:

$$X_{ij}, i, j \in 1, 2, ..., N / X_{ij} = \{0, 1\}$$

Donde X_i es un cuadrado de la matriz y N es el largo de la matriz.

Luego se tienen las restricciones, que se obtienen a partir de los x de cada fila y columna. Esto se puede definir como:

$$\sum_{i=1}^{N} X_{ij} = \sum_{i=1}^{N} R_i, j \in 1, 2, ..., N$$

$$\sum_{j=1}^{N} X_{ij} = \sum_{j=1}^{N} C_j, i \in 1, 2, ..., N$$

Donde R_{ij} se refiere a las restricciones de filas y C_{ij} se refiere a las restricciones de columnas.

5. La solución

Para resolver el problema se propuso la creación de un script en Python, al que se le entrega un archivo txt con las filas y columnas del problema y busca todas las soluciones de este.

Para esto se tienen diversos archivos, optimizados y no optimizados.

5.1. Archivos no optimizados

- Limpio.py

  1. Se hace la lectura del archivo que contiene los valores de las columnas y se guardan en arreglos de filas y columnas.
  2. Se hace una matriz de tamaño n de valores 0 que representa que no están pintados.
  3. Se obtienen las posiciones posibles a tomar como el dominio posible, se almacena como coordenada [fila, columna].
  4. Se crea un nodo por cada posición posible, luego se hace un forward-checking utilizando la clase creada para esto, entregándole como parámetro cada nodo.
  5. Finalmente, imprime las soluciones encontradas con su determinado tiempo de ejecución.

- forwardchecking.py

1. Se crea la clase Nodo con sus variables y posibles hijos para hacer el árbol.
2. Se crea una función `forwardchecking`, encargada de
   a) Tomar todos los valores del nodo en el que se encuentra.
   b) Evaluar si está en una posición sin resolver, y si la fila anterior fue resuelta continua, en caso contrario corta la rama y salta a otro nodo.
   c) Pinta el cuadro correspondiente con un 1 y llama a una función para revisar si existen inconsistencias con las restricciones.
   d) Revisa si encontró una solución iterando sobre filas y columnas, en caso de encontrarla guarda esta y corta la rama para saltar a otro nodo. En caso contrario crea un nodo con los posibles valores para el dominio y llama a `forwardchecking` recursivamente.

- Dominio.py

1. Crea la función `Inconsistencias`, encargada de:
   a) Suma el valor de la fila y ve si supera el valor máximo de cuadros pintados que puede tener, en caso de superarlo corta la rama y continua, en caso contrario continua con normalidad.
   b) Luego hace un conteo de las secuencias y compara si se superó el valor de la fila, en cuyo caso corta la rama.
   c) Bloquea valores del dominio que no se puedan tomar a futuro.
   d) Revisa si tomando ese valor queda resuelta la fila y columna para marcar como resuelta la posición actual.

5.2. Archivos optimizados/Técnica de selección de variables

- Recalculado.py

1. En lugar de comenzar de forma secuencial, se toma la fila o columna con mayor valor.
2. Se crea una variable global `row_order`, que contiene la suma de los valores de las filas, donde el índice será el índice de la fila.
3. Utilizando `row_order` se crea una variable global llamada `row_pos` que indicará en qué orden iterar sobre las filas.
4. Se utiliza la función `next_pos` para comenzar a iterar y se crea el nodo para utilizar la función `forwardchecking`

- forwardchecking.py

1. En la versión optimizada, esta utiliza una función llamada `Pre_Inconsistencias` para evaluar de distinta forma las filas y columnas, además de no revisarlas de forma secuencial.

2. Además de crear una nueva rama, se creará una nueva rama con `next_pos`.

- Dominio.py

1. Se crea la función `Pre_Inconsistencias`, encargada de:
   a) Evaluar las inconsistencias futuras de la posición actual.
   b) Comprueba si la fila o columna actual fue resuelta y la marca de acuerdo a su estado.
2. Se crea la función `Next_pos`, encargada de:
   a) Ver la siguiente posición de la línea actual

Al finalizar los códigos se mostrarán todas las soluciones encontradas y con el tiempo total, aparte cada vez que encuentre una solución mostrara el tiempo transcurrido de la ejecución que demoró en encontrarlo.

