Instituto Politécnico Nacional

Unidad Profesional Interdisciplinaria de Ingeniería

Campus Zacatecas.

Reporte de Proyecto.

"Algoritmo de las N reinas".

Materia y Grupo:

Análisis y Diseño de Algoritmos. 3CM2.

Profesora a cargo:

Erika Paloma Sánchez-Femat.

Alumno:

Jorge Ulises Zapata Arteaga.

1 Introducción.

En este reporte se analizará y recopilará información recabada al momento de realizar la práctica, donde se plasmarán los pasos a seguir para realizar las especificaciones requeridas, es decir, las estrategias para hacer más eficiente el algoritmo. Esto implica que se verán 3 códigos con diferente enfoque; se verá el algoritmo normal de las n-reinas, con estrategia de PODA y estratégia de Heurísticas. Además se analizará la complejidad de cada estrategia y una comparación con gráficas.

2 Desarrollo.

- **2.3. Análisis de Complejidad.**

Complejidad de la implementación inicial.

En el algoritmo se utiliza una función llamada *"encontrarSoluciones"* para encontrar todas las soluciones posibles. Comienza colocando una reina en la primera fila y luego intenta colocar una reina en cada columna de la fila actual. Si una posición es válida, se coloca una reina y se llama recursivamente a la función para la siguiente fila. Si se ha colocado una reina en cada fila, se ha encontrado una solución. De esto podemos concluir que la complejidad temporal del algoritmo es de O(N!) en el peor de los casos. Esto significa que el tiempo de

ejecución del algoritmo aumenta de manera factorial a medida que aumenta el tamaño del tablero. La razón por la cual la complejidad es de O(N!) se debe a que, en cada fila, hay N opciones para colocar una reina. En la siguiente fila, hay N-1 opciones, y así sucesivamente. Esto resulta en un total de N * (N-1) * (N-2) * ... * 1 = N! posibles combinaciones de reinas en el tablero.
Como podemos ver, la eficiencia de esta implementación no es muy favorable, ya que para valores de N>10 max 12, el tiempo de ejecución del algoritmo se vuelve muy grande; se puede implementar, pero debido a la naturaleza de la complejidad, las soluciones serían cada vez mayores y eso relentizaría mucho el proceso, conllevando a un aumento de recursos de memoria, lo que implica menos eficiencia.

Complejidad de la implementación con Poda.

En esta implementación, la complejidad está definida principalmente por O(N!), ya que en el peor de los casos hay N posiciones para la primera reina, N-1 para la segunda, N-2 para la tercera y así sucesivamente; sin embargo, la poda por factiblidad nos permite reducir bastante las posibilidades de colocación de reinas en general. Obviamente, esto no quita el hecho de que sigue siendo un algoritmo un poco más eficiente que la implementación anterior, con una mejoría considerable.
Considerando también la creación del tablero, podemos

decir que la complejidad espacial es O(N^2) debido a la creación de una matriz NxN, sin olvidar que la llamada de la funcion *encontrarSolucionesconPoda* es de complejidad O(N), dado que tiene que hacer N evaluaciones (una evaluación por fila del tablero). Con esto podemos concluir que la complejidad temporal es O(N!) y la complejidad espacial es O(N^2).

Complejidad de la implementación con Heurísticas.

Dada la representación del tablero, este ocupa espacio proporcional a N^2, ya que es una matriz NxN.
Por otro lado, los arreglos *tiempos y solucionesEncontradas* almacenan información para generar la gráfica. En el peor caso, ambas listas podrían llegar a contener 100 elementos, y por lo tanto, la complejidad espacial es O(100).
Por esto podemos concluir que la complejidad espacial es de O(N^2).
En este caso, podemos decir que esta estrategia puede ser mucho más eficiente cuando se trate de encontrar cierto número de soluciones en el menor tiempo posible, debido a su naturaleza de posicionamiento aleatorio.

- **2.4.1 Implementación incial del Algoritmo.**

Paso 1.- Para iniciar con esta implementación del código

inicial de las n-reinas, definimos el tamaño del tablero con la variable N, inicializamos el contador de soluciones = 0 y creamos un arreglo para almacenar los tiempos de ejecución de cada solución.

