Facultad De Ciencias - UCV
Escuela de computación
CI: 30391915
Nombre: Rafael Eduardo Contreras

Tarea #2 - N-reinas++ y solución probabilística

Problema N-reinas

El problema N-reinas es un puzzle matemático y de ajedrez que desafía a colocar N reinas en un tablero de ajedrez de N×N sin que ninguna reina amenace a otra. Las reinas son piezas de ajedrez que pueden moverse cualquier número de casillas verticalmente, horizontalmente, o diagonalmente. La amenaza se define por la capacidad de una reina de capturar a otra en su siguiente movimiento según las reglas del ajedrez.

El objetivo es encontrar una disposición de N reinas en el tablero tal que ninguna reina esté en posición de amenazar a otra. Esto implica que no puede haber más de una reina en cualquier fila, columna, o diagonal.

Solución estandar

La solución determinística para el problema N-reinas utiliza un enfoque de backtracking. En nuestra solución, en cada llamada pasamos por argumento el índice de la fila donde colocaremos una reina, y decidimos en qué columna ponerla.

Clases:

- NQueensBase: Utilizamos una clase base (NQueensBase) que implementa la función principal para la solución. Esta es la función “bool solve(int step)”. Esta es una clase abstracta en donde no está implementada la función “vector<int> candidates()”, esta función debe retornar las columnas candidatas donde agregar reinas en cada llamada.
- NQeensFull: Esta clase implementa la función “candidates” de manera que se intente usar todas las columnas posibles para encontrar una solución. A su vez, hereda de NQueensBase y se puede crear una instancia para resolver de forma estándar el problema de N reinas.

Estructura de Datos: Utilizamos varios arreglos para llevar un registro del estado del tablero:

- vector<bool> row: Indica si una fila está ocupada por una reina.
- leftD[] y rightD[]: Representan las diagonales del tablero. Estos arreglos ayudan a verificar rápidamente si una diagonal está ocupada. Son vectores de booleanos.
- vector<int> mat: Guarda las posiciones de las reinas en el tablero, “mat[i]” sería equivalente a la columna donde está la reina, e “i” a su fila.

La función solve() es recursiva y trabaja fila por fila. Para cada fila, intenta colocar una reina en una columna segura (no amenazada por otra reina). Utiliza la función validPos(int step,int i) para comprobar si la posición deseada no está amenazada. Si una reina se coloca con éxito, la función procede a la siguiente fila llamando “solve” con “step + 1”. Si llega a la fila N (base del caso recursivo), significa que todas las reinas han sido colocadas correctamente.

Cuando no es posible colocar una reina en ninguna columna de la fila actual, la función retrocede, eliminando la reina de la última posición segura y probando la siguiente.

Solución tomando un 30% aleatorio

La solución no determinística para el problema N-reinas usa el mismo enfoque y pasos que la solución determinística. La diferencia es que en la implementación de “candidates”, esta selecciona de manera aleatoria el 30% de candidatos posibles. Solo es necesario implementar una clase que herede NQueensBase e implementar esta función.

Clases

- NQeensPartial: Esta clase implementa la función “candidates” de manera que se seleccionen solo un 30% de las columnas posibles. Este porcentaje puede ser modificado en el constructor. Hereda de NQueensBase y resuelve el problema tomando solo el 30% de candidatos (aleatorios) en cada llamada.

Funciones importantes:

- ran3: Se utiliza esta implementación de una función pseudoaleatoria.
- nextPos: Esta función utiliza ran3 para generar un número random entre 0 y 1. Luego de esto se multiplica por 100000 (N nunca será mayor a 100000) y se extrae el módulo N. De esta manera obtenemos un número random entre 0 y n.
- candidates: crea una lista con el tamaño de 0.3 * N. Además crea un vector de tamaño N para saber cuáles números ya hemos generado, los ya generados se descartan para no repetir ese camino.

La probabilidad con la que se encuentra una solución depende de que tan grande sea N. En el archivo “results.csv” se encuentran los resultados de varias pruebas dadas con diferentes N. De n=8 a n=30 se tienen 50 intentos y su tiempo promedio. De n=36 a n=40 se tienen 20 intentos a causa del largo tiempo que tarda en terminar la corrida de los benchmarks.

Resultados

Revisar el archivo results.csv para los resultados de la solucion aleatoria. Revisar el archivo standard.csv para los resultados de la solucion estandar.

Al ver los resultados nos damos cuenta de que para N < 20 la probabilidad de éxito es muy baja. Aunque para 20 <= N <= 23 es considerable, y para N >= 24 es 100%. Evidentemente, al solo tomar en cuenta el 30% de los resultados, la solución es mucho más rápida que la tradicional, aunque no tan precisa. Aun así el tiempo para encontrar una solución también crece de manera exponencial en la solución que toma el 30% aleatorio.