6. Analisis

Si bien el algoritmo forward checking ayuda a buscar las soluciones de forma mas rápida dado que va viendo las futuras restricciones y bloqueando posibles valores futuros a tomar.
También se puede mejorar mucho mas el tiempo de búsqueda con la técnica de selección de variables dado que las primeras ramas ramas del árbol son las que tienden a fallar mas rápido o ser la solución, esto optimiza mucho el tiempo de búsqueda de una solución o de no solución dado que se cortan rápidamente las ramas que no son solución, se puede ver en el siguiente ejemplo de la tarea:

1. Limpio.py

```
Resuelto - Time =  6.6435980796813965  segundos
--------Soluciones--------
[0, 0, 0, 1, 1, 1, 1, 0, 0, 0]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 0, 1, 0, 1, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 1, 0, 1, 0, 1]
[1, 0, 1, 1, 1, 1, 1, 1, 0, 1]
[0, 0, 1, 1, 1, 1, 1, 1, 0, 0]
[0, 0, 1, 1, 0, 0, 1, 1, 0, 0]
[0, 0, 0, 1, 1, 1, 1, 0, 0, 0]
[0, 0, 0, 0, 1, 1, 0, 0, 0, 0]
-----------------------------
Termino de ejecucion:  12.20419955253601  segundos
```

Figura 2: Solución secuencial

2. Recalculado.py

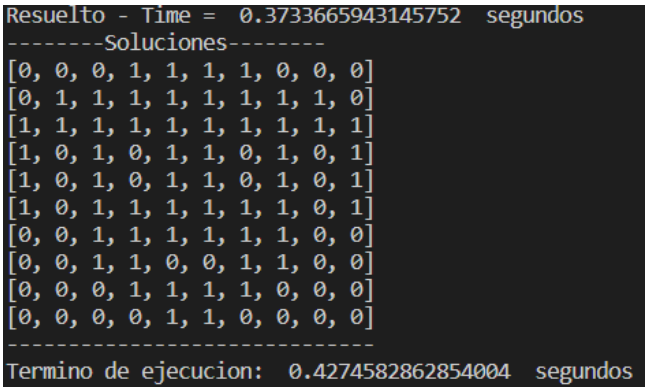

Figura 3: Solución con selección de variables

Cabe recalcar que los valores 0 son los puntos del puzzle **no pintados** y los valores 1 son los puntos del puzzle **pintados**.
Hay una gran diferencia en tiempos, pero también esta el caso de otro puzzle2.txt en donde solo encuentra la solución el algoritmo de selección de variables dado que el otro demora tanto tiempo que termina llenando la memoria RAM como para continuar después de los 15 minutos de búsqueda, en este caso el limitante es la memoria RAM.

```
2 3 2,4 8 10 3,3 1,5 1,6 1,1,3 3,2
2,3 4,1 4,1 4,1 3,3 7 8 9 1,3 1,3
```

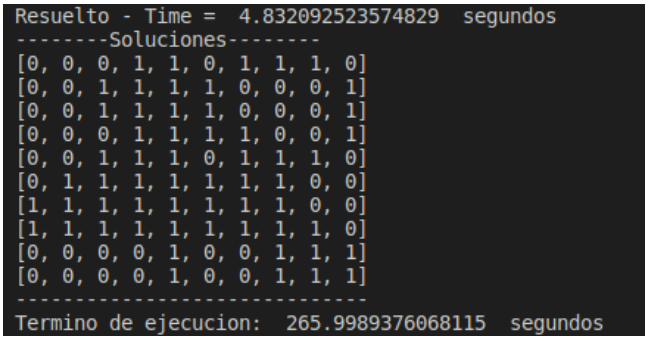

Figura 4: Solución del puzzle dado