Paso 2.- Creamos la función *"imprimirTablero"* para imprimir el tablero (que recibe como parámetro una matriz llamada "tablero") en el que podremos visualizar las soluciones encontradas. La función utiliza dos bucles FOR anidados para iterar sobre cada fila y columna del tablero. En cada posición, imprime el valor contenido en esa posición (0 ó 1, donde el 1 representa que hay una reina en la casilla), seguido de un espacio en blanco. Después de imprimir toda una fila, se agrega una nueva línea con (print()), lo que representa visualmente un tablero tradicional de ajedrez.

Paso 3.- Creamos una función *"esPosicionValida"* que nos permita verificar si se puede colocar una reina en la posicion (fila, columna) del tablero (estos son los tres parámetros que recibe la función), haciendo primero que verifique si ya hay una reina en la misma columna, recorriendo las filas y así hacer la verificación, retornando *false* si encuentra una reina (1) en la posición [i][col].

Después verificamos si hay alguna reina en la diagonal principal superior usando la función *zip* para iterar simultaneamente entre las filas y las columnas de la diagonal y si encuentra una reina (1) en la posición [i][j] retorna *false* ya que no puede poner una reina en esa posición.

Por consiguiente también tenemos que verificar la diagonal secundaria superior, usando la misma lógica que la verificación anterior, igualmente utilizando la función *zip* para iterar simultáneamente. Si esto se cumple, retornamos *true* para concluir que podemos poner una reina en la posición (fila, columna) luego de verificar que no se ataquen.

Paso 4.- Continuamos creando funciones; esta vez una para encontrar las soluciones. Iniciando verificando si ya se colocó una reina en cada una de las filas del tablero, si es así, significa que se ha encontrado una solución y el contador de las soluciones incrementa.

Continuando con colocar una reina correspondiente a las columnas de la fila actual que se está iterando y una vez que la pone pasa a la siguiente fila, todo esto lo hace verificando la función *"esPosicionValida"* recibiendo como parámetros el tablero, la fila y el iterador i del for, cabe aclarar que el for se itera dentro del rango del tamaño del tablero (N). También en este punto, medimos el tiempo en la que el algoritmo tarda en encontrar cada solución importando primeramente la librería *time* y agregando las soluciones en una variable global *"tiemposPorSolucion"* para poder graficarlos posteriormente.

Paso 5.- Por último, definimos la parte principal del programa, el cual utiliza todas las funciones definidas anteriormente para poder encontrar las soluciones al problema. Inicializando el tablero (NxN) en una matriz de ceros, para después medir el tiempo global del algoritmo,

es decir, el tiempo que tarda en ejecutarse para encontrar cada una de las soluciones y se hace una evaluación con un IF, el cual verifica: si la variable soluciones = 0, nos dirá que no encontró soluciones, si esto no es así, imprime la cantidad de soluciones que se encontraron y el tiempo global o total que habíamos medido con anterioridad.
Para terminar, graficamos las soluciones encontradas usando la librería *matplotlib.pyplot as plt* definiendo que en el eje x de la gráfica estará el numero de solución encontrada y en el eje y el tiempo que se tardó.

- **2.4.2. Implementación con Poda por Factibilidad.**

¿Qué es la Poda por Factibilidad?
El método de "poda por factibilidad" se refiere a una técnica utilizada en el contexto de la optimización y la resolución de problemas multi-objetivo. Este método es útil para reducir significativamente la cantidad de soluciones obtenidas en un problema multi-objetivo. Se aplica en el ámbito de algoritmos de ramificación y poda, los cuales son utilizados para resolver cuestiones o problemas de optimización. La técnica de ramificación y poda se interpreta como un árbol de soluciones, donde cada rama conduce a una posible solución posterior a la actual, y la "poda por factibilidad" es uno de los criterios

de poda utilizados en este contexto

Paso 1.- Para iniciar con la implementación de esta estrategia, definiremos el tamaño del tablero con la variable N, inicializamos el contador de soluciones = 0 y creamos un vector para almacenar los tiempos de ejecución de cada solución encontrada.
Paso 2.- Después, como en la implementación anterior, utilizaremos una función para imprimir el tablero, lo que nos permitirá visualizar las soluciones encontradas.
Paso 3.- Prosiguiendo, usaremos la función ya utilizada anteriormente *"esPosicionValida"* para corroborar si podemos poner una reina en la posición (fila, columna) del tablero, analizando las columnas y las diagonales; sin embargo, hay una modificación notable que nos ayudará a reducir las veces que el algoritmo analiza cada fila buscando reinas ya colocadas. Para esto, verificaremos si ya hay una reina en la fila actual (si hay un '1'), si esto es así, ya no analiza esa fila y pasa a la siguiente; por consiguiente, si no hay una reina en la fila actual, el algoritmo hará la iteración simultánea entre filas y columnas del tablero para colocar la reina adecuadamente.
Paso 4.- Posteriormente recurriremos a la creación de la función *encontrarSolucionesconPoda* para encontrar las soluciones que cumplen con las evaluaciones antecesoras; para eso, usaremos la mismo lógica que en la implementación inicial verificando si ya se colocó una reina en cada una de las filas del tablero, si es así, significa que

se ha encontrado una solución y el contador de las soluciones incrementa y, obviamente hacemos que se coloque una reina adecuadamente en cada fila para que encontremos una solución válida.