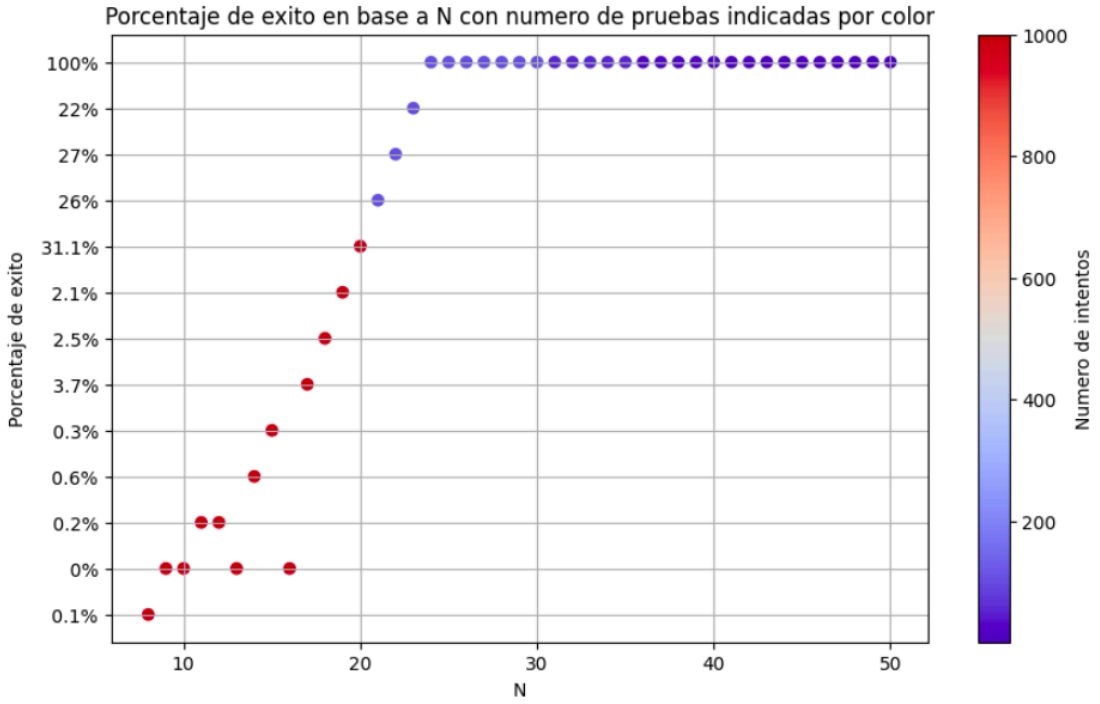
La razón por la cual la probabilidad de encontrar una solución aumenta con N es debido a que la cantidad de soluciones candidatas también crece exponencialmente con la cantidad de combinaciones de reinas posibles. Mientras N crece, el espacio de soluciones posibles también, lo que incrementa la probabilidad de que al tomar el 30% de columnas candidatas al azar se consiga alguna que sea una solución correcta.

En la siguiente tabla se aprecia como crece con N el espacio de soluciones posibles.

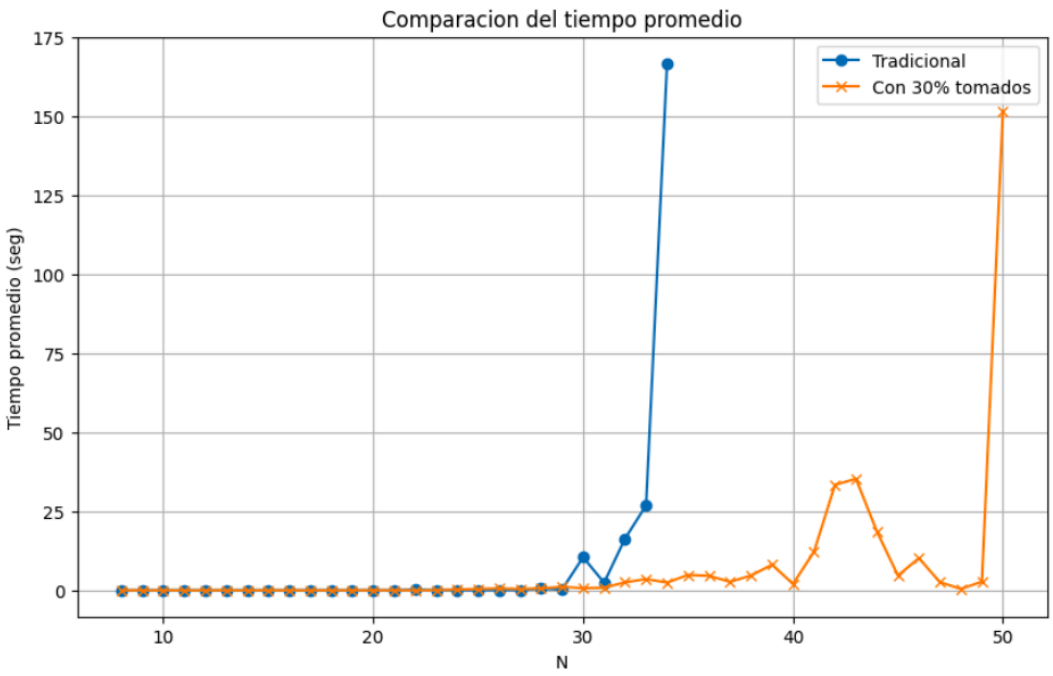
N	Soluciones candidatas
8	92
10	724
12	14,200
15	2,279,184
20	39,029,188,884
25	2,207,893,435,808,352

En el siguiente gráfico se muestra como crece la probabilidad de encontrar una solución mientras crece N. En el eje Y se tiene la probabilidad, en el X se tiene N. Además los puntos cambian de color dependiendo de cuantas pruebas fueran realizadas para este N, podría decirse que muestra que tan fiable es este dato.

Se puede apreciar como la probabilidad de encontrar una solución aumenta exponencialmente. Incluso podría reducirse la cantidad de candidatos a 10% y aún encontrar soluciones para los números más altos.



En el siguiente gráfico tenemos una comparación del tiempo de ejecución de ambos algoritmos. Aquí se puede apreciar la naturaleza exponencial del problema, donde los tiempos crecen de manera muy rápida a medida que el espacio de búsqueda se agranda. Además, podemos ver que la solución tomando solo un 30% es mucho más rápida y es capaz de abordar un N más grande. Aun así, el tiempo que toma la solución más rápida sigue creciendo de manera exponencial mientras N aumenta.


Comparacion del tiempo promedio
Tradicional
Con 30% tomados
Tiempo promedio (seg)
N
175
150
125
100
75
50
25
0
10
20
30
40
50