INTELIGENCIA ARTIFICIAL

Trabajo práctico Búsqueda local

Curso 2018-2019

Autores:
Rubén Rodríguez Esteban 737215

1. Resumen

El trabajo ha consistido en la realización de dos tareas. La primera tarea tiene por objeto familiarizarse con el código en java para resolver problemas de búsqueda local y problemas de optimización. Se ha empleado el código aima-core facilitado por el profesorado de la asignatura, y que ha sido utilizado también en prácticas anteriores. El trabajo ha consistido básicamente en la familiarización con el algoritmo Hill-Climbing y en la implementación de sudokus.

Para ello se ha tomado como punto de partida del código de la clase NQueensDemo.java en el que se muestra la resolución del problema de las 8-reinas mediante distintos algoritmos de búsqueda en profundidad, DLS, IDS, Enfriamiento Simulado (*SimulatedAnnealing*), Escalada (*Hill-Climbing*), y Algoritmos genéticos (*GeneticAlgorithm*). Sin embargo dicho código ha tenido que ser modificado para adaptarse a las especificaciones del trabajo a realizar.

2. El problema de las N-Reinas

Una vez comprendida la implementación del fichero *NQueensDemo.java* se ha procedido a reformular la definición del problema de acuerdo a las normas de especificación del trabajo. En primer lugar se ha creado un fichero denominado *NQueensTP6P1.java,* donde se ha copiado y pegado el código del fichero *NQueensDemo.java.* Posteriormente en la construcción del problema se ha modificado de manera que en vez de crearse los estados iniciales de manera incremental, es decir, añadiendo paulatinamente reinas sobre un estado vacio, *NQueensFunctionFactory.getIActionsFunction(),* se ha empleado el método *NQueensFunctionFactory.getCActionsFunction()* para partir de un estado con todas las reinas generadas ya que se debe comenzar la búsqueda desde un estado inicial completo.

Para construir los tableros iniciales se ha modificado el fichero *NQueensBoard.java*, de manera que los tableros se generan tal y como muestra el siguiente fragmento de código escrito en lenguaje java:

```
// Creación de los tableros
public NQueensBoard(int n) {
	size = n;
	squares = new int[size][size];
	for (int i = 0; i < size; i++) {
		for (int j = 0; j < size; j++) {
			squares[i][j] = 0;
		}
	}
```

```
        boolean vacio;
        Random r = new Random();

        // Inserción de las reinas, una por fila y columna aleatoria
        for (int i = 0; i < size; i++) {
                vacio = false;
                while (!vacio) {
                        int c = r.nextInt(size);
                        if (squares[i][c] != 1) {
                                vacio = true;
                                squares[i][c] = 1;
                        }
                }
        }
}
```

De acuerdo con el código mostrado anteriormente, para insertar las reinas primero se rellena el tablero vacío y posteriormente se insertan las reinas a razón de una por fila y en columnas aleatorias. La razón por la que se ha usado este método es porque inicialmente se insertaban las reinas en columnas y fila aleatorias, sin embargo, la tasa de error era del 95% aproximadamente mientras que con la otra técnica es de torno a un 84%.

Para evitar que se generarán tableros repetidos se ha usado una LinkedList donde se han ido almacenando todos los tableros generados. De esta manera al generar un tablero se compara con todos los tableros empleados en métodos anteriores comprobando las posiciones, fila y columna, de todas las reinas puestas en el tablero.

Una vez generados los tableros se ha procedido a implementar dos métodos denominados, *nQueensHillClimbingSearch_Statistics(intnumExperiments) y un método adicional muy parecido* llamado *nQueensRandomRestartHillClimbing(). El primer método* realiza *numExperimets* y muestre el porcentaje de éxitos, fallos, media de pasos al fallar y media de pasos en éxito, concretamente se han realizado 10000 experimentos a partir de estados iniciales aleatorios no repetidos de las 8-reinas con la búsqueda *HillClimbingSearch(),* y el segundo método reinicia el estado inicial hasta que se obtiene el éxito.

Un factor importante a tener en cuenta es que la comprobación de los tableros supondría un coste sumamente elevado ya que tiene que comparar los 10000 tableros sino fuera por la improbabilidad de la repetición de todas las posiciones de las reinas en los tableros.

En los ficheros java se ha comentado el código que se ha implementado durante la primera parte del trabajo y como trabaja, por lo que no se ha hecho más hincapié en este apartado.

3. Sudokus en clips

Para elaborar la implementación de los sudokus se ha partido de una plantilla en clips proporcionada por el profesorado de la asignatura. En dicha plantilla se han especificado todos los módulos y todas las restricciones empleadas en la implementación del código.

En dicha parte del trabajo se han implementado los patrones de búsqueda para poder lograr la solución de los sudokus. Paralelamente se han programada un conjunto de reglas o estrategias de resolución que permiten alcanzar un estado final, de esta forma se logra la resolución de sudokus mediante la propagación de restricciones. En el fichero *sudokus.clp* se ha descrito detalladamente el código de búsqueda y las estrategias de resolución.

4. Trazas de la solución

Esta salida se corresponde con una solución al ejecutar el problema de las N-Reinas

```
NQueensHillClimbing con 10000 estados iniciales diferentes -->
Fallos: 8543
Coste medio fallos: 3,08
Exitos: 1457
Coste medio exitos: 4,10

SOLUTION_FOUND
------Q-
---Q----
-Q------
----Q---
-------Q
Q-------
--Q-----
-----Q--

Numero de intentos: 7
Fallos: 6
Coste medio de fallos: 3,50
Coste exito: 3,00
```

Seguidamente se han mostrado capturas de dos de los 54 sudokus de ejemplo empleados para comprobar el buen funcionamiento del programa implementado

```
CLIPS> (resuelve "043080250600000000000001094900004070000608000010200003820500000000000005034090710")
Estado inicial
+---+---+---+
| 43| 8 |25 |
|6  |   |   |
|   |  1| 94|
+---+---+---+
|9  |  4| 7 |
|   |6 8|   |
| 1 |2  |  3|
+---+---+---+
|82 |5  |   |
|   |   |  5|
| 34| 9 |71 |
+---+---+---+
Estado final
+---+---+---+
|143|986|257|
|679|425|381|
|285|731|694|
+---+---+---+
|962|354|178|
|357|618|942|
|418|279|563|
+---+---+---+
|821|567|439|
|796|143|825|
|534|892|716|
+---+---+---+
```

```
CLIPS> (resuelve "48.3.....71.2.....7.5....6....2..8.....1.76...3.....4......5....")
Estado inicial
+---+---+---+
|48 |3  |   |
|   |   | 71|
| 2 |   |   |
+---+---+---+
|7 5|   | 6 |
|   |2  |8  |
|   |   |   |
+---+---+---+
|  1| 76|   |
|3  |   |4  |
|   | 5 |   |
+---+---+---+
Estado final
+---+---+---+
|487|312|695|
|593|684|271|
|126|597|384|
+---+---+---+
|735|849|162|
|914|265|837|
|268|731|549|
+---+---+---+
|851|476|923|
|379|128|456|
|642|953|718|
+---+---+---+
```