Paso 5.- Por último, definimos la parte principal del programa, en la cual utilizamos todas las funciones definidas anteriormente para poder encontrar las soluciones al problema. Inicializando el tablero (NxN) en una matriz de ceros, para después medir el tiempo global del algoritmo, es decir, el tiempo que tarda en ejecutarse para encontrar cada una de las soluciones, procediendo a hacer una evaluación con un IF, el cual verifica: si la variable soluciones = 0, nos dirá que no encontró soluciones, si esto no es así, imprime la cantidad de soluciones que se encontraron y el tiempo global o total que habíamos medido con anterioridad.

Para culminar la implementación, graficaremos el tiempo de cada solución, definiendo nuestro eje X de la gráfica como el número de soluciones encontradas y en el eje Y el tiempo de ejecución.

- **2.4.3. Implementación con Heurística/Algoritmo "minConflicts".**

Pero, ¿qué es la heurística Min-Conflicts?

Una heurística es un procedimiento que intenta producir soluciones buenas o aproximadas a un problema de forma rápida pero que carece de garantías teóricas. En el contexto de la resolución de un MIP, la heurística es un método que puede producir una o varias soluciones, cumpliendo con todas las restricciones y con todas las condiciones de integralidad, pero que carece de una indicación de si se ha encontrado la mejor solución posible.

Estas soluciones de heurística integradas en la ramificación y corte obtienen las mismas ventajas en cuanto a la prueba de optimalidad que cualquier solución producida por la ramificación, y en muchos casos, pueden acelerar la prueba de optimalidad final, o bien pueden proporcionar una solución subóptima pero de alta calidad en menor cantidad de tiempo que la que lleva solamente la ramificación. Con valores de parámetro predeterminados, CPLEX invoca automáticamente a la heurística cuando parece ser beneficiosa.

Min-Conflicts. *Dada esta breve introducción, el algoritmo de Min-Conflicts es un algoritmo de búsqueda o método heurístico para resolver problemas de satisfacción de restricciones. Dada una asignación inicial de valores a todas las variables de un problema de satisfacción de restricciones, el algoritmo selecciona aleatoriamente una variable del conjunto de variables con conflictos que vi-*

olan una o más de sus restricciones. Luego asigna a esta variable el valor que minimiza el número de conflictos. Si hay más de un valor con un número mínimo de conflictos, elige uno al azar. Este proceso de selección de variables aleatorias y asignación de valores de min-conflict se iterará hasta que se encuentre una solución o se alcance un número máximo de iteraciones preseleccionado.

Paso 1.- Comenzaremos la implementación de la tercera estretegia de implementación, definiendo el tamaño del tablero con la variable N, inicializaremos el contador de soluciones = 0, y declareremos dos arreglos; el primero, para guardar los tiempos de ejecuión; el segundo, para guardar las soluciones encontradas.
Paso 2.- Posteriormente, nuevamente, utilizaremos la función de imprimir el tablero.
Paso 3.- Construimos la función *contarConflictos*, en donde contabilizamos la cantidad de reinas que entran en conflicto entre sí dada una posición específica. Para esto definimos una variable conflictos = 0, para después evaluar si una reina se encuentra en la misma fila, columna o diagonal, utilizando la función *abs* para calcular el valor absoluto o, en este caso, la distancia o diferencia de posición entre la reina en i y la reina en fila.
Luego, verificamos si hay una reina en la misma columna (col), si tablero[i][col] = 1 significa que hay una reina en

