2021

Problema de las 8 reinas MEMORIA

PARADIGMA PROBABILISTA

MARCOS MOLINA RUBIO
ALFREDO LLOPIS

Índice

1. **Código C# explicado**

Archivo “Program.cs”:

Nuestro código usa el paradigma probabilístico para mejorar la velocidad de resolución del problema de las 8 reinas. Para ello, consta de una lista de tuplas para poder guardar cada una de las coordenadas X e Y de cada reina que sabemos que no se amenazan entre si. Las funciones usadas son:

- static List<Tuple<int, int>> Bacotraco(List<Tuple<int, int>> coordenadas, int x)
  Este es el algoritmo de Backtracking, devuelte la lista de tuplas con las reinas usadas. Primero, comprueba si la lista tiene 8 elementos, en cuyo caso ya se han encontrado las 8 reinas. Si no, realiza dos bucles anidados que inicializa la X en la posición siguiente a la última posición asignada e inicializa la Y a 0 para recorrer todo el tablero. Luego llama a ComprobarPosicion para saber si la posición dada por el bucle amenaza a alguna reina del tablero. En caso de que no, guarda la lista antigua, añade a la lista esa posición, comprueba si ya están todas las reinas y si no hace la llamada recursiva para encontrar el siguiente elemento. Después de la llamada, el programa comprueba si la lista que recibe de la llamada recursiva es igual a la lista antigua guardada; en caso de que así sea, significa que no ha encontrado las 8 reinas con esa ruta por lo que se elimina las nuevas coordenadas de la lista. En caso de que no se cumpla esa condición se comprueba otra vez si la lista tiene 8 reinas y si si se devuelve la nueva lista recibida. En caso de terminar el bucle, comprobamos si hay 8 reinas en la lista y en caso de que no eliminamos un elemento de la lista y lo devolvemos, para indicar a la llamada anterior que no hemos encontrado la solución por esa ruta ya que será la misma que la lista de reinas antigua de la llamada anterior.
- static void PrintCoordenadas(List<Tuple<int, int>> coordenadas)

Tan solo es una función que escribe por consola las coordenadas finales de la solución del algoritmo de forma clara, ya que la función Bacotraco escribe todas las posibilidades que ha barajado. De esta forma la solución queda clara al final del programa.

- static bool ComprobarPosicion(List<Tuple<int, int>> CoordenadasReinas, int x, int y)
  Llama a las 3 funciones a continuación. Hace un bucle que recorre las coordenadas que ya están guardadas y comprueba si estas amenazan alguna reina de las guardadas. En caso de que si devuelve True, en caso de que no choque con ninguna, devuelve False.
  - static bool ComprobarReinas(Tuple<int, int> reina, int x, int y)
    Comprueba si las coordenadas X e Y dadas coinciden con la posición de la reina dada en la tupla.
  - static bool ComprobarDiagonales(Tuple<int, int> reina, int x, int y)
    Comprueba si la posición (X,Y) está en la diagonal de las coordenadas de la reina dada. Está implementado con un cuatro bucles que manejan dos variables cada uno, son cuatro porque es uno por cada dirección diagonal (superior izquierda, derecha, inferior izquierda y derecha).
  - static bool ComprobarFilasYColumnas(Tuple<int, int> reina, int x, int y)
    Comprueba si las filas y columnas de las coordenadas de la reina dada coinciden con la posición X e Y dada.
- static void Main(string[] args)
  La función Main() primero pregunta al usuario cuantas posiciones aleatorias quiere crear antes de realizar la resolución con backtracking. Por defecto es 2 ya que es la solución más óptima como podemos ver en este gráfico:

No Reinas	*p*	*v*	*f*	*t*
0	1,0000	114,00	0,00	114,00
1	1,0000	39,63	0,00	39,63
2	0,8750	22,53	39,67	28,20
3	0,4931	13,48	15,10	29,01
4	0,2618	10,31	8,79	35,10
5	0,1624	9,33	7,29	46,92
6	0,1357	9,05	6,98	53,50
7	0,1293	9,00	6,97	55,93
8	0,1293	9,00	6,97	55,93

p-> probabilidad de éxito
v-> número esperado de nodos a visitar en caso de éxito
f-> número esperado de nodos en caso de fracaso

Después de elegir cuantas coordenadas se resolverán de forma aleatoria, genera este numero de posiciones aleatorias e inicia el algoritmo de backtracking, llama a PrintCoordenadas para mostrar la solución y, por último, ejecuta el form del tablero pasándole las coordenadas solucion halladas anteriormente para representarlo graficamente.

Archivo "Form1.cs":

Este Form recibe las coordenadas de las reinas en el tablero y las representará en un tablero de ajedrez.

