Trabajo TP6-1: Lenguajes de Reglas y Propagación de restricciones

Javier Herrer Torres

Noviembre 2020

1. Introducción

El trabajo consta de tres tareas. La primera tarea tiene por objeto familiarizarse con en el desarrollo de programas en CLIPS tanto escribiendo código como depurando programas. La segunda tarea consistirá en la resolución de sudokus mediante la propagación de restricciones. Finalmente, la tercera tarea combina la idea de búsqueda local con la propagación de restricciones en el algoritmo `min-conflicts`.

2. Primera Tarea

2.1. Problema 1 CLIPS

Se ha partido del programa `PuzzleAEstrella2.clp` presentado en las transparencias, y se ha modificado la representación del estado y los operadores. Los operadores `mover-dcha` y `mover-izq` reflejan la restricción de que como máximo se puede saltar sobre dos fichas.

```
(defrule OPERADORES::mover-dcha
   (nodo (estado $?a H $?b&:(<= (length$ $?b) 2) ?c $?d)
          (camino $?movimientos)
          (clase cerrado)
          (coste ?coste))

=>
   (bind $?nuevo-estado (create$ $?a ?c $?b H $?d))
   (assert (nodo
              (estado $?nuevo-estado)
              (camino $?movimientos (implode$ $?nuevo-estado))
              (coste (+ ?coste 1))
              (heuristica (heuristica $?nuevo-estado)))))

(defrule OPERADORES::mover-izq
```

```
   (nodo (estado $?a ?b $?c&:(<= (length$ $?c) 2) H $?d)
          (camino $?movimientos)
          (clase cerrado)
          (coste ?coste))

=>
   (bind $?nuevo-estado (create$ $?a H $?c ?b $?d))
   (assert (nodo
              (estado $?nuevo-estado)
              (camino $?movimientos (implode$ $?nuevo-estado))
              (coste (+ ?coste 1))
              (heuristica (heuristica $?nuevo-estado)))))
```

2.2. Problema 3 CLIPS

Se ha partido del programa `PuzzleCU.clp` presentado en las transparencias, y se ha modificado la representación del estado y los operadores. La definición del problema ha sido consultada en las últimas transparencias de la primera lección («Resolución de Problemas y Búsqueda»). Los operadores `Andar` y `Saltar` reflejan las restricciones de que no puede haber más acciones de saltar que de andar para llegar a un estado, y que no se permiten las acciones que mueven más allá de la casilla objetivo.

```
(defrule OPERADORES::Andar
   (nodo (casilla ?c&:(< ?c 8))
          (pasos ?p)
          (saltos ?s)
          (camino $?movimientos)
          (coste ?coste)
          (clase cerrado))
=>
   (bind ?nueva-casilla (+ ?c 1))
   (bind ?nuevos-pasos (+ ?p 1))
   (bind ?nuevos-saltos ?s)
   (assert (nodo
               (casilla ?nueva-casilla)
               (pasos ?nuevos-pasos)
               (saltos ?nuevos-saltos)
               (camino $?movimientos (
                       implode$ (create$ ?nueva-casilla
                                         ?nuevos-pasos
                                         ?nuevos-saltos)))
               (coste (+ ?coste 1))
          )
   )
)
```

```
(defrule OPERADORES::Saltar
   (nodo (casilla ?c&:(< ?c 5))
          (pasos ?p)
          (saltos ?s&:(< ?s ?p))
          (camino $?movimientos)
          (coste ?coste)
          (clase cerrado))
=>
   (bind ?nueva-casilla (* ?c 2))
   (bind ?nuevos-pasos ?p)
   (bind ?nuevos-saltos (+ ?s 1))
   (assert (nodo
               (casilla ?nueva-casilla)
               (pasos ?nuevos-pasos)
               (saltos ?nuevos-saltos)
               (camino $?movimientos (
                       implode$ (create$ ?nueva-casilla
                                         ?nuevos-pasos
                                         ?nuevos-saltos)))
               (coste (+ ?coste 2))
         )
   )
)
```