esa posición y no podremos ponerla.
Por otro lado, usaremos nuevamente la función *abs* para calcular la diferencia entre las posiciones de las columnas entre la reina en la fila *i* y la reina entre la reina en *fila*, luego obtenemos la columna en donde se encuentra la reina en la fila *i*. En resumen, la función *contarConflictos* utiliza el valor absoluto para calcular diferencias en posiciones y verifica si hay reinas en la misma columna o en diagonales y devuelve, si existen, los conflictos encontrados para cada reina en la posición especificada.
Paso 4.- Se definen una nueva funcion *encontrarMejorPosicion* en donde principalmente se declaran dos variables *conflictosMinimos y posicionesMinimas (esta última como un arreglo),* estas variables se utilizan para mantener un seguimiento de la cantidad mínima de conflictos encontrada hasta el momento y las posiciones que tienen ese mínimo número de conflictos. Inicialmente, se establece *conflictosMinimos* en infinito (float('inf')) para que sí o sí cualquier cantidad de conflictos encontrada sea menor. Después iteramos sobre todas las posibles posiciones en la fila actual del tablero (N), para así poder llamar a la función *contarConflictos* para obtener la cantidad de conflictos que se obtendrían por colocar una reina en la posición i de la fila *fila (conflictos = contarConflictos(tablero, fila, i)).*
Prosiguiendo a hacer unas comparaciones que tienen la siguiente lógica:
Si conflictos es menor que conflictosMinimos (es decir si

conflictos $< \infty$) actualiza *conflictosMinimos* a conflictos y reinicia el arreglo *posicionesMinimas* con la posición i. Por otro lado, si conflictos = *conflictosMinimos* se agrega la posición *i* a la lista *posicionesMinimas* ya que es una posición en la que no tiene conflictos. Por último el algoritmo retorna/elige una posición aleatoria de las *posicionesMinimas* en el tablero (con la funcion *random.choice*).

Paso 5.- Declaramos una función *inicializarTablero*, la cual devuelve una matriz de tamaño NxN representando un tablero de ajedrez, donde cada celda contiene aleatoriamente un 0 o un 1 y esto lo logramos con la función *random.choice*.

Paso 6.- Creamos una nueva función *minConflicts* la cual resuelve el problema de las N reinas utilizando la heurística de Min-conflicts.Para lograr esto, inicializa un tablero aleatorio y realiza una serie de movimientos para minimizar los conflictos en cada fila. Se inicializa un contador global soluciones para rastrear el número de soluciones encontradas. Se inicializa una variable *tablero* con la llamada a la función *inicializarTablero*. Después se ejecuta un bucle externo que controla el número de movimientos que se realizarán para minimizar conflictos en el tablero. Dentro de la función, también se itera sobre cada fila del tablero y se encuentra la mejor posición para la reina en esa fila utilizando la función *encontrarMejorPosicion*. Se actualiza la fila en el tablero para colocar

la reina en la mejor posición encontrada. Después de realizar los movimientos, se verifica si el tablero resultante es válido utilizando la función *esTableroValido*, la cual vamos a explicar en un momento. Si el tablero es válido, se imprime y se incrementa el contador de soluciones y se devuelve el tablero. En caso contrario, se devuelve *None*.

La función *esTableroValido* se utiliza un bucle anidado para iterar sobre todas las celdas del tablero (filas y columnas). Si encuentra una reina en una posición (fila, columna), es decir *tablero[fila][col] == 1* y hay al menos un conflicto en esa posición (según la función contarConflictos) o sea *contarConflictos(tablero, fila, col) ¿ 0*, retorna False,es decir, no son validas esas posiciones. Caso contrario, si no encuentra ningún conflicto en todo el tablero, retorna True, lo cual indica que son posiciones válidas para colocar reinas.

Paso 7.- Finalmente definimos la parte principal del programa, en la cual utilizamos todas las funciones definidas anteriormente para poder encontrar las soluciones al problema. Aquí se ejecuta un bucle hasta que se encuentren las soluciones que nosotros indiquemos (en este caso 100), al problema de las N reinas utilizando la heurística propuesta Min-conflicts. En cada iteración, se llama a la función *minConflicts* para encontrar una solución y se asigna el tablero resultante a la variable tablero. Culminando con la gráfica de cada solución encontrada.

- **2.5 Comparación de rendimiento con Gráficas".**

El funcionamiento de las estretégias para distintos valores de N crecen de manera acelerada, dependiendo del valor N; funcionando optima y eficientemente con valores entre 3 < N < 13. Por ejemplo: con N = 4, hay 2 soluciones; con N = 8, 92 soluciones; con N = 10, 724 soluciones; y con N = 12, 14200 soluciones. Como podremos deducir, con valores mayores que 12, las soluciones aumentarán, lo que ocasionará que el programa falle o se tarde un tiempo excesivamente largo.