En un único bucle, generará casillas de ajedrez que estarán vacías a no ser que en esa coordenada haya una reina. Estando vacía cargará un elemento Panel. Habiendo una reina cargará un elemento PictureBox con el png correspondiente. Ambos serán del mismo tamaño.

Luego, según su posición en el tablero de ajedrez, asignará un color de fondo a cada elemento según corresponda: blanco o gris oscuro.

2. Explicación del paradigma

La programación probabilística (PP) es un paradigma de programación en el que se especifican modelos probabilísticos y la inferencia a partir de estos modelos se realiza automáticamente. Representa un intento de unificar el modelado probabilístico y la programación tradicional de propósito general para hacer que el primero sea más fácil y más aplicable. Este paradigma se puede usar para crear sistemas que ayuden a tomar decisiones ante la incertidumbre.

Esto, aplicado al problema que hemos realizado nosotros, significa que a un algoritmo de back-tracking que sabemos que tardaría un montón en resolver un problema le añadimos un factor de aleatoriedad, es decir, conseguir unas pocas soluciones de forma aleatoria y a partir de ahí empieza el back-tracking. De esta forma conseguimos reducir el tiempo que se tarda en realizar el problema y dependiendo de cuantas soluciones aleatorias resuelvas la probabilidad de encontrar la solución puede ser casi igual que si solo usásemos un algoritmo clásico de back-tracking.

3. Complejidad computacional del algoritmo

Al igual que todos los algoritmos de backtracking que hemos dado, este tiene complejidad ^n. Debido a que hay un numero variable de opciones que se hayan de forma probabilística, introduciremos la variable p para indicar el número de coordenadas encontradas de forma aleatoria. Dado que hay una reina por cada fila/columna, podemos entender que hay 8 opciones, por lo tanto, la complejidad final de este algoritmo sería:

$$\theta(n) = 8^{(n-p)}$$

El caso peor sería que p = 0, donde la complejidad sería la más alta, 8^n.

En este paradigma, el caso mejor también es el caso menos probable, donde p = 8 y con un 0,1293 de probabilidad consigue encontrar la solución son una complejidad igual a 1.

4. **Conclusiones del alumno (riesgos y ventajas de cada paradigma)**

En general, hemos visto que implementar un algoritmo de este tipo es como realizar un algoritmo de backtracking normal y añadirle la opción de crear de forma aleatoria un numero pequeño de soluciones aleatorias mejorando el rendimiento del programa.

En cuanto a las ventajas y riesgos de este paradigma, vemos claramente (y por eso nuestro programa tiene la opción de elegir cuantos elementos aleatorios escoger) que cuantas más posiciones aleatorias elijas, como hemos podido ver en la imagen del apartado 1. De hecho, a partir de 2 elementos aleatorios la probabilidad de encontrar una solución cae drásticamente y con más de 4 elementos es prácticamente imposible. Por lo tanto, aunque aumenta la velocidad de encontrar una solución, hay que tener cuidado porque a veces no encuentra ninguna solución.

El otro paradigma, el de Ramificación y Poda, se encarga de ir eliminando opciones que sabemos que no van a resultar en la solución final. Nosotros no hemos realizado esto de la manera tradicional, pero nuestros bucles anidados que recorren el tablero no lo recorren entero, ya que si hay elementos por encima esos no los baraja como opciones. Aunque una forma más clásica de implementar este paradigma sería guardar las posiciones x e y donde podemos colocar mas reinas y solo comprobar esas opciones. Se nos ocurre que quizá dos arrays de 8 elementos que cada vez se reduce su tamaño podría ser una opción. De esta forma, aunque realizas mas operaciones de manera individual, en cada llamada hay que hacer una comprobación, a nivel general ganas tiempo ya que no pruebas tantas opciones. El objetivo de este algoritmo es descartar las

opciones que de antemano sabemos que no van a funcionar, por ello se llama poda.

La manera más óptima sería poder implementar ambos paradigmas para poder hacer un programa lo mas eficiente posible y que encuentre una solución válida lo mas cercana posible a la solución óptima lo mas rápido que se pueda.

5. Bibliografía

"ALGORITMOS DE LAS VEGAS - webdelprofesor.ula.ve."
http://www.webdelprofesor.ula.ve/ingenieria/ibc/dyaa/c26lasVegas.pdf.

"Programación probabilística - Wikipedia, la enciclopedia libre."
https://es.wikipedia.org/wiki/Programaci%C3%B3n_Probabil%C3%ADstica.

“Console Output forma WinForms - csharp411.com”
https://www.csharp411.com/console-output-from-winforms-application/

“Creating a Chess Board using Windows Forms - stackoverflow.com”
https://stackoverflow.com/questions/6733310/creating-a-chess-board-using-windows-forms