Además, dos estados $S_i = (C_i, P_i, S_i)$ y $S_j = (C_j, P_j, S_j)$ son iguales si $C_i = C_j$ y $P_i - S_i = P_j - S_j$. Si son iguales, se descartará el nodo de mayor coste.

```
(defrule RESTRICCIONES::repeticiones-de-nodo
    (declare (auto-focus TRUE))
    ?nodo-i <- (nodo (casilla ?c-i) (pasos ?p-i)
                     (saltos ?s-i) (coste ?coste-i))
    ?nodo-j<- (nodo (casilla ?c-i)
                    (pasos ?p-j)
                    (saltos ?s-j&:(eq (- ?p-i ?s-i) (- ?p-j ?s-j)))
                    (coste ?coste-j&:(> ?coste-i ?coste-j)))
    test (neq  ?nodo-i ?nodo-j))
 =>
   (retract ?nodo-i))
```

3. Segunda Tarea - resolución de sudokus mediante propagación de restricciones y búsqueda

Se ha añadido a la clase `SudokuVariable` el valor de la celda y las coordenadas x e y, además del nombre de la variable. Se ha escogido como `SolutionStrategy` la más completa: `MRV + DEG + AC3 + LCV`. Que entrelaza la búsqueda (**backtracking**), algoritmos de propagación de restricciones considerando diferente número de variables, como las restricciones sobre los valores de dos variables (**AC-3**, Arc consistency version 3), y **heurísticas de propósito general** como elegir primero las variables con menor número de valores posibles.

Se adjunta la traza de ejecución de la aplicación con un resultado de **156 sudokus solucionados** (`trazaSudokus.txt`). A continuación, se muestra el método `resolverSudoku`:

```
private boolean resolverSudoku(Sudoku sudoku, SolutionStrategy strategy) {
    sudoku.imprimeSudoku();

    if (sudoku.completo()) {
        System.out.println("SUDOKU COMPLETO");
    } else {
        try {
            System.out.println("SUDOKU INCOMPLETO - Resolviendo");
            SudokuProblem problem =
                new SudokuProblem(sudoku.pack_celdasAsignadas());

            strategy.addCSPStateListener(new CSPStateListener() {
                @Override
                public void stateChanged(Assignment assignment, CSP csp) {
//                  System.out.println("Assignment evolved : " + assignment);
                }

                @Override
                public void stateChanged(CSP csp) {
//                  System.out.println("CSP evolved : " + csp);
                }
            });

            double start = System.currentTimeMillis();
            Assignment sol = strategy.solve(problem);
            double end = System.currentTimeMillis();

            System.out.println(sol);
            System.out.println("Time to solve = " + ((end - start) / 1000) +
                               "segundos");
```

```
            System.out.println("SOLUCION:");

            Sudoku solucion = new Sudoku(sol);
            solucion.imprimeSudoku();
            if (solucion.correcto()) {
                System.out.println("Sudoku solucionado correctamente");
                return true;
            } else {
                System.out.println("Sudoku solucionado incorrectametente");
                return false;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    return false;
}
```

4. Tercera Tarea - Propagación de restricciones y búsqueda local

Se ha definido el problema en la clase `NQueensProblem`. A diferencia del problema anterior, se parte de un estado completo en el que se asigna una variable (reina) a cada columna (Q1, Q2, ..., Qn). Una `NQueensVariable` cuenta con un nombre Qi y con una columna. Después, a cada variable Qi se le asigna un dominio $[1, N]$.

Por último, se establece una restricción entre cada par de reinas Qi y Qj de forma que no se puedan atacar. Qi es una reina en la columna i, y Qj es una reina en la comuna j. **El valor de Qi y Qj son las filas en las que van a ser colocadas.** Para evitar que se ataquen entre ellas, en `NQueensConstraint` se evitan tres tipos de ataque:

1. Ataque vertical: es imposible ya que cada par de reinas está colocado en columnas distintas.

2. Ataque horizontal: se atacarán cuando estén en la misma fila ($Qi \neq Qj$).

3. Ataque diagonal: no puede haber la misma diferencia en columnas y en filas ($|i - j| \neq |Qi - Qj|$).

Se adjunta la traza de ejecución de la aplicación con un resultado de 50 nqueens solucionados (`trazaNQueens.txt`).

El **algoritmo min-conflicts**, dado una asignación inicial de valores a todas las variables del problema, selecciona aleatoriamente una variable que está violando una restricción y le asigna un valor que minimiza el número de conflictos.