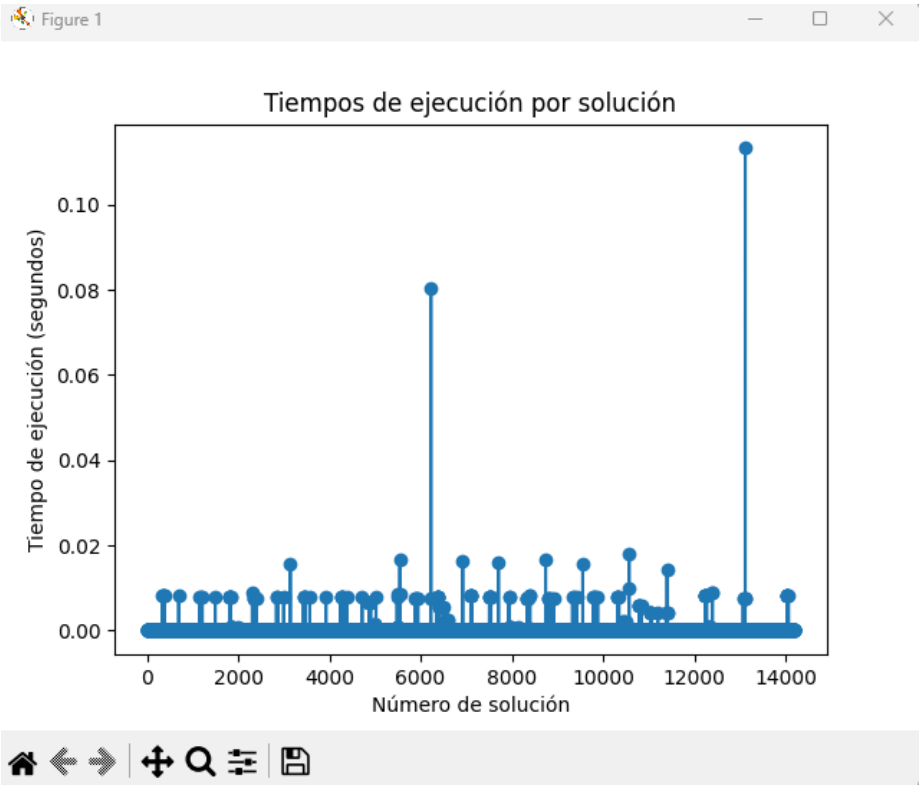

Figure 2: Gráfica de implementación inicial. Aprox 11 segundos.

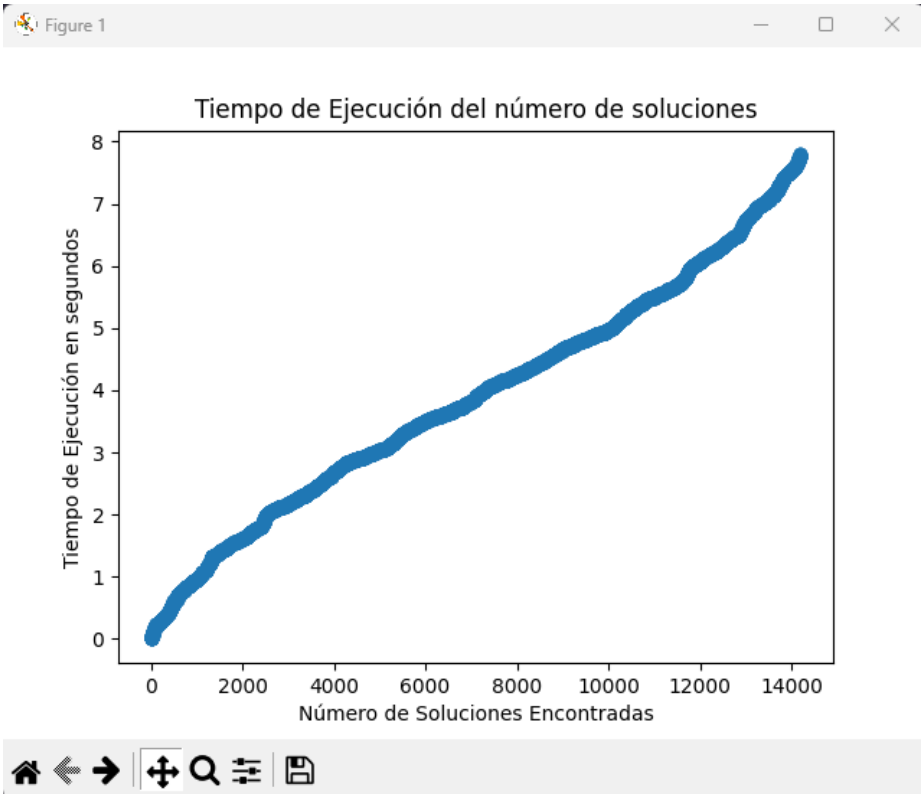

Figure 3: Gráfica de implementación con Poda. Aprox 8 segundos.

Nota: Si la gráfica usa un trazo ancho, es por la cantidad de puntos graficados.

3 Conclusión.

Como hemos podido observar en el anterior reporte, el problema de las N-Reinas puede ser resuelto de diferentes maneras, algunas más eficientes que otras, pero al final de cuentas hay una resolución; para poder elegir la que más nos convenga, es necesario examinar las complejidades de cada estrategia, sin embargo, también

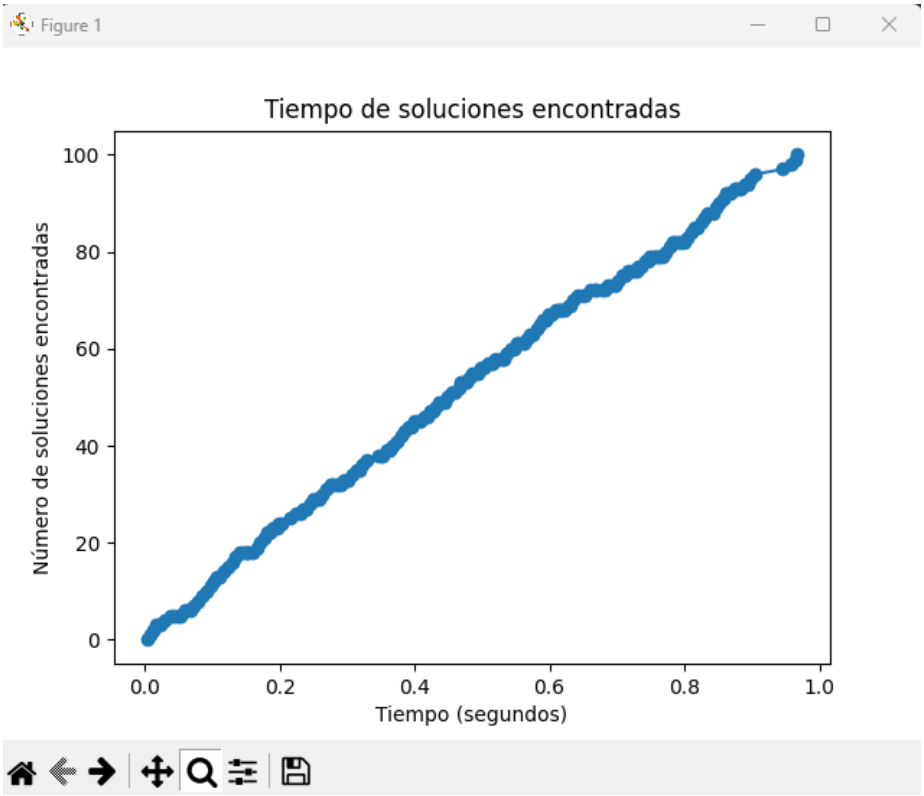

Figure 4: Gráfica de implementación con Min-Conflicts. Aprox 1 segundos.

hay que considerar el tamaño del tablero, ya que, como hemos visto, con la implementación inicial del algoritmo es difícil encontrar todas las soluciones de un tablero de N $>$ 10 MAX 12, ya que tardaría muchísimo tiempo en encontrar todas las soluciones gracias a la complejidad O(N!), lo cual hace que crezcan rapidísimo el número total de soluciones.

4 Referencias Bibliográficas.

[1]. https://www.ibm.com/docs/es/icos/12.9.0?topic=heuristics-what-are

[2]. Federico Barber y Miguel A. Salido. Introducción a la Programación de Restricciones. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial. Nº 20, pp. 13-30, 2003

[3]. Minton, Steven; Mark D. Johnston; Andrew B. Philips; Philip Laird (1990). "Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method" (PDF). Eighth National Conference on Artificial Intelligence (AAAI-90), Boston, Massachusetts: 17–24. Retrieved 27 March 2013.

[4]. http://portal.amelica.org/ameli/journal/595/5952866029/html/